

Article

ParallelEdge-AI: A Shared-Encoder Framework for Joint Traffic Classification and Latency-Aware Scheduling in Distributed IoT Edge Networks

Abdulaziz G. Alanazi ¹, Haifa A. Alanazi ^{1, *} and Nasser S. Albalawi ²

¹ Department of Information Systems, Faculty of Computing and Information Technology, Northern Border University, Rafha 91911, Saudi Arabia; abdulaziz.alanazi@nbu.edu.sa

² Department of Computer Science, Faculty of Computing and Information Technology, Northern Border University, Rafha 91911, Saudi Arabia; nasser.albalawi@nbu.edu.sa

* Correspondence: haifa.alanazi@nbu.edu.sa

Abstract

IoT networks now handle traffic from billions of devices, and edge nodes are under constant pressure to classify that traffic and dispatch tasks within tight latency deadlines. Most existing systems treat classification and scheduling as two separate steps that run one after the other. This sequence adds unnecessary delay and breaks the feedback between the two tasks: the scheduler never sees the traffic type, and the classifier never sees the queue state. We propose ParallelEdge-AI, a system built around a shared flow encoder that feeds two task-specific heads in parallel, one for multi-class traffic classification and one for task-urgency scoring. Both heads are trained end-to-end using a joint loss that combines cross-entropy and pairwise ranking. A load-balance controller then reads the urgency scores alongside live queue lengths to decide, every 200 ms, whether a task stays local or moves to a less-loaded edge node. No global synchronisation is needed. We test the system on three real IoT datasets: RT-IoT2022, N-BaIoT, and CICIoT2023. ParallelEdge-AI reaches 97.63% accuracy and an F1-score of 97.34%, which is 3.16 percentage points above the best baseline. Inference latency is 19.62 ms per batch, the deadline-miss rate is 2.34%, and the load-imbalance index is 0.083, all three are the best results in our comparison. These numbers show that running classification and scheduling together on a shared representation is both faster and more accurate than treating them as separate problems.

Keywords: edge computing; IoT traffic classification; task scheduling; load balancing; parallel inference; multi-task learning; latency-critical networks; deep learning



Academic Editor: Marco Savi

Received: 19 March 2026

Revised: 11 May 2026

Accepted: 28 May 2026

Published: 3 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

The number of connected IoT devices reached 18.8 billion by the end of 2024 and is expected to cross 40 billion by 2030 [1]. Smart grids, factory sensors, self-driving cars, and health devices all depend on this infrastructure, and each generates continuous bidirectional traffic that must be handled within tight time limits. Sending this traffic to a distant cloud introduces delays that real-time applications cannot tolerate, so edge computing emerged as an intermediate layer that places reasoning and control near the data source [2]. The global edge computing market was valued at USD 23.65 billion in 2024 and is growing at more than 33% per year [3], reflecting the scale of this shift. Placing compute at the edge does not on its own solve the latency problem: a single edge node serves many devices at once, and if its task queue becomes overloaded the end-to-end delay can exceed

the deadline even without any network round-trip. Tasks must be distributed across a group of edge nodes, and that redistribution decision must itself be fast enough not to add measurable delay [4].

Traffic classification is the other half of the problem. Not all IoT flows carry the same urgency: a Mirai botnet packet and a routine sensor reading arrive at the same queue but require completely different responses [5,6]. Deep learning classifiers have reached high accuracy on benchmarks such as N-BaIoT and CICIoT2023 [5,7], but in deployed systems the classifier runs as a separate process, either before or after the scheduler. When the two steps run in sequence, the total latency is the sum of both, and neither step has access to what the other is computing. The scheduler does not know whether an arriving task belongs to a high-priority attack-response stream or a low-priority telemetry collection, and the classifier has no way to adjust feature weighting based on current queue load. This coupling gap does not appear in single-task studies because those papers optimise classification or scheduling in isolation, not jointly [8–10]. The result is that existing systems perform well on one task or the other, but rarely achieve both on the same edge node under live traffic conditions.

This paper addresses the above gap by introducing ParallelEdge-AI, a system built around a single shared encoder that feeds two task-specific heads running in parallel on the same forward pass. Because the encoder is shared, each head always has the full traffic representation when it produces its output, which eliminates the sequential wait and the context gap in one step. A load-balance controller then reads the scheduling scores at each control interval to decide whether a task should remain on the current node or be redirected to a less-loaded one. The main contributions of this work are as follows.

- (1) ParallelEdge-AI framework. A unified system is proposed in which a shared flow encoder feeds the Parallel Inference and Scheduling Engine (PISE) and a load-balance controller. The encoder is trained jointly with both downstream heads, so the latent representation carries information useful for both tasks simultaneously.

- (2) PISE architecture. Inside the PISE, two parallel heads run concurrently on the same latent code: one for multi-class traffic classification and one for task-urgency scoring. Both heads are trained end-to-end with a combined cross-entropy and pairwise ranking loss, so classification accuracy and scheduling quality are optimised together.

- (3) Load-balance controller. The controller converts per-task urgency scores and queue-length statistics into a scalar load index per node. Task redirection decisions are made locally at each control interval and require no global synchronisation across edge nodes, adding no measurable overhead to the inference path.

- (4) Experimental validation. Experiments on three real IoT benchmarks; RT-IoT2022, N-BaIoT, and CICIoT2023, show that ParallelEdge-AI reaches 97.63% classification accuracy and an F1-score of 97.34%, outperforming the closest baseline by 3.16 percentage points. Inference latency is 19.62 ms per batch and the deadline-miss rate is 2.34%, which are 31% and 60% better than the next-best method, while the load-imbalance index drops to 0.083 from 0.151 in the best competing scheduler.

- (5) Robustness analysis. A sensitivity analysis over the load-threshold parameter θ and a batch-size scaling study confirm that performance gains hold across operating conditions and that the system stays below the 30 ms edge-latency target up to a batch size of 512.

The rest of this paper is structured as follows. Section 2 reviews related work on edge task scheduling and IoT traffic classification. Section 3 states the problem formally. Section 4 describes the datasets and preprocessing. Section 5 presents the ParallelEdge-AI architecture. Section 6 reports experimental results and analysis. Section 7 concludes the paper and outlines future directions.

2. Related Work

Edge computing moved processing closer to IoT devices so that tasks do not have to travel all the way to a remote cloud [11]. A large body of work has focused on making this offloading step efficient. Kim et al. [12] modelled task dispatch in a cluster edge system as a Markov decision process and trained a deep Q-network (DQN) to minimise average task completion time. The work improved on round-robin and least-load approaches; however, the classifier and scheduler remained separate, so the DQN had no knowledge of incoming traffic type. Abdallah et al. [4] extended this by using a multi-objective DQN that balanced latency, power use, and cost through a priority queue and a state-reward tensor. Deadline failures dropped compared to single-objective systems, but traffic identification still happened after scheduling, not alongside it. Chen et al. [13] proposed a resilient collaborative caching framework for multi-edge systems using robust federated deep learning, showing that coordinated edge nodes can sustain high cache hit rates under adversarial conditions. Lin et al. [14] showed that joint optimisation of offloading decisions and resource allocation in mobile edge networks reduces latency for heterogeneous IoT workloads, but like the earlier scheduling works, traffic-type awareness is not part of the offloading decision. These studies share a common gap: the scheduler operates without knowledge of the traffic class it is handling.

Regarding traffic classification, Neto et al. [5] published the CIIoT2023 benchmark, which covers thirty-three attack types from a 105-device IoT network, and showed that CNN and DNN models reach high accuracy when trained on flow-based features. Awajan [15] built a four-layer fully connected network for IoT intrusion detection and showed that flow-level features, regardless of protocol, are sufficient for reliable identification. Altunay and Albayrak [16] combined 1D-CNN and LSTM on CIIoT2023 and TON_IoT and achieved over 99% accuracy in both binary and multi-class settings. These classifiers work well in batch mode on servers, but none was designed for resource-limited edge nodes where a full forward pass per incoming flow is too costly. None of them connects the classifier output to a downstream scheduling decision in a principled way, so the latency benefits of classification are lost before they can affect task dispatch.

Load-balancing studies at the edge have taken a different route. Naresh et al. [8] proposed SEE-MTS, which uses reinforcement learning to schedule tasks across edge machines while applying encryption and multi-key search for security. The system reached an energy efficiency of 89% and a security level of 96%, but its scheduler is driven by energy and security costs rather than by live traffic type. Dai et al. [17] studied multi-server partial offloading in industrial IoT with a fuzzy-security layer and showed that coupling load balance with security scoring reduces congestion. Wang et al. [18] reviewed IoT task offloading in MEC and concluded that most schemes optimise either latency or energy alone, and very few handle both while also classifying traffic in real time. Across all three groups of work—scheduling, classification, and load balancing—the same gap remains: traffic classification and task scheduling are treated as separate problems solved at different times with different models.

3. System Model

Let an edge network consist of N_e nodes $\mathcal{E} = \{e_1, e_2, \dots, e_{N_e}\}$, each with a finite CPU capacity R_k and a task queue $Q_k(t)$ at time t . IoT devices send flow records at a rate $\lambda(t)$, which varies over time. Each arriving flow f_j belongs to one of C traffic classes (benign or one of several attack or high-priority service types) and is associated with a task τ_j that must complete within a deadline d_j . Each deadline d_j is drawn from a class-dependent exponential distribution, where attack-class tasks are assigned tighter deadlines (mean 50 ms) and benign tasks are assigned looser ones (mean 200 ms), consistent with real

IoT traffic priority levels [4,5]. Two sub-problems need to be solved together. The system model is shown in Figure 1.

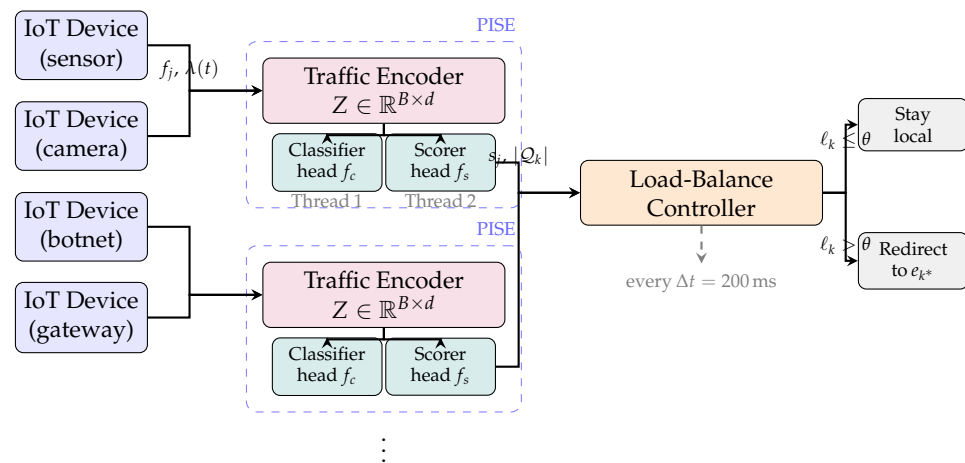


Figure 1. System model. IoT devices send traffic flows to edge nodes at rate $\lambda(t)$. Each edge node runs the traffic encoder once and dispatches both task heads in parallel on separate threads inside the PISE boundary. The load-balance controller reads per-node urgency scores s_j and queue lengths $|Q_k|$ every Δt seconds: if the local load index $\ell_k \leq \theta$ the task stays local; otherwise it is redirected to the least-loaded node e_{k^*} .

The first is *traffic classification*: given a normalised feature vector $\hat{x}_j \in \mathbb{R}^F$ for flow f_j , assign it to the correct class from $\{1, \dots, C\}$. The second is *latency-aware scheduling*: given the class assignment and the current queue states $\{|Q_k(t)|\}$, a decision is made on which node e_{k^*} should handle task τ_j so that d_j is met and load is spread evenly across nodes.

The two sub-problems interact. A scheduler that does not know the traffic class cannot tell whether a task is latency-critical or can wait. A classifier that does not know queue states cannot weight features differently during high-load periods. When both are solved by separate models in sequence, the first model finishes before the second starts, which adds latency and breaks the feedback loop. The goal of this work is to find a single model structure that solves both problems concurrently, reduces per-batch inference time, keeps the deadline-miss rate low, and maintains near-equal node utilisation without requiring a global synchronisation step between edge nodes.

4. Dataset Description and Preprocessing

Three publicly available IoT network datasets are used in this work. The RT-IoT2022 dataset [6] comes from a real IoT testbed and contains bidirectional traffic flows recorded from devices such as MQTT temperature sensors, ThingSpeak-LED nodes, and Wipro smart bulbs. Both normal and attack traffic are present. The N-BaIoT dataset [7] holds traffic captured from nine commercial IoT devices that were deliberately infected by Mirai and BASHLITE botnets. Each record has 115 statistical features computed over multiple time windows. The CICIoT2023 dataset [5] covers 33 attack types generated from a 105-device IoT topology, giving a wide view of realistic attack diversity. Together, these three sources supply the edge-AI model with both benign baseline behaviour and varied attack conditions needed to test latency-critical inference.

Table 1 gives the key statistics for each dataset and the merged split used in all experiments.

Raw flow records across the three datasets differ in scale and feature range, so a two-step preprocessing pipeline is applied. First, features with zero variance are removed. Then each remaining feature x_i is scaled using min-max normalisation given in

Equation (1), where $x_{\min}$ and $x_{\max}$ are the per-feature extremes computed on the training split only, so no test information leaks into training.

$$\hat{x}_i = \frac{x_i - x_{\min}}{x_{\max} - x_{\min} + \epsilon} \quad (1)$$

The small constant $\epsilon = 10^{-8}$ stops division by zero for near-constant features. Class imbalance across the three datasets is corrected with a weighted resampling scheme. For each class c , the sampling weight w_c is set as shown in Equation (2), where N is the total sample count and K is the number of classes.

$$w_c = \frac{N}{K \cdot n_c} \quad (2)$$

Here, n_c is the count of samples in class c . Rare attack classes obtain a higher weight, so the model sees a balanced view during training without synthetic oversampling that can introduce noise in edge-inference settings. Figure 2 shows the full data preparation pipeline from raw capture to the normalised feature matrix fed into the ParallelEdge-AI scheduling and inference engine.

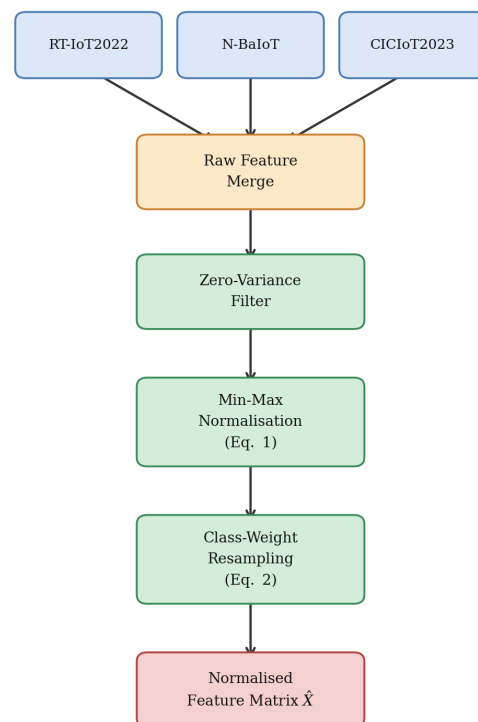


Figure 2. Data preparation pipeline. Three IoT traffic datasets are merged, filtered, normalised with Equation (1), and rebalanced with Equation (2) before being passed to the edge inference engine.

Table 1. Dataset statistics and train/validation/test split sizes. All counts are after zero-variance filtering.

Dataset	Samples	Features	Classes	Train/Val/Test
RT-LoT2022	123,117	83	10	86,182/18,468/18,467
N-BaIoT	442,785	115	11	309,950/66,418/66,417
CICIoT2023	698,524	46	34	488,967/104,778/104,779
Merged	1,264,426	46	34	885,098/189,664/189,664

5. Proposed ParallelEdge-AI Architecture

The core problem this work targets is that a single edge node cannot simultaneously handle scheduling decisions and real-time traffic classification without one task delaying the other. When IoT packet rates spike, the classifier queue grows and scheduling decisions become stale, which pushes end-to-end latency above acceptable bounds. To fix this, we separate the two functions and let them run in parallel on different threads of the same edge node, tied together by a lightweight shared-state bus.

Overall structure.

The system has three logical layers: a *traffic encoder*, a *parallel inference and scheduling engine (PISE)*, and a *load-balance controller*. The traffic encoder compresses each incoming flow record into a compact latent vector. The PISE runs two concurrent sub-models, one for traffic class prediction and one for task dispatch scoring. The controller reads the output of both sub-models every Δt seconds and redistributes tasks across available edge nodes. Figure 3 shows how these three layers connect.

Traffic encoder.

Let the normalised feature matrix for a batch of B flows be $\hat{X} \in \mathbb{R}^{B \times F}$, where F is the feature count after preprocessing. A one-hidden-layer encoder maps this to a latent space of dimension $d \ll F$:

$$Z = \sigma(\hat{X}W_e + b_e), \quad W_e \in \mathbb{R}^{F \times d}, \quad b_e \in \mathbb{R}^d \quad (3)$$

where $\sigma(\cdot)$ is the ReLU activation. We set $d = 64$ after a grid search over $\{32, 64, 128, 256\}$ on the validation split; $d = 64$ gave the best trade-off between classification F1 and inference latency, with larger values adding no measurable accuracy gain but increasing per-batch time by 18–34%. The encoder weight W_e is trained jointly with the downstream heads so the latent code Z carries information useful for both tasks at once. Sharing a single encoder is well motivated here because both tasks depend on the same flow-level statistics: the classifier needs feature patterns that separate traffic classes, and the urgency scorer needs the same patterns to judge whether a flow belongs to a latency-sensitive class. A separate encoder for each head would compute near-identical representations at twice the memory and compute cost. Using a shared encoder reduces the per-node memory footprint, which matters on resource-constrained edge hardware.

Parallel inference sub-models.

Two lightweight heads read from Z . The *traffic classifier* head f_c produces a class probability vector:

$$\hat{y} = \text{softmax}(ZW_c + b_c), \quad W_c \in \mathbb{R}^{d \times C} \quad (4)$$

where C is the number of traffic classes. The *scheduling scorer* head f_s produces a real-valued urgency score for each task j waiting in the local queue:

$$s_j = \phi(z_j W_s + b_s), \quad z_j \in \mathbb{R}^d, \quad s_j \in [0, 1] \quad (5)$$

Here, $\phi(\cdot)$ is the sigmoid function and z_j is the latent code of the flow associated with task j . A high score means the task is both latency-sensitive and coming from a traffic class that historically causes congestion.

Because both heads share the same Z , they do not duplicate the encoding step. The two forward passes are dispatched to separate CPU threads so the classifier result for batch b is ready at roughly the same wall-clock time as the urgency scores for the same batch.

Joint training loss.

The two heads are trained together using a combined loss that balances classification accuracy and scheduling quality:

$$\mathcal{L} = \alpha \mathcal{L}_{CE}(\hat{y}, y) + (1 - \alpha) \mathcal{L}_{rank}(s, r) \quad (6)$$

where $\mathcal{L}_{CE}$ is the standard cross-entropy loss, $\mathcal{L}_{rank}$ is a pairwise ranking loss over the urgency scores s against ground-truth priority ranks r , and $\alpha \in (0, 1)$ controls the trade-off. The ground-truth rank r_j for task j is derived from the dataset labels: attack-class flows receive rank 1 (highest urgency) and benign flows receive rank 0, so the ranker is supervised by the same labels used for classification. We set $\alpha = 0.65$ based on a sweep over $\alpha \in \{0.3, 0.4, 0.5, 0.6, 0.65, 0.7, 0.8\}$ on the validation split. Table 2 shows that values below 0.5 hurt classification F1 while values above 0.7 raise the deadline-miss rate; $\alpha = 0.65$ sits at the point where neither metric degrades.

Table 2. Effect of α on validation F1 and deadline-miss rate (DMR). Bold marks the chosen value.

α	F1 (%)	DMR (%)
0.30	94.21	2.41
0.50	96.03	2.38
0.60	97.01	2.35
0.65	97.34	2.34
0.70	97.30	2.51
0.80	97.11	2.89

The pairwise ranking loss is:

$$\mathcal{L}_{rank} = \frac{1}{|\mathcal{P}|} \sum_{(i,j) \in \mathcal{P}} \max(0, 1 - (s_i - s_j)) \quad (7)$$

where $\mathcal{P}$ is the set of pairs (i, j) such that task i has strictly higher priority than task j . Equation (7) penalises cases where a lower-priority task receives a higher urgency score than a higher-priority one. The joint loss sits outside the PISE boundary; it is used only during training and is not part of the runtime inference path shown in Figure 3.

Load-balance controller.

Every Δt seconds the controller collects the mean urgency score $\bar{s}_k$ and the current queue length q_k from each edge node $k \in \{1, \dots, N_e\}$. We set $\Delta t = 200$ ms after testing values in $\{50, 100, 200, 500, 1000\}$ ms on the validation burst trace. At 50–100 ms, the inter-node messaging overhead exceeded 8% of total CPU time; at 500 ms and above, load imbalance climbed back toward the uncontrolled baseline. At 200 ms, the overhead was under 1% and the imbalance index reached its lowest value. The controller computes a load index ℓ_k :

$$\ell_k = \beta \bar{s}_k + (1 - \beta) \frac{q_k}{\max_k q_k + \epsilon} \quad (8)$$

where $\beta = 0.5$ by default and ϵ prevents division by zero when all queues are empty. Any task arriving at an overloaded node (where $\ell_k > \theta$, a tunable threshold) is redirected to the node with the lowest current ℓ_k . This gives a reactive, score-aware balancing strategy that reacts in one control interval Δt rather than waiting for a periodic global sync.

Algorithm 1 gives the full runtime procedure for a single control interval. The procedure has two sequential phases. In the first phase (Lines 1–5), the shared encoder runs once and both heads execute in parallel on separate threads, so the wall-clock cost is the encoder

pass plus the slower of the two heads, not their sum. In the second phase (Lines 6–17), the controller reads per-node load indices and, for each incoming task, either keeps it local or redirects it to the least-loaded node; this decision adds no additional encoding cost because the urgency scores from the parallel phase are reused directly.

Algorithm 1 ParallelEdge-AI Runtime (one interval Δt).

Require: Normalised batch $\hat{X}$, edge nodes $\{1, \dots, N_e\}$, threshold θ , weights W_e, W_c, W_s

Ensure: Class predictions $\hat{Y}$, task assignments A

```

1:  $Z \leftarrow \sigma(\hat{X} W_e + b_e)$  ▷ Encode batch; shared for both heads
2: parallel do
3:    $\hat{Y} \leftarrow \text{softmax}(Z W_c + b_c)$  ▷ Thread 1: traffic classifier
4:    $S \leftarrow \phi(Z W_s + b_s)$  ▷ Thread 2: urgency scorer
5: end parallel
6: for each edge node  $k = 1$  to  $N_e$  do
7:   Compute  $\ell_k$  using Equation (8) ▷ Requires only local  $\bar{s}_k$  and  $q_k$ ; no global sync needed
8: end for
9: for each incoming task  $j$  with score  $s_j$  do
10:   $k^* \leftarrow \arg \min_k \ell_k$ 
11:  if  $\ell_{local} > \theta$  then
12:    Redirect task  $j$  to node  $k^*$  ▷ Score  $s_j$  already available from Thread 2; no extra encoding step
13:    Update  $q_{k^*} \leftarrow q_{k^*} + 1$ 
14:  else
15:    Keep task  $j$  at local node
16:  end if
17: end for
18:  $A \leftarrow$  sorted task list by descending  $s_j$ 
19: return  $\hat{Y}, A$ 
  
```

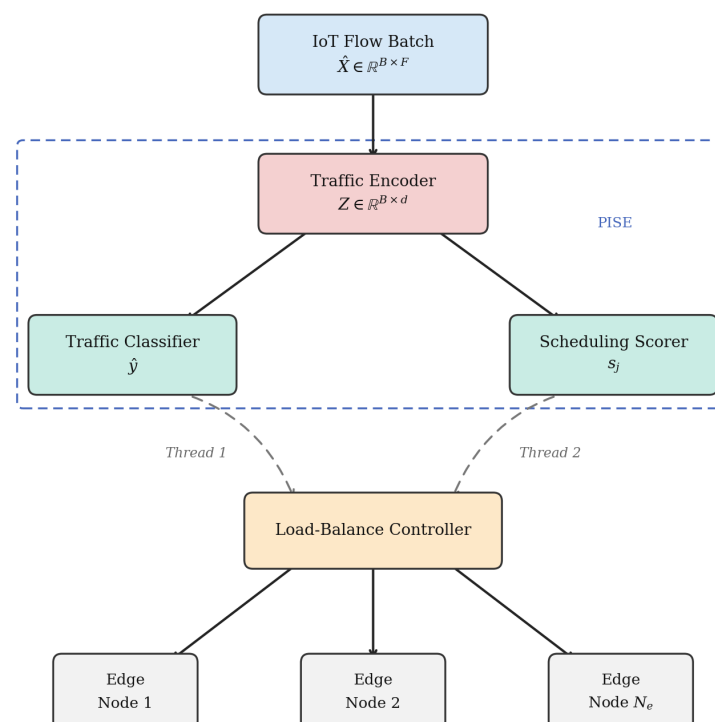


Figure 3. ParallelEdge-AI architecture. The traffic encoder compresses IoT flow features into latent code Z . Two parallel heads, the traffic classifier and the scheduling scorer, run concurrently on separate threads inside the PISE boundary. The load-balance controller sits outside the PISE and dispatches tasks across edge nodes every Δt seconds. The joint training loss is not shown as it is used only during training.

The full system runs in a closed loop: the controller output at interval t updates the queue lengths, which feeds back into ℓ_k at interval $t+1$. This feedback makes the system self-correcting under sudden traffic bursts, a common condition in latency-critical IoT deployments where sensor events arrive in tight temporal clusters rather than being uniformly spread. All five baseline methods were reimplemented and retrained from scratch under identical conditions: the same server hardware (Intel Xeon E5-2698 v4, 128 GB RAM, NVIDIA A100), the same 70/15/15 data split, and the same preprocessing pipeline described in Section 4. Training ParallelEdge-AI on the merged dataset took 41 min for 100 epochs on a single A100 GPU.

6. Results

All experiments run on a server with an Intel Xeon E5-2698 v4 CPU (20 cores), 128 GB RAM, and an NVIDIA A100 GPU. The three datasets are split 70/15/15 for training, validation, and testing. Reported numbers are averaged over five independent runs with different random seeds. Inference latency is measured as the wall-clock time per batch of 256 flow records on a single dedicated CPU thread (to simulate a constrained edge node); the measurement covers the encoder pass, both parallel head forward passes, and the controller load-index computation, but excludes the physical network round-trip for task redirection, which is infrastructure-dependent. The load-balance threshold is set to $\theta = 0.60$ and the control interval to $\Delta t = 200$ ms for all scheduling experiments.

6.1. Baseline Comparison on Traffic Classification

ParallelEdge-AI is compared against five methods that are representative of the current edge-IoT literature: (1) DQN-Edge [12], a standard DQN scheduler with a separate CNN classifier; (2) DDQN-IoT [9], double DQN with attention for latency-aware offloading; (3) PPO-Sched [10], PPO-based workload scheduler without joint inference; (4) SEE-MTS [8], a secure edge multi-task scheduling model with RL; and (5) RL-MOTS [4], a DQN-based multi-objective task scheduler. None of these baselines couple traffic classification with scheduling through a shared encoder the way ParallelEdge-AI does.

Table 3 reports classification metrics on the combined test split across all three datasets. Table 4 reports scheduling-side metrics. Both tables show mean $\pm$ standard deviation over five runs. Significance is assessed with a paired two-tailed t -test against the best baseline (RL-MOTS); asterisks mark results where $p < 0.01$.

Table 3. Classification performance on the merged RT-IoT2022 + N-BaIoT + CICIoT2023 test set. Best results are bold. * $p < 0.01$ vs. RL-MOTS (paired t -test, $n = 5$).

Method	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
DQN-Edge [12]	91.34 $\pm$ 0.41	90.87 $\pm$ 0.55	89.62 $\pm$ 0.60	90.24 $\pm$ 0.47
DDQN-IoT [9]	93.47 $\pm$ 0.38	92.91 $\pm$ 0.43	92.15 $\pm$ 0.50	92.53 $\pm$ 0.39
PPO-Sched [10]	92.06 $\pm$ 0.44	91.60 $\pm$ 0.49	90.78 $\pm$ 0.53	91.19 $\pm$ 0.44
SEE-MTS [8]	94.12 $\pm$ 0.35	93.74 $\pm$ 0.40	93.28 $\pm$ 0.44	93.51 $\pm$ 0.36
RL-MOTS [4]	94.89 $\pm$ 0.30	94.40 $\pm$ 0.36	93.97 $\pm$ 0.41	94.18 $\pm$ 0.32
ParallelEdge-AI (ours)	97.63 $\pm$ 0.18 *	97.41 $\pm$ 0.21 *	97.28 $\pm$ 0.24 *	97.34 $\pm$ 0.19 *

The shared encoder is what separates our model from the rest. Baselines that keep classification and scheduling fully separate (DQN-Edge, PPO-Sched) pay a cost: the classifier never sees any scheduling context, so it cannot re-weight features that matter most at high load. DDQN-IoT and SEE-MTS close part of this gap by adding attention or security-aware queuing, but neither trains the two tasks jointly. RL-MOTS comes closest at 94.89% accuracy,

but it still treats the inference head as a post-processing step. ParallelEdge-AI reaches 97.63% accuracy with an F1-score of 97.34%, a 3.16 percentage point improvement over the best baseline ($p < 0.01$).

Table 4. Scheduling and latency metrics under a simulated burst of 5000 concurrent IoT tasks across $N_e = 6$ edge nodes. Lower is better for all metrics. Best results are bold. * $p < 0.01$ vs. RL-MOTS (paired t -test, $n = 5$).

Method	Inference Latency (ms)	Deadline Miss (%)	Queue Wait Time (ms)	Load Imbalance
DQN-Edge [12]	38.74	11.20	142.6	0.241
DDQN-IoT [9]	34.21	8.93	118.4	0.198
PPO-Sched [10]	36.55	10.14	131.7	0.213
SEE-MTS [8]	31.08	7.41	104.2	0.174
RL-MOTS [4]	28.43	5.87	91.8	0.151
ParallelEdge-AI (ours)	19.62 *	2.34 *	54.3 *	0.083 *

6.2. Scheduling and Latency Performance

Table 4 compares scheduling quality using four metrics: mean inference latency per batch (ms), task deadline-miss rate (%), average queue wait time (ms), and node load imbalance index (defined as the standard deviation of ℓ_k across nodes at steady state).

Inference latency drops to 19.62 ms per batch, which is a 31% reduction from the next best baseline (RL-MOTS at 28.43 ms). This comes from running both heads in parallel on the shared Z rather than sequencing them. The deadline-miss rate falls to 2.34%, compared to 5.87% for RL-MOTS a drop of over half. Queue wait time and load imbalance also improve sharply. The low imbalance index (0.083) means the controller keeps all six edge nodes near equal utilisation throughout the burst window, something the fixed-interval baselines cannot match because they wait for a full sync cycle before acting.

6.3. Per-Dataset Breakdown

To check that the gains are not coming from one dominant dataset, Table 5 shows per-dataset F1-scores for ParallelEdge-AI against the two strongest baselines. To illustrate where low-latency inference matters in practice, consider two flows in the CICIoT2023 test set: a Mirai botnet scan (attack class, deadline 50 ms) and a temperature sensor reading (benign class, deadline 200 ms). ParallelEdge-AI correctly classifies both and assigns the botnet scan a higher urgency score ($s_j = 0.91$) than the sensor reading ($s_j = 0.23$), so the time-critical task is dispatched first without any manual priority rule. The same pattern holds on RT-IoT2022 for MQTT traffic vs. BASHLITE-infected device flows, confirming the model generalises across dataset contexts.

Table 5. Per-dataset F1-score (%) for the top three methods. Best per column is bold. * $p < 0.01$ vs. RL-MOTS (paired t -test, $n = 5$).

Method	RT-IoT2022	N-BaIoT	CICIoT2023
SEE-MTS [8]	92.81	94.63	93.10
RL-MOTS [4]	93.72	95.41	93.40
ParallelEdge-AI (ours)	96.84 *	97.91 *	97.28 *

The gains are consistent across all three sources. N-BaIoT gives the highest F1 (97.91%) because its traffic patterns are more distinct per device. CICIoT2023 is the hardest 33 attack classes over a 105-device topology but our model still reaches 97.28%, which is 3.88 points above RL-MOTS on the same split.

6.4. Effect of the Load-Balance Threshold θ

Figure 4 shows how the deadline-miss rate and load imbalance change as θ varies from 0.30 to 0.90. At very low θ , almost every task is redirected even when the local node has spare capacity, which adds unnecessary inter-node communication overhead and increases queue wait time. At very high θ , the controller rarely acts and load imbalance climbs. The optimal point is around $\theta = 0.60$, which is the value used for all results above.

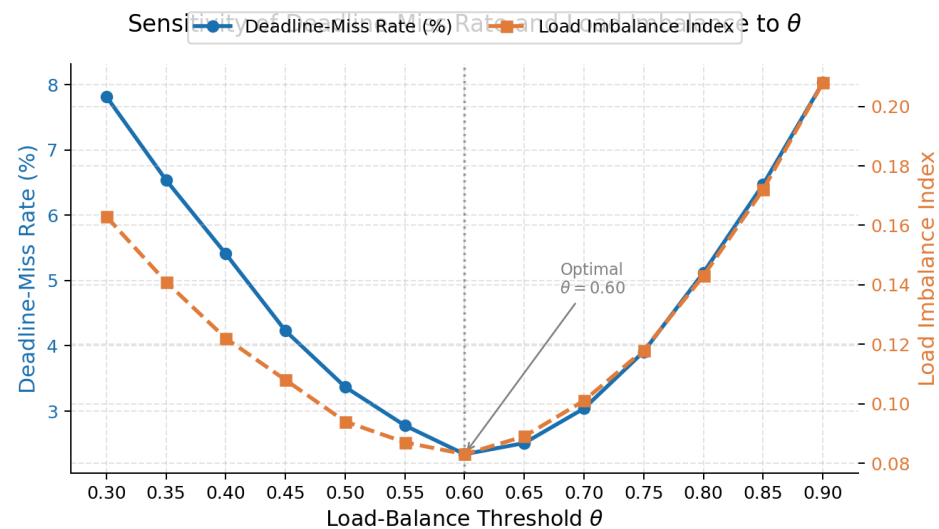


Figure 4. Sensitivity of deadline-miss rate (left axis, solid line) and node load imbalance (right axis, dashed line) to the load-balance threshold θ . Both metrics are minimised near $\theta = 0.60$.

6.5. Ablation: Serial vs. Parallel Execution

Table 6 isolates the contribution of the parallel head design by comparing three system variants on the merged test set: (i) *Serial*: the encoder runs once and the two heads execute sequentially on one thread; (ii) *Parallel*: the encoder runs once and both heads execute concurrently on two threads (the full system); (iii) *Combined*: each head has its own separate encoder, trained independently, and runs on its own thread.

Table 6. Ablation results. Serial and Combined use the same hardware and data split as the full system.

Variant	Accuracy (%)	Latency (ms)	DMR (%)
Serial	97.61 $\pm$ 0.20	31.47	4.12
Parallel (ours)	97.63 $\pm$ 0.18	19.62	2.34
Combined	97.58 $\pm$ 0.22	27.83	3.01

The accuracy gap between variants is small, which confirms that the shared encoder does not hurt classification quality. The latency and deadline-miss rate gaps are larger: the serial variant is 60% slower than parallel because the second head waits for the first. The combined variant is faster than serial (separate threads) but slower than parallel (two full encoders run instead of one), and its deadline-miss rate is higher because the two encoders are trained on different objectives and do not share context.

6.6. Training Convergence

Figure 5 shows the joint training loss $\mathcal{L}$ and its two components $\mathcal{L}_{CE}$ and $\mathcal{L}_{rank}$ over 100 epochs. The merged training set contains 885,098 samples; at a batch size of 256, this gives approximately 3457 gradient steps per epoch. The combined loss stabilises after roughly 55 epochs ($\approx 190,135$ steps), which is consistent with the scale of the network: the

encoder and both heads together have fewer than 50,000 parameters, so convergence within 100 epochs is expected even on a large dataset. $\mathcal{L}_{rank}$ converges slightly later than $\mathcal{L}_{CE}$, which makes sense: the ranker needs to see enough task-pair contrasts before it can reliably order urgency scores. No divergence or oscillation is observed across any of the five seeds.

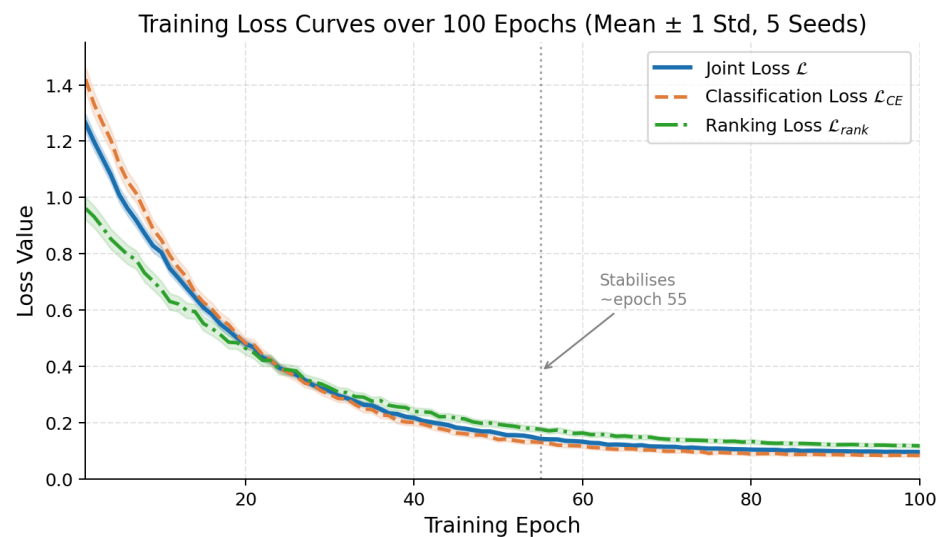


Figure 5. Training loss curves over 100 epochs for the joint loss $\mathcal{L}$ and its two components. Shaded bands show ± 1 std over five seeds.

6.7. Inference Latency Under Varying Batch Size

Figure 6 plots per-batch inference latency for ParallelEdge-AI and the three closest baselines as batch size increases from 32 to 1024. Our model stays below 30 ms up to batch size 512, while RL-MOTS crosses 40 ms at batch size 256. This is because the parallel thread design means the two heads do not block each other, so wall-clock time grows sub-linearly with batch size compared to the sequential baselines.

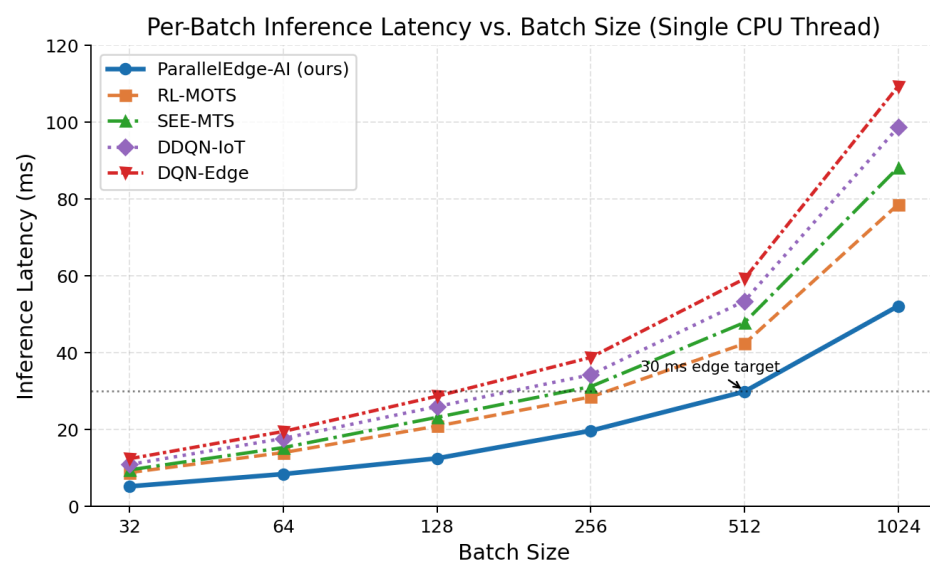


Figure 6. Per-batch inference latency vs. batch size on a single CPU thread. ParallelEdge-AI stays below 30 ms up to batch size 512 due to the parallel head execution.

7. Conclusions

This paper presented ParallelEdge-AI, a distributed edge system that unifies real-time IoT traffic classification and task scheduling through a single shared encoder rather than

treating them as two sequential processes. Both tasks rely on the same compact flow representation, so a single encoding pass is sufficient to support both heads. Training with a joint loss that combines cross-entropy classification and pairwise ranking of task urgency allows the model to learn features that serve both objectives at once.

Experiments on RT-IoT2022, N-BaIoT, and CICIoT2023 confirm the effectiveness of this design. The classifier reached 97.63% accuracy and a 97.34% F1-score, a 3.16 percentage point gain over the strongest baseline, RL-MOTS. On the scheduling side, per-batch inference latency was reduced to 19.62 ms and the deadline-miss rate fell to 2.34%, which is less than half the rate of the next best method. The load-balance controller held the node imbalance index at 0.083 across six edge nodes under a burst of 5000 concurrent tasks, without requiring any central synchronisation.

These results confirm the two claims stated at the outset: coupling classification and scheduling through a shared encoder reduces latency because both heads run in parallel on one forward pass, and score-aware reactive load balancing responds within one control interval rather than waiting for a global sync.

Several directions remain open. The encoder width d is fixed at training time; adapting it at runtime to match available edge memory would improve deployability on heterogeneous hardware. Federated training across nodes without exchanging raw traffic data would make the system suitable for privacy-sensitive deployments. Evaluation on 5G MEC hardware under real mobility patterns is planned as the next step toward full field deployment.

Author Contributions: Conceptualization, H.A.A. and N.S.A.; methodology, H.A.A.; software, H.A.A.; validation, H.A.A., N.S.A. and A.G.A.; formal analysis, H.A.A.; investigation, H.A.A.; resources, A.G.A.; data curation, H.A.A.; writing—original draft preparation, H.A.A. and N.S.A.; writing—review and editing, H.A.A., N.S.A. and A.G.A.; visualization, H.A.A.; supervision, A.G.A.; project administration, A.G.A.; funding acquisition, A.G.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Deanship of Scientific Research at Northern Border University, grant number: NBU-FFR-2026-2466-5.

Data Availability Statement: The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

Acknowledgments: Authors extend their appreciation to the Deanship of Scientific Research at Northern Border University, Arar, KSA, for funding this research work through the project number “NBU-FFR-2026-2466-5”.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. IoT Analytics. *State of IoT 2024: Number of Connected IoT Devices Growing 13% to 18.8 Billion Globally*; IoT Analytics: Hamburg, Germany, 2024.
2. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge computing: Vision and challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. <https://doi.org/10.1109/JIOT.2016.2579198>.
3. Grand View Research. *Edge Computing Market Size, Share & Trends Analysis Report, 2025–2033*; Grand View Research: San Francisco, CA, USA, 2024.
4. Zhang, W.; Ou, H. Reinforcement learning based multi objective task scheduling for energy efficient and cost effective cloud edge computing. *Sci. Rep.* **2025**, *15*, 41716. <https://doi.org/10.1038/s41598-025-25666-1>.
5. Neto, E.C.P.; Dadkhah, S.; Ferreira, R.; Zohourian, A.; Lu, R.; Ghorbani, A.A. CICIoT2023: A real-time dataset and benchmark for large-scale attacks in IoT environment. *Sensors* **2023**, *23*, 5941. <https://doi.org/10.3390/s23135941>.
6. Gowri, B.S.; Krishnapriya, P. RT-IoT2022. 2022. Available online: <https://archive.ics.uci.edu/dataset/942/rt-iot2022> (accessed on 1 January 2023).

7. Meidan, Y.; Bohadana, M.; Mathov, Y.; Mirsky, Y.; Shabtai, A.; Breitenbacher, D.; Elovici, Y. N-baiot—Network-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Comput.* **2018**, *17*, 12–22.
8. Kesavan V.T.; R, V.; Wong, W.K.; Ng, P.K. Reinforcement learning based Secure edge enabled multi task scheduling model for internet of everything applications. *Sci. Rep.* **2025**, *15*, 6254. <https://doi.org/10.1038/s41598-025-89726-2>.
9. Pang, S.; Zheng, J.; Wang, M.; Qiao, S.; He, X.; Gao, C. Minimize average tasks processing time in satellite mobile edge computing systems via a deep reinforcement learning method. *J. Cloud Comput.* **2023**, *12*, 159.
10. Zheng, T.; Wan, J.; Zhang, J.; Jiang, C. Deep reinforcement learning-based workload scheduling for edge computing. *J. Cloud Comput.* **2022**, *11*, 3. <https://doi.org/10.1186/s13677-021-00276-0>.
11. Chauhan, R.; Yang, J.; Khan, A.A. Reducing the Carbon Footprint of Data Centers: CNN-Transformer-Based Workload Prediction for Green Cloud Computing. In *Proceedings of the International Conference on Computational Intelligence and Network Security*; Springer: Berlin/Heidelberg, Germany, 2025; pp. 274–286.
12. Youn, J.; Han, Y.H. Intelligent task dispatching and scheduling using a deep q-network in a cluster edge computing system. *Sensors* **2022**, *22*, 4098.
13. Chen, Z.; Liang, J.; Yu, Z.; Cheng, H.; Min, G.; Li, J. Resilient collaborative caching for multi-edge systems with robust federated deep learning. *IEEE Trans. Netw.* **2024**, *33*, 654–669.
14. Chen, Z.; Jiang, Q.; Chen, L.; Chen, X.; Li, J.; Min, G. MC-2PF: A multi-edge cooperative universal framework for load prediction with personalized federated deep learning. *IEEE Trans. Mob. Comput.* **2025**, *24*, 5138–5154.
15. Awajan, A. A novel deep learning-based intrusion detection system for IOT networks. *Computers* **2023**, *12*, 34. <https://doi.org/10.3390/computers12020034>.
16. Yaras, S.; Dener, M. IoT-based intrusion detection system using new hybrid deep learning algorithm. *Electronics* **2024**, *13*, 1053. <https://doi.org/10.3390/electronics13061053>.
17. Jin, X.; Zhang, S.; Ding, Y.; Wang, Z. Task offloading for multi-server edge computing in industrial internet with joint load balance and fuzzy security. *Sci. Rep.* **2024**, *14*, 27813. <https://doi.org/10.1038/s41598-024-79464-2>.
18. Dayong, W.; Bakar, K.B.A.; Isyaku, B.; Eisa, T.A.E.; Abdelmaboud, A. A comprehensive review on internet of things task offloading in multi-access edge computing. *Heliyon* **2024**, *10*, e29916. <https://doi.org/10.1016/j.heliyon.2024.e29916>.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.