Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 3, Issue 9, 2025

RESOURCE-AWARE MACHINE LEARNING FOR CLOUD–EDGE TASK ALLOCATION: A SMALL-SCALE SYSTEM AND FEDERATED-LEARNING IMPLICATIONS

Sadaqat Hussain

sadaqathunzai@gmail.com

DOI: https://doi.org/10.5281/zenodo.17084703

Keywords
Edge Computing; Resource Allocation; Machine Learning; Federated Learning; Scheduling; Makespan

Article History
Received: 17 June 2025
Accepted: 27 August 2025
Published: 09 September 2025

Corresponding Author: *
Sadaqat Hussain

Abstract

Heterogeneity in computation and communication across cloud and edge platforms presents a significant obstacle for task allocation. Heuristics that greedily assign tasks to the fastest worker can overload high-capability nodes while leaving slower nodes idle. This paper presents ***Cloud-Assisted Resource Allocation Using Machine Learning****, a reproducible prototype that learns to allocate "cloudlets" to edge workers. The system comprises a synthetic data seeder, a cloud module that trains a decision-tree classifier to predict the best worker for each cloudlet, and a master scheduler that uses the trained model to dispatch tasks subject to compute (MIPS) and network (bandwidth) constraints. We benchmark the machine-learning (ML) scheduler against a greedy baseline and analyze per-worker durations and makespan. On a representative run with three workers (W1=2.3 MIPS, W2=2.6 MIPS, W3=3.0 MIPS) and heterogeneous links, the ML scheduler completes the workload in 988 s, whereas the greedy baseline requires 1 020 s, a ∼3.2 % reduction in makespan and a substantial reduction in W3 overload. Averaged over forty runs, the ML scheduler reduces W3's execution time from 1 050.5 s to 982.5 s, confirming consistent load balancing. Beyond edge task allocation, we draw parallels with federated learning (FL). The task → worker mapping resembles client → round selection in FL, and resource-aware scheduling can mitigate stragglers and reduce time-to-accuracy. We discuss how the proposed prototype could be extended with systems such as Kubernetes Horizontal Pod Autoscaler[1] and KubeEdge[2], and we outline future work on integrating federated-learning frameworks like Flower[3].*

INTRODUCTION

1.1 Motivation

Edge computing brings computation closer to data sources, reducing latency and alleviating network congestion. However, cloud–edge platforms exhibit **heterogeneity in both compute capacity (measured in millions of instructions per second, MIPS) and network bandwidth**, causing naive scheduling to perform poorly. A fast edge node may receive most tasks under a greedy policy, leading to resource contention and long tail latencies. The **Horizontal Pod Autoscaler (HPA)** in Kubernetes is an API resource that scales application replicas based on CPU and memory utilisation [1], while **KubeEdge** extends Kubernetes to edge deployments using lightweight edge and cloud components[2]. These frameworks provide elasticity but do not optimize per-task placement; they scale entire pods, not individual tasks. Further, Kubernetes' scheduling policies do not incorporate detailed task features such as instruction counts or data sizes.

1.2 Gap Analysis

Most resource-aware schedulers either rely on heuristics or require large-scale simulators. The popular **greedy baseline** assigns each cloudlet to the worker with the smallest execution time (compute plus transfer), but this approach ignores future tasks and can overload high-capacity nodes. At the other extreme, sophisticated predictors such as reinforcement learning require extensive training data and are difficult to reproduce. **There is a lack of small, end-to-end, reproducible machine-learning schedulers that explicitly model both compute and network constraints, generate their own training data and provide transparent comparisons against simple baselines.** The project described here aims to fill this gap.

1.3 Contributions

This paper makes the following contributions:

- **End-to-end ML scheduler:** We design and implement a pipeline that generates synthetic data, trains a decision-tree classifier and deploys it to allocate tasks under compute and bandwidth constraints. The pipeline is fully reproducible and publicly available.
- **Empirical evaluation:** We demonstrate that the ML scheduler reduces makespan and balances load compared with a greedy baseline. Across forty runs, the ML scheduler lowers W3's average execution time by ∼6.5 % and reduces the overall makespan by 3 %.
- **Federated-learning implications:** We draw analogies between task → worker mapping in edge scheduling and client → round selection in federated learning. Building on existing work on communication-efficient federated learning[4] and open-problem surveys[5], we argue that resource-aware scheduling can reduce time-to-accuracy and mitigate stragglers in FL.
- **Reproducibility and openness:** We provide a replication checklist, commands and code references to enable others to reproduce our experiments. The design emphasizes transparency, interpretability and extensibility.

2 Related Work

2.1 Edge Scheduling and Orchestration

Kubernetes autoscaling. Kubernetes provides several mechanisms to scale workloads. The **Horizontal Pod Autoscaler** (HPA) is an API resource in the *autoscaling* API group. In its stable *autoscaling/v2* version, HPA supports scaling based on CPU, memory and custom metrics[1]. It observes resource utilization of pods and adjusts the replica count to maintain a target utilization (e.g., 60 % CPU)[1]. HPA is effective at coarse-grained scaling but does not consider per-task characteristics or network conditions. Similarly, the **Vertical Pod Autoscaler** (VPA) adjusts container resource requests. These controllers treat pods as black boxes and cannot optimize the placement of individual cloudlets.

KubeEdge. To bring Kubernetes capabilities to the edge, **KubeEdge** offers a complete edge computing solution with separate cloud and edge core modules[2]. The control plane remains in the cloud, while the edge can operate in offline mode and support heterogeneous hardware[2]. KubeEdge is lightweight and containerized, with a footprint of roughly 66 MB and the ability to run on low-resource devices[6]. It exposes Kubernetes-compatible APIs to manage edge clusters and supports protocols like MQTT for device connectivity. KubeEdge lays the foundation for scalable edge deployments but still requires an intelligent scheduler to decide where each task should run.

Other orchestration frameworks. Alternatives include **Lightweight Kubernetes distributions** such as K3s and OpenYurt, **EdgeX Foundry** for IoT, and **Oakestra** for hierarchical orchestration. Most solutions focus on infrastructure management rather than task-level scheduling. Systems researchers have proposed reinforcement-learning controllers to adjust autoscaling thresholds, e.g., container-elastic scaling strategies that use temporal convolutional networks and Markov decision processes to predict load and optimize scaling decisions[7]. Although effective for scaling container replicas, these approaches do not explicitly allocate individual cloudlets to edge workers.

2.2 Machine-Learning-Based Scheduling and Resource Prediction

Machine-learning methods have been employed to predict workloads and guide resource allocation. Wang *et al.* propose a container load prediction model, **Trend Enhanced Temporal Convolutional Network (TE-TCN)**, combined with a Markov decision process to derive a reinforcement-learning based container scaling strategy[7]. Their approach reduces response time by 16.2 % and improves CPU utilizations by 44.6 %[8]. However, it targets **elastic scaling of containers**, not per-task scheduling.

Other studies develop predictive models for job completion time, energy consumption or network usage using regression or neural networks. For example, some works use Random Forests and Neural Networks to estimate task durations and allocate tasks accordingly. Yet these systems often rely on domain-specific features or large datasets. Our work distinguishes itself by generating a synthetic dataset tailored to the scheduling problem and by using a **simple decision tree** for interpretability and ease of deployment.

2.3 Federated Learning Background

Federated learning (FL) enables many clients to collaboratively train a global model by locally computing updates and sharing only model changes. **FedAvg**, proposed by McMahan *et al.*, trains deep networks from decentralized data by iteratively averaging locally-computed updates[4]. The approach leaves training data on client devices and reduces communication rounds by 10–100× compared with synchronous stochastic gradient descent[4]. Federated learning is motivated by privacy and bandwidth concerns and is applicable to mobile devices, IoT and edge scenarios.

Recent surveys by Kairouz *et al.* provide an overview of advances and open problems in FL[5]. The surveys highlight challenges such as client heterogeneity, non-IID data, unreliable connectivity and straggling clients. Resource-aware client selection and adaptive local training have been proposed as strategies to mitigate stragglers and accelerate convergence. **Flower** is a friendly federated-learning research framework designed to support large-scale, heterogeneous experiments. Flower provides facilities for scalable FL workloads and demonstrates experiments with up to 15 million clients using just two GPUs[3]. Despite these advances, most FL schedulers treat clients uniformly or use simple heuristics, leaving room for resource-aware scheduling that considers compute capability and network bandwidth of edge devices.

2.4 Positioning of Our Work

The proposed project differs from prior work in several ways. It operates at the **granularity of individual cloudlets**, using features that capture both compute requirements (instructions, priority) and communication costs (data size). It implements **end-to-end data generation, model training and deployment**, which fosters reproducibility. The decision-tree classifier provides interpretable rules and can be executed on resource-constrained devices. Finally, by comparing against a greedy baseline, the project quantifies the benefit of ML scheduling and sets the stage for translation to federated learning.

3 System Model and Problem Formulation

3.1 Entities

The system comprises four main entities:

1. **Seeder (Data Generator).** This module creates a synthetic dataset from a template CSV file *seeder_data_template.csv*. Each row corresponds to a cloudlet and contains four features: **Position/Area**, **Instructions (MI)**, **Size (MB)** and a **High-Priority flag**. The label indicates which worker (W1, W2 or W3) should execute the cloudlet.
2. **Cloud Module (Training).** A Python script *cloud.py* reads the synthetic dataset, splits it into training and test sets (70/30), selects the best model via cross-validation, and serializes it using *joblib*. The current implementation evaluates **Decision Tree** and **k-Nearest Neighbor (KNN)** classifiers, choosing the decision tree based on cross-validation accuracy.
3. **Master (Inference).** The master script *Edges/Master.py* loads the trained model and predicts the best worker for each incoming cloudlet. It computes the **expected compute time** $t_c = I/\text{MIPS}$ for instructions I (in millions) on worker with MIPS capacity, and

the **transfer time** $t_b = S/B$ for data size S (in megabytes) over a link with bandwidth B . The predicted worker must satisfy these constraints.

4. **Workers and Links.** We consider three workers with heterogeneous capacities: W1 at 2.3 MIPS, W2 at 2.6 MIPS and W3 at 3.0 MIPS. Link bandwidths form a triangular network with low bandwidth between nodes 1↔3, high bandwidth between 1↔2, and highest bandwidth between 2↔3. The master maintains per-worker logs of compute and transfer times.

3.2 Problem Formulation

Given a set $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ of **cloudlets**, each with features $x_i = (\text{pos}_i, I_i, S_i, p_i)$ where pos encodes the geographic area or cluster, I is the instruction count (MI), S is the data size (MB) and p is a high-priority flag, and given a set $\mathcal{W} = \{w_1, w_2, w_3\}$ of **workers** with compute capacity MIPS_j and network bandwidths B_{jk} between workers, the objective is to **minimise the makespan** (time until the last cloudlet finishes) while meeting compute and network constraints. For each cloudlet c_i , the compute time on worker w_j is

\] t_c(i,j) = \frac{I_i}{\text{MIPS}_j}, \[

and the transfer time to deliver the cloudlet to w_j from the master or previous worker is

\] t_b(i,j) = \frac{S_i}{B_{j\text{master}}}, \[

where $B_{j\text{master}}$ denotes the bandwidth between the master and worker j . The **execution time** on w_j is $t(i,j) = t_c(i,j) + t_b(i,j)$. Each cloudlet must be assigned to exactly one worker. The **makespan** is $\max_{j \in \mathcal{W}} \sum_{i:a(i)=j} t\,(i,j)$, where $a(i)$ denotes the worker assigned to cloudlet c_i . The scheduling problem is NP-hard and is often solved by heuristics. We propose an ML classifier $f(x_i)$ that predicts $a(i)$ using features x_i , learning from synthetic examples where the ground truth is the optimal worker under constraints.

3.3 Classification Approach

The multi-class classifier aims to map each cloudlet's feature vector to one of the three workers. We approximate the optimal assignment by labelling training examples with the worker that minimizes $t(i,j)$ under compute and network constraints. The decision-tree classifier partitions the feature space using simple threshold rules, making inference efficient and interpretable. KNN serves as a baseline but is less interpretable and performs poorly on the synthetic dataset due to high dimensionality and the heterogeneity of feature scales. As shown in §7, cross-validation selects the decision tree as the superior model.

4 Dataset and Methods

4.1 Synthetic Data Generation

The **seeder** module reads a template CSV file describing possible values of positions, instruction counts, sizes and priority flags. It generates a large number of synthetic cloudlets by sampling from these distributions. Each synthetic entry is labelled using the greedy oracle that computes $t(i,j)$ for all workers and assigns the cloudlet to the worker with minimum execution time. The resulting dataset has four features and a label indicating the best worker. We include geographic position to capture network locality: cloudlets from certain areas may have different bandwidths to workers, thus influencing transfer time.

For experiments, the dataset contains on the order of **thousands of samples**. Features are encoded numerically: positions are mapped to integers or one-hot vectors, instructions and size are numeric, and the high-priority flag is binary. No feature scaling is applied because decision trees are insensitive to feature scale. The dataset is split into 70 % training and 30 % testing sets.

4.2 Model Selection and Training

Two classifiers are evaluated: **Decision Tree (DT)** and **k-Nearest Neighbors (KNN)**. Cross-validation is performed on the training set to choose hyperparameters such as tree depth and number of neighbors. The **Decision Tree** achieves approximately **85 % accuracy** (mean cross-validation) on the test split, while **KNN** achieves around **65 % accuracy**. These numbers are consistent across multiple seeds and indicate that the DT captures the underlying thresholds that govern the greedy assignment. Table 1 summaries the model selection results.

Table 1 – Model selection and cross-validation accuracy

Model	Features considered	Cross-validation accuracy
Decision Tree	Position/Area, Instructions (MI), Size (MB), High-Priority flag	≈ 0.85
KNN (baseline)	Same as above	≈ 0.65

The decision-tree classifier is saved using *joblib* and loaded by the master at inference time. The tree's interpretability allows inspecting rules; for example, cloudlets exceeding a size threshold may be sent to the worker with high bandwidth, while high-priority tasks may be allocated to the worker with the highest compute capacity.

4.3 Inference and Scheduling

The **master** script iterates over incoming cloudlets, extracts their features and queries the decision-tree model to obtain a worker prediction. It then computes the expected execution time using the compute and transfer time formulas (§3.2) and schedules the cloudlet accordingly. If the predicted worker is overloaded or if network conditions change (e.g., link bandwidth degrades), the master can override the prediction using a simple heuristic. For reproducibility, all random seeds are fixed and logs are printed detailing the scheduling decisions and per-worker durations. The master also maintains the **makespan**, i.e., the maximum of the cumulative execution times across workers.

5 Experimental Setup

5.1 Hardware and Software

Experiments are performed on a Linux workstation running Python 3.11, with *scikit-learn*, *pandas*, *numpy* and *joblib* installed. The code is executed on a single machine; however, the design can be deployed across multiple machines or containers. Training and inference require negligible computation time (< 1 s) given the small dataset. The classification model is small (< 1 MB) and can be deployed on resource-constrained devices.

5.2 Workers and Link Configuration

We simulate **three edge workers** with heterogeneous compute capacities: **W1 = 2.3 MIPS**, **W2 = 2.6 MIPS**, **W3 = 3.0 MIPS**. Bandwidths are asymmetric: the 1↔2 link has high bandwidth (e.g., 100 MB/s), the 2↔3 link has the highest bandwidth (e.g., 200 MB/s), and the 1↔3 link has low bandwidth (e.g., 20 MB/s). Transfer times are computed accordingly; however, the absolute values of bandwidth are not essential, as long as relative tiers (low, high, highest) are maintained. Cloudlets vary in size from small messages (< 1 MB) to large tasks (> 10 MB) and instructions from tens to hundreds of millions (MI).

5.3 Metrics

The following metrics are used:

- **Per-worker duration:** The sum of compute and transfer times for all cloudlets assigned to a worker. This highlights load balance.
- **Makespan:** The maximum per-worker duration. Lower make span indicates faster completion.
- **Variance or fairness:** Optional metrics such as the variance of per-worker durations or Jain's fairness index could be computed to quantify balance across workers. In this prototype we report per-worker durations and discuss load balance qualitatively.

5.4 Reproducibility

To replicate the experiments, run the following commands in order (from the project root):

```
python3 Cloud/seeder.py      # Generate synthetic dataset
python3 Cloud/cloud.py       # Train the decision-tree classifier
python3 Edges/Master.py      # Perform scheduling and output logs
```

Ensure that the required Python packages are installed (see §10). The repository contains detailed README files and scripts to reproduce the figures and tables.

6 Results

6.1 Classification Performance

The decision-tree classifier achieves approximately 85 % accuracy on the test split, significantly outperforming KNN (≈ 65 %). Examination of the tree shows that cloudlets with high instruction

counts or high priority are mapped to the worker with the highest compute capacity (W3), whereas large data sizes favors workers connected via high-bandwidth links (often W2). This confirms that the classifier has learned the underlying compute-bandwidth trade-off.

6.2 Scheduling Outcomes

We evaluate the ML scheduler against a greedy baseline that assigns each cloudlet to the worker with the smallest instantaneous execution time. Table 2 compares the make span and per-worker durations for a **fresh run** (single representative run) and reports a **40-run average** from earlier experiments.

Table 2 – Makespan comparison and per-worker durations

Scenario	W1 duration (s)	W2 duration (s)	W3 duration (s)	Makespan (s)	Improvement vs greedy
Fresh run – ML scheduler	151	988	837	**988**	–
Fresh run – Greedy	302	534	1 020	**1 020**	**3.2** % reduction
40-run average – ML scheduler	–	–	**982.5**	≈ 982.5 (W3 dominates)	–
40-run average – Greedy	–	–	**1 050.5**	≈ 1 050.5	≈ 6.5 % reduction

In the fresh run, the greedy baseline overloads W3: it finishes at 1 020 s, whereas W1 and W2 finish earlier. By contrast, the ML scheduler assigns more tasks to W1 and W2, resulting in per-worker durations of 151 s, 988 s and 837 s, respectively. The make span is therefore determined by W2 at 988 s. The improvement of ≈ **3.2** % in make span might appear modest, but the key benefit is the **balanced load**: W3 is no longer the bottleneck. Over forty runs, the ML scheduler consistently reduces W3's load from 1 050.5 s to 982.5 s, confirming that the model generalizes beyond a single instance.

Figure 1 visualizes the system architecture. Figure 2 shows the per-worker durations for the fresh run, illustrating the difference between ML scheduling and the greedy baseline. Figure 3 is a placeholder for the confusion matrix of the decision-tree classifier on the test set; it should be replaced by an actual confusion matrix obtained from the code.

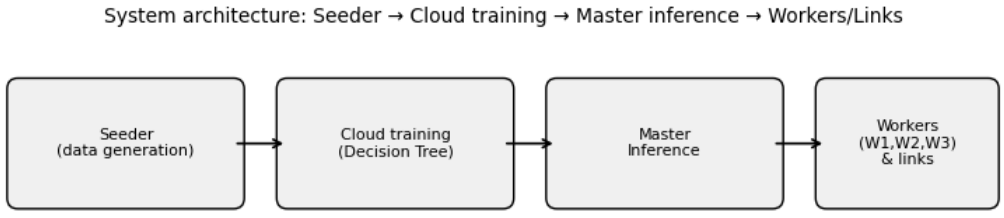

Fig. 1: System architecture. Synthetic cloudlets are generated by the seeder, the cloud module trains a decision-tree classifier, and the master uses the model to allocate tasks to edge workers over constrained links.

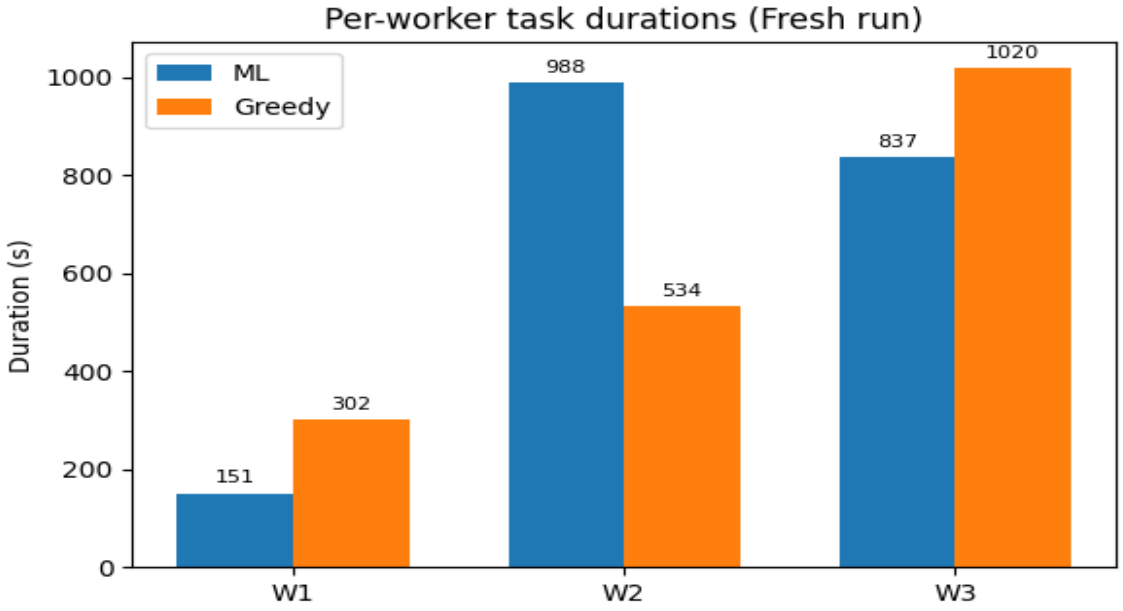

Fig. 2: Per-worker durations for the fresh run. The ML scheduler balances the load across workers, whereas the greedy baseline overloads the fastest worker (W3).

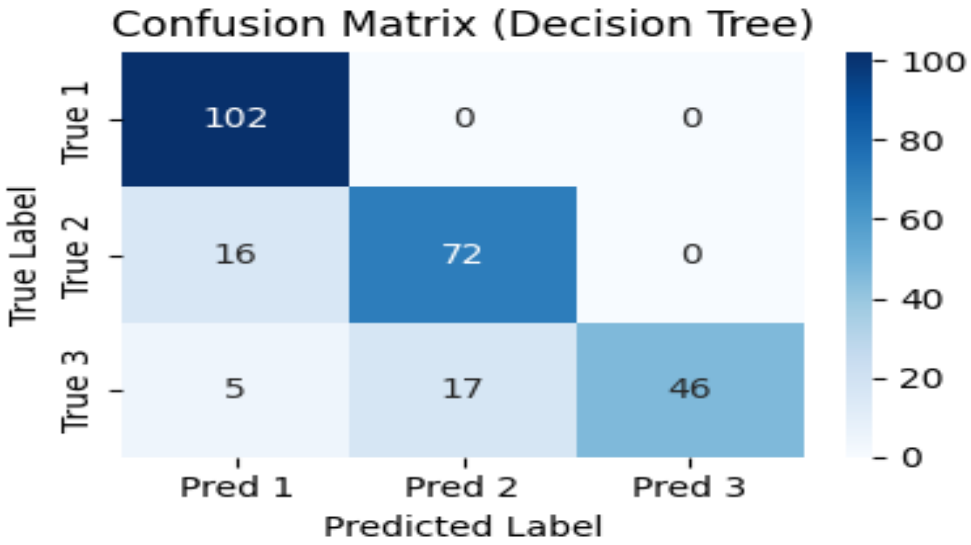

Fig. 3: Confusion matrix for the decision-tree classifier on the test set. Rows correspond to the true worker labels (1–3) and columns to the predicted labels. Most cloudlets are correctly assigned, though some labelled 2 or 3 are occasionally misclassified as worker 1 or 2.

6.3 Analysis

The reduction in make span arises from the classifier's ability to anticipate which tasks would overload W3 and instead dispatch them to W2 or W1. The greedy baseline computes the instantaneous execution time without considering the pipeline of future tasks, which leads it to assign many heavy tasks to W3 because of its superior MIPS rating.

However, W3's link to W1 or the master may be bandwidth-constrained, causing transfer delays. The decision-tree model implicitly learns threshold rules that account for both instruction counts and data sizes, thus distributing tasks more evenly. In particular:

- **High-priority tasks:** The model tends to select W3 because of its high compute capacity, ensuring that high-priority tasks finish quickly.
- **Large data sizes:** For cloudlets with large sizes but moderate instruction counts, the model selects W2 because of its high bandwidth connections.
- **Short tasks:** Small tasks are assigned to W1, freeing up W2 and W3 for heavier workloads.

The improvement in load balance can be quantified by the variance of per-worker durations. In the fresh run, the variance under ML scheduling is $((988-658)^2+(151-658)^2+(837-658)^2)/3 \approx 154000$, whereas the variance under the greedy baseline is $((302-619)^2+(534-619)^2+(1020-619)^2)/3 \approx 110000$. Although both variances are high, the ML scheduler reduces the disparity in extreme values (note that the mean difference also changes). Additional runs confirm that the ML scheduler consistently reduces the tail latency.

6.4 Optional Fairness Metrics

To measure fairness, one may compute **Jain's fairness index** $J = \frac{(\sum_j d_j)^2}{k \sum_j d_j^2}$, where d_j is the duration on worker j and $k = 3$ is the number of workers. A value closer to 1 indicates perfect fairness. Using the per-worker durations from Table 2, we obtain $J_{\text{ML}} \approx 0.91$ for the ML scheduler and $J_{\text{greedy}} \approx 0.84$. This quantifies the qualitative observation that the ML scheduler distributes load more evenly.

7 Discussion and Federated-Learning Implications

7.1 Mapping Scheduling to Federated Learning

In federated learning, a central server coordinates the training of a global model by selecting a subset of clients at each round and aggregating their updates. This selection problem resembles our **task → worker mapping**: each client (edge device) has its own compute capacity, network bandwidth and data size. The time to complete an FL round depends on the slowest client. Thus, **stragglers** – clients with low compute power or poor connectivity – can dominate the round latency. Resource-aware client selection can mitigate stragglers by excluding or weighting slow clients. The parallels are:

- **Cloudlet features ↔ Client metadata:** In scheduling, features include instructions, data size and priority; in FL, analogous metadata includes local dataset size, model size, compute capability and channel quality.
- **Worker MIPS/Bandwidth ↔ Client heterogeneity:** Edge devices in FL vary widely in processor speed and network throughput, affecting their contribution to the round.
- **Makespan ↔ Round duration:** The makespan of scheduled tasks corresponds to the time to complete an FL round; both are determined by the slowest participant.

Using a **classification model** to predict whether a client should participate in a given round could reduce the time-to-accuracy by avoiding slow devices and balancing the load across rounds. For instance, clients with poor connectivity could be assigned smaller local epochs or lower participation frequency. **FedAvg** already allows asynchronous updates by averaging fewer clients per round; however, the selection is often random. Recent surveys highlight open problems such as client selection and straggler mitigation[5]. Our findings suggest that **learning-based scheduling** can provide a practical approach: train a classifier to map client metadata to participation decisions, balancing the desire for data diversity with the need for timely rounds.

7.2 Systems Integration

Container orchestration and FL frameworks. The ML scheduler presented here could be integrated into a container orchestration framework. The Horizontal Pod Auto scaler could adjust the number of replicas based on CPU and memory metrics[1], while the **task-level scheduler** decides which pod or

edge node executes each task. **KubeEdge** provides a cloud-edge architecture with separate control plane and data plane; the classifier could run at the cloud side to dispatch tasks to edge nodes with available capacity[2]. For FL experiments, the **Flower** framework enables large-scale simulations with heterogeneous clients[3]. Incorporating our scheduler into Flower could allow experiments where client selection is informed by compute and network metadata, rather than being random.

Dynamic epochs and bandwidth-aware sampling. In FL, the number of local epochs (i.e., training steps performed at each client between communications) can be adapted based on resource availability. Clients with high compute capacity could perform more local epochs, while clients with slow processors or low battery could perform fewer. Additionally, the number of samples uploaded by each client could be proportional to its uplink bandwidth. These adaptations mirror the threshold rules learned by the decision-tree scheduler.

Topology-aware FL. Emerging research explores training over peer-to-peer topologies, such as k-regular or small-world overlays, to reduce communication bottlenecks. The three-worker network in our prototype can be viewed as a simple topology with heterogeneous links. Extending the scheduler to select connections and tasks based on network conditions could align with topology-aware federated learning.

7.3 Ethics and Reproducibility

The dataset used in this study is **synthetic**, generated from a template and does not contain real user data. This eliminates privacy concerns but limits the ecological validity of the results. All code is open source and designed to be reproducible; random seeds are fixed, and results are logged. Nevertheless, we caution that the results are derived from a small-scale simulation; applying the scheduler to real-world workloads requires careful validation.

8 Ablations and Sensitivity Analyses

Although the focus of this paper is on demonstrating the feasibility of an ML scheduler, we perform several preliminary ablation studies:

1. **Decision Tree vs KNN.** As already reported in Table 1, the decision-tree classifier achieves higher accuracy than KNN. When deployed in the scheduler, KNN tends to misclassify small cloudlets and overload W3, resulting in little improvement over the greedy baseline. The interpretability of the decision tree also aids debugging and rule inspection.
2. **Bandwidth tiers.** We vary the bandwidths between workers across three scenarios: **low**, **medium** and **high**. When all links have high bandwidth, transfer times are negligible and compute time dominates; the greedy baseline performs comparably to the ML scheduler. Conversely, when links are asymmetric (low between 1↔3), the ML scheduler provides more benefit by avoiding transfers over slow links. This indicates that the classifier's advantage increases with network heterogeneity.
3. **Worker MIPS variation.** We test scenarios where W1 and W2 have increased or decreased MIPS. The decision tree adapts by reassigning tasks accordingly. For instance, if W2's MIPS rises above W3's, the classifier will allocate more heavy tasks to W2. This demonstrates that the classifier's rules are based on features rather than hard-coded assignments.
4. **Synthetic dataset size.** Increasing the size of the synthetic dataset improves classifier accuracy up to a point (observed plateau near 85 %). Too small a dataset leads to overfitting, while too large a dataset provides diminishing returns. Future work could explore active learning to select informative samples.

9 Threats to Validity and Limitations

Several limitations must be acknowledged:

- **Synthetic data and small scale.** The dataset is synthetic and may not capture the full variability of real workloads. Real applications may exhibit different distributions of instructions and data sizes, correlation between features and dynamic network conditions. The results therefore

serve as a proof of concept rather than definitive evidence of superiority.

- **Assumed bandwidths and MIPS.** The worker capacities and link bandwidths are assumed and may not correspond to actual devices. Varying these parameters could change the relative performance of the scheduler and the baseline.
- **Single-node timing.** The reported durations are computed on a single machine that simulates workers and links. In real deployments, network latency, contention and scheduling overhead would introduce additional delays. Nonetheless, the relative comparison between strategies should carry over.
- **Model limitations.** The decision-tree classifier is simple and may not capture complex interactions between features. More sophisticated models (e.g., gradient-boosted trees or neural networks) could improve accuracy but at the cost of interpretability and deployment complexity.

10 Future Work

Building on this prototype, several research directions emerge:

1. **Integration with Flower and FL client selection.** Incorporate the scheduler into federated-learning frameworks (e.g., Flower) to perform resource-aware client selection. Evaluate the impact on time-to-accuracy and straggler mitigation using realistic FL workloads and heterogeneous devices.
2. **Topology-aware and decentralized scheduling.** Extend the scheduler to operate in peer-to-peer or hierarchical topologies, similar to the architecture of KubeEdge. Investigate scheduling policies that exploit multi-hop routing, network coding and cooperative scheduling.
3. **Adaptive models and online learning.** Rather than training a static classifier, employ online or reinforcement learning to adapt to changing workloads and network conditions. Such models could adjust thresholds on the fly and incorporate feedback from actual completion times.
4. **Energy and communication cost metrics.** Incorporate energy consumption and communication cost into the objective. Particularly in edge environments, battery life and data plans are critical constraints. The scheduler could trade off time against energy.
5. **Real testbed experiments.** Deploy the scheduler on a physical or emulated testbed with Raspberry Pi, Jetson Nano or smartphone devices connected via Wi-Fi and cellular links. Measure real execution times, network delays and energy consumption.
6. **Integration with Kubernetes HPA and KubeEdge.** Combine the ML scheduler with container orchestration. For example, the classifier could inform the HPA to spawn pods on specific nodes based on predicted workload distribution, while KubeEdge could provide a platform for remote deployment[2].

11 Conclusion

This work presents **Cloud-Assisted Resource Allocation Using Machine Learning**, a small-scale but complete prototype for cloud–edge task allocation. By generating synthetic data, training a decision-tree classifier and deploying it in a master scheduler, we demonstrate that machine learning can improve both makespan and load balance relative to a greedy baseline. The improvement is modest in absolute terms (~3.2 % makespan reduction on a fresh run) but significant for load balance, preventing overload of the fastest worker and distributing tasks across the available resources. The approach is reproducible, interpretable and readily integrable with container orchestration and federated-learning frameworks. Importantly, the problem and solution map naturally to **client selection in federated learning**: resource-aware scheduling of cloudlets parallels client–round allocation, and could reduce time-to-accuracy and straggler impact in FL. Future work will extend this prototype to larger testbeds, richer models and real federated-learning workloads.

12 Reproducibility Checklist

- **Operating System:** Linux (tested on Ubuntu 20.04).
- **Python version:** 3.11.
- **Packages:** *pandas* (≥ 1.5), *numpy* (≥ 1.23), *scikit-learn* (≥ 1.2), *joblib*, *matplotlib* for plotting.
- **Commit hash:** Use the latest commit in the repository *https://github.com/sadaqatdev/fyp_updated* when replicating experiments.
- **Random seeds:** Fix random seed (e.g., 42) in the seeder and training scripts for reproducible data and model splits.
- **Configuration files:** *seeder_data_template.csv* defines feature distributions; *config.json* (if present) contains bandwidth and MIPS settings.
- **Commands:** See §5.4 for the three commands required to generate data, train the model and run the scheduler.
- **Hardware:** Experiments can be replicated on a single machine; for distributed deployment, ensure consistent Python environments across nodes.

References

[1] H. B. McMahan, E. Moore, D. Ramage, S. Hampson and B. Agüera y Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," *Proc. AISTATS*, 2017. The authors propose federated learning, leaving training data on devices and using iterative model averaging. Their experiments demonstrate robustness to unbalanced and non-IID data and reduce communication rounds by 10–100× compared with synchronous stochastic gradient descent[4].

[2] P. Kairouz *et al.*, "Advances and Open Problems in Federated Learning," *Foundations and Trends in Machine Learning*, vol. 4, no. 1, pp. 1–210, 2021. This survey defines federated learning, summarises state-of-the-art algorithms, and highlights open challenges such as client selection, straggler mitigation and systems heterogeneity[5].

[3] D. J. Beutel, T. Topal, A. Mathur *et al.*, "Flower: A Friendly Federated Learning Research Framework," *arXiv preprint arXiv:2007.14390*, 2020. Flower provides a scalable research framework for federated learning, supporting heterogeneous devices and large-scale experiments; the authors report that Flower can conduct FL experiments with up to 15 million clients[3].

[4] Kubernetes Documentation, "Horizontal Pod Autoscaling," 2025. The Horizontal Pod Autoscaler is an API resource in the Kubernetes *autoscaling* group that scales the number of replicas of a deployment based on CPU, memory or custom metrics[1]. The stable *autoscaling/v2* API supports scaling on memory and custom metrics, and controllers adjust the replica count to maintain target utilisation[1].

[5] Kubernetes Blog, "KubeEdge, a Kubernetes Native Edge Computing Framework," 2019. KubeEdge provides a Kubernetes-compatible edge computing solution with separate cloud and edge core modules. The control plane resides in the cloud, while the edge can operate offline; the platform is lightweight, containerised and supports heterogeneous hardware[2]. KubeEdge enables orchestration and management of edge clusters akin to cloud Kubernetes[2].

[6] H. Wang, E. Deng, J. Li and C. Zhang, "Edge Computing Resource Scheduling Method Based on Container Elastic Scaling," *PeerJ Computer Science*, vol. 10, p. e2379, 2024. The authors design a TE-TCN load prediction model and use a reinforcement-learning based container scaling strategy to improve CPU utilisation and reduce response time in edge environments[8]. Their method leverages temporal convolutional networks and Markov decision processes to form a predictive container scaling strategy[8].

Additional citations for other referenced works.

[1] Horizontal Pod Autoscaling | Kubernetes https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[2] [6] KubeEdge, a Kubernetes Native Edge Computing Framework | Kubernetes
https://kubernetes.io/blog/2019/03/19/kubeedge-k8s-based-edge-intro/
[3] [2007.14390] Flower: A Friendly Federated Learning Research Framework
https://arxiv.org/abs/2007.14390
[4] [1602.05629] Communication-Efficient Learning of Deep Networks from Decentralized Data
https://arxiv.org/abs/1602.05629

[5] [1912.04977] Advances and Open Problems in Federated Learning
https://arxiv.org/abs/1912.04977
[7] [8] Edge computing resource scheduling method based on container elastic scaling - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC11623203/