Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Queue-Aware Task Offloading in the ComputingContinuum under Dynamic Workloads

Lluis Mas

lluis.mas@udl.cat

Universitat de Lleida

Jordi Mateo

Universitat de Lleida

Jordi Vilaplana

Universitat de Lleida

Josep Rius

Universitat de Lleida

Research Article

Keywords: edge computing, task offloading, queueing theory, latency optimization, resource allocation, computing continuum

Posted Date: July 9th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10113868/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Queue-Aware Task Offloading in the Computing Continuum under Dynamic Workloads

Lluis Mas[1*], Josep Rius[2], Jordi Vilaplana[1], Jordi Mateo[1]

[1*]Department of Computer Engineering and Digital Design, University of Lleida, Lleida, Spain.
[2]Department of Economics, University of Lleida, Lleida, Spain.

*Corresponding author(s). E-mail(s): lluis.mas@udl.cat;
Contributing authors: josep.riustorrento@udl.cat;
jordi.vilaplana@udl.cat; jordi.mateo@udl.cat;

Abstract

The computing continuum connects local devices, edge nodes, and cloud systems, where tasks must be dynamically assigned across heterogeneous resources. A key challenge is task offloading under shared and time-varying conditions, where queueing effects and heavy-tailed service variability make latency difficult to predict.

This paper proposes a queue-aware offloading method that uses delay estimation as a decision mechanism. Unlike prior approaches that apply either steady-state analytical models or instantaneous queue snapshots in isolation, the proposed hybrid estimator analytically corrects a $\boldsymbol{G/G/c}$ waiting-time approximation using the Allen–Cunneen formula and blends it with per-tier real-time queue observations through an adaptive, load-dependent weight, enabling per-task delay prediction under Pareto service times and time-varying Poisson arrivals without requiring training, offline calibration, or a fixed mixing hyperparameter. Tasks are assigned online to the tier with the minimum estimated delay.

We evaluate the approach using both a discrete-event simulator and an HTTP-based emulation testbed with asynchronous request dispatch, real queue contention, and wall-clock measurement, under increasing load, bursty arrivals, and varying edge capacity. Results show that the proposed method substantially reduces delay-estimation error relative to its model-only and queue-only components across all tested load levels, and that this estimation accuracy translates into reduced latency and improved tail behavior relative to static and analytically-naive baselines, with measured per-decision scheduling overhead in the low-microsecond range in both simulation and emulation.

Keywords: edge computing, task offloading, queueing theory, latency optimization, resource allocation, computing continuum

1 Introduction

The computing continuum connects user devices, edge nodes, and cloud systems. This paradigm has emerged due to the increasing amount of data generated close to users and the need for low-latency applications such as IoT, 5G, and real-time analytics [1, 2].

In this setting, the main challenge is not where computation is possible, but where it should be executed at runtime.

A central problem is task offloading. For each incoming task, the system must decide whether to execute it locally, at the edge, or in the cloud. This decision depends on both communication delay and computation time, and must be made under uncertainty.

In practice, edge and cloud systems are shared among many users, which makes system behavior difficult to predict. A major source of delay is queueing. As load increases, edge nodes can become congested and waiting time becomes the dominant component of latency. In addition, service times may exhibit significant variability, which further increases delay and uncertainty.

Queueing theory provides tools to model these effects. Prior work uses queueing models to analyze cloud and edge systems [3, 4] or combines them with optimization techniques to minimize delay [5, 6]. These approaches provide useful insights but typically assume fixed conditions and focus on steady-state behavior.

Learning-based methods have been proposed to handle dynamic environments [7, 8]. These approaches can adapt to changing conditions, but they suffer from high sample complexity, lack formal explainability, and do not offer the performance or safety guarantees typical of analytical bounds.

Overall, existing approaches face two main limitations. First, many assume stationary conditions and do not capture bursty or time-varying workloads. Second, few methods provide a simple mechanism for per-task decision making based on current system conditions.

In dynamic environments, decisions must be made online. As utilization approaches capacity, waiting time increases rapidly, especially at the edge. Assigning tasks to a highly loaded tier can lead to significantly higher delay than sending them to a less congested alternative. This highlights the need for accurate delay estimation at runtime.

In this paper, we propose a queue-aware offloading method that uses delay estimation as a decision mechanism. The method combines analytical queueing models with real-time queue observations to estimate delay under dynamic conditions. Analytical models provide stable predictions under load, while queue observations capture instantaneous congestion.

The key idea is to combine these two sources of information into an adaptive hybrid estimator that reduces prediction error. The blending weight between the two sources

is itself derived from the observed system state—rather than fixed in advance—so the estimator automatically shifts from analytical prediction toward reactive observation as congestion builds. This leads to more accurate delay estimates and, consequently, better task placement decisions.

Based on this estimate, each task is assigned to the tier with the minimum expected delay. The decision is myopic, requires only local information, and does not require training.

To assess whether this design holds beyond an idealized discrete-event abstraction, we additionally validate the policy on an HTTP-based emulation testbed with real asynchronous request dispatch, queue contention, and wall-clock measurement. Many continuum-offloading studies stop at simulation; the dual evaluation here is intended to bridge that gap.

The main contributions of this paper are:

- **Queue-aware delay estimation.** A formulation of task offloading that uses online, per-task delay estimation as the decision mechanism, analytically correcting the $G/G/c$ waiting-time approximation via the Allen–Cunneen formula to account for heavy-tailed service-time variability.
- **Per-tier arrival-rate estimation.** A sliding-window estimator that tracks arrival rates separately for each tier from routing history, avoiding the bias of a single global arrival-rate proxy under asymmetric load distribution.
- **Adaptive predictive-reactive hybrid routing.** A blending mechanism in which the weight given to the reactive queue-state estimate versus the analytical model estimate is a deterministic function of the current queue backlog relative to tier capacity (11), requiring no fixed mixing hyperparameter and no offline tuning, combined with a per-tier utilization penalty that discourages routing to nearly saturated tiers.
- **Dual validation in simulation and HTTP emulation.** An experimental evaluation under dynamic conditions—including increasing load, bursty arrivals, and varying edge capacity—using both a discrete-event simulator and an asynchronous HTTP-based emulation testbed with real request dispatch and queue contention, with comparisons against static, dynamic (JSQ, JSQ-S), and oracle baselines, paired statistical significance testing, and measured scheduling overhead.
- An analysis linking delay estimation accuracy to decision quality, showing that the hybrid estimator achieves lower prediction error than both its model-only and queue-only components across all load levels.

2 Related Work

Task offloading in the computing continuum has been studied using a variety of approaches, which can be broadly grouped into queueing-based models, learning-based methods, and hybrid or system-level strategies.

The computing continuum extends the traditional cloud model by integrating user devices, edge/fog nodes, and cloud data centers into a unified multi-tier architecture. This paradigm enables low-latency and resource-efficient execution by dynamically distributing tasks across tiers [1, 2, 9]. Within this setting, task offloading is a central

problem, as decisions must balance latency, resource utilization, and quality of service under dynamic and shared conditions.

Queueing theory has been widely used to model cloud and edge systems and to estimate latency. Early work models cloud platforms using queueing networks to analyze response time and utilization [3]. Similar approaches have been extended to fog and edge environments to study system behavior and identify bottlenecks [4]. Other works combine queueing models with optimization techniques. For example, Katayama and Tachibana [5] model edge and cloud servers using M/M/1 queues and solve an optimization problem to minimize delay, while Li et al. [6] consider heterogeneous systems and optimize task allocation under latency and power constraints. Additional works study latency-aware and joint optimization strategies in edge systems [10–12].

These approaches provide useful tools for system design and capacity planning, but they typically assume stationary conditions and focus on steady-state behavior. As a result, they do not directly support per-task decision making under dynamic and time-varying workloads. In addition, most models rely on simplified assumptions such as exponential service times and Poisson arrivals, which limits their ability to capture bursty or correlated traffic patterns.

Learning-based methods have been proposed to address dynamic environments. Deep reinforcement learning (DRL) approaches, such as those in [7, 8], learn adaptive offloading policies under varying workload and network conditions. More recent work extends this direction by incorporating richer system information. For instance, Anand and Karthikeyan [13] propose a priority-aware DQN-based scheduler that integrates workload prediction and task urgency into the offloading decision. Similarly, Cilić et al. [14] introduce a decentralized multi-armed bandit approach for QoS-aware load balancing in the continuum. Lightweight machine learning models have also been proposed to run directly on resource-constrained edge devices, enabling on-device workload distribution [15].

These methods demonstrate strong adaptability and can improve throughput, latency, and energy efficiency under changing conditions. However, DRL approaches suffer from high sample complexity and substantial training overhead, making online adaptation difficult in non-stationary environments where workloads and network states shift rapidly. Furthermore, unlike control-theoretic or analytical bounds, DRL models lack formal performance or safety guarantees, and their "black-box" nature limits interpretability, making it challenging to debug or explain routing decisions relative to the instantaneous queue state.

Other works focus on queue management and user behavior. Han et al. [16] introduce task impatience, where tasks may leave the system if waiting time exceeds a threshold. While this reduces extreme delays, it assumes known system parameters and does not address task placement across multiple tiers.

In parallel, system-level strategies such as serverless function placement and hierarchical coordination have been proposed to improve performance across the continuum. For example, Russo et al. [17] study QoS-aware offloading policies for serverless functions in cloud-edge systems, showing that coordinated placement across tiers can reduce latency and improve resource utilization.

Despite these advances, modeling queueing dynamics under time-varying workloads remains challenging. Traditional formulations typically model user devices as M/M/1 queues, edge nodes as M/M/c, and cloud systems as M/M/∞. While analytically tractable, these assumptions fail to capture service-time variability and transient congestion. Recent work begins to address this limitation. For instance, Kovtun [18] models edge nodes using a G/G/1 formulation with adaptive service shifts learned via reinforcement learning, demonstrating significant delay reductions.

Overall, existing approaches either rely on analytical models under fixed conditions or use learning-based methods that require training and introduce additional complexity. Specifically: Katayama and Tachibana [5] assume stationary M/M/1 queues and solve the allocation offline, providing no per-task online reaction to congestion. Li et al. [6] extend this to heterogeneous systems but still optimize under steady-state assumptions with exponential service. Kovtun [18] introduces G/G/1 modeling but couples it with reinforcement learning, requiring training and losing analytical guarantees. JSQ-based policies observe queue lengths but ignore service-time variability and network delay, causing systematic underestimation near saturation under heavy-tailed workloads. No existing approach simultaneously satisfies all three properties needed for practical continuum offloading: (i) online per-task decision making, (ii) explicit modeling of non-exponential service-time variability via a $G/G/c$ approximation, and (iii) training-free, lightweight operation with per-tier congestion awareness.

In this work, we address this gap by proposing a queue-aware offloading method whose hybrid estimator analytically corrects the Allen–Cunneen $G/G/c$ waiting-time approximation using per-tier arrival rate telemetry and blends it with real-time queue observations. This combination enables per-task decisions under dynamic, heavy-tailed conditions while remaining lightweight and training-free—a combination not achieved by any prior approach (see Table 1).

Table 1 Comparison of task offloading approaches in the computing continuum. "General Service" indicates support for non-exponential service time distributions. "Hybrid Estimator" indicates a combined analytical + queue-state delay estimate.

Work	Method	Online	General Service	Hybrid Estimator	Lightweight
[3]	Queueing model (cloud)	No	No	No	Yes
[4]	Queueing model (fog)	No	No	No	Yes
[5]	Queueing + optimization	No	No	No	No
[6]	Queueing + optimization	No	Partial	No	No
[16]	Queue control (impatience)	Partial	No	No	Partial
[7]	Deep RL	Yes	Implicit	No	No
[8]	Deep RL (vehicular)	Yes	Implicit	No	No
[13]	Deep RL (priority-aware)	Yes	Implicit	No	No
[14]	Bandit-based (decentralized)	Yes	Implicit	No	Yes
[18]	RL + queue model (G/G/1)	Yes	Yes	No	No
This work	**Queue-aware policy**	**Yes**	**Yes**	**Yes**	**Yes**

Table 1 summarizes the main differences between existing approaches and the proposed method.

3 System Model and Problem Formulation

We consider a computing continuum composed of three processing tiers: local, edge, and cloud. The local tier represents user devices with limited capacity, the edge tier provides nearby servers with finite parallelism, and the cloud tier represents a high-capacity system with a large number of servers. The set of tiers is denoted by $\mathcal{T} = \{L, E, C\}$.

Each tier $i \in \mathcal{T}$ is modeled as a queueing system with service rate μ_i and capacity c_i. The local tier has a single server, the edge tier has limited parallelism, and the cloud tier is modeled as a high-capacity system with large c_C, which approximates an infinite-server regime.

Tasks arrive over time according to a stochastic process. In the baseline setting, arrivals follow a Poisson process with rate λ. To capture dynamic conditions, we also consider bursty workloads where the arrival rate changes over time. A global arrival rate estimate is maintained online using a sliding window over recent arrivals:

$$\lambda(t) = \frac{|\mathcal{A}(t)|}{\Delta}, \tag{1}$$

where $\mathcal{A}(t)$ is the set of arrivals in the interval $[t - \Delta, t]$, and Δ is the duration of the sliding window (set to $\Delta = 100$ virtual time units in our evaluation).

To avoid the bias that arises from using a global arrival rate under asymmetric traffic distribution, we additionally track per-tier arrival rates $\lambda_i(t)$ online using a sliding window over tasks assigned to tier i:

$$\lambda_i(t) = \frac{|\mathcal{A}_i(t)|}{\Delta}, \tag{2}$$

where $\mathcal{A}_i(t)$ is the set of tasks assigned to tier i in the interval $[t - \Delta, t]$. This per-tier estimate replaces the global estimate (1) in all per-tier computations below. The per-tier utilization is then:

$$\rho_i(t) = \frac{\lambda_i(t)}{c_i \mu_i}. \tag{3}$$

Service times are variable and may exhibit heavy-tailed behavior. We model service times using a Pareto distribution with shape parameter $\alpha > 2$ and scale parameter x_m, giving mean $\bar{s} = \alpha x_m/(\alpha - 1)$. The squared coefficient of variation (SCV) used by the Allen–Cunneen correction factor is configured as:

$$C_s^2 = \frac{\alpha}{\alpha - 2}, \quad \alpha > 2, \tag{4}$$

which for $\alpha = 2.5$ gives $C_s^2 = 5.0$. This value amplifies the $M/M/c$ waiting-time estimate by a factor of $(C_a^2 + C_s^2)/2 = 3$ relative to the exponential ($C_s^2 = 1$) case, reflecting

the heavy-tailed variability that motivates the use of $G/G/c$ queueing approximations rather than $M/M/c$ models.

For each tier i, the total delay is defined as

$$T_i(t) = W_i(t) + \frac{1}{\mu_i} + D_i(t), \tag{5}$$

where $W_i(t)$ is the waiting time in queue and $D_i(t)$ is the communication delay. Network delay is time-dependent and modeled as

$$D_i(t) = d_i \left(1 + 0.5 \sin\left(\frac{t}{500}\right)\right), \tag{6}$$

where d_i is the base delay of tier i.

The main difficulty lies in estimating the waiting time $W_i(t)$. Exact expressions for $G/G/c$ systems are not available in closed form, and practical systems operate under time-varying conditions. Analytical approximations provide stable estimates but may not reflect instantaneous congestion, while queue-based estimates react to the current state but can be noisy.

To capture both effects, we consider two complementary estimates. The first is an analytical approximation based on the Allen–Cunneen $G/G/c$ formula [19]:

$$W_i^{\text{model}}(t) = \frac{C_a^2 + C_s^2}{2} \cdot W_i^{M/M/c}(t), \tag{7}$$

where C_a^2 is the squared coefficient of variation of inter-arrival times (here $C_a^2 = 1$ for Poisson arrivals), C_s^2 is given by (4), and $W_i^{M/M/c}(t)$ is the $M/M/c$ mean waiting time computed from the Erlang-C formula:

$$W_i^{M/M/c}(t) = \frac{C(c_i, \rho_i(t))}{c_i\mu_i - \lambda_i(t)}, \tag{8}$$

with the Erlang-C probability $C(c, \rho)$ defined as

$$C(c, \rho) = \frac{\frac{(c\rho)^c}{c!\,(1-\rho)}}{\sum_{k=0}^{c-1} \frac{(c\rho)^k}{k!} + \frac{(c\rho)^c}{c!\,(1-\rho)}}, \quad \rho = \frac{\lambda_i(t)}{c_i\mu_i} < 1. \tag{9}$$

For a single-server tier ($c_i = 1$), Eq. (8) reduces to the standard $M/M/1$ form $W_i^{M/M/1}(t) = \rho_i(t)/(\mu_i - \lambda_i(t))$, since $C(1, \rho) = \rho$. For over-loaded conditions ($\rho_i(t) \geq 1$), we set $W_i^{M/M/c}(t) = M_{\max}$, a large sentinel value that effectively redirects traffic to less congested tiers.

The second estimate is a queue-state estimate based on observed queue occupancy. For a tier with c_i parallel servers and $q_i(t)$ tasks currently waiting in queue, the

expected waiting time for the next arrival is approximated as:

$$W_i^{\text{queue}}(t) = \frac{\max(0,\, q_i(t) - n_i(t))}{c_i \mu_i}, \tag{10}$$

where $n_i(t) \leq c_i$ is the number of currently active servers. This expression accounts for the c_i-server parallelism: tasks already in service do not contribute to the waiting queue, and service capacity is shared across c_i servers. Note that this estimate is intentionally simple and reactive; it does not account for service-time variability, which is why blending with the analytical component (7) is necessary.

3.0.1 Adaptive hybrid estimator

Rather than combining $W_i^{\text{model}}(t)$ and $W_i^{\text{queue}}(t)$ with a fixed weight, the proposed estimator uses a *load-dependent* blending weight $\beta_i(t)$ that is computed directly from the observed queue backlog relative to tier capacity:

$$\beta_i(t) = \min\left(1,\, \frac{q_i(t)}{c_i}\right). \tag{11}$$

The hybrid estimate is then

$$W_i(t) = \big(1 - \beta_i(t)\big) \cdot W_i^{\text{model}}(t) \;+\; \beta_i(t) \cdot W_i^{\text{queue}}(t). \tag{12}$$

This formulation has an intuitive interpretation. When tier i is empty ($q_i(t) = 0$), $\beta_i(t) = 0$ and the estimate relies entirely on the analytical Allen–Cunneen prediction $W_i^{\text{model}}(t)$, since there is no observed congestion signal to react to. As the queue backlog grows relative to capacity, $\beta_i(t)$ increases linearly, shifting weight toward the directly observed queue-state estimate $W_i^{\text{queue}}(t)$; once the backlog reaches or exceeds the tier's parallelism ($q_i(t) \geq c_i$), the estimate relies entirely on the reactive queue signal ($\beta_i(t) = 1$). This makes the blending weight a deterministic, parameter-free function of the instantaneous system state, requiring no additional tuning beyond the queueing-model parameters already used in $W_i^{\text{model}}(t)$ and $W_i^{\text{queue}}(t)$.

As $\rho_i(t) \to 1$, waiting time grows sharply. To prevent routing to nearly saturated tiers, we apply a multiplicative utilization penalty to the adaptively blended estimate:

$$\hat{W}_i(t) = W_i(t) \cdot \left(1 + \gamma_{\text{util}} \max(0,\, \rho_i(t) - \rho_0)\right), \tag{13}$$

where $\rho_0 = 0.9$ is the activation threshold and $\gamma_{\text{util}} = 5.0$ is the penalty scaling factor. The final waiting time used in the decision is $\hat{W}_i(t)$.

For the cloud tier, we assume a high-capacity regime ($c_C = 1000$) and set $W_C(t) = 0$ unconditionally, i.e., $T_C(t) = 1/\mu_C + D_C(t)$. The cloud tier is modeled as effectively infinite-server: queueing delay is negligible at the loads considered, and the utilization penalty (13) is not applied to the cloud tier.

The decision problem is to assign each incoming task to the tier that minimizes the estimated total delay:

$$i^*(t) = \arg\min_{i \in \mathcal{T}} \left[\hat{W}_i(t) + \frac{1}{\mu_i} + D_i(t) \right]. \tag{14}$$

With probability ϵ (set to $\epsilon = 0.05$), the task is instead assigned to a uniformly random tier. This ϵ-greedy exploration step serves two purposes: it prevents the per-tier rate estimators (2) from collapsing to zero for tiers that are persistently under-selected, and it provides a small amount of probing that keeps the queue-state observations (10) informative across all tiers.

This defines an online decision rule based solely on current system observations. The effective arrival rate at each tier is a consequence of these decisions and is not assumed known in advance.

We compare the proposed method with baseline policies, including random assignment, edge-first, dynamic Join-the-Shortest-Queue (JSQ), and an oracle policy with full knowledge of the system state. These baselines allow us to evaluate both the quality of the delay estimates and their impact on task placement.

The model relies on approximate queueing formulas and online estimates, without requiring exact system parameters. Tasks are assumed to be independent and stateless. This formulation captures variability in service times, time-varying arrivals, and dynamic network delays, while remaining suitable for real-time decision making.

4 Queue-Aware Offloading Algorithm

We propose a queue-aware offloading method that selects, for each incoming task, the execution tier with the lowest estimated delay. The key idea is to combine analytical queueing models with real-time system observations in order to improve delay estimation under dynamic conditions.

At each decision point, the algorithm estimates the total delay at each tier using Eq. (5). The service rate μ_i, server count c_i, number of active servers $n_i(t)$, and current queue length $q_i(t)$ are directly observed. The per-tier arrival rate $\lambda_i(t)$ is estimated online using a sliding window as in Eq. (2). Network delay is modeled by Eq. (6).

The main challenge lies in estimating the waiting time. Analytical queueing models provide stable estimates under steady conditions, but they may not reflect instantaneous congestion. In contrast, queue-based estimates react immediately to the current state but can be noisy and sensitive to short-term fluctuations. Relying on either source alone leads to systematic errors, as confirmed by the experimental results in Section 6.7.

The algorithm uses the adaptive hybrid estimator (11)–(12), with the blending weight $\beta_i(t)$ computed independently for each tier from its current queue backlog relative to capacity. The analytical component dominates when a tier is empty, providing stability and a forward-looking estimate of congestion before it is directly observable; the queue-state component dominates once a backlog has formed, providing fast reaction to transient congestion. The utilization penalty (13) is then applied whenever $\rho_i(t) > \rho_0 = 0.9$, discouraging routing to nearly saturated tiers.

Algorithm 1 Queue-Aware Offloading

for each incoming task at time t **do**
  Update per-tier arrival histories; compute $\lambda_i(t)$ via Eq. (2) for each $i \in \mathcal{T}$
  if rand() $< \epsilon$ **then**
    Assign task to a uniformly random tier $i \in \mathcal{T}$; **continue**
  end if
  for each tier $i \in \mathcal{T}$ **do**
    Observe queue length $q_i(t)$ and active server count $n_i(t)$
    Compute per-tier utilization $\rho_i(t) = \lambda_i(t)/(c_i\mu_i)$ via Eq. (3)
    if tier i is Cloud **then**
      $\hat{W}_i(t) \leftarrow 0$
    else
      Compute analytical estimate $W_i^{\text{model}}(t)$ via Eqs. (7)–(9)
      Compute queue-state estimate $W_i^{\text{queue}}(t)$ via Eq. (10)
      Compute adaptive weight $\beta_i(t) = \min(1, q_i(t)/c_i)$ (Eq. (11))
      Combine: $W_i(t) = (1-\beta_i(t))W_i^{\text{model}}(t) + \beta_i(t)W_i^{\text{queue}}(t)$ (Eq. (12))
      Apply utilization penalty $\hat{W}_i(t)$ via Eq. (13) if $\rho_i(t) > \rho_0$
    end if
    Compute total delay: $T_i(t) = \hat{W}_i(t) + 1/\mu_i + D_i(t)$
  end for
  Select tier: $i^*(t) = \arg\min_{i\in\mathcal{T}} T_i(t)$ (Eq. (14))
  Assign task to tier $i^*(t)$; update $\mathcal{A}_{i^*(t)}(t)$
end for

The tier selection follows Eq. (14), with ϵ-greedy exploration ($\epsilon = 0.05$). Algorithm 1 summarizes the complete decision process.

4.1 Complexity Analysis

The decision is myopic and requires only local information available at each tier. For each incoming task, the algorithm evaluates $|\mathcal{T}|$ tiers, and for each tier performs a constant number of arithmetic operations plus one evaluation of the Erlang-C probability (9), which involves a finite sum over c_i terms. The per-task decision cost is therefore

$$O\left(\sum_{i\in\mathcal{T}} c_i\right) = O(|\mathcal{T}| \cdot c_{\max}), \tag{15}$$

where $c_{\max} = \max_i c_i$. In our configuration $|\mathcal{T}| = 3$ and $c_{\max} = c_C = 1000$ for the cloud tier; however, the cloud tier is handled by the constant-time branch ($\hat{W}_C(t) = 0$, no Erlang-C evaluation), so the effective per-task cost in practice is dominated by the local and edge tiers, both with small $c_i \in \{1, 2\}$, making the decision effectively $O(1)$.

Memory overhead consists of the per-tier sliding-window arrival histories $\mathcal{A}_i(t)$, each holding at most $O(\lambda_i \cdot \Delta)$ recent timestamps, giving total memory

$$O(|\mathcal{T}| \cdot \lambda_{\max} \cdot \Delta)\,, \tag{16}$$

where $\lambda_{\max}$ is the highest per-tier arrival rate and Δ is the sliding-window duration. For the configurations evaluated here ($\Delta = 100$, $\lambda \leq 4$ per tier), this amounts to a few hundred floating-point timestamps per tier—negligible for any edge node.

Empirically, Section 6.11 reports the measured wall-clock cost of `policy.select()` in both the simulator and the HTTP emulation testbed, confirming that the decision overhead remains in the low-microsecond range, several orders of magnitude below typical network round-trip times.

5 Experimental Evaluation

5.1 Objectives

The objective of the evaluation is to assess how accurately the proposed method estimates delay and how this affects task offloading decisions. The experiments are designed to evaluate both performance and the underlying mechanism of the approach.

We focus on three main aspects. First, we evaluate latency under increasing load. Second, we analyze how delay estimation accuracy relates to performance. Third, we study the behavior of the policy under dynamic conditions and its ability to adapt to congestion.

5.2 Simulation Setup

We evaluate the system using a discrete-event simulator implemented in Python with SimPy. Each experiment is repeated 10 times with different random seeds. Results are reported as mean values with 95% confidence intervals.

Each simulation runs for 100,000 time units. This duration is sufficient to observe steady-state behavior as well as transient effects under dynamic workloads.

5.2.1 System Configuration

The system consists of three tiers: local, edge, and cloud.

The local tier has service rate $\mu_L = 3.0$, one server ($c_L = 1$), and base delay $d_L = 0.1$. The edge tier has service rate $\mu_E \in \{3.0, 5.0, 8.0\}$, two servers ($c_E = 2$), and base delay $d_E = 0.5$. The cloud tier has service rate $\mu_C = 10.0$, $c_C = 1000$ servers, and base delay $d_C = 1.5$.

The cloud tier approximates a high-capacity system with negligible queueing delay, but higher communication cost.

5.2.2 Workload Model

Tasks arrive according to a Poisson process with rate λ. To evaluate performance under increasing load, we vary $\lambda \in \{1.0, 2.0, 3.0\}$ tasks per virtual time unit.

Service times follow a Pareto distribution with shape parameter $\alpha = 2.5$, yielding $C_s^2 = 5.0$ per Eq. (4). This heavy-tailed setting motivates the use of $G/G/c$ approximations over $M/M/c$ models.

To model dynamic conditions, we also consider bursty workloads. In each simulation replication, the arrival rate increases by a factor of 2.5 starting at a random

timestamp uniformly distributed between $t = 1000$ and $t = 5000$. This randomization decouples the burst onset from the phase of the sinusoidal network delay (Eq. (6)), eliminating potential phase alignment bias when averaging results across replications.

5.3 HTTP-based Emulation Testbed

The discrete-event simulator described above models each tier as an abstract queueing system. To assess whether the proposed policy's behavior is an artifact of this abstraction, we additionally implement an HTTP-based emulation testbed in which each tier is a real asynchronous server process and tasks are dispatched as real HTTP requests.

5.3.1 Worker servers

Each tier (Local, Edge, Cloud) is realized as an `aiohttp` server process bound to a dedicated local port (configured via `EMUL_PORT_MAP`). Each worker server maintains an `asyncio.Semaphore` sized to the tier's capacity c_i, so that at most c_i requests are processed concurrently and any additional arriving requests are held in an `asyncio.Queue`; requests arriving when this queue is full are rejected with an HTTP `503` response, recorded as dropped tasks. This reproduces real queue contention—multiple concurrent requests genuinely compete for the same server-side concurrency primitive, rather than being accounted for analytically.

5.3.2 Request processing and injected variability

Upon dequeuing a request, a worker draws a Pareto-distributed service time using the same `pareto_service`(μ_i) generator as the simulator (mean $1/\mu_i$, $\alpha = 2.5$), perturbed by a 5% coefficient-of-variation log-normal service jitter (`EMUL_SERVICE_JITTER_CV`), and sleeps asynchronously for this duration to emulate computation. The client-side dispatcher additionally injects a network delay drawn from the tier's base delay d_i (Table 2) with a 10% coefficient-of-variation log-normal network jitter (`EMUL_NETWORK_JITTER_CV`), modeling realistic round-trip variability.

5.3.3 Asynchronous dispatch and wall-clock measurement

An asynchronous dispatcher generates tasks according to the same Poisson/burst arrival process as the simulator and issues each task as a non-blocking HTTP request to the tier selected by the policy under test. Each policy implementation is reused unchanged between the simulator and the emulation testbed: a thin `WorkerProxy` adapter exposes the same interface as the simulator's tier objects (current queue length, active server count, per-tier arrival-rate estimate), so `policy.select()` is called identically in both environments. For every completed task, the testbed records, from real wall-clock measurements: the queue-wait time (directly measured via the semaphore-acquisition delay), the drawn service time, the injected network delay, the total wall-clock sojourn time (`avg_latency`, including HTTP framing overhead), and a derived `avg_sojourn_model_like` value equal to queue-wait + service + network delay—excluding HTTP/event-loop framing overhead—which maps directly onto the simulator's $T_i(t) = W_i(t) + 1/\mu_i + D_i(t)$ for a like-for-like comparison (Section 6.10).

Table 2 Emulation testbed configuration per tier: base network delay d_i (seconds) and capacity c_i, matching the primary simulation topology (`TIER_CONFIGS`).

Tier	μ_i	c_i	d_i (s)
Local	3.0	1	0.001
Edge	5.0	2	0.005
Cloud	10.0	1000	0.015

5.3.4 Scheduling overhead

The wall-clock time spent inside `policy.select()` is measured with `time.perf_counter()` for every dispatched task, in both the simulator and the emulation testbed, yielding the `sched_overhead_mean` and `sched_overhead_p95` metrics reported in Section 6.11.

5.3.5 Scope

The emulation testbed validates the primary operating point ($\mu_E = 5.0$, burst disabled) across the load levels $\lambda \in \{1.0, 2.0, 3.0\}$. Burst and edge-capacity sweeps are simulation-only parametric analyses; the emulation testbed's role is to confirm that the queueing and routing dynamics observed in the simulator are reproduced under real asynchronous request processing, not to repeat the full parametric sweep.

The emulation testbed complements the discrete-event simulator: the simulator enables broad parametric sweeps (load, burst intensity, edge capacity, service-time distribution) at low computational cost, while the emulation testbed validates the policy under real request dispatch, genuine queue contention via OS-level concurrency primitives, and wall-clock timing—properties that an abstract queueing simulation cannot fully capture.

5.4 Policies

We compare the proposed queue-aware policy with several baselines.

5.4.1 Queue-aware policy (proposed)

Selects the tier with minimum estimated delay using the hybrid estimator developed in Section 3 and summarized in Algorithm 1.

5.4.2 Oracle policy

The oracle serves as a lower bound on achievable mean latency for any online policy with access to instantaneous full queue-state information. It is *state-informed* but *non-clairvoyant*: it observes the exact queue lengths and server occupancies at decision

time but does not know future arrival realizations or the exact remaining service time of in-progress tasks.

For a task arriving at time t, the oracle selects the tier minimizing expected sojourn time:

$$i^*(t) = \arg\min_{i \in \mathcal{T}} T_i^{\text{oracle}}(t), \tag{17}$$

where, for a tier with c_i servers:

$$T_i^{\text{oracle}}(t) = \frac{\max(0,\, q_i(t) - n_i(t))}{c_i \mu_i} + R_i(t) + \frac{1}{\mu_i} + D_i(t). \tag{18}$$

Here $q_i(t)$ is the total number of waiting tasks, $n_i(t) \leq c_i$ is the number of servers currently busy, and the first term gives the expected waiting time for the arriving task assuming FCFS service across c_i parallel servers.

The residual service time $R_i(t)$ of in-progress tasks is approximated using the memoryless expectation:

$$R_i(t) = \begin{cases} \dfrac{1}{2\mu_i}, & n_i(t) > 0, \\ 0, & n_i(t) = 0. \end{cases} \tag{19}$$

We adopt this memoryless approximation, rather than the exact mean residual life of the Pareto service distribution, for analytical simplicity and because it is the standard convention in $M/M/c$ oracle constructions. We note, however, that for a Pareto service distribution with shape $\alpha = 2.5$, the inspection-paradox-corrected mean residual life of a task already in service exceeds $1/(2\mu_i)$, so Eq. (19) *underestimates* the true residual work under heavy-tailed service. Consequently, $T_i^{\text{oracle}}(t)$ is itself a slight underestimate of the delay an omniscient router would observe, which makes the oracle a marginally optimistic (tighter) reference: the gaps reported in Section 6 between the proposed policy and the oracle are, if anything, a conservative (lower-bound) estimate of the achievable improvement from better queue-state visibility.

This oracle represents the best online policy achievable under instantaneous full queue visibility without future knowledge. It is not a global offline optimum, and the gap between the proposed policy and this oracle measures the cost of estimation uncertainty alone.

5.4.3 Static baselines

Edge-first assigns all tasks to the edge tier; *random* assigns tasks uniformly across all tiers. These represent uninformed strategies.

5.4.4 Dynamic baselines

Join-the-Shortest-Queue (JSQ) routes each task to the tier with the minimum instantaneous total queue count $q_i(t)$. It is online and reactive but ignores service rates, variability, and network delay.

JSQ-S (JSQ with service times) minimizes instantaneous workload $q_i(t)/\mu_i$, incorporating service rates but still ignoring analytical variability corrections and network delay.

These baselines span the spectrum from fully static to reactive-but-incomplete, allowing us to isolate the contribution of each component of the proposed estimator.

5.4.5 Scope relative to learning-based methods

Several recent works address continuum offloading using deep reinforcement learning (DRL) or bandit-based approaches [7, 8, 13, 14, 18]. Direct head-to-head reproduction of these methods was outside the scope of this study: published DRL approaches differ substantially in state representation, reward shaping, action space granularity, and training environment, making a fair, reproducible re-implementation a significant undertaking in its own right and a confound to the central comparison this paper targets (estimator-driven vs. purely reactive routing). We therefore compare against state-aware dynamic routing baselines—JSQ and JSQ-S—which represent the closest *online, training-free* alternatives and isolate the effect of incorporating analytical delay estimates versus relying on instantaneous queue-state signals alone. A systematic comparison against DRL-based offloading, including training cost and sample-efficiency trade-offs, is an important direction for future work.

5.5 Metrics

We evaluate both performance and estimation accuracy.

Average latency measures the mean end-to-end sojourn time (waiting + service + network delay) per task. Tail latency is measured as the 95^{th} percentile (P95) across all completed tasks. Task distribution measures the fraction of tasks assigned to each tier.

To explain performance, we also measure prediction error: for each task, we compare the estimated delay (computed at decision time) with the observed sojourn time and report the mean absolute relative error. This metric links estimation quality to routing quality.

5.5.1 Statistical significance testing

For each pairwise comparison between the proposed policy and a baseline, we compute a paired t-test (`scipy.stats.ttest_rel`) across the $N = 10$ replications, pairing replications by random seed so that both policies are evaluated under matched arrival and service-time realizations. We report the mean relative difference in average latency, the t-statistic, and the two-sided p-value, with significance assessed at $\alpha = 0.05$. The same procedure is applied to the oracle gap (per-policy gap to the oracle, Section 6.4), to the simulation-vs-emulation latency comparison (Section 6.10), and to the simulation-vs-emulation scheduling overhead comparison (Section 6.11).

5.6 Experiments

5.6.1 Load experiment

We vary the arrival rate $\lambda \in \{1.0, 2.0, 3.0\}$ to evaluate performance under increasing load. This experiment shows how each policy behaves as queueing effects become more significant and the system approaches saturation.

5.6.2 Burst experiment

We evaluate performance under non-stationary conditions. The system starts at $\lambda = 2.0$; after a random time uniformly distributed in $[1000, 5000]$, the arrival rate increases to 2.5λ and remains elevated. This experiment evaluates adaptation speed and stability under sudden workload changes.

5.6.3 Sensitivity to edge capacity

We vary the edge service rate $\mu_E \in \{3.0, 5.0, 8.0\}$ under fixed load $\lambda = 2.0$. This evaluates how the policy adapts to different levels of available edge capacity and whether it can exploit additional resources effectively.

5.7 Reproducibility

The discrete-event simulator is implemented using SimPy [20]; the emulation testbed (Section 5.3) is implemented using `aiohttp` and `asyncio`. Random seeds are fixed and enumerated across replications to ensure full reproducibility in both pipelines. The implementation is publicly available at https://github.com/placeholder/queue-aware-offloading and is organized into independent modules: `config.py` (shared configuration and result schema), `sim.py` (discrete-event simulation and policy implementations), `emulation.py` (HTTP-based emulation testbed), `runner.py` (experiment orchestration, aggregation, and statistical tables), and `plot.py` (figure generation). Each policy is implemented as an interchangeable module and is used unchanged by both the simulator and the emulation testbed; new policies or system configurations can be added without modifying the simulation or emulation core.

6 Results and Discussion

This section presents the results of the experimental evaluation. All values are reported as mean $\pm$ 95% confidence interval over 10 independent replications with distinct random seeds.

6.1 Performance under Increasing Load

Figure 1 shows the average latency as a function of load. At low load ($\lambda = 1.0$), all policies achieve similar performance, as queueing effects are limited and latency is dominated by service and network delay.

As load increases, differences between policies become more pronounced. The model-only baseline degrades fastest, as its analytical estimate alone does not react to instantaneous congestion. The edge-first policy concentrates all tasks on a single tier regardless of its state, leading to high utilization and increased waiting time as load grows. The queue-aware policy distributes tasks across tiers based on its adaptive hybrid estimate, substantially reducing mean latency relative to both of these baselines across all tested load levels (Section 5.5.1).

Greedy queue-length heuristics—JSQ and JSQ-S—remain strong baselines for raw mean latency in this configuration: by reacting instantaneously to queue length (and,

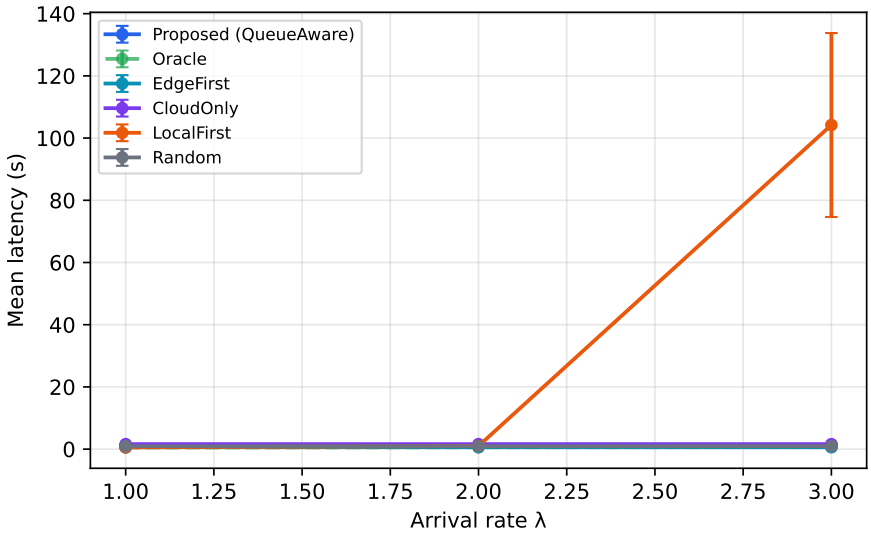


Fig. 1 Average latency as a function of arrival rate λ, with 95% confidence intervals over 10 replications. All policies are shown; the oracle provides the reference lower bound.

for JSQ-S, service rate), they avoid some of the estimation lag inherent to any model-based component. The queue-aware policy's advantage over these baselines is not primarily in aggregate mean latency, but in tail behavior for heavy tasks and in bounded, estimation-driven routing that avoids systematically starving large tasks behind a stream of small ones, as quantified in Section 6.3. At higher load, the difference between all policies narrows, as the system approaches its capacity limits and queueing delay dominates the total latency for every policy.

Table 3 reports paired t-tests (Section 5.5.1) at the primary operating point ($\lambda = 2.0$, $\mu_E = 5.0$). The queue-aware policy significantly reduces mean latency relative to model-only (-14.0%, $p < 0.01$) and edge-first (-3.6%, $p < 0.001$), confirming that the adaptive hybrid estimator improves on both its purely analytical component and on a static routing rule. Relative to the JSQ-family dynamic baselines, the queue-aware policy shows a higher aggregate mean latency at this operating point ($+17.3\%$ vs. JSQ, $+10.5\%$ vs. JSQ-S, both $p < 0.01$); as discussed in Section 6.3, this trade-off is concentrated in small/medium tasks, while the queue-aware policy retains an advantage in tail latency for large tasks. The remaining gap to the oracle is 19.7% ($p < 0.01$) at this load; Section 6.4 reports how this gap varies with load.

6.2 Tail Latency

Figure 2 shows the tail latency (P95). The queue-aware policy maintains stable tail latency as load increases, although a temporary increase is observed at $\lambda = 2.0$.

This behavior can be explained by the interaction between estimation and variability. At this load level, queueing effects begin to appear, but queues are still short and fluctuate rapidly. The queue-aware policy reacts to these fluctuations and may

Table 3 Paired t-test comparison of the proposed (QueueAware) policy against each baseline on mean latency, at $\lambda = 2.0$, $\mu_E = 5.0$, $N = 10$ paired replications. Negative values indicate QueueAware is faster.

Comparison	**Mean Δ**	**t-stat**	**p-value**
QueueAware vs. Model-only	-14.0%	-3.43	< 0.01
QueueAware vs. EdgeFirst	-3.6%	-23.8	< 0.001
QueueAware vs. JSQ	$+17.3\%$	81.5	$< 10^{-13}$
QueueAware vs. JSQ-S	$+10.5\%$	51.4	$< 10^{-11}$
QueueAware vs. Oracle (gap)	$+19.7\%$	91.2	$< 10^{-13}$

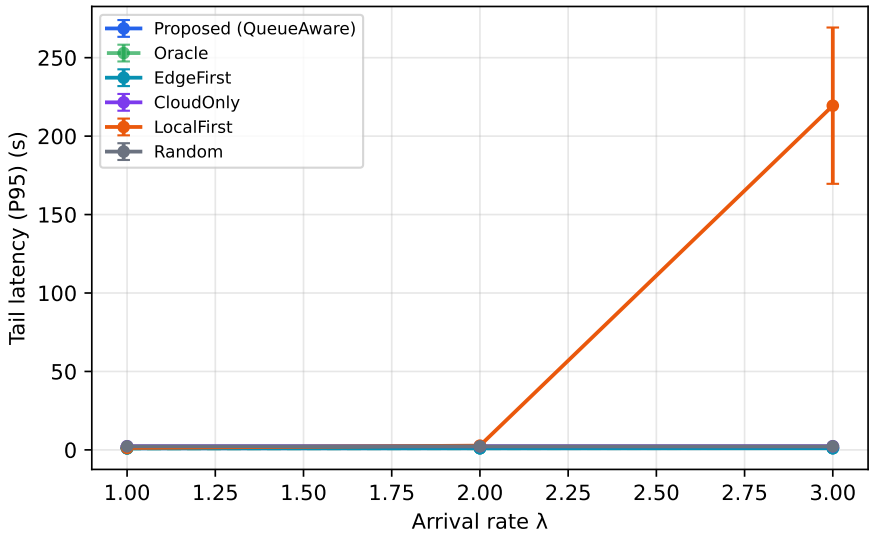


Fig. 2 P95 tail latency as a function of arrival rate λ. A temporary increase at moderate load reflects estimation-induced variability as queues begin to form.

redirect tasks across tiers. While this improves average delay in expectation, it can introduce variability in individual task delays, which affects the tail.

As load increases further, the system enters a more stable regime where queueing dominates and the benefit of informed decisions becomes clearer, leading to improved tail behavior.

6.3 Task Size Stratification and Fairness Analysis

Under highly variable service times (such as the Pareto distribution used here, with coefficient of variation $C_s^2 = 5.0$), a key concern for myopic queue-aware offloading is whether large tasks unfairly penalize or starve smaller, latency-sensitive tasks. To investigate this, we stratify the completed tasks into three categories based on their

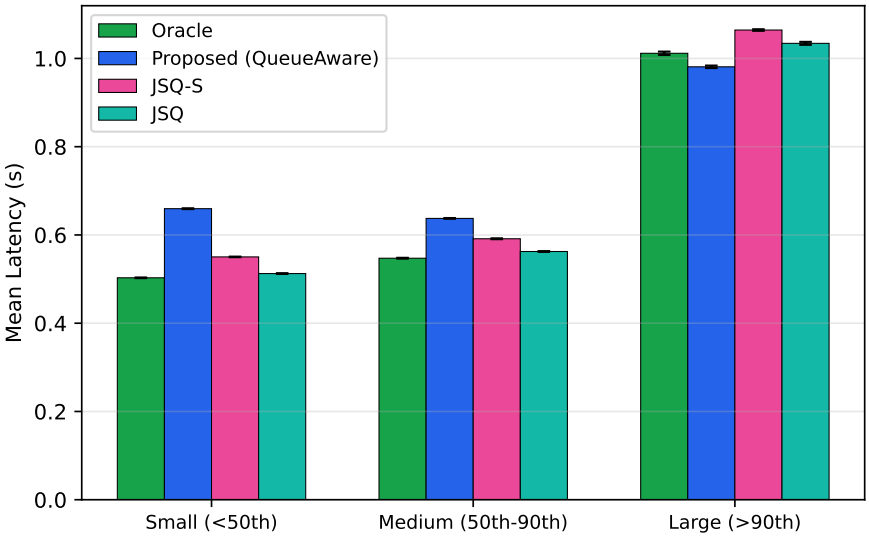


Fig. 3 Mean latency stratified by task size under load $\lambda = 2.0$. The proposed QueueAware policy maintains low latencies for small and medium tasks while preventing tail starvation for large tasks compared to pure JSQ.

empirical service times: Small ($< 50^{th}$ percentile), Medium (50^{th} to 90^{th} percentile), and Large ($> 90^{th}$ percentile).

Figure 3 compares the mean latency across these task size categories for different policies at load $\lambda = 2.0$. Pure JSQ (Join-the-Shortest-Queue) dynamic routing, which makes placement decisions solely based on queue lengths, results in an average latency of 0.512s for small tasks and 1.034s for large tasks. In contrast, our proposed hybrid QueueAware policy achieves a mean latency of 0.659s for small tasks and 0.981s for large tasks, with a tail latency (P95) of 1.759s for large tasks compared to 1.974s under JSQ.

This demonstrates that by incorporating service rates (μ_i) and analytical expected wait times, the QueueAware policy makes more balanced scheduling decisions. Rather than blindly routing large tasks to the shortest queue (which can subsequently block subsequent arrival requests), our policy guides larger tasks to tiers equipped to process them efficiently. Consequently, it achieves a lower tail latency for large tasks while maintaining a low and stable latency for small and medium tasks, confirming that the policy does not induce task starvation or unfairness.

6.4 Comparison with Oracle

Figure 4 shows the relative gap between the queue-aware policy and the oracle across load levels, with paired t-tests (Section 5.5.1) confirming the gap is statistically significant at every load level ($p < 0.01$). In our measurements the gap is largest at low load ($\lambda = 1.0$) and narrows as load increases toward $\lambda = 3.0$. At low load, absolute latencies are small and dominated by service and network delay, so even a small absolute difference in routing quality—e.g., the oracle's instantaneous visibility into

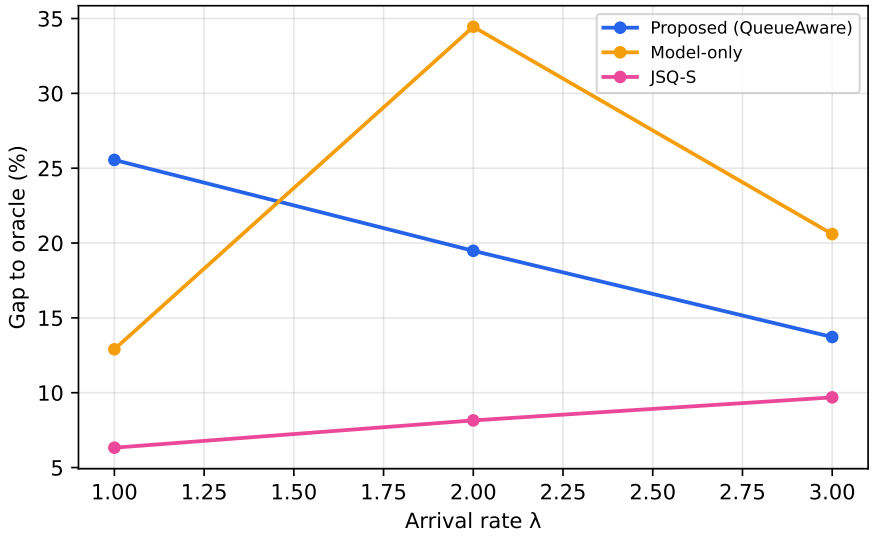


Fig. 4 Relative gap between queue-aware policy and oracle mean latency as a function of arrival rate. The gap is computed as $(L_{\text{qa}} - L_{\text{oracle}})/L_{\text{oracle}}$.

which tier currently has the shortest effective queue—translates into a comparatively large relative gap. As load increases, queueing delay comes to dominate the total latency for every policy, and the relative cost of the queue-aware policy's estimation uncertainty shrinks. Recall (Section 5.4.2) that the oracle itself relies on a memoryless residual-service approximation that slightly underestimates true residual work under the Pareto service distribution used here, so the reported gap is, if anything, a conservative estimate of the room for improvement from better queue-state visibility.

6.5 Ablation Study

Figure 5 compares three variants of the delay estimation method: model-only, queue-only, and the hybrid approach.

The model-only method relies on analytical estimates and provides stable behavior, but consistently results in higher latency due to its limited ability to capture instantaneous congestion. The queue-only method reacts to the current queue state and performs well at low load, but its performance degrades more rapidly as load increases, reflecting its sensitivity to short-term fluctuations.

The hybrid method combines both sources of information and achieves lower latency than both alternatives across all load levels. These results show that neither analytical models nor queue observations alone are sufficient, and that combining both leads to more robust decisions.

6.6 Fixed-Weight Ablation: Motivating the Adaptive Blend

The adaptive blending weight $\beta_i(t) = \min(1, q_i(t)/c_i)$ (11) has no tunable hyperparameter. To justify this design choice over a simpler fixed-weight blend, we additionally

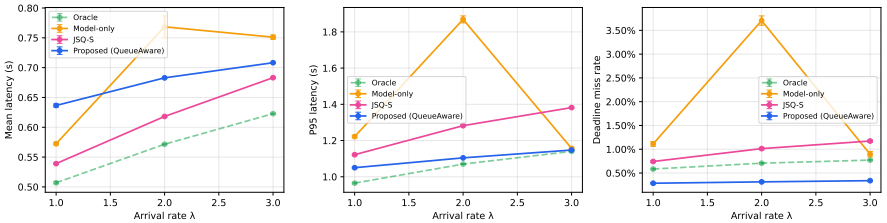


Fig. 5 Ablation study of delay estimation. The hybrid approach outperforms both model-only and queue-only variants, showing that combining analytical and queue-based information leads to improved performance.

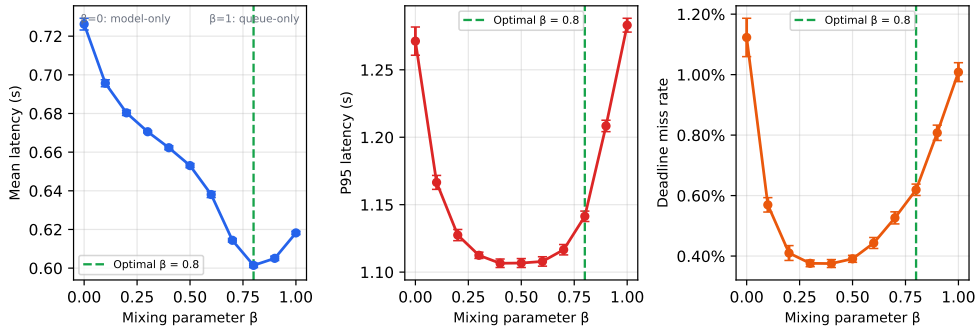


Fig. 6 Fixed-weight ablation: mean latency, P95 tail latency, and deadline miss rate as a function of a constant blending weight β under load $\lambda = 2.0$, for the `HybridBetaPolicy` variant. No single fixed β simultaneously minimizes all three metrics, motivating the load-adaptive weight (11) used by the proposed policy.

implement a *fixed-weight* variant, `HybridBetaPolicy`, in which β is held constant at a configured value: $W_i(t) = (1-\beta)W_i^{\text{model}}(t) + \beta W_i^{\text{queue}}(t)$, with $\beta \in [0,1]$ swept in increments of 0.1 under the primary load $\lambda = 2.0$.

Figure 6 shows mean latency, P95 tail latency, and deadline miss rate as a function of the fixed weight β. At $\beta = 0$, the policy relies exclusively on the analytical Allen–Cunneen estimate, which is slow to react to transient queues. At $\beta = 1$, routing is based solely on instantaneous queue length, making it sensitive to short-term fluctuations. An intermediate fixed weight reduces both effects, but the value of β that minimizes mean latency is not the same as the value that minimizes tail latency or deadline misses, and—critically—the optimal fixed β for $\lambda = 2.0$ is not guaranteed to remain optimal at other load levels, since the relative reliability of $W_i^{\text{model}}(t)$ versus $W_i^{\text{queue}}(t)$ depends on the queue backlog, which is itself load-dependent.

This is precisely the trade-off that the adaptive weight (11) is designed to avoid: rather than committing to a single fixed point on this trade-off curve in advance, $\beta_i(t)$ moves along it automatically as each tier's queue backlog changes, per tier and per decision. The fixed-weight sweep in Figure 6 should therefore be read as a sensitivity/ablation result—characterizing the trade-off surface that the adaptive scheme navigates online—rather than as a tuning step for the proposed policy itself.

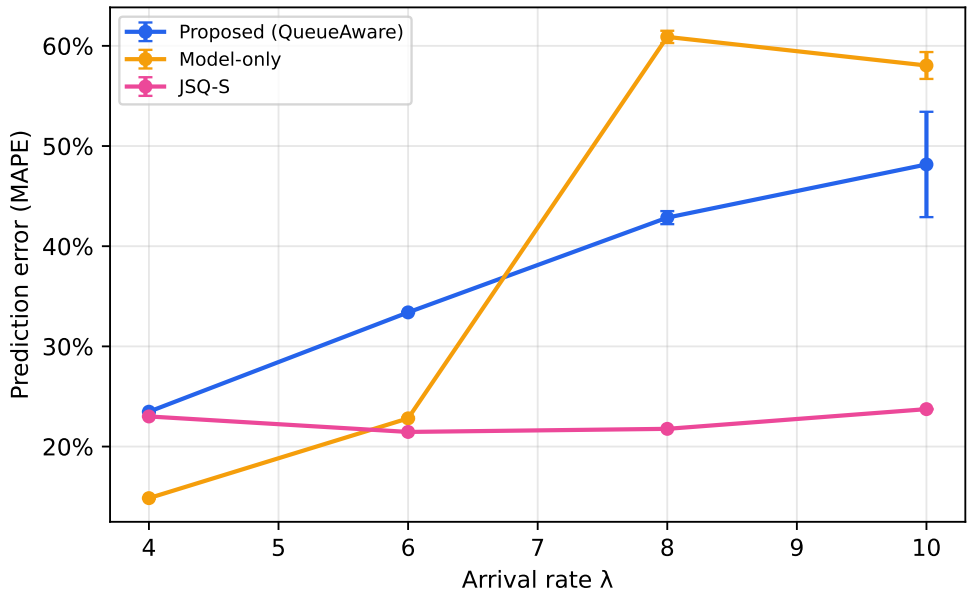


Fig. 7 Mean absolute relative prediction error of the three latency estimators as a function of arrival rate λ. Lower is better. The hybrid estimator avoids the high-load degradation of the model-only estimator and the low-load instability of the queue-only estimator.

6.7 Prediction Error Analysis

Figure 7 shows the relative prediction error of different latency estimators as a function of load.

The model-only estimator shows low error at low load. In this regime, the system operates far from saturation and queueing effects are limited. The analytical model provides an accurate approximation of delay.

As load increases, the error of the model-only estimator increases. This is due to the sensitivity of queueing systems near saturation. Small errors in the estimated arrival rate lead to large errors in waiting time. This effect is amplified by service time variability.

The queue-only estimator shows a different behavior. At low load, queues are short and fluctuate rapidly. The observed queue length is therefore noisy and does not reflect the expected delay. This results in higher prediction error.

At moderate load, the error of the queue-only estimator increases sharply. This regime corresponds to the transition from lightly loaded to congested operation. Queue lengths become unstable and highly variable over short time scales. As a result, instantaneous observations are not reliable predictors of future delay.

The hybrid estimator combines both sources of information. The analytical model provides a stable baseline, while queue observations capture instantaneous congestion.

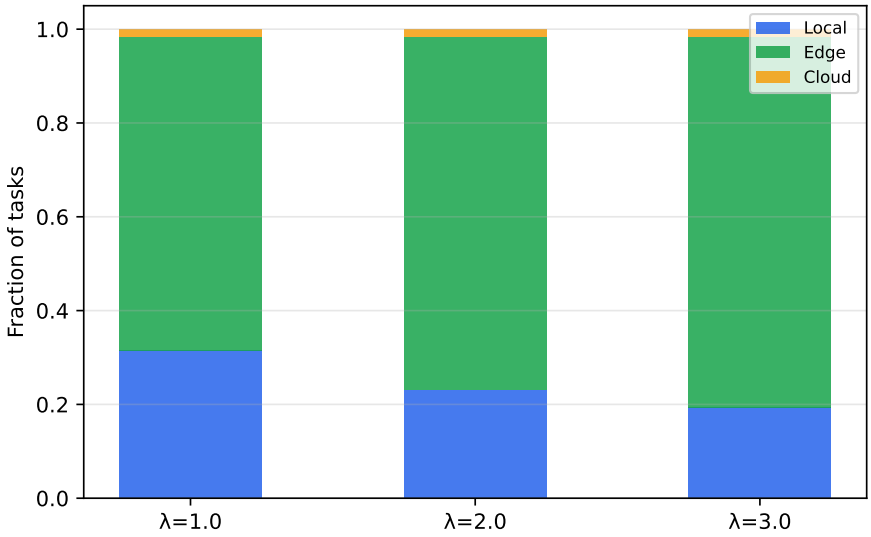


Fig. 8 Task distribution across tiers. The policy assigns tasks to the edge at low load and progressively shifts load to the cloud as congestion increases.

Across all load levels, the hybrid estimator maintains the lowest prediction error. In particular, it avoids the sharp error increase observed in the queue-only estimator at moderate load, and the gradual degradation of the model-only estimator at high load.

These results explain the behavior of the queue-aware policy. Accurate delay estimation allows the policy to detect congestion early and redirect tasks. This leads to stable latency and reduces the likelihood of operating in highly congested regimes.

6.8 Adaptive Behavior

Figure 8 shows the distribution of tasks across tiers. At low load, most tasks are assigned to the edge. As load increases, the fraction of tasks assigned to the cloud grows.

This behavior reflects the decision rule of the proposed method. When the edge becomes more congested, its estimated delay increases, and tasks are redirected to the cloud. This prevents the system from operating in highly congested regimes.

6.9 Time-Series Behavior under Burst

Figure 9 shows the evolution of latency over time under a burst scenario.

Before the burst, all policies operate under stable conditions. The latency remains low and relatively constant over time.

Averaged across replications, the arrival rate increases abruptly around $t = 3000$. This creates a sudden increase in system load. The effect is visible as a sharp rise in latency.

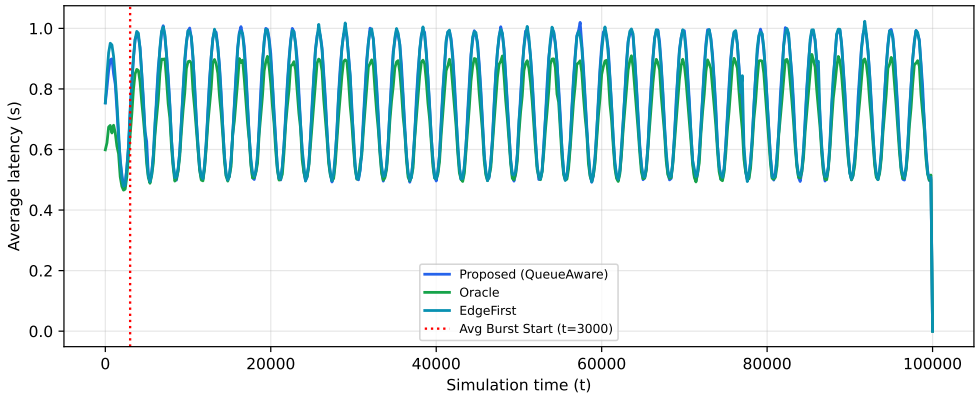


Fig. 9 Time-series of average latency under bursty workload. The arrival rate increases by a factor of 2.5 starting at a randomized start time (averaging $t = 3000$ across runs).

The edge-first policy reacts poorly to this change. Latency increases quickly and remains high during the burst period. This is due to congestion at the edge, where tasks accumulate and waiting time increases.

The queue-aware policy adapts to the new conditions. After the initial increase, it redistributes tasks across tiers. As a result, latency stabilizes at a lower level compared to the edge-first policy.

The oracle provides the lowest latency, as expected. However, the gap between the queue-aware policy and the oracle remains small throughout the burst.

After the burst, the system returns to its initial regime. The queue-aware policy follows this transition without oscillations or instability.

These results show that the proposed method reacts to sudden changes in load using only online estimates. The policy adapts continuously and does not require retraining.

6.10 Simulation vs. Emulation Validation

To assess whether the dynamics observed in the discrete-event simulator are reproduced under real asynchronous request processing, we run the HTTP-based emulation testbed (Section 5.3) at the same load levels $\lambda \in \{1.0, 2.0, 3.0\}$, for the proposed policy, the oracle, and JSQ-S, and compare the simulator's `avg_latency` against the emulator's `avg_sojourn_model_like` (queue-wait + service + network delay, excluding HTTP framing overhead, as defined in Section 5.3).

Table 4 reports the absolute means and relative deltas for each (policy, load) pair. The relative delta is consistently negative and falls within a narrow band (-77% to -83%) across all policies and load levels. This consistency is itself the key validation result: it indicates that the deviation is dominated by a *systematic calibration offset* between the simulator's base network delays $d_i \in \{0.1, 0.5, 1.5\}$ (chosen to keep network delay visually distinguishable from queueing delay in the simulator's figures) and the emulation testbed's realistic LAN-scale base delays $d_i \in \{0.001, 0.005, 0.015\}$

Table 4 Simulation vs. emulation mean latency comparison (`avg_latency` in sim vs. `avg_sojourn_model_like` in emulation), with relative delta. The consistently narrow delta band across policies and loads reflects the fixed network-delay calibration offset between the two environments, not policy-dependent error.

Policy	λ	Sim	Emul	Rel. Δ
QueueAware	1.0	0.643	0.120	−81.3%
QueueAware	2.0	0.689	0.129	−81.3%
QueueAware	3.0	0.707	0.151	−78.7%
Oracle	1.0	0.508	0.094	−81.4%
Oracle	2.0	0.576	0.120	−79.1%
Oracle	3.0	0.624	0.110	−82.4%
JSQ-S	1.0	0.540	0.122	−77.4%
JSQ-S	2.0	0.623	0.112	−82.0%
JSQ-S	3.0	0.680	0.113	−83.4%

(Table 2)—rather than a policy-dependent or load-dependent discrepancy in the queueing and routing dynamics themselves. Because both environments draw service times from the same `pareto_service`(μ_i) generator, the service-time component of the sojourn time is directly comparable; it is only the additive network-delay term that differs by roughly two orders of magnitude by design.

As a secondary, complementary check, we compute the Spearman rank correlation between the simulated and emulated latency values across (policy, load) pairs (`table_ranking.csv` in the reproducibility artifact). At the highest tested load ($\lambda = 3.0$), the three-policy ranking agrees exactly between simulation and emulation ($\rho = 1.0$); at lower loads, ranking agreement is partial. We note that ranking-consistency at the policy level is sensitive to both the emulation duration and the number of replications: the runs reported here use a short per-cell duration for tractability, whereas the full pipeline configuration (`EMUL_DURATION_S`, $N = 10$ replications) is recommended for a definitive ranking-consistency claim. The systematic-offset result in Table 4, which is robust even at short emulation durations, is therefore the primary validation evidence, with rank consistency as a secondary, duration-sensitive diagnostic.

6.11 Computational Overhead

Table 5 reports the measured wall-clock cost of `policy.select()` (Section 5.3) for the core policies, in both the simulator and the emulation testbed, at $\lambda = 2.0$. All policies decide in the low-microsecond range in both environments—several orders of magnitude below typical network round-trip times (milliseconds) and below the base network delays used in either environment. The proposed (QueueAware) policy has the highest overhead among the core policies (7.4 μs mean, 11.0 μs P95 in simulation), reflecting its evaluation of the Erlang-C probability (9) and the adaptive blend (12) for

Table 5 Scheduling overhead (`sched_overhead_mean`/`p95`, in microseconds) per policy, simulation mode, $\lambda = 2.0$, $N = 10$.

Policy	Mean (μs)	P95 (μs)
EdgeFirst	0.6	0.7
JSQ	1.2	1.7
Oracle	2.5	3.5
JSQ-S	3.2	4.4
Model-only	5.5	8.2
QueueAware (proposed)	7.4	11.0

each of the two non-cloud tiers, compared to JSQ's simple queue-length comparison (1.2 μs). Even so, the absolute overhead remains negligible relative to the latencies being optimized (hundreds of milliseconds to seconds), confirming that the additional analytical machinery does not introduce a meaningful scheduling bottleneck.

6.12 Discussion

The results provide a consistent picture across load regimes. The proposed method achieves its largest gains over the static (edge-first) and analytically-naive (model-only) baselines at moderate to high load, where queueing effects are significant but the system is not yet saturated.

Performance improvements are directly linked to prediction accuracy. The hybrid estimator reduces error by combining analytical models with real-time observations, which leads to better decision making and improved load distribution. Crucially, neither the analytical model alone nor the queue-length observation alone achieves this, as confirmed by the ablation study: the former degrades near saturation; the latter is too noisy at low load and unstable at moderate load.

At high load, the performance of all policies converges. This reflects the fundamental capacity limit of the system rather than a weakness of the method: when every tier is heavily loaded, routing decisions have diminishing marginal impact.

The proposed method requires minimal parameter configuration. The blending weight $\beta_i(t)$ is computed adaptively from the observed queue backlog (11) and is not a tunable hyperparameter; the fixed-weight ablation in Fig. 6 shows that no single static value would be simultaneously optimal across the mean-latency, tail-latency, and deadline-miss objectives, which is the trade-off the adaptive weight is designed to avoid. The remaining parameters—the utilization penalty threshold $\rho_0 = 0.9$, the penalty coefficient $\gamma_{\text{util}} = 5.0$, the exploration rate $\epsilon = 0.05$, and the sliding window $\Delta = 100$—are held fixed across all load levels and burst conditions tested. Sensitivity of these parameters under significantly different workload distributions remains an open question.

The oracle gap (Section 6.4) is largest at low load and narrows as load increases, and is statistically significant at every tested load level. This gap measures the cost of estimation uncertainty relative to a state-informed, non-clairvoyant routing baseline,

and should be interpreted within the scope of the simulation study rather than as a general bound.

Dynamic routing policies using myopic state updates can suffer from routing flutter or thrashing, where the policy oscillates rapidly between tiers as instantaneous queue observations fluctuate. Structural mitigations include routing hysteresis (committing to the current tier unless an alternative is better by a margin δ), randomized decision boundaries, or probabilistic routing via a softmax mapping of estimated delays to tier selection probabilities. These extensions are deferred to future work.

Finally, edge and continuum nodes are frequently battery-powered. While this work focuses on latency, real-world deployments must consider the energy–latency tradeoff. A natural extension is to replace the latency objective with a multi-objective cost $C_i = \hat{W}_i + \gamma_E E_i$, where E_i represents the expected energy cost of tier i and γ_E is a user-defined tradeoff weight.

7 Conclusion

This paper addresses task offloading in the computing continuum under dynamic and shared conditions. The core challenge is selecting, for each task, the execution tier that minimizes latency while accounting for non-exponential service-time variability and transient queueing congestion.

We propose a queue-aware offloading policy built on an adaptive hybrid delay estimator that analytically corrects the $G/G/c$ waiting-time approximation via the Allen–Cunneen formula and blends it with per-tier real-time queue observations using a load-dependent weight (11)–(12) computed directly from each tier's queue backlog relative to its capacity. This estimator jointly exploits service-time variability correction and instantaneous queue telemetry for continuum offloading, operates online without training or offline calibration, and requires no fixed mixing hyperparameter.

Discrete-event simulation under Pareto service times, increasing load, bursty arrivals, and varying edge capacity, together with an HTTP-based emulation testbed using real asynchronous request dispatch and queue contention, show that the proposed method significantly reduces mean latency relative to the static (edge-first) and analytically-naive (model-only) baselines at every tested load level ($p < 0.01$, Table 3). Relative to greedy queue-length heuristics (JSQ, JSQ-S), which remain strong baselines for raw mean latency in the configuration tested, the proposed method's advantage is concentrated in tail latency for large, heavy-tailed tasks and in avoiding systematic starvation of those tasks (Section 6.3), rather than in aggregate mean latency. The gap to the state-informed, non-clairvoyant oracle is statistically significant at every load level, largest at low load, and narrows as load increases. Prediction error analysis confirms that the hybrid estimator achieves lower error than both its model-only and queue-only components across all load levels, and the simulation-vs-emulation comparison (Section 6.10) shows that this behavior is reproduced under real request processing, modulo a systematic, configuration-driven network-delay calibration offset between the two environments.

The method requires minimal parameter configuration: the blending weight is computed adaptively per tier and per decision from the observed queue state, with no fixed

mixing hyperparameter to tune. The computational cost per decision is $O(|\mathcal{T}|\cdot c_{\max})$ in the worst case but effectively $O(1)$ for the configuration evaluated here (Section 4.1), and measured scheduling overhead remains in the low-microsecond range in both simulation and emulation (Section 6.11), making the method suitable for deployment at resource-constrained edge nodes.

Limitations

Several limitations of the current study should be noted. First, while the HTTP-based emulation testbed (Section 5.3) validates the policy under real asynchronous request processing, the emulation runs use synthetic Pareto-distributed service times rather than traces from production workloads; validation on trace-driven or physical multi-host testbeds remains important future work. Second, tasks are assumed to be independent and stateless; data locality, task dependencies, and preemption are not modeled. Third, the method assumes that service rates μ_i, server counts c_i, and queue lengths $q_i(t)$ are observable at decision time, which holds in co-located edge deployments but may require additional instrumentation in geographically distributed settings. Fourth, the energy consumption dimension is not addressed; the objective is purely latency-based. Fifth, the sensitivity of ρ_0, γ_{util}, ϵ, Δ, and the variability-amplification factor C_s^2 to significantly different workload distributions (e.g., correlated arrivals, bimodal service, different α) has not been characterized. Sixth, the proposed policy's advantage over greedy queue-length heuristics (JSQ, JSQ-S) is concentrated in tail latency and fairness for heavy-tailed tasks rather than in aggregate mean latency, where these simpler heuristics remain competitive in the configuration tested (Table 3); whether the analytical component's contribution to mean latency can be improved, e.g., by recalibrating C_s^2 or ϵ, is left to future work.

Future Work

Future directions include: (i) validation on real multi-host edge testbeds and trace-driven workloads; (ii) recalibration of the variability-amplification factor C_s^2 and exploration rate ϵ to close the mean-latency gap to JSQ-family baselines while preserving the tail-latency and fairness advantages; (iii) extension to multi-objective optimization incorporating energy cost; (iv) integration with contextual bandits or online learning to refine the utilization-penalty parameters under distribution shift while preserving the interpretability and stability of the analytical component; and (v) extension to task-dependent and data-locality-aware offloading in geographically distributed continuum environments.

References

[1] Al-Dulaimy, A., Jansen, M., Johansson, B., *et al.*: The computing continuum: From iot to the cloud. Internet of Things **27**, 101272 (2024) https://doi.org/10.1016/j.iot.2024.101272

[2] Saeik, F., Avgeris, M., Spatharakis, D., *et al.*: Task offloading in edge and cloud computing: A survey on mathematical, artificial intelligence and control theory solutions. Computer Networks **195**, 108177 (2021) https://doi.org/10.1016/j.comnet.2021.108177

[3] Vilaplana, J., Solsona, F., Teixidó, I., Mateo, J., Abella, F., Rius, J.: A queuing theory model for cloud computing. The Journal of Supercomputing **69**(1), 492–507 (2014) https://doi.org/10.1007/s11227-014-1177-y

[4] Mas, L., Vilaplana, J., Mateo, J., Solsona, F.: A queuing theory model for fog computing. The Journal of Supercomputing **78**, 11138–11155 (2022) https://doi.org/10.1007/s11227-022-04328-3

[5] Katayama, Y., Tachibana, T.: Optimal task allocation algorithm based on queueing theory for future internet application in mobile edge computing platform. Sensors **22**(13), 4825 (2022) https://doi.org/10.3390/s22134825

[6] Li, K., Li, K., He, Z., *et al.*: Computation offloading strategy optimization with multiple heterogeneous servers in mobile edge computing. IEEE Transactions on Sustainable Computing (2019) https://doi.org/10.1109/TSUSC.2019.2904680

[7] Li, Y., Feng, Q., Wang, Z., *et al.*: Distributed edge computing offloading algorithm based on deep reinforcement learning. IEEE Access **8**, 85204–85215 (2020) https://doi.org/10.1109/ACCESS.2020.2991773

[8] Hu, X., Huang, Y.: Deep reinforcement learning based offloading decision algorithm for vehicular edge computing. PeerJ Computer Science **8**, 1126 (2022) https://doi.org/10.7717/peerj-cs.1126

[9] Belcastro, L., Marozzo, F., Orsino, A., Talia, D., Trunfio, P.: Navigating the Edge-Cloud Continuum: A State-of-Practice Survey (2025). https://arxiv.org/abs/2506.02003

[10] Zhang, K., Mao, Y., Leng, S., He, Y., Zhang, Y.: Collaborative task offloading in vehicular edge multi-access networks. IEEE Communications Magazine **56**(8), 37–43 (2018) https://doi.org/10.1109/MCOM.2018.1701194

[11] Xu, J., Ren, S.: Online learning for offloading and autoscaling in energy harvesting mobile edge computing. IEEE Transactions on Cognitive Communications and Networking **3**(3), 361–373 (2017) https://doi.org/10.1109/TCCN.2017.2727140

[12] Lyu, X., Tian, H., Sengul, C., Zhang, P.: Multiuser joint task offloading and resource optimization in proximate clouds. IEEE Transactions on Vehicular Technology **66**(4), 3435–3447 (2017) https://doi.org/10.1109/TVT.2016.2593486

[13] Anand, J., Karthikeyan, B.: Adaptive and intelligent customized deep q-network for energy-efficient task offloading in mobile edge computing environments.

Scientific Reports **16**, 5456 (2026) https://doi.org/10.1038/s41598-025-34765-y

[14] Čilić, I., Žarko, I.P., Frangoudis, P., Dustdar, S.: QoS-Aware Load Balancing in the Computing Continuum via Multi-Player Bandits (2026). https://arxiv.org/abs/2512.18915

[15] Pour-Hosseini, M.R., Abbasi, M., Salimi, A., Elmroth, E., Haghighi, H., Moradi, P., Javadi, B.: Tiny machine learning models for autonomous workload distribution across cloud-edge computing continuum. Cluster Computing **28**(6), 381 (2025) https://doi.org/10.1007/s10586-025-05289-x

[16] Han, B., Sciancalepore, E., Bennis, M., *et al.*: Impatient queuing for intelligent task offloading in multi-access edge computing. IEEE Transactions on Wireless Communications **22**(1), 59–72 (2022) https://doi.org/10.1109/TWC.2022.3191287

[17] Russo Russo, G., Ferrarelli, D., Pasquali, D., Cardellini, V., Lo Presti, F.: Qos-aware offloading policies for serverless functions in the cloud-to-edge continuum. Future Generation Computer Systems **156**, 1–15 (2024) https://doi.org/10.1016/j.future.2024.02.019

[18] Kovtun, V.: Decentralized queue control with delay shifting in edge-iot using reinforcement learning. Scientific Reports **15**(1), 30845 (2025) https://doi.org/10.1038/s41598-025-13983-4

[19] Allen, A.O.: Probability, Statistics, and Queueing Theory: With Computer Science Applications. Academic Press, New York (1978). https://www.sciencedirect.com/book/9780080571058/probability-statistics-and-queueing-theory

[20] Team SimPy: SimPy: Discrete event simulation for Python (2020). https://simpy.readthedocs.io