

SafeTail: Tail Latency Optimization in Edge Service Scheduling via Redundancy Management

Jyoti Shokhanda, Utkarsh Pal, Aman Kumar, Soumi Chattopadhyay, *Senior Member, IEEE*,
Arani Bhattacharya, *Member, IEEE*

Abstract—Optimizing tail latency while efficiently managing computational resources is crucial for delivering high-performance, latency-sensitive services in edge computing. Emerging applications, such as augmented reality, require low-latency computing services with high reliability on user devices, which often have limited computational capabilities. Consequently, these devices depend on nearby edge servers for processing. However, inherent uncertainties in network and computation latencies—stemming from variability in wireless networks and fluctuating server loads—make service delivery on time challenging. Existing approaches often focus on optimizing median latency but fall short of addressing the specific challenges of tail latency in edge environments, particularly under uncertain network and computational conditions. Although some methods do address tail latency, they typically rely on fixed or excessive redundancy and lack adaptability to dynamic network conditions, often being designed for cloud environments rather than the unique demands of edge computing. In this paper, we introduce SafeTail, a framework that meets both median and tail response time targets, with tail latency defined as latency beyond the 90th percentile threshold. SafeTail addresses this challenge by selectively replicating services across multiple edge servers to meet target latencies. SafeTail employs a reward-based deep learning framework to learn optimal placement strategies, balancing the need to achieve target latencies with minimizing additional resource usage. Through trace-driven simulations, SafeTail demonstrated near-optimal performance and outperformed most baseline strategies across three diverse services.

Index Terms—Tail Latency, Redundant Scheduling, Reward-based deep learning, Edge Computing.

I. INTRODUCTION

In the realm of edge computing [1], latency-sensitive applications play a crucial role in providing seamless and high-quality user experiences. Technologies such as augmented reality (AR) [2], virtual reality (VR) [3], and real-time video conferencing demand exceptionally low latency to ensure responsiveness and fluid interaction [4]. For example, AR applications used in interactive gaming or navigation require near-instantaneous processing to align digital overlays with the real world, while VR experiences depend on minimal latency to create immersive, lag-free environments. Real-time video conferencing tools also require rapid data transmission to maintain clear and uninterrupted communication. These applications often run on user devices with limited computation power, relying on nearby edge servers for efficient processing.

Conversely, some latency-sensitive applications can tolerate higher median latency but still require stringent control over tail latency. For instance, in batch processing systems for data analytics, large-scale data analysis or scientific simulations may process data in batches and provide results at periodic intervals rather than in real-time [5]. While the system can accept higher median latency for routine tasks, it is crucial to manage tail latency carefully, especially for urgent queries or emergency analyses. Ensuring that tail latency remains within acceptable limits is vital for maintaining the system's effectiveness and responsiveness during critical instances, as excessive delays in these scenarios could significantly impact the application's performance and reliability. Thus, while overall throughput and accuracy are paramount, controlling tail latency is essential to meet the performance requirements of latency-sensitive applications.

A major difficulty in supporting latency-sensitive applications is ensuring the timely delivery of services with an acceptable stable median latency and minimal deviation from that latency. However, consistently achieving acceptable latency levels remains challenging. Service requests involve both network and computation latencies, each with inherent uncertainties that make it difficult to meet target latencies reliably. Consequently, most existing research focuses on optimizing median or mean latencies with edge servers but often overlooks higher percentiles of latency, such as the 90th, 95th, and 99th percentiles. These higher percentiles are crucial for delivering a high-quality user experience in latency-sensitive applications [6], [7]. Many studies on edge service scheduling fail to address these tail latencies effectively. The issue of tail latency is particularly pronounced in edge computing [8], where both the computation on edge servers and communication over wireless networks are subject to significant variability. Therefore, addressing high tail latency through the effective use of edge servers is essential for maintaining service quality in latency-sensitive applications.

Traditional methods often rely on strong assumptions about the underlying latency distributions (e.g., assuming a known statistical model such as Weibull, exponential, Pareto tails, or normal distribution) [9], [10] and static resource allocation rules. However, in real-world edge computing environments, latency patterns are highly dynamic and influenced by heterogeneous factors such as fluctuating network conditions, varying server loads, and service-specific computational demands. Rigid heuristic-based methods often fail to adapt effectively under such variability, leading to suboptimal latency performance. Prior research has explored task scheduling on edge

Jyoti Shokhanda, Utkarsh Pal, Aman Kumar and Arani Bhattacharya are affiliated to Indraprastha Institute of Information Technology Delhi, New Delhi, email: {jyotis, utkarsh20144, aman20279, arani}@iitd.ac.in. Soumi Chattopadhyay is with the Department of Computer Science and Engineering, Indian Institute of Technology Indore, India, e-mail: soumi@iiti.ac.in. Final version of this paper is available at IEEEXplore.

servers using deep reinforcement learning (DRL) [11]. These studies demonstrate that DRL can reduce task completion times by rewarding schedules with lower latency. However, they focus primarily on standard tasks offloaded from smartphones, rather than latency-sensitive tasks. Latency-sensitive applications impose strict constraints not only on median or mean latency but also on its entire distribution. Inadequate handling of these constraints can result in significant safety issues [12]. Consequently, these studies do not address the impact of DRL on tail latency.

One of the primary techniques for reducing tail latency involves introducing redundancy. For instance, a user device may submit a service to multiple edge servers, allowing it to utilize the fastest response among them. While this redundancy improves tail latencies, it can also increase the utilization of edge computing resources, such as network bandwidth and costs. Therefore, it is crucial to manage redundancy carefully to minimize tail latency while controlling resource usage.

However, determining where to place services with controlled redundancy remains complex. As redundancy increases, the number of potential scheduling options grows exponentially. For a single-end device with n available edge servers, the number of possible schedules is $2^n - 1$, making exhaustive search impractical for service request execution.

We tackle this challenge by first identifying a target latency for each service and then designing our reward to minimize the actual difference from the target. SafeTail, our framework, uses redundant scheduling combined with a reward-based deep-learning approach to address tail latency. This method involves assigning the same service request to multiple edge servers and enabling the system to learn from experience and interaction, considering both computation and network latencies. Unlike methods that rely on fixed redundancy or focus solely on optimizing one type of latency, SafeTail dynamically adjusts redundancy based on real-time conditions. This strategy reduces overall latency by effectively managing both network and computation latency, allowing the user device to select the fastest response and thus improving overall system responsiveness.

Our experiments begin with an in-depth tail latency analysis of the YOLOv5 object detection service on our own WiFi network. We observed that network latency is influenced by factors like the number of active users, while computation latency is affected by compute resource availability, memory, and server configuration. These variables introduce uncertainty in service latency [13], further compounded by varying workloads on edge servers and network unpredictability [14].

Our experiments depend on the collection of our own WiFi network and compute traces. We then assessed the performance of our reward-based deep learning mechanism on the median as well as tail latency values for three distinct services: (i) Object detection using YOLOv5, (ii) Instance segmentation of images, and (iii) Removal of noise from audio. These three services are all latency-sensitive, and some of them are often used in real-time applications such as virtual reality and video conferencing. We evaluated SafeTail using trace-driven simulations on a system with five edge servers and compared its performance against four baseline techniques.

Our findings show that SafeTail effectively reduces both median and tail latency compared to these baseline methods. We now summarize our contributions as follows:

(i) We tackle the challenge of reducing service latency in edge computing, where environmental uncertainties, such as fluctuating wireless network conditions and varying computational loads, are significantly more pronounced than in cloud settings. These uncertainties impact both network latency (transmission and propagation delays) and computation latency. While similar concerns have been explored in cloud computing, the dynamic and resource-constrained nature of edge environments demands a more tailored strategy for latency-sensitive services. SafeTail addresses this by prioritizing latency reduction particularly tail latency over resource usage. Rather than aiming for a globally optimal solution, SafeTail dynamically applies controlled redundancy based on real-time server conditions to achieve near-optimal latency, balancing responsiveness with efficient resource utilization.

(ii) We propose a reward-driven deep learning framework that adaptively manages service redundancy to optimize latency. By dynamically adjusting redundancy based on system conditions, the framework effectively balances meeting target latency requirements with minimizing resource consumption.

(iii) We developed a prototype testbed with real-world characteristics and collected execution traces from three distinct applications: (a) single-shot object detection from images, (b) instance segmentation of images, and (c) noise removal from audio under varying network and edge load conditions. Using these traces for simulation, we demonstrate that our reward-based deep learning framework significantly optimizes both median and tail latency.

II. PROBLEM FORMULATION

In this section, we formulate our problem mathematically. Our framework has the following inputs:

- A set of homogeneous edge servers $\mathcal{E} = \{e_1, e_2, \dots, e_n\}$.
- The dynamic state of each edge server $e_i \in \mathcal{E}$ at any timestep t is identified by 5-tuple: $(\lambda_i^{u(t)}, \lambda_i^{d(t)}, \mathcal{M}_i^t, \mathcal{U}_i^t, \ell_i^t)$, where $\lambda_i^{u(t)}, \lambda_i^{d(t)}, \mathcal{M}_i^t, \mathcal{U}_i^t, \ell_i^t$ denote the uplink and downlink bandwidths, memory and CPU utilization, and number of active users accessing e_i at timestep t , respectively. In the rest of this paper, however, we omit the superscript t from each symbol to represent the values of the above parameters as experienced by the user when the service is actually requested.
- A user at a specific location with the requirement of edge servers for the execution of a service s_j .
- A user is denoted by a 3-tuple: $U = (L, \Lambda^u, \Lambda^d)$, where L, Λ^u, Λ^d represent the location, upload and download bandwidths of the user device, respectively.
- Each service s_j is characterized by 3-tuple: $(\mathcal{I}_j, \mathcal{O}_j, \Gamma_j)$, where \mathcal{I}_j represents a set of input parameters required by the service, \mathcal{O}_j denotes a set of output parameters generated by the service after execution, and Γ_j includes the characteristics of the input parameters of that influence the computation time of s_j . It may be noted that Γ_j varies across different services. The objective of our work is to reduce the overall latency for the service s_j . Before going to discuss the details of our

framework, we first explain the key components of latency. Multiple events are associated with the execution of a service on an edge server, which actually contributes to latency computation [15], [16], as discussed below.

- *Transfer of service-input*: A set of input parameters \mathcal{I}_j for the service s_j must be transferred from the user device to the edge server for execution. This transfer time includes both transmission and propagation latency. The transfer process involves three key events: (i) uploading the input file from the user device, (ii) propagating the file from the user device to the edge server (represented by a function of edge server and user location $\rho_i(L)$), and (iii) downloading the input file to the edge server.

- *Execution of service*: Once an edge server receives the input of a service, the service is executed on it. The computation time $C(e_i, s_j)$ at e_i is a function of various parameters of s_j and the dynamic state of the edge server e_i .

- *Transfer of service output*: The output parameters \mathcal{O}_j of the service s_j are transferred from the edge server to the user device, similar to the process of transfer of the service input. Considering a service is executed in e_i , service latency is mathematically defined as follows. The symbols used in Eq. (1) are defined in brackets after introducing each term above.

$$\mathcal{L}_i = \underbrace{\left(\underbrace{\frac{\text{Size}(\mathcal{I}_j)}{\min(\lambda_i^u, \Lambda^u)}}_{\text{uploading}} + \underbrace{\rho_i(L)}_{\text{propagation latency}} + \underbrace{\frac{\text{Size}(\mathcal{I}_j)}{\min(\lambda_i^d, \Lambda^u)}}_{\text{downloading}} \right)}_{\text{Transfer of service input}} + \underbrace{C(e_i, s_j)}_{\text{Computation latency}} + \underbrace{\left(\underbrace{\frac{\text{Size}(\mathcal{O}_j)}{\min(\lambda_i^u, \Lambda^d)}}_{\text{uploading}} + \underbrace{\rho_i(L)}_{\text{propagation latency}} + \underbrace{\frac{\text{Size}(\mathcal{O}_j)}{\min(\lambda_i^u, \Lambda^d)}}_{\text{downloading}} \right)}_{\text{Transfer of service output}} \quad (1)$$

In this paper, we have the following assumptions:

(i) *Capacity-aware admission model*: Each edge server can accept a certain number of requests. Beyond this limit, the edge server refuses to accept additional services.

(ii) Each edge server adopts a capacity-aware admission model, allowing only a limited number of concurrent service requests. Once a request is admitted, it shares compute resources with others, and any delay due to contention is implicitly reflected in the observed execution time. Queueing delays prior to admission are not explicitly modeled.

(iii) The number of edge servers (n) reachable from a user is reasonably small, which is most likely fewer than 10 in number. This is because the network used has a relatively smaller range, making it realistic to assume that only a small number of edge servers would be within this range. Today's 5G networks also support receiving signals from at most 10 cell towers from a user device [17].

In this paper, we address the challenge of long tail latency in edge computing by introducing controlled redundancy to improve response times. As a fallback strategy, if a service request fails to receive a response from the edge server within a reasonable time, a lightweight version of the service

is executed locally on the user's device. This local version ensures bounded latency, as its execution time defines the maximum tolerable delay. However, due to its reduced accuracy compared to the full version, it is used only when necessary. Our primary goal remains to obtain responses from the edge server, where the standard, high-accuracy version of the service runs.

III. ANALYZING TAIL LATENCY: EMPIRICAL STUDIES

In this section, we first characterize tail latency through experimental studies by observing it under different edge computing environments, including variations in compute and network loads. Here, we select a widely used application in autonomous systems—object detection using YOLOv5—and run it on a desktop machine equipped with an Intel Core i7-11700 processor, 16GB of RAM, and a total of 16 virtual cores. We disabled the background processes for this experiment. We used the tools stress-ng [18] to generate workload and task set [19] to vary the number of cores allocated to the process from the total cores available on the system. Fig. 1 illustrates the different percentiles of latency and its various components (i.e., transmission latency, propagation latency, and computation latency) under varying network and compute parameters. Unless otherwise specified, we maintain the RAM at 16GB, the number of cores at 16, and the CPU background workload at 0%. We then varied a single parameter for each experiment, where the parameters are: (a) amount of available RAM, (b) CPU background workload, (c) number of available cores, and (d) number of devices communicating over the network. We repeated each experiment 1000 times and reported the results. We now discuss our empirical findings concerning tail latency.

Effect of Changes in Available RAM: In this experiment, we study the behavior of computation latency with respect to RAM availability. We varied the RAM utilization from 5000 to 11000 MB using stress-ng to generate workloads. Figure 1(a) shows the median and tail computation latencies (at the 90th, 95th, and 99th percentiles) for different levels of RAM utilization. As indicated in Fig. 1(a), the gap between median latency and tail latencies increased as RAM availability decreased. Moreover, higher percentile latencies were more significantly impacted by lower RAM availability compared to median latency.

Effect of CPU Background Workload: In this experiment (refer to Fig. 1(b)), we varied background workload on CPUs from 20% to 80%, and observed the changes in computation latency. We found a similar trend in median and tail latencies for this experiment as well.

Effect of Number of Cores: For this experiment, we varied the number of available cores from 2 to 16 and studied the behavior of computation latency (refer to Fig. 1(c)). We observed that computation latency decreased with an increase in the number of cores up to a certain threshold (here, up to 8 cores). Beyond this threshold, additional cores did not affect the computation latency.

Effect of Network Load: We generated network load by using additional Raspberry Pi nodes that sent data via iPerf 3

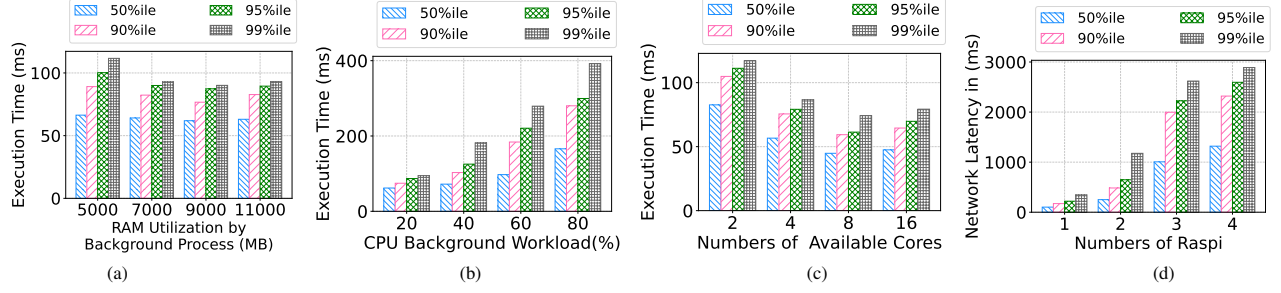


Fig. 1. Latency characteristics for the YOLOv5 object detection service: Illustration of variation in latency with changes in (a) RAM utilization by background processes, (b) CPU background workload, (c) the number of available cores, and (d) the number of Raspberry Pi devices to simulate varying network load.

[20]. We then sent ping probes from our machine to a server over the same WiFi network to obtain our network latency. Figure 1(d) shows how network latencies change with varying computational resources (i.e., the number of Raspberry Pi nodes). We observed that latency increases only slightly with a small number of Raspberry Pi nodes (up to 2) but rises rapidly beyond this threshold. Other observations were consistent with the first and second experiments.

The Challenge of Optimizing Tail Latency: The above experiments demonstrate that tail latency is unevenly influenced by various resource parameters. For instance, parameters like the availability of CPU cores impact both median and tail latency similarly. In contrast, factors such as RAM availability, background workload, and network load have a more pronounced effect on tail latency compared to median latency. Therefore, we design a framework focused on minimizing tail latency, taking into account the varying sensitivities of these resource parameters.

To tackle this challenge, we employ several strategies. First, we use redundant service scheduling to mitigate latency unpredictability. Second, we develop a placement mechanism based on reward-based deep learning that learns from the dynamic state of edge servers to determine where to place the service. This approach ensures that latency-sensitive services are not reliant on simplistic scheduling models.

IV. FRAMEWORK AND METHODOLOGY

In this section, we introduce SafeTail, our framework that utilizes a reward-based deep learning approach inspired by reinforcement learning. SafeTail is a user-centric service allocation framework that aims to reduce service latency by incorporating redundancy into the process. The primary motivation for our work is based on the premise that if one edge server fails to deliver a service response within the promised time, other edge servers may still be able to deliver the output on time. However, replicating the service execution to all edge servers can lead to increased network congestion and unnecessary resource utilization. Therefore, our objective in this paper is to minimize the service latency by determining the reasonable number of redundancies for a service execution across various servers, depending on the dynamic state of the edge servers, without significantly increasing resource utilization or compromising network traffic. We now discuss the details of our framework, starting with an introduction to the concept of redundancy scheduling for a service.

The goal of redundancy scheduling is to duplicate the execution of a service s_j across multiple edge servers. By selecting a subset of edge servers $\mathcal{E}_k^t \subseteq \mathcal{E}$ at timestep t , we intend to minimize latency variability and achieve the fastest response. The rationale is that if one edge server is heavily loaded, another might have sufficient resources to execute the service within an acceptable time limit. This approach effectively addresses challenges such as uncertain propagation latencies, long transmission latencies, and variable computation latencies. The improved latency achieved by SafeTail through redundancy scheduling is demonstrated using Eq. 2.

$$\mathcal{L}_R = \min_{e_i \in \mathcal{E}_k} (\mathcal{L}_i) \quad (2)$$

We now present our framework, SafeTail, which dynamically adapts redundant scheduling. SafeTail leverages a reward-based deep learning framework to approximate complex functions and understand the relationships among the dynamic state of the edge servers at timestep t , the service requirements, and the redundant executions that need to be introduced at timestep $t + 1$. The primary idea is to comprehend the dynamic state of each server and the service requirements to determine the expected latency for the service when executed on an edge server and the uncertainty of achieving that latency. Based on this, SafeTail decides the subset of edge servers needed to achieve the desired latency. Fig. 2 presents an overview of our framework. We discuss each component of SafeTail below. We begin by discussing the *state* that depicts the dynamic condition of all edge servers, along with the components of the service on which SafeTail bases its *actions*.

Definition 4.1: State: The state of the environment at timestep t , denoted by ω^t , is defined by an $(n + 1)$ -tuple $(\Omega_1^t, \Omega_2^t, \dots, \Omega_n^t, \mathcal{P}_j, U)$, where $\forall i = \{1, \dots, n\}$:

$$\Omega_i^t = (\lambda_i^{u(t)}, \lambda_i^{d(t)}, \mathcal{M}_i^t, \mathcal{U}_i^t, \ell_i^t, \rho_i^H(L)); \mathcal{P}_j = (\text{Size}(\mathcal{I}_j), e\text{Size}(\mathcal{O}_j), \Gamma_j)$$

where the first five elements of Ω_i^t captures the dynamic state of edge server e_i at timestep t . $\rho_i^H(L)$ represents the median propagation latency from the user device to e_i . \mathcal{P}_j denotes the properties of the service s_j that may influence the latency. Here, $\text{Size}(\mathcal{I}_j)$ represents the size of \mathcal{I}_j and $e\text{Size}(\mathcal{O}_j)$ represents the estimated size of \mathcal{O}_j . ■

It is important to note that various parameters in ω^t contribute to estimating different components of the latency. For instance, $\lambda_i^{u(t)}, \lambda_i^{d(t)}$ in Ω_i^t and $\text{Size}(\mathcal{I}_j), e\text{Size}(\mathcal{O}_j)$ in \mathcal{P}_j helps estimate

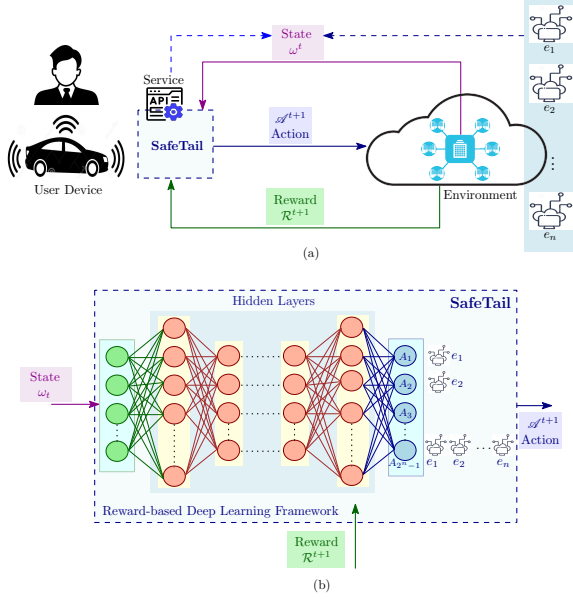


Fig. 2. Overview of (a) SafeTail; (b) Reward-based Deep Learning Framework

the transmission latency, while $\rho_i^\mu(L)$ in Ω_i^t enables to measure the propagation latency. Finally, $\mathcal{M}_i^t, \mathcal{U}_i^t, \ell_i^t$ in Ω_i^t and Γ_j in \mathcal{P}_j lead towards approximating the computation latency.

Once our learning network receives the state ω^t at t , it determines an appropriate action at $(t+1)^{th}$ timestep, denoted by \mathcal{A}^{t+1} , by selecting a subset of edge servers for redundancy scheduling for s_j . For n edge servers, there are $2^n - 1$ possible actions, denoted as $A = \{A_1, A_2, \dots, A_{2^n-1}\}$. Each action $A_i \in A$ is an element of $\mathbb{P}(\mathcal{E}) \setminus \phi$, where $\mathbb{P}(\mathcal{E})$ represents the powerset of \mathcal{E} . SafeTail employs an ϵ -greedy exploration and exploitation strategy to choose an action.

$$\mathcal{A}^{t+1} = \begin{cases} \text{Random}(A_i) & \text{with probability } \epsilon \\ \arg \max_{A_i \in A} (A_i) & \text{with probability } (1 - \epsilon) \end{cases} \quad (3)$$

where ϵ is a parameter that controls the balance between exploration and exploitation. Initially, ϵ is set to 1 when the network is untrained. SafeTail begins with 100% exploration, where a random action is chosen. Each time SafeTail actuates an action, a reward is given based on the effectiveness of the action in optimizing tail latency and resource utilization. The state-action-reward tuple, i.e., $(\omega^t, \mathcal{A}^{t+1}, \mathcal{R}^{t+1})$ is stored to train the network. Once a sufficient number of tuples, denoted as κ , have been collected, we start training the network.

Each time the network is partially trained, ϵ is decreased by a certain quantity, denoted as ϵ_{decay} , until it reaches a minimum value, denoted as ϵ_{min} . As ϵ is decreased, SafeTail continues its exploration and exploitation κ times before re-training the network. During exploitation, SafeTail invokes the reward-based learning network to return the action determined by the network, i.e., the second case of Eq. (3).

Before discussing the architecture of the reward-based learning network, we first address the reward function of SafeTail. As previously mentioned, the reward is decided based on how effectively the action optimizes tail latency and resource utilization. Ideally, the highest reward is given when optimal

latency is achieved with minimal resource usage. However, determining the optimal latency value without executing the service on all edge servers is infeasible. Therefore, we define a *target latency*, which is a heuristic value that we approximate. In calculating the reward, we consider the achieved latency relative to this target latency. To proceed with defining the reward function, we first need to establish the target latency.

Definition 4.2: Target Latency: The target latency, denoted as τ , is mathematically defined as:

$$\tau(\omega^t) = \left(\frac{\text{Size}(\mathcal{I}_j)}{\Lambda^u} + \rho^\mu(L) + \frac{\text{Size}(\mathcal{I}_j)}{\Lambda^u} \right) + \mathcal{C}(s_j) + \left(\frac{e\text{Size}(\mathcal{O}_j)}{\Lambda^d} + \rho^\mu(L) + \frac{e\text{Size}(\mathcal{O}_j)}{\Lambda^d} \right) \quad (4)$$

where $\rho^\mu(L) = \text{median}_{e_i \in \mathcal{E}}(\rho_i^\mu(L))$ and $\mathcal{C}(s_j)$ is the median computation latency on an edge server when a certain number (say, ν) of instances of the service is executed on it. ■

It is important to note from Eq. (4) that to compute the transmission latency, we use only the user's uplink and downlink bandwidths. This is a reasonable assumption because, in general, the bandwidth of the edge server is significantly higher than that of the user device. As a result, when computing the minimum value between the user's uplink bandwidth and the edge server's downlink bandwidth, the user's uplink bandwidth typically provides the minimum value.

Since the actual size of the output parameters is not available during the computation of the target latency, we use the estimated size of the output parameters in the computation of the transfer time for the service output.

We choose the median propagation latency across all edge servers because each server, regardless of its propagation latency, has a non-zero probability of providing the least service latency. Therefore, instead of choosing the least propagation latency, we consider the median value across all servers.

We select the value of ν based on profiling the computation time with varying numbers of parallel instances of the service. Initially, adding more parallel instances typically results in only minor increases in execution time. However, beyond a certain threshold, additional parallel instances cause a significant rise in execution time. We choose ν to strike a balance, ensuring sufficient parallelism to prevent a drastic increase in execution time while reflecting the fact that each edge server often handles multiple parallel instances.

We finally define the reward function.

Definition 4.3: Reward: The reward, denoted as $\mathcal{R}^{t+1}(\omega^t, \mathcal{A}^{t+1})$, is a function computed at timestep $t+1$ and is mathematically defined as:

$$\mathcal{R}^{t+1}(\omega^t, \mathcal{A}^{t+1}) = \begin{cases} 0 & \text{if } \mathcal{L}_R = \tau(\omega_t) \\ 0 & \text{if } \mathcal{L}_R < \tau(\omega_t) \text{ and } |\mathcal{E}_k| = 1 \\ 0 & \text{if } \mathcal{L}_R > \tau(\omega_t) \text{ and } |\mathcal{E}_k| = n \\ -\delta \cdot e^{(n-|\mathcal{E}_k|)} & \text{if } \mathcal{L}_R > \tau(\omega_t) \text{ and } |\mathcal{E}_k| < n \\ -\delta \cdot e^{(\mathcal{L}_R - \tau(\omega_t))} & \text{if } \mathcal{L}_R < \tau(\omega_t) \text{ and } |\mathcal{E}_k| > 1 \end{cases} \quad (5)$$

where \mathcal{E}_k is the subset of edge servers chosen in action \mathcal{A}^{t+1} , and \mathcal{L}_R is the latency achieved as a result of the action \mathcal{A}^{t+1} , calculated according to Eq. (2). The variable n denotes the total number of edge servers and δ is an externally tunable positive real-valued hyperparameter used to adjust the value of \mathcal{R}^{t+1} , either scaling it up or down.

Property 4.1: Upper Bound: The upper bound of the reward function is 0. ■

Property 4.2: Conditions to Achieve Maximum Reward: The maximum reward is attained when:

- The achieved latency is exactly equal to the target latency irrespective of resource utilization.
- The achieved latency is less than the target latency, and the resource utilization is minimal.
- The achieved latency is higher than the target latency, but the resource utilization is maximum. It is important to note that, although our objective is not fulfilled in this scenario, SafeTail cannot select a better action to satisfy the objective under these conditions. ■

Property 4.3: Conditions to Obtain Negative Reward: A negative reward is attained when:

- The achieved latency is higher than the target latency, however, there is a scope to increase the redundancy. In this case, the reward is determined according to the fourth case of Eq. (5), where it is inversely proportional to the potential increase in redundancy that remains achievable.
- The achieved latency is lower than the target latency; however, there is still potential to reduce redundancy. In this case, the reward is chosen in the fifth case of Eq. (5), where it is inversely proportional to the duration by which the target latency exceeds the achieved latency. ■

Property 4.4: Characteristics of Reward Function: In designing the reward function, we prioritize meeting the target latency over reducing resource utilization. This is reflected in the fourth and fifth cases of Eq. (5). Notably, when the achieved latency exceeds the target and there is potential to increase redundancy, the penalty is more significant compared to situations where latency requirements are met but with higher resource usage. This characteristic of the reward function effectively optimizes not only the median latency but also the tail latency. ■

We now discuss the working principle of SafeTail. Fig. 2 (a) illustrates its structure. SafeTail uses a reward-based deep learning framework, with a feed-forward neural network with back propagation (FNN) as its core component, as presented in Fig. 2 (b). The FNN takes the state ω^t as input, therefore, the number of nodes in the input layer corresponds to $|\omega^t|$. The number of nodes in the output layer is equal to the number of possible actions, i.e., $2^n - 1$. In our experiment, the FNN comprises 5 hidden layers with ReLU activations and a Softmax output layer. It is optimized using Adam with categorical cross-entropy as the loss function.

The reward-based learning network in SafeTail is trained on an episode-wise basis. At the start of each episode, we randomly initialize the load, i.e., the number of active users on each edge server, and then begin the simulation. Each episode is divided into a certain number of steps (denoted as α_s). In each step, a user requests a service execution, and the SafeTail framework selects a set of edge servers to execute the service, using either exploration or exploitation with a probability ϵ as described in Eq. 3. Once the user receives the service response, a reward is calculated based on the actual service latency and

the target latency heuristically computed for that service. The state-action-reward $(\omega^t, \mathcal{A}^{t+1}, \mathcal{R}^{t+1})$ tuple from each step is stored to train our reward-based learning framework, which is updated after every interval of κ steps.

The reward \mathcal{R}^{t+1} in the $(\omega^t, \mathcal{A}^{t+1}, \mathcal{R}^{t+1})$ tuple is translated to represent the target vector \mathcal{V}^t for the FNN, which is then used to calculate the loss function for training the FNN. The target vector has a length equal to the number of output nodes in the FNN, i.e., $2^n - 1$. In typical classifiers, the target vector is a one-hot vector. However, in our framework, due to the lack of prior knowledge about the appropriate action for a given state, our target vector is not a one-hot vector. Instead, we ensure that the sum of all elements in the target vector equals 1. If the reward \mathcal{R}^{t+1} for a particular action \mathcal{A}^{t+1} is zero, the corresponding element of A_k (assuming $\mathcal{A}^{t+1} = A_k$) in \mathcal{V}^t is set to 1, and the rest of the elements are set to 0. For other cases where \mathcal{R}^{t+1} is negative, we start by initializing all elements of \mathcal{V}^t with an equal value, i.e., $\frac{1}{2^n - 1}$. We then adjust the values for each element in \mathcal{V}^t . The values corresponding to A_k and all other actions A_j that involve edge servers \mathcal{E}_j , where \mathcal{E}_j is a subset of \mathcal{E}_k , which correspond to A_k , are set as follows:

$$\mathcal{V}^t(j) = \max\left(0, \left(\frac{1}{2^n - 1} + \mathcal{R}^{t+1}\right)\right), \quad \forall j \text{ where } \mathcal{E}_j \subseteq \mathcal{E}_k \quad (6)$$

Once all the elements in \mathcal{V}^t corresponds to the actions A_j , $\forall j$ where $\mathcal{E}_j \subseteq \mathcal{E}_k$, are adjusted, the remaining value, i.e., $\left(1 - \sum_{\forall j, \mathcal{E}_j \subseteq \mathcal{E}_k} \mathcal{V}^t(j)\right)$, is equally distributed among the remaining elements of \mathcal{V}^t .

Our reward-based learning network is trained for a predefined number of episodes (denoted as α_e) until the training process is terminated.

In the next section, we discuss our experimental setup and analyze the performance of SafeTail.

V. IMPLEMENTATION & EXPERIMENTAL ANALYSIS

A. Experimental Setup

Our simulator was built using YAFS [21] to model the topology of edge servers and user devices. We modified YAFS to be compatible with Python 3.11, incorporated uncertainties in the wireless network and execution time, enabled service scheduling on duplicate edge servers, and integrated our SafeTail framework to allow a user to schedule all the services they wish to invoke. We implemented our reward-based deep learning model using Keras with the TensorFlow backend (Keras v3.0.5 and TensorFlow 2.16.1).

Use Case Selection for Evaluating SafeTail's Performance: To demonstrate the generaliability of SafeTail, we selected three diverse use cases: (i) object detection on images using YOLOv5 [22], (ii) instance segmentation on images [23], and (iii) noise removal from audio signals [24]. These applications differ significantly in computation time, transmission latency, overall latency, and input modality. Object detection has low overall latency and moderate computational cost with minimal transmission overhead. Noise removal, though computationally light, involves high transmission latency due to large audio inputs. Instance segmentation is the most latency-intensive, with

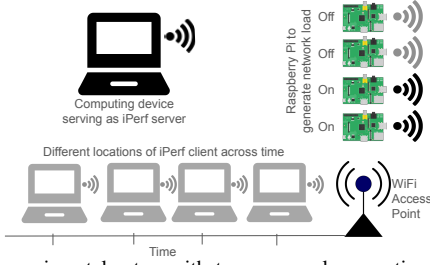


Fig. 3. Our experimental setup with two personal computing devices, four Raspberry Pi's, and one access point.

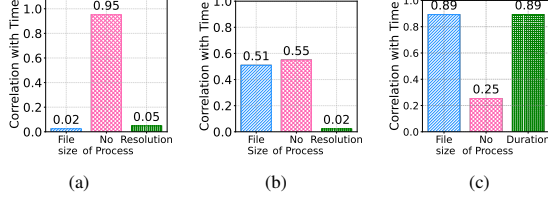


Fig. 4. Correlation between computation latency and different input parameters for (a) YOLOv5, (b) Instance segmentation, (c) Audio noise removal

high computational demands and moderate transmission latency. This diverse selection allows us to evaluate and highlight SafeTail's adaptability across a broad spectrum of latency-sensitive workloads. These services are core to autonomous and real-time systems, where minimizing tail latency is critical. The use cases span a wide latency range—from hundreds of milliseconds to several seconds—and cover two distinct input modalities: images and audio. We tested using benchmark datasets: the COCO dataset for images [25], and a Kaggle benchmark for audio files [26].

To make our experiments realistic, we collected both network traces and execution traces, as illustrated below:

Modeling Transmission Latency: We first collected traces of network data over WiFi to obtain the distribution for the uplink and downlink bandwidths of both the server and client. To do this, we set up an access point (AP) and connected two personal computing devices to it via WiFi, as shown in Fig. 3. One device acted as the iPerf3 server, while the other served as the client. We collected network throughput data (in bits per second) reported by iPerf3, repeating the experiment 1000 times to gather each throughput value. To introduce additional network congestion, we added 1-4 Raspberry Pi (Model 3B) devices to the same WiFi network, simulating the presence of additional users. The varying number of Raspberry Pis captured the uncertainty in the uplink and downlink bandwidths of the edge servers. Since our framework assumes homogeneous edge server settings, a single experiment involving one edge server can be used to model the initial uplink and downlink bandwidths for all edge servers in the network. However, as the load on each edge server changes over time, the uplink and downlink bandwidths per user may vary at later timesteps.

Modeling Propagation Latency: In a similar networking setup, we sent 500 ping packets from one computing device to another to model propagation latency. We repeated this experiment by placing the client device at varying distances from the AP to capture propagation latency under different signal strengths, recording the round trip times reported by ping for each packet.

Modeling Computation Latency: To collect the execution traces, we run 1-5 instances of the chosen service in parallel on an Intel Core i7-11700 processor (8 cores at 2.5 GHz base frequency) and 16 GB RAM. For each instance, we note down the execution time of the service using the time command.

For each service considered in this paper, we identified the characteristics of the input parameters that influence computation latency by analyzing its correlations with various parameters. For example, we examined the relationship between execution time and parameters like file size, resolution/duration, and the number of processes. In image-based services, resolution is the relevant parameter, whereas duration applies to audio noise removal.

We first prepared a dataset with file sizes, computation latency, and resolutions, converting the latter into numerical formats for consistency. We then calculated the Spearman correlation coefficients to quantify these relationships and visualized them using bar plots, as shown in Fig. 4.

In the case of object detection, we found that execution time is strongly correlated only with the number of processes, while file size and resolution showed no significant correlation. For instance segmentation, both file size and the number of processes had moderate correlations with execution time, but resolution exhibited a minimal correlation. Similarly, for noise removal from audio signals, there was a strong correlation with file size and duration and a moderate correlation with the number of processes. Parameters with small correlations were discarded, while those with stronger correlations were retained for further analysis.

After identifying all the parameters that influence the computation latency of a service, we created a dataset by executing each service on the benchmark test dataset. During each execution, we recorded the values of the selected parameters. This dataset was then used to train our regression model.

These datasets are subsequently used to formulate a regression model capable of predicting the execution time for each service. We employed a multilayer perceptron for this purpose, where computation latency serves as the response variable, and the selected parameters for each service act as the regressors.

To introduce uncertainty, we implemented the following strategy. For each number of processes (i.e., 1 to 5 processes that are executed in parallel) of a service, we obtained the standard deviation, denoted as σ_k (where k is the number of parallel processes), of the computation latency. After the regression model generates an output, denoted as \hat{y} , for k parallel processes, we further introduced Gaussian noise with a mean of \hat{y} and a standard deviation of σ_k .

Modeling Load on Each Edge Server: We utilized the dataset [27] to model the load variations for each base station. Starting with an initial number of active users on each edge server, we then adjusted the load according to the dataset's distribution.

B. Comparative Methods

We implemented and evaluated four baseline methods, each employing different strategies to optimize latency and two past works having a similar goal. We compared the total elapsed time to complete the service by each baseline method

with SafeTail. For a fair comparison, these methods also incorporate redundancy. We select fixed redundancy levels with $x \in \{1, 2, 3\}$ and limit x to 3, as higher levels significantly increase resource consumption. The four strategies are:

(i) *Oracle (Optimal)*: This method executes the service on all connected edge servers and records the fastest response, ensuring minimal latency.

(ii) *Random (Rand- x)*: This method involves using one or more randomly selected edge servers for service execution, depending on the allowed redundancy. In our experiment, we implement three sub-methods: Rand-1, Rand-2, and Rand-3, where the number indicates the level of redundancy.

(iii) *Edge Server with Minimum Propagation Latency (MinProp- x)*: This method directs the service to one or more edge servers with the lowest propagation latency, depending on the allowed redundancy, which is denoted by x .

(iv) *Edge Server with Minimum Load (MinLoad- x)*: This method sends the service to one or more edge servers that are running the fewest number of services, with the number of edge servers selected based on the redundancy level x .

(v) *Deep Reinforcement Learning with Linear Model (DRL-Linear)* [28]: This work minimizes execution delay and energy in multi-user settings via joint offloading and resource allocation, using a Q-learning and DQN-based strategy. We adapt this framework to SafeTail by modifying Deep Q-learning to suit its specific goals and constraints.

(vi) *TLORA* [9]: TLORA reduces tail latency in edge networks by modeling latency with a Weibull distribution and dynamically allocating resources. We extend this method to align with SafeTail's objectives and constraints.

C. Performance Metrics

The following metrics are used to measure the performance of SafeTail in comparison to the baseline methods.

(i) **Access Rate**: The access rate of a method is defined as the ratio between the number of redundant edge servers selected for service execution and the total number of edge servers reachable from the user device.

(ii) **Latency Deviation**: The latency deviation is the difference between the target latency and the achieved latency.

In our experimental analysis, we present the results using the following visualizations for each use case:

(i) **Average access rate versus average episode number**: This visualization illustrates how the access rate fluctuates in response to the dynamic state of edge servers over time.

(ii) **Average latency deviation versus average episode number**: This visualization demonstrates the characteristics of latency deviation as the number of episodes increases.

(iii) **Average absolute value of reward versus average episode number**: This visualization illustrates that the absolute value of the reward decreases and eventually stabilizes as the number of episodes increases.

(iv) **Comparison with baseline methods in terms of median and tail latency**: This analysis compares SafeTail with baseline methods in terms of median and higher percentiles (90th, 95th, and 99th) of latency, also known as tail latency. We first compare the latency achieved by SafeTail to the optimal

TABLE I
CONFIGURATION OF REWARD-BASED LEARNING FRAMEWORK

Parameter	UC-1: Value	UC-2: Value	UC-3: Value
δ	0.003	0.0015	0.003
$\epsilon_{\text{decay}}, \epsilon_{\text{min}}$	$3.3 \times 10^{-6}, 0.01$	$3.3 \times 10^{-6}, 0.01$	$3.3 \times 10^{-6}, 0.01$
α_s, α_e	256, 50	256, 50	256, 50
κ, β	128, 200	128, 120	128, 200

UC: Use case

latency. Next, we assess the average target latency relative to both the optimal latency and the latency achieved by SafeTail. Finally, we demonstrate that SafeTail outperformed all baseline methods in most cases.

We analyze SafeTail's theoretical time complexity in Appendix A. A comparative analysis of algorithmic analysis with the baseline methods is also presented in Appendix A.

In our analysis, we average each performance metric over β steps and present the results. In our plots, the average episode number refers to the ratio of the total number of steps to the parameter β . In the plot of the average absolute value of reward versus episode number, we display the training reward (for $50 \times \beta$ steps) followed by the testing reward (for $20 \times \beta$ steps). Furthermore, in our comparison plots, we have included the average value of the target latency.

D. Configuration of the Parameters

In our experiment, we consider 5 edge servers ($n = 5$), each with a maximum load capacity of 4 concurrent service requests. Appendix B of supplementary file shows the comparative performance of SafeTail when $n = 10$. Table I presents the configuration of the hyperparameters used in our reward-based learning framework. The values of the hyperparameters were determined experimentally using the validation dataset.

E. Experimental Analysis

As discussed earlier, we now analyze the performance of SafeTail across the following three different use cases.

1) *Use Case-1: Object Detection using YOLOv5*: Fig. 5 illustrates the performance of SafeTail on the YOLOv5 service. Our observations are as follows:

(i) **Improvement in median latency**: SafeTail attained a higher latency than Oracle's Optimal value, though with a small margin (5(a)). Specifically, Oracle's Optimal attained a speed-up in median latency of 1.08x compared to SafeTail. A more detailed discussion between the performance of Oracle (Optimal) and SafeTail is given in Appendix C in the supplementary material. However, as evident from the figure, this median latency is 1.08x, 1.19x, 1.72x, 1.11x, and 1.13x faster than DRL-Linear, TLORA, Rand-1, MinLoad-1, and MinProp-1, respectively.

(ii) **Improvement in tail latency**: As observed in Fig. 5(a), although SafeTail attained higher latency compared to Oracle's Optimal by a small margin, it achieved significantly better tail latency compared to the baseline methods. For the 90th, 95th, and 99th percentiles, SafeTail had speed-ups in tail latencies of 0.93x, 0.92x, and 0.79x compared to Oracle's Optimal value, respectively. However, these tail latencies are faster by 1.48x,

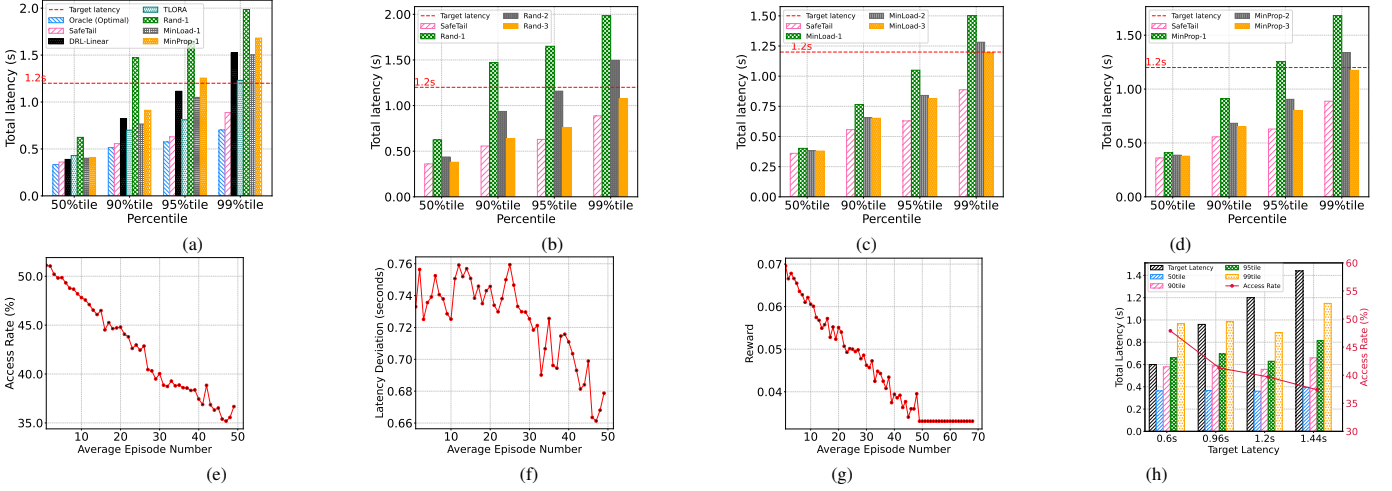


Fig. 5. Use Case-1: (a) Latency comparison among SafeTail, target latency and baseline methods, (b) SafeTail vs. Rand-x, (c) SafeTail vs. MinLoad-x, (d) SafeTail vs. MinProp-x, (e) Access rate of SafeTail, (f) Latency Deviation of SafeTail, (g) Reward distribution of SafeTail and (h) Impact of target latency.

1.77x and 1.72x, compared to DRL-Linear, 1.26x, 1.29x and 1.39x compared to TLORA, 2.64x, 2.62x, and 2.24x compared to Rand-1, 1.37x, 1.66x, and 1.69x compared to MinLoad-1, and 1.63x, 1.99x, and 1.89x compared to MinProp-1.

(iii) Achieved latency with respect to target latency: As evident from Fig. 5(a), SafeTail consistently achieved latency below the target latency, including for tail latency cases.

(iv) Comparison with baselines with higher access rates: We further analyzed the performance of each baseline method by increasing redundancy up to 3, as shown in Fig. 5(b)-(d). The results indicate that even with higher access rates, the latency values of all baseline strategies remain higher compared to SafeTail. This trend is consistent across all percentile values, thus showing that SafeTail is effective. We did not consider Redundancy levels 4 and 5 because level 5 is equivalent to Oracle's Optimal latency, and level 4 closely approximates it. Additionally, we note that the access rate of SafeTail varies between 1 to 3 edge servers.

(v) Variation in access rate: As noted from Fig. 5(f), the latency deviation of SafeTail decreases and stabilizes with an increasing number of episodes. This is because, as observed in Fig. 5(a), SafeTail consistently maintains latency below the target, even in tail latency scenarios. To further reduce access rate, as shown in Fig. 5(e), SafeTail reduces the latency deviation while still keeping the latency value within the target limits, by compromising slightly on the latency value.

(vi) Latency deviation: As noted from Fig. 5(f), the latency deviation obtained by SafeTail gradually decreased with an increasing number of episodes.

(vii) Stabilization of reward function: We show in Fig. 5(g) that the absolute value of the average reward curve eventually converges to a consistent value, indicating that SafeTail closely approximates the optimal latency and provides stable performance relative to the target latency.

(viii) Impact of target latency: Our experimental results (Fig. 5(h)) demonstrate the trade-off between target latency and resource usage, as captured by the access rate. When the target latency was the tightest (e.g., 0.60s), SafeTail reduced the achieved latency accordingly. While the median and 90th

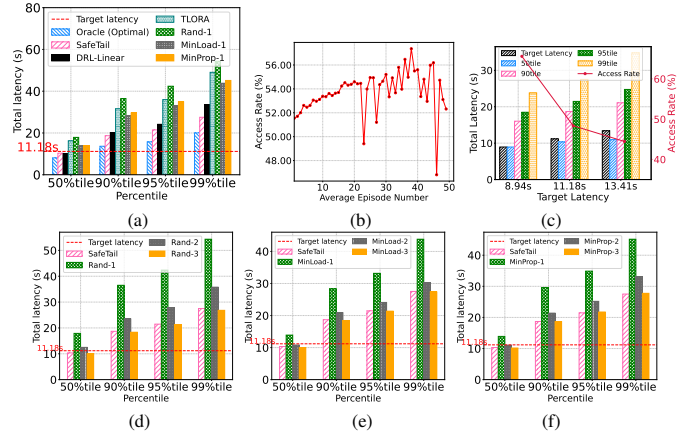


Fig. 6. Use Case-2: (a) Latency comparison: SafeTail vs. other methods, (b) Access rate of SafeTail, (c) Impact of target latency, (d) SafeTail vs. Rand-x, (e) SafeTail vs. MinLoad-x, (f) SafeTail vs. MinProp-x.

percentile latencies stayed below the target, the 95th and 99th percentiles slightly exceeded it, as their optimal latencies (0.57s and 0.70s, Fig. 5(a)) were inherently higher. This indicates SafeTail remains close to optimal when strict tail targets are challenging. As the target latency was relaxed (e.g., 1.2s and 1.44s), all latency metrics (median and tail) fell within the threshold, and the access rate decreased. This confirms SafeTail's ability to adaptively balance latency compliance with resource efficiency based on the target constraint.

2) *Use Case-2: Instance Segmentation of Images*: A similar trend is observed for this use case regarding median and tail latency, access rate, latency deviation, and stabilization of the reward, with a few exceptions discussed below.

As shown in Fig. 6(a), while the achieved median latency was lower than the target latency, SafeTail did not always meet the target latency in the case of tail latencies. It is worth noting that even the optimal tail latency exceeded the target latency. This discrepancy arises because the target latency is a heuristic value based on certain assumptions that may not always hold true. The target latency is primarily used for reward computation, but achieving it is not always guaranteed.

Another observation from Fig. 6(d)-(f) is that SafeTail's

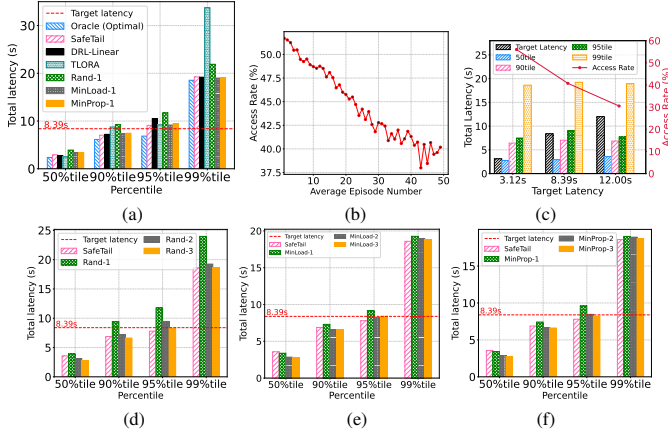


Fig. 7. Use Case-3: (a) Latency comparison: SafeTail vs. other methods, (b) Access rate of SafeTail, (c) Impact of target latency, (d) SafeTail vs. Rand-x, (e) SafeTail vs. MinLoad-x, (f) SafeTail vs. MinProp-x.

performance was comparable to Rand-3, MinLoad-3, and MinProp-3. Specifically, for Rand-3, SafeTail showed the worst degradation in median latency, with Rand-3's latency equal to 0.97x that of SafeTail. For MinLoad-3, the most significant degradation was observed in the 90th percentile, where MinLoad-3's latency was 0.90x that of SafeTail. For MinProp-3, SafeTail's worst performance was seen in the median latency, with MinProp-3 having 0.98x of SafeTail's latency. However, SafeTail outperformed MinProp-3 in other cases. This degradation may be attributed to the baseline methods using a fixed redundancy of three edge servers, leading to higher resource utilization and network congestion. In contrast, SafeTail selectively uses an average of 2 to 3 edge servers and, in some cases, just 1 edge server. This analysis shows a clear trade-off between optimizing latency and effectively utilizing resources. While the baseline methods achieve lower latency by consistently using multiple edge servers, they increase resource utilization and network congestion. SafeTail achieves comparable performance with more efficient resource usage by selectively employing fewer edge servers, thus better balancing latency optimization and resource management. The trend observed on latency for different target latencies in Fig. 6(h) aligns with that of Use Case-1. In Use Case-2, where SafeTail failed to meet the target latency, the corresponding optimal latency was either higher than or comparable to the specified target, indicating the inherent difficulty of satisfying the requirement under those conditions.

3) *Use Case-3: Removal of Noise from Audio Files*: An interesting observation can be made from Fig. 7(a), (d)-(f). SafeTail shows a slight degradation compared to the Oracle Optimal value, with Oracle's Optimal being faster by 1.50x, 1.14x, 1.12x, and 1.00x for the median, 90th, 95th, and 99th percentiles, respectively. While SafeTail utilized 1-3 edge servers, it predominantly employed only 1-2 edge servers in most cases, as shown in Fig. 7(b). It can be observed that for median and 90th percentile latency cases, the baseline methods outperformed SafeTail. However, for the 95th and 99th percentile cases, SafeTail performed better. Notably, in the median and 90th percentile cases, even though SafeTail did not surpass the baseline, it achieved latency values better

than the target latency. The primary objective of this work is to balance the trade-off between the utilization of a number of edge servers and latency optimization, with a higher priority on reducing tail latency. This result aligned with our objective. In the median and 90th percentile cases, when the achieved latency is better than the target, SafeTail focused on reducing the access rate. Conversely, in the 95th and 99th percentile cases, where SafeTail could not meet the target latency, it outperformed the baseline methods even with higher redundancy, because it prioritized latency minimization. As observed in Fig. 7(a), DRL-Linear and TORA achieved slightly lower median latency than SafeTail. However, SafeTail's median latency remained well within the target latency threshold. More importantly, SafeTail outperformed both methods in terms of tail latency, especially in cases where the tail latency surpassed the target. Furthermore, Fig. 7(c) indicates that lowering the target latency led to improved median latency, aligning it more closely with the optimal latency.

4) *Overhead of SafeTail*: To evaluate the computational overhead of identifying edge servers, we measured execution time on an Intel Core i7-11700 processor (2.5 GHz, 8 cores). For a scenario with 5 edge devices in the vicinity, SafeTail required approximately 40 ms, which is comparable to DRL-Linear at 42 ms, indicating that SafeTail introduces minimal additional overhead despite its more expressive decision-making model. As the number of nearby edge devices increases to 10, the time taken by SafeTail rises to 45 ms due to the exponential growth in the scheduling space. However, this overhead is typically masked in practice by parallel execution alongside other system services and thus has a negligible impact on the end-to-end service latency.

In summary, SafeTail always achieved near-optimal latency while effectively managing resource utilization across all three use cases. In most cases, it outperformed baseline methods without redundancy, and when redundancy was introduced, it delivered competitive median and tail latencies compared to the baseline methods, while controlling the number of edge servers used. SafeTail's dynamic and selective utilization of edge servers offered a major advantage in resource optimization. Despite minor latency degradations in certain scenarios, SafeTail maintained a stable performance, showcasing a clear balance between latency and resource efficiency.

VI. RELATED WORK

Existing studies fall into four categories, as discussed below: **Optimization of Latency-Sensitive Services on Edge Servers**: Latency-sensitive services are used by applications such as video conferencing, virtual reality and even Industry 4.0. These applications all require optimization of tail latency. A number of other works optimize the latency for critical applications over wireless networks using edge computing [29], [30], [31]. The work iSapiens [29] provides a distributed platform to run different smart city applications. The work [30] schedules tasks for vehicular control on the mobile edge, with suitable slicing of the network resources. [31] utilizes software-defined networking to schedule tasks related to controlling a vehicle. Although these works focus on controlling

latency-sensitive applications, unlike SafeTail, they all focus on optimization of median latency and not tail latency.

A few other works also specifically focus on optimizing the computation latency of latency-sensitive services. For example, NeuOS [32] is a system solution that runs multi-DNN workloads within autonomous systems. TailGuard [7] ensures that tail latency SLOs are met even under varying workloads by using predictive modeling, real-time monitoring, and adaptive scheduling. Unlike SafeTail, none of them consider the uncertainties present in both the wireless network and the computation time on edge servers. Furthermore, these works also do not utilize redundancy for scheduling services.

Learning-based Scheduling for Edge Services: Learning-based scheduling of edge services is often used to optimize response times and other quality of service parameters. The survey [33] provides a detailed discussion of the variety of approaches used. For example, the work [34] uses prioritization of packets for mobile hotspots to schedule tasks offloaded from mobile devices. The work [35] utilizes deep reinforcement learning to balance the workload across the edge servers. Finally, in the context of edge computing, deep reinforcement learning (DRL) has been employed for caching data in proximity to users [36] and for computation offloading [28], [37]. Notably, [28] utilizes DRL to identify the components to offload, departing from traditional optimization methods. This approach assumes a linear relationship when mathematically modeling the edge compute system. These works focus on caching data, data offloading, and optimizing latency. However, different from SafeTail, they do not focus on scheduling of services to reduce tail latency.

Utilization of Redundancy to Reduce Latency: The concept of redundancy to enhance reliability is widely employed in data centers [38], [39]. For example, the work [40] analyzes the root cause of high tail latency in longer jobs and identify that it is mostly caused by high server utilization. This insight enables the prediction of potential slower jobs early into their computation, enabling efficient management and optimization. In [41], an algorithm is proposed to estimate the latency and cost tradeoff based on the empirical distribution of task computation time. This shows that a modest level of task replication can diminish both latency and the expenses associated with computing resources. Furthermore, [42] balances the requirement of tail latency with that of reduction of redundancy, similar to ours. These works focus on cloud computing, and not on edge computing, where network latencies are higher and more uncertain. These techniques are suitable in situations with abundant compute and network resources, stable network conditions, and the ability to absorb the overheads of aggressive duplication, predictive scaling, or speculative redundancy. These assumptions do not hold in edge computing scenarios. On the other hand, SafeTail is explicitly designed for decentralized, resource-constrained edge environments, where excessive redundancy and overprovisioning are infeasible.

Comparative Analysis of Tail Latency Optimization Across Different System Models: Several existing methods, such as EdgeTuner [43], TLORA [9], and EcoEdgeInfer [44], also aim to reduce tail latency but differ significantly from SafeTail in control assumptions and operational scope. EdgeTuner [43]

operates at the orchestration layer with full control over scheduling policies, enabling global job scheduling optimization. TLORA [9] assumes centralized control in fog-based 5G networks to dynamically reallocate resources by assuming Weibull latency distribution. EcoEdgeInfer [44] focuses on optimizing local inference on resource-constrained edge devices, controlling both hardware and software settings. In contrast, SafeTail assumes no control over edge server internals. Instead, it guides service placement from the user side, leveraging observed system conditions to manage redundancy and minimize tail latency. Unlike the others, SafeTail passively adapts to dynamic environments, making it well-suited for decentralized, user-constrained edge computing.

Duplication [45] is used on edge servers to shorten the transmission time and improve the quality of service (QoS). Duplicate aware scheduling was proposed to duplicate every task. The work [46] proactively senses the network and computes resources available to ascertain the need for duplication before scheduling services on edge servers. Further, [47] explored the use of computation duplication to edge servers to accelerate the download of results. The work [48] focuses on reducing latency and reliability in edge computing with inherent system uncertainty. Unlike SafeTail, these techniques do not have a specific *target* latency. Our notion of target latency ensures that the level of duplication remains limited and thus, reduces the amount of resource utilization.

VII. LIMITATION AND FUTURE WORK

SafeTail currently has the following limitations. First, it has been evaluated on a homogeneous set of edge servers, where all servers have identical computing and network resources. While this is common in prior studies, some research has shown that resource heterogeneity in edge servers can help reduce latencies. However, our framework captures the dynamic state of each server, which varies across the edge servers, and thus, our approach can be extended to heterogeneous settings as well. In future work, we intend to expand our experiments to consider heterogeneous environments. Additionally, our current framework assumes homogeneous services for execution, a constraint we also plan to relax in future work.

Second, our framework is currently user-centric, with SafeTail operating on the user side to decide redundancy levels based on the user's needs. However, this approach may occasionally increase overall resource consumption, although we mitigate this by reducing redundancy when possible. In the future, we plan to optimize tail latency across services by considering the needs of all users in the network.

Third, we did not model the initial waiting time for each server. We assume that a server that cannot accept a request simply discards it. Note that the subsequent waiting time of processes due to resource preemption and/or contention is considered to be part of execution time in our experiments. Addressing this limitation will be a focus of our future work.

VIII. CONCLUSION

In this paper, we introduced SafeTail, a reward-based deep learning framework designed to tackle the challenge of reducing tail latency in latency-sensitive services. SafeTail aims

to optimize service execution latency through adaptive redundancy, intelligently managing the use of additional edge servers to achieve this goal. The framework dynamically adapts to the changing conditions of edge servers and service demands, deploying redundancy only when necessary to minimize tail latency while avoiding excessive resource use and network congestion.

Our experimental results revealed that SafeTail significantly improved both median and tail latency compared to existing baseline methods. We performed extensive trace-driven simulations across diverse applications, including object detection, image segmentation, and audio noise removal. These simulations demonstrated that SafeTail effectively reduced service latency, particularly tail latency, while skillfully balancing latency and resource utilization.

ACKNOWLEDGMENTS

This work is supported by the Science and Engineering Research Board, Department of Science and Technology, Government of India, under Grant CRG/2022/005096.

REFERENCES

- [1] K. Jiang, H. Zhou, X. Chen, and H. Zhang, "Mobile edge computing for ultra-reliable and low-latency communications," *IEEE Communications Standards Magazine*, vol. 5, no. 2, pp. 68–75, 2021. DOI: 10.1109/MCOMSTD.001.2000045.
- [2] Y. Siriwardhana, P. Porambage, M. Liyanage, and M. Ylianttila, "A survey on mobile augmented reality with 5g mobile edge computing: Architectures, applications, and technical aspects," *IEEE Communications Surveys and Tutorials*, vol. 23, no. 2, pp. 1160–1192, 2021. DOI: 10.1109/COMST.2021.3061981.
- [3] L. Wang, L. Jiao, T. He, J. Li, and M. Mühlhäuser, "Service entity placement for social virtual reality applications in edge computing," *IEEE International Conference on Communications (INFOCOM)*, pp. 468–476, 2018. DOI: 10.1109/INFOCOM.2018.8486411.
- [4] X. Zhang, H. Chen, Y. Zhao, Z. Ma, Y. Xu, H. Huang, H. Yin, and D. O. Wu, "Improving cloud gaming experience through mobile edge computing," *IEEE Wireless Communications*, vol. 26, no. 4, pp. 178–183, 2019. DOI: 10.1109/MWC.2019.1800440.
- [5] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications," *IEEE International Conference on Cloud Computing (CLOUD)*, pp. 82–89, 2018. DOI: 10.1109/CLOUD.2018.00018.
- [6] X. Chen, H. Song, J. Jiang, C. Ruan, C. Li, S. Wang, G. Zhang, R. Cheng, and H. Cui, "Achieving low tail-latency and high scalability for serializable transactions in edge computing," *ACM European Conference on Computer Systems (EuroSys)*, p. 210–227, 2021. DOI: 10.1145/3447786.3456238.
- [7] Z. Wang, H. Li, L. Sun, T. Rosenkrantz, H. Che, and H. Jiang, "Tailguard: Tail latency slo guaranteed task scheduling for data-intensive user-facing applications," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2023, pp. 898–909. DOI: 10.1109/ICDCS57875.2023.00042.
- [8] M. S. Elbamby, C. Perfecto, C.-F. Liu, J. Park, S. Samarakoon, X. Chen, and M. Bennis, "Wireless edge computing with latency and reliability guarantees," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1717–1737, 2019. DOI: 10.1109/JPROC.2019.2917084.
- [9] S. Zheng, Z. Gao, X. Shan, W. Zhou, and Y. Wang, "Tail latency optimized resource allocation in fogbased 5g networks," in *IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 249–254. DOI: 10.1109/ISCC.2018.8538463.
- [10] W. Zhang, M. Feng, and M. Krunz, "Latency estimation and computational task offloading in vehicular mobile edge computing applications," *IEEE Transactions on Vehicular Technology*, vol. 73, no. 4, pp. 5808–5823, 2024. DOI: 10.1109/TVT.2023.3334192.
- [11] S. P. Panda, A. Banerjee, and A. Bhattacharya, "User allocation in mobile edge computing: A deep reinforcement learning approach," in *IEEE ICWS*, 2021, pp. 447–458. DOI: 10.1109/ICWS53863.2021.00064.
- [12] N. Khan and S. Coleri, "Resource allocation for ultra-reliable low-latency vehicular networks in finite blocklength regime," in *IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, 2022, pp. 322–327. DOI: 10.1109/MeditCom55741.2022.9928733.
- [13] R. Bi, T. Peng, J. Ren, X. Fang, and G. Tan, "Joint service placement and computation scheduling in edge clouds," in *IEEE International Conference on Web Services (ICWS)*, 2022, pp. 47–56. DOI: 10.1109/ICWS55610.2022.00022.
- [14] Y.-W. Hung, Y.-C. Chen, C. Lo, A. G. So, and S.-C. Chang, "Dynamic workload allocation for edge computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 519–529, 2021. DOI: 10.1109/TVLSI.2021.3049520.
- [15] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communication Surveys and Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017. DOI: 10.1109/COMST.2017.2745201.
- [16] Y. Zhang, C. Chen, L. Liu, D. Lan, H. Jiang, and S. Wan, "Aerial edge computing on orbit: A task offloading and allocation scheme," *IEEE Transactions on Network Science and Engineering*, vol. 10, no. 1, pp. 275–285, 2023. DOI: 10.1109/TNSE.2022.3207214.
- [17] "3GPP: Release 18," <https://www.3gpp.org/specifications-technologies/releases/release-18>, 2024, accessed: 2025-06-20.
- [18] "Ubuntu manpage: stress-ng," <https://manpages.ubuntu.com/manpages/xenial/man1/stress-ng.1.html>.
- [19] "Linux manpage: taskset," <https://man7.org/linux/manpages/man1/taskset.1.html>.
- [20] M. Mortimer, "iperf3 documentation," 2018.
- [21] I. Lera, C. Guerrero, and C. Juiz, "Yafs: A simulator for iot scenarios in fog computing," *IEEE Access*, vol. 7, pp. 91 745–91 758, 2019. DOI: 10.1109/ACCESS.2019.2927895.
- [22] <https://github.com/ultralytics/yolov5>, accessed on June 20, 2024.
- [23] Ultralytics, <https://docs.ultralytics.com/tasks/segment/>, accessed on June 20, 2024.
- [24] B. McFee, M. McVicar, D. Faronbi, I. Roman, M. Gover, S. Balke, S. Seyfarth, A. Malek, C. Raffel, V. Lostanlen, B. van Niekirk, D. Lee, F. Cwitkowitz, F. Zalkow, O. Nieto, D. Ellis, J. Mason, K. Lee, B. Steers, E. Halvachs, C. Thomé, F. Robert-Stöter, R. Bittner, Z. Wei, A. Weiss, E. Battenberg, K. Choi, R. Yamamoto, C. Carr, A. Metsai, S. Sullivan, P. Friesch, A. Krishnakumar, S. Hidaka, S. Kowalik, F. Keller, D. Mazur, A. Chabot-Leclerc, C. Hawthorne, C. Ramaprasad, M. Keum, J. Gomez, V. Monroe, V. A. Morozov, K. Eliasi, nullmightybofo, P. Biberstein, N. D. Sergin, R. Hennequin, R. Naktinis, beantowel, T. Kim, J. P. Åsen, J. Lim, A. Malins, D. Hereñú, S. van der Struijk, L. Nickel, J. Wu, Z. Wang, T. Gates, M. Vollrath, A. Sarroff, Xiaoming, A. Porter, S. Kranzler, VoodooHop, M. D. Gangi, H. Jinzo, C. Guerrero, A. Mazhar, toddrme2178, Z. Baratz, A. Kostin, X. Zhuang, C. T. Lo, P. Campr, E. Semeniuc, M. Biswal, S. Moura, P. Brossier, H. Lee, and W. Pimenta, "librosa/librosa: 0.10.2.post1," May 2024. DOI: 10.5281/zenodo.11192913.
- [25] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European Conference on Computer Vision (ECCV) 2014*, pp. 740–755. DOI: 10.1007/978-3-319-10602-1_48.
- [26] "Mnist audio dataset." [Online]. Available: <https://github.com/Jakobovski/free-spoken-digit-dataset/tree/master>
- [27] Y. Li, A. Zhou, X. Ma, and S. Wang, "Profit-aware edge server placement," *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 55–67, 2021.
- [28] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for mec," in *IEEE Wireless Communications Networks and Communications (WCNC)*, 2018, pp. 1–6. DOI: 10.1109/WCNC.2018.8377343.
- [29] F. Cicirelli, A. Guerrieri, G. Spezzano, and A. Vinci, "An edge-based platform for dynamic smart city applications," *Future Generation Computer Systems*, vol. 76, pp. 106–118, 2017. DOI: 10.1016/j.future.2017.05.034.
- [30] L. Li, Y. Li, and R. Hou, "A novel mobile edge computing-based architecture for future cellular vehicular networks," in *IEEE Wireless Communications Networks and Communications (WCNC)*, 2017, pp. 1–6. DOI: 10.1109/WCNC.2017.7925830.
- [31] J. Liu, J. Wan, B. Zeng, Q. Wang, H. Song, and M. Qiu, "A scalable and quick-response software defined vehicular network assisted by mobile edge computing," *IEEE Communications Magazine*, vol. 55, no. 7, pp. 94–100, 2017. DOI: 10.1109/MCOM.2017.1601150.

- [32] S. Bateni and C. Liu, “Neuos: A latency-predictable multi-dimensional optimization framework for dnn-driven autonomous systems,” in *USENIX Annual Technical Conference (ATC)*, USA, 2020.
- [33] M. Raeisi-Varzaneh, O. Dakkak, A. Habbal, and B.-S. Kim, “Resource scheduling in edge computing: Architecture, taxonomy, open issues and future research directions,” *IEEE Access*, vol. 11, pp. 25 329–25 350, 2023. DOI: 10.1109/ACCESS.2023.3256522.
- [34] W. Zhou, L. Fan, F. Zhou, F. Li, X. Lei, W. Xu, and A. Nallanathan, “Priority-aware resource scheduling for uav-mounted mobile edge computing networks,” *IEEE Transactions on Vehicular Technology*, vol. 72, no. 7, pp. 9682–9687, 2023. DOI: 10.1109/TVT.2023.3247431.
- [35] T. Zheng, J. Wan, J. Zhang, and C. Jiang, “Deep reinforcement learning-based workload scheduling for edge computing,” *Journal of Cloud Computing*, vol. 11, 2022. DOI: 10.1186/s13677-021-00276-0.
- [36] H. Zhu, Y. Cao, X. Wei, W. Wang, T. Jiang, and S. Jin, “Caching transient data for internet of things: A deep reinforcement learning approach,” *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2074–2083, 2019. DOI: 10.1109/JIOT.2018.2882583.
- [37] J. Wu, H. Lin, H. Liu, and L. Gao, “A deep reinforcement learning approach for collaborative mobile edge computing,” in *IEEE International Conference on Communications (ICC) 2022*, pp. 601–606. DOI: 10.1109/ICC45855.2022.9839202.
- [38] H. M. Bashir, A. B. Faisal, M. A. Jamshed, P. Vondras, A. M. Iftikhar, I. A. Qazi, and F. R. Dogar, “Reducing tail latency using duplication: A multi-layered approach,” in *ACM International Conference on emerging Networking Experiments and Technologies (CoNEXT)*, 2019, p. 246–259. DOI: 10.1145/3359989.3365432.
- [39] M. Primorac, K. Argyraki, and E. Bugnion, “When to hedge in interactive services,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021, pp. 373–387.
- [40] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, “Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters,” *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 91–104, 2019. DOI: 10.1109/TSC.2016.2611578.
- [41] D. Wang, G. Joshi, and G. W. Wornell, “Efficient straggler replication in large-scale parallel computing,” *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 4, no. 2, apr 2019.
- [42] B. Cai, K. Li, L. Zhao, and R. Zhang, “Less provisioning: A hybrid resource scaling engine for long-running services with tail latency guarantees,” *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1941–1957, 2022. DOI: 10.1109/TCC.2020.3016345.
- [43] S. Wen, R. Han, C. H. Liu, and L. Y. Chen, “Fast drl-based scheduler configuration tuning for reducing tail latency in edge-cloud jobs,” *Journal of Cloud Computing*, vol. 12, no. 1, p. 90, 2023.
- [44] S. P. Rachuri, N. Shaik, M. Choksi, and A. Gandhi, “Ecoedgeinfer: Dynamically optimizing latency and sustainability for inference on edge devices,” in *IEEE/ACM Symposium on Edge Computing (SEC)*, 2024, pp. 191–205. DOI: 10.1109/SEC62691.2024.00023.
- [45] W.-C. Chang and P.-C. Wang, “Adaptive replication for mobile edge computing,” *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 11, pp. 2422–2432, 2018. DOI: 10.1109/JSAC.2018.2874140.
- [46] B. Choudhury, S. Choudhury, and A. Dutta, “A proactive context-aware service replication scheme for adhoc iot scenarios,” *IEEE Transactions on Network Service and Management*, vol. 16, no. 4, pp. 1797–1811, 2019. DOI: 10.1109/TNSM.2019.2928698.
- [47] K. Li, M. Tao, and Z. Chen, “Exploiting computation replication for mobile edge computing: A fundamental computation-communication tradeoff study,” *IEEE Transactions on Wireless Communications*, vol. 19, no. 7, pp. 4563–4578, 2020. DOI: 10.1109/TWC.2020.2985039.
- [48] Q. Li, X. Ma, A. Zhou, C. Joo, and S. Wang, “Online service request duplicating for vehicular applications,” *IEEE Transactions on Mobile Computing*, vol. 22, no. 7, pp. 4168–4180, 2023. DOI: 10.1109/TMC.2022.3148170.

APPENDIX A

ALGORITHMIC COMPLEXITY ANALYSIS

The algorithmic complexity of the SafeTail framework consists of two key components: the redundant scheduling space and the deep learning-based scheduling model.

Given n edge servers, the total number of possible non-empty subsets for redundant scheduling is: $2^n - 1$. This leads to an exponential search space, making exhaustive enumeration

impractical. For instance, when $n = 5$, there are $2^5 - 1 = 31$ possible combinations. Thus, the scheduling complexity in the worst case is: $O(2^n)$. However, in practical scenarios, such as within a confined area of approximately 2.5 kilometers in radius, the value of n is unlikely to exceed 10 [17].

State and Action Space Complexity: Each environment state ω_t includes the dynamic features of all edge servers and user-device characteristics. The dimensionality of the state vector grows linearly with the number of edge servers n , i.e., State Complexity: $O(n)$.

The action space corresponds to the set of all non-empty subsets of edge servers, which results in: Action Space Size: $O(2^n)$.

Neural Network Forward Pass Complexity SafeTail uses a feed-forward neural network (FNN) for policy learning. Let:

- d : input dimension ($O(n)$)
- h : number of neurons per hidden layer
- l : number of hidden layers
- $o = 2^n - 1$: output dimension (number of actions)

Then, the complexity of a single forward pass through the FNN is: $O(dh + (l - 1)h^2 + ho)$.

Training Complexity: Training occurs episodically. Let κ be the batch size (number of state-action-reward tuples), then the training complexity per update step is: $O(\kappa \cdot (dh + (l - 1)h^2 + ho))$.

TABLE II
COMPLEXITY SUMMARY OF SAFETAIL FRAMEWORK

Component	Complexity
Redundant scheduling space	$O(2^n)$
State dimensionality	$O(n)$
Action space size	$O(2^n)$
Neural network forward pass	$O(dh + (l - 1)h^2 + ho)$
Training (per batch)	$O(\kappa \cdot (dh + (l - 1)h^2 + ho))$

While SafeTail’s worst-case scheduling complexity remains exponential due to the redundant action space, the use of a deep learning-based approach circumvents exhaustive search. This enables near-optimal scheduling decisions in real-time for small to moderately sized edge networks ($n \leq 10$). For larger networks, further optimization techniques (e.g., hierarchical action pruning) may be required.

The comparative analysis reveals that SafeTail achieves a practical balance between expressiveness and tractability by leveraging a combinatorial DRL-based policy while constraining the number of edge servers to $n \leq 10$. This allows it to make latency-aware, near-optimal scheduling decisions without incurring intractable computational costs. In contrast, traditional heuristics such as Rand-x and MinLoad-x offer low complexity but lack adaptivity, resulting in suboptimal tail latency performance. DRL-Linear [28] introduces more expressive control through joint offloading and resource allocation but suffers from an exponential action space that severely limits scalability. On the other hand, EdgeTuner [43] provides excellent scalability and fast training by fixing the action space to scheduler configurations and leveraging a simulator for offline learning. However, it is inherently limited to tuning configuration parameters rather than making fine-grained, redundancy-aware scheduling decisions like SafeTail.

TABLE III
COMPARATIVE ALGORITHMIC COMPLEXITY ANALYSIS OF SAFETAIL AND BASELINES

Method	Complexity	Remarks
Oracle (Optimal)	$O(n)$	Executes on all n servers; yields minimum latency but with maximum resource consumption.
Rand-x (Random Selection)	$O(x)$	Simple and fast heuristic; selects x servers randomly without latency or load awareness.
MinLoad-x	$O(n \log x)$	Selects x least-loaded servers based on server queue or utilization metrics; ignores network delay.
MinProp-x	$O(n \log x)$	Picks x servers with lowest propagation latency; does not adapt to server-side load variations.
DRL-Linear [28]	Action: $O(2^n \cdot m^n)$ DQN Inference: $O(dh + (l-1)h^2 + ho)$; m = Number of discrete bins used for resource levels in joint offloading models	Joint binary offloading + resource allocation leads to exponential action space; scalability limited for larger user sets.
TLORA [9]	$O(T \times n \times M)$; T = number of time slots; M = number of historical latency samples stored per server	Statistical method using Weibull-based tail estimation; low online cost, but adaptivity limited by model fit.
EdgeTuner [43]	Action: $O(1)$ (fixed-size config space) State: $O(n + T)$ Inference: $O(dk + (l-1)h^2 + ho)$	DRL-based scheduler configuration tuning; highly scalable via simulator-based training and compact action space.
SafeTail (Ours)	Scheduling: $O(2^n)$; NN Inference: $O(dh + (l-1)h^2 + ho)$; Training: $O(\kappa \cdot \text{inference})$	Redundancy-based DRL policy with combinatorial action space; scalable for $n \leq 10$ in practical edge deployments.

TABLE II summarizes the complexity of SafeTail and all the baseline methods along with their limitations.

APPENDIX B QUANTITATIVE ANALYSIS OF EDGE SERVER REACHABILITY CONSTRAINT

The SafeTail framework assumes that the number of edge servers reachable from a user is fewer than 10 [17]. This assumption plays a critical role in making the framework computationally efficient and practically deployable in edge environments.

Effect on Action Space Size: SafeTail considers all non-empty subsets of reachable edge servers for redundancy scheduling. The number of possible actions is given by: $|\mathcal{A}| = 2^n - 1$.

This exponential growth justifies constraining $n < 10$ to ensure tractable action selection.

Effect on Neural Network Complexity: The output layer of the deep neural network in SafeTail must match the number of actions, i.e., $2^n - 1$. As n increases, the number of output neurons grows exponentially, significantly increasing the computational burden of both inference and training.

Comparative Analysis: Table IV presents a comparative analysis of SafeTail's performance as the number of edge servers increases from $n = 5$ to $n = 10$.

TABLE IV
COMPARISON OF REALISTIC SCENARIOS WITH DIFFERENT NUMBER OF REACHABLE EDGE SERVERS

Metric	$n = 5$	$n = 10$
Action space size	31	1023
Inference time (ms)	40	45
Memory usage per inference (KB)	75.2	1228
Training episodes required	10,000	30,000
Avg. access rate (post-convergence)	1.5–2.5	4–5
Tail latency improvement (over Rand-1)	1.7–2.6×	1.5–2.2×

The assumption that fewer than 10 edge servers are reachable from a user is both realistic and strategic. It makes

the SafeTail framework scalable, efficient, and well-suited for deployment in dynamic, resource-constrained edge environments.

APPENDIX C OPTIMALITY GAP ANALYSIS

In this section, we present an empirical analysis of the optimality gap for SafeTail, demonstrating that it consistently produces solutions close to the optimal across all scenarios.

While the Oracle baseline provides the minimum achievable latency by scheduling a request on all edge servers, it incurs the highest resource usage. To assess the trade-off, we compute the relative optimality gap of SafeTail as:

$$\text{Gap}_{\text{rel}} = \frac{L_{\text{SafeTail}} - L_{\text{Oracle}}}{L_{\text{Oracle}}} \times 100\%$$

Table V compares SafeTail and Oracle across all use cases.

TABLE V
OPTIMALITY GAP BETWEEN SAFETAIL AND ORACLE

Use Case	Target Latency (s)	Percentile	Oracle (s)	SafeTail (s)	Gap (%)	Access (%)
Object Detection	1.20	50%	0.33	0.36	8.33	39.7
		99%	0.70	0.89	27.14	
Segmentation	8.94	50%	8.09	8.94	10.50	65.6
		99%	20.03	23.73	18.47	
Noise Removal	3.12	50%	2.37	3.09	30.38	38.91
		99%	18.54	19.29	4.04	

Despite a modest increase in latency (maximum 30%), SafeTail significantly reduces resource usage by selectively choosing 1–3 edge servers, compared to Oracle's use of all servers. This validates that SafeTail achieves near-optimal performance while maintaining efficiency in edge environments.