Tail-Latency Aware Scheduler For Inference Workloads

Khelifa Saif eddine*, Miloud Bagaa*, Sihem Ouahouah§, Messaoud Ahmed Ouameur * and Adlen Ksentini¶
* Université du Québec à Trois-Rivières, Trois-Rivières, QC, Canada.
Emails: {Saif.Eddine.Khelifa, miloud.bagaa, messaoud.ahmed.ouameur}@uqtr.ca
, §Aalto University, Otakaari 24, 02150 Espoo FINLAND (e-mail: sihem.ouahouah@aalto.fi),
¶EURECOM, Campus SophiaTech, France (e-mail: adlen.ksentini@eurecom.fr)

***Abstract*—In recent years, AI inference has seen widespread adoption across fields like finance and healthcare, driving significant demand for high-performing applications. This demand brings about a complex relationship between inference application types, such as real-time applications, and their specific service level objectives (SLOs), like tail-latency. Tail-latency is a metric requiring a defined percentage of requests to meet a maximum response time, which is crucial for applications where delays can impact user experience or decision-making. This dependency creates a challenging research problem in scheduling inference workloads. The core question becomes: How can we deploy AI workloads in a way that minimizes SLO violations?**

Specifically, we worked on real-time applications that require tail-latency guarantees. To address this, we developed a tail-latency-aware scheduler designed for resource-constrained devices. Our scheduler employs advanced machine learning techniques to optimize task placement, aiming to minimize SLO violations and enhance performance for latency-sensitive applications. We have developed and integrated our custom scheduler into Kubernetes, which operates on a specially configured cluster designed to test its performance. This cluster features diverse computing capabilities, enabling a comprehensive evaluation of the scheduler's effectiveness. The experimental results highlight that our proposed scheduler outperforms the native Kubernetes scheduler in terms of efficiency.

***Index Terms*—Cloud-Edge-Computing-Continuum , Scheduling , Kubernetes**

I. Introduction

The rapid adoption of machine learning (ML) inference across diverse domains—such as healthcare [1] and finance [2]—has significantly increased the demand for robust inference services, particularly within cloud infrastructure. In fact, inference now constitutes more than 90% of AWS infrastructure costs [1], and platforms like Facebook perform tens of trillions of inferences daily [3] [4]. However, running these workloads solely in the cloud introduces latency and data privacy challenges, creating obstacles for industrial verticals that require high data rates and/or low latency. In contrast, deploying inference services closer to the user at the edge significantly enhances data privacy and improves the user experience. Thus, to address these issues, computation has been extended to edge devices and the Internet of Things (IoT), positioning inference tasks closer to end users. This shift minimizes communication delays and enhances data privacy. Consequently, deploying inference tasks has been extended across the entire continuum—from cloud to edge to IoT—requiring diverse Service Level Objectives (SLOs), such as latency, throughput, and energy efficiency at different levels of the continuum. These SLO requirements create a challenging scheduling problem. This problem has been tackled by the state-of-the-art [5]–[10] using various AI-based schedulers aimed at placing tasks on optimal nodes to minimize SLO violations. In contrast, and to the best of our knowledge, these studies have not fully considered an important aspect of inference workloads. Specifically, AI model characteristics which impact factors such as latency and energy.These factors are critical for real-time applications where responsiveness is essential. To overcome this gap, we have proposed a new framework that considers both AI model features and hardware profiles to account for the environment state. We hypothesize that this combination will lead to better scheduling decisions. Moreover, we prioritize tail-latency as a target metric for our AI-based scheduling decision, as it represents the "worst-case" performance for a high percentage of queries, typically at the 90th or 99th percentile [11].

In summary, our work focuses on real-time applications deployed on resource-constrained devices with heterogeneous computing characteristics (e.g., edge and IoT), emphasizing tail-latency as the key performance metric (KPM). The result is a tail-latency-aware scheduler that predicts tail-latency based on model and hardware characteristics, optimizing inference task placement in these complex environments. Our framework is structured as follows.

- **Framework Design:** We developed a two-stage framework with online and offline phases, supplemented by a monitoring system to collect real-time data on model performance and hardware status.
- **AI-Driven Scoring Function:** At the scheduling level, we introduce a scoring function that uses AI-based regression models to predict tail-latency, selecting the optimal computational node for each task based on predicted tail-latency performance.
- **Model Training and Fine-Tuning:** The regression models are trained on data collected by the monitoring system. We select the best-performing model and apply grid search to fine-tune it, increasing its accuracy.
- **Evaluation Against Kubernetes Scheduler:** Finally, we evaluate our scheduler compared to the native Kubernetes scheduler, analyzing the performance and efficiency gains in managing real-time inference workloads.

[1]https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf

The remainder of the paper is organized as follows. The next section presents the related work. Meanwhile, the problem is formulated in Section III. The framework design is presented in Section IV. Section V summarizes the experimental results, concludes our findings, and paves the way for future work.

II. Related works

Recent research has incorporated advanced AI techniques into scheduling algorithms to optimize the placement of microservice tasks on diverse computational nodes, using metrics such as latency and energy consumption to achieve optimal task distribution and meet Service Level Objectives (SLO) [12]. These efforts have evolved to include DNN-based workloads [10], given the increasing relevance of AI, although most studies mainly focus on the scheduling of DNN training workloads [12]. Few works in the literature specifically target the placement of inference workloads. For instance, authors in [5] explore scheduling ML inference across a cloud-to-things continuum, considering factors like CPU characteristics, energy efficiency, available cores, and network latency to meet task requirements. Similarly, the work [7] schedules inference across diverse edge processors: CPUs, GPUs, and DSPs, matching tasks to suitable hardware for latency and resource efficiency.

In the realm of cloud environments, [8] introduces DQoES, a system that allows users to set Quality of Experience (QoE) targets, such as response time for various applications, dynamically adjusting resources to meet these goals. These approaches tackle the problem of AI inference scheduling in diverse ways but often lack consideration of model-specific characteristics, such as parameters or FLOPs, which significantly impact energy, latency, and quality of experience (QoE). In contrast, some studies, such as [6], [9], [13], begin to address model information, though only at a generalized level using a binary vector to represent model types. For instance, [6] uses multi-agent reinforcement learning with a heterogeneous graph attention network to optimize task placement in edge clusters, balancing resource utilization and latency. [13] employs deep reinforcement learning to dynamically adapt job placement based on model type and resource needs. Meanwhile, the solution [9] predicts DNN response times across nodes by incorporating hardware configurations to determine if SLAs can be met under specific workloads.

Our work builds upon these efforts by introducing a tail-latency-aware scheduler within Kubernetes, designed for inference workloads in resource-constrained environments like edge and IoT. Our approach uniquely combines both hardware and model-specific information, capturing latent relationships to predict P90 or tail-latency accurately, which is critical for real-time applications. This predicted latency serves as a scoring factor for optimal node selection, improving the placement of the inference task.

III. Problem Formulation

To address the scheduling challenges discussed in previous sections, this section defines the real-time inference task scheduling and latency prediction problems on heterogeneous, resource-constrained systems.

A. Scheduling Problem Statement

In our scheduling problem, we have a computing cluster $N = \{n_1, n_2, \ldots, n_m\}$ that consists of multiple resource-constrained computing nodes, each characterized by distinct hardware properties, such as different CPU capabilities (e.g., clock speed and core count). Each node $n_i \in N$ represents these characteristics as a vector, where $n_i = [\text{clock speed}_i, \text{core count}_i]$. In addition, it is worth noting that these capabilities change over time due to resource contention, where multiple tasks compete for finite processing capabilities.

This diversity introduces computational heterogeneity across the cluster, creating a scheduling challenge in which the primary task of the scheduler is to place Deep Neural Network (DNN) inference tasks on suitable nodes to ensure minimal response time for queries, thereby meeting the tail-latency service level objective.

Moreover, in this scheduling problem, the DNN task set $X = \{x_1, x_2, \ldots, x_n\}$ is defined such that each task $x_i = \{M_i, F_i\}$ consists of the Model Size M_i and the Model FLOPs F_i. FLOPs F_i (Floating Point Operations Per Second) is a measure of computational workload commonly used to evaluate the efficiency of operations in neural networks. Model size M_i is determined by extracting key metrics (e.g., layer parameters, activation functions, kernel size) from the DNN model using tools like Torchinfo and converting the model to megabytes.

FLOPs are calculated at different stages. For instance, the FLOPs for a convolutional layer are calculated in equation 1:

$$\text{FLOPs} = 2 \times IC \times OC \times KH \times KW \times OH \times OW \quad (1)$$

where IC represents the input channels (i.e., the number of channels in the input feature map), and OC represents the output channels (i.e., the number of channels in the output feature map). KH and KW represent the kernel height and width, defining the dimensions of the convolution kernel or filter. OH and OW denote the output height and width, representing the dimensions of the output feature map after applying the convolution.

For fully connected (dense) layers, with N_{in} input neurons and N_{out} output neurons, the FLOPs are calculated in equation 2:

$$\text{FLOPs} = 2 \times N_{\text{in}} \times N_{\text{out}} \quad (2)$$

where N_{in} is the number of input neurons and N_{out} is the number of output neurons. The factor of 2 accounts for both multiplication and addition operations per neuron.

For activation functions (e.g., ReLU, Sigmoid, Tanh), which are typically applied element-wise, the FLOPs for an activation layer are simply the number of neurons in the layer, as shown in equation 3:

$$\text{FLOPs} = N_{\text{neurons}} \quad (3)$$

where N_{neurons} is the number of neurons in the layer.

Pooling layers, such as max pooling or average pooling, generally do not involve multiplications and are computationally less intensive, often counted as one operation per element in the pooling region. Thus, for a pooling layer with an output

feature map of size $H \times W$ and C channels, the FLOPs are calculated via equation 4:

$$\text{FLOPs} = H \times W \times C \tag{4}$$

The total FLOPs ($\text{FLOPs}_{\text{layer}}$) for a deep neural network are computed by summing the FLOPs across all layers, which are calculated using the previous equations 1, 2, 3, 4. This total is calculated via equation 5:

$$\text{Total FLOPs} = \sum_{\text{layer}=1}^{L} \text{FLOPs}_{\text{layer}} \tag{5}$$

This general approach ensures a comprehensive calculation of FLOPs for any CNN-based deep neural network model.

On the other hand, during the resource contention mentioned earlier, the hardware vector n_i keeps changing; thus, a real-time calculation is necessary. This is done using the following equations:

$$CPU_{eff} = CPU \times (1 - U_C) \tag{6}$$

$$FREQ_{eff} = FREQ \times (1 - U_F) \tag{7}$$

$$\text{Available Capacity} = \{CPU_{eff}, FREQ_{eff}\} \tag{8}$$

These equations describe how these changes are handled. Equation 6 calculates the effective core count CPU_{eff}, representing how many cores remain available each time a new task arrives. This is achieved by calculating U_C, which denotes the fraction of cores currently in use, and multiplying it by the total CPU cores CPU of the current computational node. For the frequency, we follow the same logic by calculating U_F, which denotes the fraction of frequency in use, and multiplying it by the node's frequency $FREQ$. Combining these results, the Available Capacity is calculated via equation 8.

Finally, the scheduling problem is based on a target P90 latency (p_i), where p_i is the latency threshold within which 90% of queries for task x_i must complete. To measure P90 latency, we gather a list of response times for all queries associated with a given task x_i on node n_j, represented as:

$$T_{i,j} = [t_1, t_2, \ldots, t_n] \tag{9}$$

where each entry in $T_{i,j}$ denotes the response time for a query q, with $q \in [1, N]$ for task x_i on node n_j. Using equation 9, the P90 tail-latency P_{90} is then calculated via equation 10:

$$P_{90} = T_{i,j}\left[\lceil 0.9 \times n \rceil\right] \tag{10}$$

This value represents the response time at the 90th percentile, meaning that 90% of the queries for task x_i complete within the P_{90} latency. The scheduling objective is to allocate each task x_i to a suitable node n_j such that its P90 latency constraint p_i is met, represented by equation 11:

$$\mathbb{P}(T_{i,j} \leq p_i) \approx 0.9 \tag{11}$$

Here, $T_{i,j}$ denotes the set of response times for task x_i on node n_j, ensuring that 90% of these queries meet the latency threshold p_i. This probabilistic approach enables predictable performance for the majority of queries, which is crucial for maintaining a stable real-time response. This predictable P90 latency is used as a target metric, while other equations for calculating model information and CPU contention will be used as inputs for the prediction problem statement, which will be explained in a subsequent section.

B. Prediction Problem Statement

In this context, the focus is on predicting the 90th percentile latency (P90) of DNN tasks to support tail-latency-aware scheduling across the cluster. The goal is to develop a prediction model capable of accurately estimating the P90 latency of tasks on various nodes within the cluster. To facilitate this, we define a dataset as:

$$D = \{N_D, X_D, Y_D\} \tag{12}$$

where $N_D \subset N$ is a subset of nodes, $X_D \subset X$ is a subset of DNN tasks, and $Y_D \subset Y$ represents the observed P90 latencies of tasks in X_D measured on each node in N_D. With this dataset, we establish a hybrid model- and hardware-conditioned prediction model as follows:

$$f(x_i, n_i; \gamma) : X \times N \to \mathbb{R} \tag{13}$$

where $x_i \in X$ is characterized by Model Size M_i and FLOPs F_i, calculated via the previous equation 5, representing the computational requirements. The computing node $n_i \in N$ is characterized by CPU features or available capacity $(\text{Clock}_{n_i}, \text{Cores}_{n_i})$ under contention, detailing clock speed and core count, which are calculated via equation 8. Here, γ represents the regressor $f(\cdot)$ parameters.

The objective is to learn a regressor $f(x_i, n_i; \gamma)$ that accurately predicts the P90 latency $y \in Y_D$ by minimizing the empirical loss L, such as the Residual Mean Squared Error (RMSE), between predicted and observed values from the dataset D:

$$\min_{\gamma} L(f(x_D, n_D; \gamma), y_D) \tag{14}$$

The regressor described in equation 14 relies on DNN model characteristics while taking underlying system hardware characteristics into account. These inputs outline the intersection between hardware features and model features, aiming to capture system stability through the prediction of P90 latency. This prediction enables the scheduler to allocate resources effectively, meeting latency constraints and ensuring reliable and efficient task processing across the cluster.

IV. Framework Design

This section introduces our framework design illustrated via the Figure 1 based on the de facto orchestration platform known as Kubernetes. The framework aims to solve the task placement problem within a Kubernetes cluster where the nodes are heterogeneous and resource-constrained. To tackle this challenge,We designed two phases: an offline phase, where a regression model is trained to accurately predict the P90 latency, and an online phase, where this model is integrated into the scheduling framework. This integration is exemplified by incorporating the model into custom scoring logic. The model predicts the P90 latency, which is then used to score nodes and assign tasks to the most suitable node. These phases are explained in detail in the following subsections.

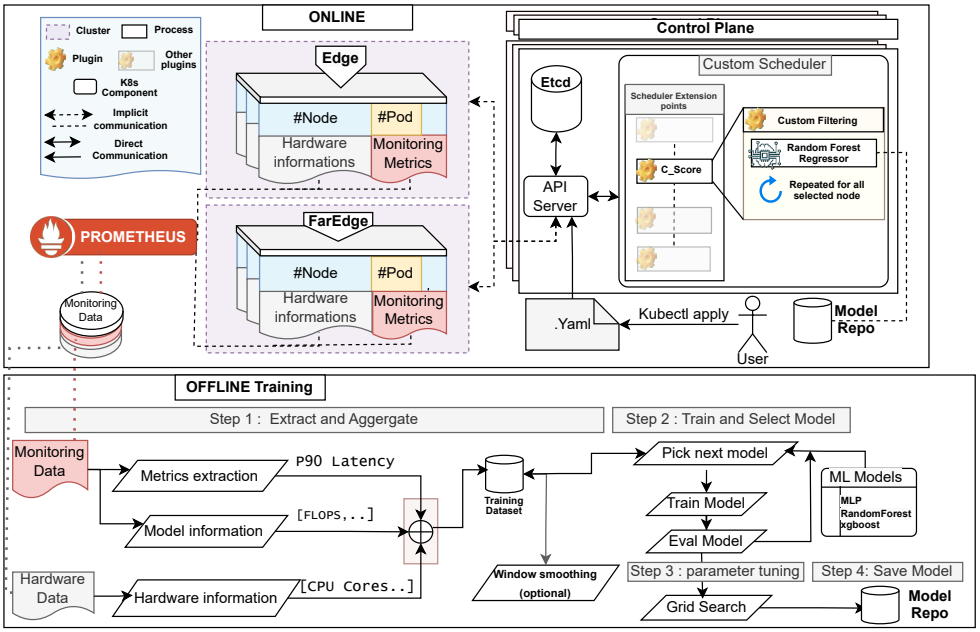


Fig. 1: Figure explains our conducted study with two phases online and offline

A. *Offline: Training*

The methodology for identifying the suitable regressor to serve as the core of our scoring function is illustrated in Figure 1. This approach centers on offline training to get the best model for predicting the P90 latency. Specifically, this is done first by using performance metrics and hardware specifications. The process begins with a data extraction and aggregation phase.

This phase involves extracting the monitoring data, which includes metrics such as P90 latency. This data is gathered from the system using Prometheus[2]. In addition to P90 latency, hardware data (CPU cores, clock speeds) are also extracted. Since hardware information is often textual, a preprocessing step then converts it into a structured format for further analysis.

Finally, the model information is extracted via a Python script that inspects the model using the aforementioned equations 1, 2, 3, 4, 5. These data points are aggregated based on common timestamps to ensure temporal alignment, forming our training dataset. The training dataset includes both the hardware information and the model information as inputs, while the P90 latency serves as the target. Moreover, before the training step, an optional window smoothing step is applied to reduce noise in the dataset, using the formula:

$$M_i = \frac{1}{\min(w,i)} \sum_{j=\max(1,i-w+1)}^{i} \log(S_j + 1), \qquad (15)$$

where M_i represents the smoothed value at the i-th time step, S_j is the original data point at time step j, and w is the size of the smoothing window. This log-based transformation minimizes the impact of outliers in time-series data, thereby stabilizing it for model training.

[2]https://prometheus.io/

Following that, a set of machine learning models are trained on the resulting training dataset. The models used in this phase include Multi-Layer Perceptrons (MLP), Random Forest, and XGBoost, each evaluated based on Root Mean Square Error (RMSE)—a metric that measures the average magnitude of the errors between predicted and actual values, giving greater weight to larger errors. Once the models are trained, the best-performing model is selected, followed by a parameter-tuning phase using grid search [14] to optimize hyperparameters (e.g., learning rate). This fine-tuning step ensures optimal prediction accuracy.

The final model is then saved to a model repository, where it can be accessed and deployed for real-time inference tasks. Furthermore, saved models are periodically updated with new cluster performance data, facilitating continuous improvements in predictive performance.

B. *Online : Scheduling*

In the online scheduling phase, as illustrated in Figure 1, the process begins with the user creating a deployment YAML file to deploy their application. This YAML file contains essential information for the inference task, such as the URL for model storage. The scheduler uses this URL to extract model-specific details, such as model size and FLOPs, as explained in the offline stage. Hardware information is provided by an agent deployed using a DaemonSet, which retrieves this information from the cluster nodes and saves it into a ConfigMap, stored at the etcd level for access during the scheduling phase.

The main objective of the scheduler is to determine which node r will be assigned to a task x. To do so, it goes through several phases, starting with filtering, where nodes that cannot support the job are excluded using the pre-defined default scheduling algorithm.

As described in Algorithm 1, the scoring function iterates over the nodes selected in the filtering phase. Each selected

node is evaluated using a pre-trained regressor from the offline phase, accepting as input the node vector V_r, which includes CPU clock speed and CPU cores, as well as the task vector V_x, containing model size and model FLOPs. The node with the lowest predicted tail-latency, denoted as P90, is then assigned to the task x. To complete the process, a monitoring phase is launched to gather metrics on the deployed task using Prometheus. These metrics are stored in a database for offline training to refine the regressor further, aiming to minimize Service Level Objective (SLO) violations as much as possible.

Algorithm 1 Latency-Aware Scheduler

Input: Deployment manifest, task characteristics V_x, hardware characteristics V_r, latency regressor f , tail-latency threshold : $P90_T$

Output: Job manifest with assigned node and estimated latency

▷ *PL: is the Predicted latency based on task and hardware characteristics*
Initialize CandidateNodes $\leftarrow \emptyset$
for each node n_i in available nodes **do**
    PL$= f(V_x, V_r)$
    Add (n_i, PL) to CandidateNodes
(SelectedNode) $\leftarrow \arg\min_{n_i \in \text{CandidateNodes}} \text{PL}(n_i, V_x, V_r)$
Update job manifest with SelectedNode.
Submit the updated manifest and monitor job execution

V. Preliminary results

During our preliminary results, we used the setup presented in Table I, and the candidate models are listed in Table II with their respective model sizes and FLOPs. We chose different models for different tasks, such as ResNet-50 [15], which is a deep convolutional neural network designed for image classification, and RetinaNet [16], which is a one-stage object detection model that identifies and localizes objects within images. We used these models along with publicly available datasets: ImageNet [17] and OpenImages [18].

Our experiment used the open-source MLPerf Inference Benchmark [11], integrated with Kubernetes. The MLPerf Benchmark provides a standardized way to evaluate model performance in various scenarios, such as single-stream, multi-stream, server, and offline modes. Prometheus calculated the P90 latency during the benchmark. We opted for the single-stream scenario, where queries are sent sequentially, as it represents our case for real-time applications [11].

TABLE I: Cluster setup

Node Name	CPU Cores	CPU Clock Speed	RAM
controller	16	2.30 GHz	16 GiB
node1	4	2.30 GHz	6 GiB
node2	8	2.30 GHz	8 GiB
node3	12	2.30 GHz	16 GiB
node4	16	2.30 GHz	12 GiB

TABLE II: Model Sizes and FLOPs

Model	Size (MB)	FLOPs
ResNet-50	97.66 MB	4 billion FLOPs
RetinaNet	129.7 MB	3.8 bilion FLOPs

We evaluate the effectiveness of our custom scheduler, which uses a Random Forest Regressor, XGBoost Regressor, and MLP Regressor to predict tail-latency for scheduling tasks in a multi-node environment. The evaluation scenario, referred to as the immediate scheduling scenario [19], simulates real-world conditions where inference tasks arrive sequentially, one after another, and are processed immediately without delay. This immediate scheduling scenario demonstrates the need to handle single-query, single-stream workloads typical of real-time applications. In such scenarios, tasks are processed as soon as they arrive, requiring highly accurate latency predictions to optimize scheduling decisions. Following this rationale, a total of 10 pods is deployed across the nodes. Each task consists of a single inference query with a batch size of 1, which arrives and is scheduled sequentially. Once all pods are assigned to nodes, we initiate query execution in a single stream, processing a total of 1024 queries per pod—the recommended number of queries from the MLPerf Inference benchmark [11].

During the scheduling process, regression models predict P90 latency, which is eventually used by the scheduler to assess the placement choice of pods. This prediction is based on input features such as core count, frequency, and model size and FLOPs. The models' performance is assessed during offline (training) and online (inference) phases using the Root Mean Squared Logarithmic Error (RMSLE) and Root Mean Squared Error (RMSE), as presented in Table III.

TABLE III: Regressor performance metrics

Model	RMSLE	RMSE
Random Forest Regressor	0.206	1.833
XGBoost Regressor	0.202	1.784
MLP Regressor	0.281	2.326

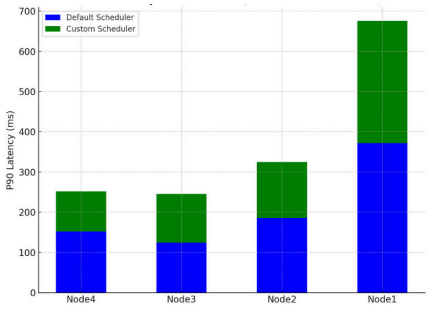
The relatively low RMSE values reflect the nature of our single-query, single-stream workload. Given the limited range of latency values, which are inherently small due to the single-sample batch size (fixed at 1 sample per query), the overall prediction task remains less complex. The performance of the models is further influenced by the relatively small size of the machine learning models used—such as ResNet-50 and RetinaNet—which simplifies the prediction process.

Using the models in Table II, we proceeded to test our solution. The metrics were collected using Prometheus[3], applying the equation 16:

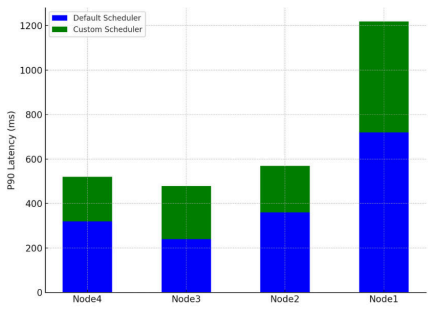
$$\text{AVG\_NP90} = \frac{\sum (\text{P90 Latencies for all pods on the node})}{\text{Number of pods on the node}} \tag{16}$$

The **AVG_NP90** equation calculates the average P90 latency across all pods deployed on a particular node. By summing the P90 latencies of each pod and dividing by the number of pods on that node, we obtain a representative latency value that accounts for the resource contention and workload assigned to that node. This calculation is executed after every 1024 queries. For the first 1024 queries, we plot **AVG_NP90**, as depicted in Figure 2, which is a stacked bar chart comparing the default Kubernetes scheduler and

[3]https://prometheus.io/



(a) AVG NP90 for ResNet-50



(b) AVG NP90 for RetinaNet

Fig. 2: Figure shows AVG NP90 for each candidate model

our custom scheduler.Our custom scheduler uses predicted latencies to make more informed scheduling decisions, reducing **AVG_NP90** latency across the cluster. Moreover, in the plot, the default Kubernetes scheduler (represented by blue bars) does not account for predicted tail-latency or resource contention, leading to suboptimal pod distribution and higher, more variable P90 latencies. This inefficiency is particularly pronounced on nodes with fewer cores and lower frequencies, where contention further exacerbates latency. In contrast, the green bars illustrate the significant improvements achieved with our custom scheduler. By using regression models to predict P90 latency, our scheduler makes more efficient task allocations, reducing both mean and variance of latency. This optimization is especially beneficial for nodes with limited resources, where accurate latency predictions are crucial.

VI. Conclusion

Our experimental results highlight the effectiveness of using predictive models—particularly Random Forest and XGBoost Regressors—for inference scheduling, leading to lower average P90 latency across deployed inference tasks. This validation underscores the importance of hardware-aware and model-aware scheduling mechanisms in modern inference-serving systems. However, this study primarily focused on single-stream workloads, providing a foundation for future research. In subsequent work, we aim to extend this scheduler to support multi-stream scenarios, varying batch sizes, and dynamic scheduling strategies, further enhancing adaptability to real-world cloud-edge computing environments. By bridging the gap between model characteristics, hardware constraints, and inference scheduling, our work lays the groundwork for more efficient, scalable, and latency-aware scheduling frameworks, ensuring optimal performance for real-time AI applications.

References

[1] H. Abdel-Jaber, D. Devassy, A. Al Salam, L. Hidaytallah, and M. EL-Amir, "A review of deep learning algorithms and their applications in healthcare," *Algorithms*, vol. 15, no. 2, p. 71, 2022.

[2] A. Arévalo, J. Niño, G. Hernández, and J. Sandoval, "High-frequency trading strategy based on deep neural networks," in *Intelligent Computing Methodologies*, D.-S. Huang, K. Han, and A. Hussain, Eds. Cham: Springer International Publishing, 2016, pp. 424–436.

[3] U. Gupta, C.-J. Wu, X. Wang, and e. a. Naumov, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.

[4] K. Hazelwood, S. Bird, and e. a. Brooks, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[5] I. Syrigos, D. Kefalas, N. Makris, and T. Korakis, "Eelas: Energy efficient and latency aware scheduling of cloud-native ml workloads," in *2023 15th International Conference on COMmunication Systems and NETworkS (COMSNETS)*, 2023, pp. 819–824.

[6] Y. Li and X. e. a. Zhang, " Task Placement and Resource Allocation for Edge Machine Learning: A GNN-Based Multi-Agent Reinforcement Learning Paradigm ," *IEEE Transactions on Parallel & Distributed Systems*, vol. 34, no. 12, pp. 3073–3089, Dec. 2023. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TPDS.2023.3313779

[7] W. Seo, S. Cha, Y. Kim, J. Huh, and J. Park, "Slo-aware inference scheduler for heterogeneous processors in edge platforms," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, Jul. 2021. [Online]. Available: https://doi.org/10.1145/3460352

[8] Y. Mao, W. Yan, Y. Song, Y. Zeng, M. Chen, L. Cheng, and Q. Liu, "Differentiate quality of experience scheduling for deep learning inferences with docker containers in the cloud," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1667–1677, 2023.

[9] S. Shah, Y. Amannejad, and D. Krishnamurthy, "Predicting the performance of dnns to support efficient resource allocation," in *2023 19th International Conference on Network and Service Management (CNSM)*, 2023, pp. 1–7.

[10] Z. Ye and e. a. Gao, "Deep learning workload scheduling in gpu datacenters: A survey," New York, NY, USA, Jan. 2024. [Online]. Available: https://doi.org/10.1145/3638757

[11] V. J. Reddi, C. Cheng, D. Kanter, and e. a. Mattson, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[12] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," New York, NY, USA, Dec. 2022. [Online]. Available: https://doi.org/10.1145/3539606

[13] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters with heterogeneous workloads," *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, pp. 634–647, 2023.

[14] S. B H and G. Dagnew, "Grid search-based hyperparameter tuning and classification of microarray cancer data," 02 2019, pp. 1–8.

[15] N. Saleem, J. Gao, M. Irfan, E. Verdú, and J. Fuente, "E2e-v2sresnet: Deep residual convolutional neural networks for end-to-end video driven speech synthesis," *Image and Vision Computing*, vol. 119, p. 104389, 01 2022.

[16] Y. Li and F. Ren, "Light-weight retinanet for object detection," *arXiv preprint arXiv:1905.10011*, 2019.

[17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2009, pp. 248–255.

[18] A. Kuznetsova and H. R. et al, "The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale," *International Journal of Computer Vision*, 2020.

[19] R. Gu, Y. Chen, and L. et al, "Liquid: Intelligent resource estimation and network-efficient scheduling for deep learning jobs on distributed gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2808–2820, 2022.