An Empirical Design and Implementation of Job Scheduling Enhancement for Kubernetes Clusters

Van-Binh Duong
School of Electronic Engineering, Soongsil University, Seoul, Korea
binhdv@dcn.ssu.ac.kr

Duong Phung-Ha
School of Electronic Engineering, Soongsil University, Seoul, Korea
phunghaduong@dcn.ssu.ac.kr

Jangwon Lee
School of Electronic Engineering, Soongsil University, Seoul, Korea
jangwon.lee@dcn.ssu.ac.kr

Younghan Kim
School of Electronic Engineering, Soongsil University, Seoul, Korea
younghak@ssu.ac.kr

Abstract—**Batch job scheduling is a popular method in scheduling topics achieving considerable results. However, it is novel to bring those achievements to the cloud where problems of environments would limit algorithms. Although some projects have been introduced recently with benefits directly targeting machine learning jobs in cloud-native environments, there are gaps to be fulfilled. Consequently, this paper proposes an empirical design and implementation of deadline-aware enhancement of job scheduling for cloud-native environments. The proposal's target is to automatically monitor, provision, and maintain machine learning jobs for batch job scheduling in cloud-native environments. The performance evaluation shows that the proposal has succeeded in reducing the average response time of machine learning jobs scheduled by different algorithms.**

Index Terms—**deadline-aware, kubernetes, batch job scheduling, machine learning job, cloud computing**

I. Introduction

Batch job scheduling is a method of collecting a bunch of jobs to execute at a time. Those jobs would wait in a queue to be scheduled in a resource. There are two important parameters in the batch job scheduling, namely, the window time that is the amount of time the scheduler waits for the incoming jobs, and the maximum number of jobs that are contained in a collection. Those parameters could be manually fixed or dynamically changed based on the application design. One of use cases of batch job scheduling is in machine learning systems where many different proposals were proposed to improve the job completion time, as well as the accuracy of training models. However, batch job scheduling is the pre-scheduled stage where the incoming jobs are collected before being scheduled in a resource. Therefore, to optimize resource usage and management, scheduler algorithms have to be used reasonably in the scheduled stage. Especially, in cloud-native environments, resource management affects different aspects (i.e., performance, functionality, and cost) that build a system. Inefficient resource management has a direct negative effect on a system's performance and cost. It leads to indirect effects on the system's functionality. To this end, an efficient cloud system for machine learning jobs should care about both aforementioned stages.

This paper aims to build up a scheduling system for machine learning jobs in cloud-native clusters in which incoming jobs are analyzed and observed to define a suitable queue wait time (i.e., the deadline period - a job has to wait before being scheduled) and different scheduling algorithms could be used to schedule submitted jobs in a resource to efficiently make use of resources. This paper proposes a deadline-aware batch scheduler focusing on cloud-native clusters. It uses jobs' resource characteristics and historical observations from submitted jobs to calculate the suitable queue wait time for incoming jobs in the pre-scheduled stage and in the scheduled stage, different scheduling algorithms could be interchanged based on resource management strategy. The approach is to ensure Service Level Agreement (SLA) satisfaction (which is the agreement on a job's runtime between the user and the system). Furthermore, a sophisticated procedure is also developed to intervene in tasks of scheduled jobs to reach optimized performance and user SLA satisfaction while jobs are running in resources. The details of the proposal are discussed in the rest of the paper. The sequel of the paper is organized as follows: Section II provides a brief review of related works. The proposal of this paper is contained in Section III. Section IV describes experimental setup and evaluation results. Finally, some conclusion remarks are given in Section V.

II. Literature Review

In the paper [1], an introduced SLA-aware scheduling algorithm that utilized domain-specific properties of machine learning inference to batch input requests intelligently with dynamic batch size and timeout by utilizing a batch state table to track the batching status among the executing inputs. Research [2] proposed admission control and resource scheduling algorithms applying the data splitting-based method to give parallel processes on the split data sets as optimization solutions for Analytics-as-a-Service platforms. The paper [3] presented a deep learning-driven machine learning cluster scheduler to place training jobs in a resource server that minimizes interference and maximizes performance. In [4], a Shortest Processing time First scheduler combined with a task classification (i.e. small or large task) was proposed to generate scheduling improvements. In research [5], the average time was simply calculated based on the prior tasks that a user submitted. A model prediction was proposed to predict the wait time for a task queue and then classify jobs that are similar using past queue data in [6]. The paper [7] depicted queue congestion as an estimate of the state based on the degree of congestion for the queue wait time predicted at time *t* and then utilized a Markov model to anticipate the queue wait times at time *t+1*. Research [8] explored the use of machine learning to predict the queue wait time for jobs using real-world data to train prediction models.

Previous research concentrated on the pre-scheduled stage where the choice of algorithms for the scheduled stage is limited. Research on queue wait time prediction did not contain aspects of machine learning jobs on the cloud. Because of the relationship between tasks in a job, the machine learning jobs in a cloud cluster cannot be served at once which is different from pure cloud computing jobs. Therefore, each task needs to wait for their turn to be processed which brings a problem of scalability. Moreover, using a prediction model to predict the queue wait time requires large-scale data sets and the prediction phase also consumes an amount of time when scheduling a job. Consequently, a solution that could handle both stages and provide efficiency for machine learning jobs in cloud-native clusters is a necessary need. Besides, managing the jobs after being scheduled is also vital. This implies that jobs waiting in queue need to have a deadline to prevent resource starvation and running jobs also need a deadline to prevent resource over-occupation. To achieve those goals, this paper proposes a design, Deadline-aware Job Scheduling, which proactively provides a job deadline queue time based on jobs' characteristics to improve scheduling algorithms and ensure job completion satisfaction. The design provides flexibility in choosing scheduling algorithms based on resource status. As a result, this paper has following key contributions:

- A procedure to analyze job characteristics with historical data in order to define the queue wait time of a job to guarantee the user SLA and enhance scheduling algorithms.
- A cloud-based Deadline-aware Job Scheduling is developed to automatically manage and maintain machine learning jobs in a Kubernetes cluster.
- An active procedure to modify running heavy tasks of a job to reduce SLA degradation.

III. Proposal: Deadline-aware Job Scheduling

A. Problem Formulation

A machine learning job contains different tasks with different resource requirements. So the execution time (or runtime) of a job is a sum of tasks' execution time (Equation 1). The target of the problem is to satisfy the Expression 2 where T_ω is the queue wait time of a job, T_ϵ is the amount of time needed to execute the job and β is the deviation of the system. The β is calculated as 5% out of SLA_{target} which is the acceptable margin of error.

TABLE I: List of acronyms and terminologies.

β	Deviation of the system
CPU	Central Processing Unit
CRD	Custom Resource Definitions
DRF	Dominant Resource Fairness
GPU	Graphics Processing Unit
H	The heaviest task
Job CM	Job Controller Manager
K	A Kubernetes cluster
P	A profile
P_{GPU}	A profile containing GPU information
RAM	Random Access Memory
SLA	Service Level Agreement
S_{RAM}	Sum of RAM requirements
S_{CPU}	Sum of CPU requirements
S_{GPU}	Sum of GPU requirements
T_ω	Queue wait time
T_ϵ	Execution time

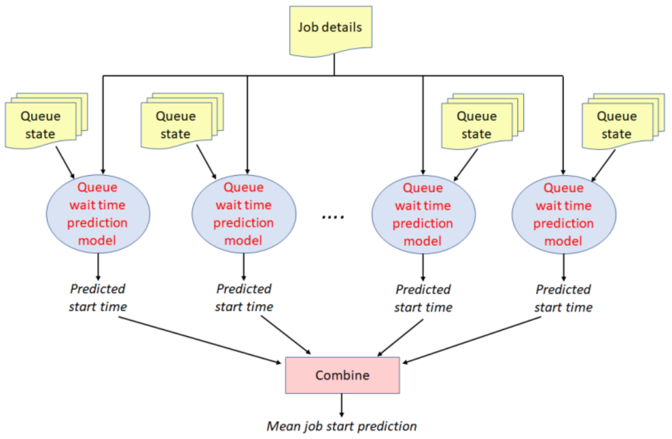
Job details
Queue state
Queue state
Queue state
Queue state
Queue wait time prediction model
Queue wait time prediction model
Queue wait time prediction model
Queue wait time prediction model
....
Predicted start time
Predicted start time
Predicted start time
Predicted start time
Combine
Mean job start prediction

Fig. 1: Illustration of stochastic approach operating over distinct randomly generated queue states using queue wait time prediction models [8].

$$T_\epsilon = \sum_{i=1}^{N} t_\epsilon \tag{1}$$

$$SLA_{target} + \beta \geq T_\omega + T_\epsilon \tag{2}$$

The proposed Diagnoser uses Algorithm 1 to calculate the queue wait time for each job by making use of past job observations to practically define the T_ω of each job. The Dominant Resource Fairness (DRF) [9] is used for determining job resource dependence to efficiently decide the most suitable queue wait time of a job.

The Algorithm 1 determines the dominant resource of each job and then uses that information to search T_ϵ from a profile. For instance, if the job's dominant resource is *CPU*, then Diagnoser finds the observation from the profile having a matched *CPU* (or the nearest higher one). For the *RAM*, the

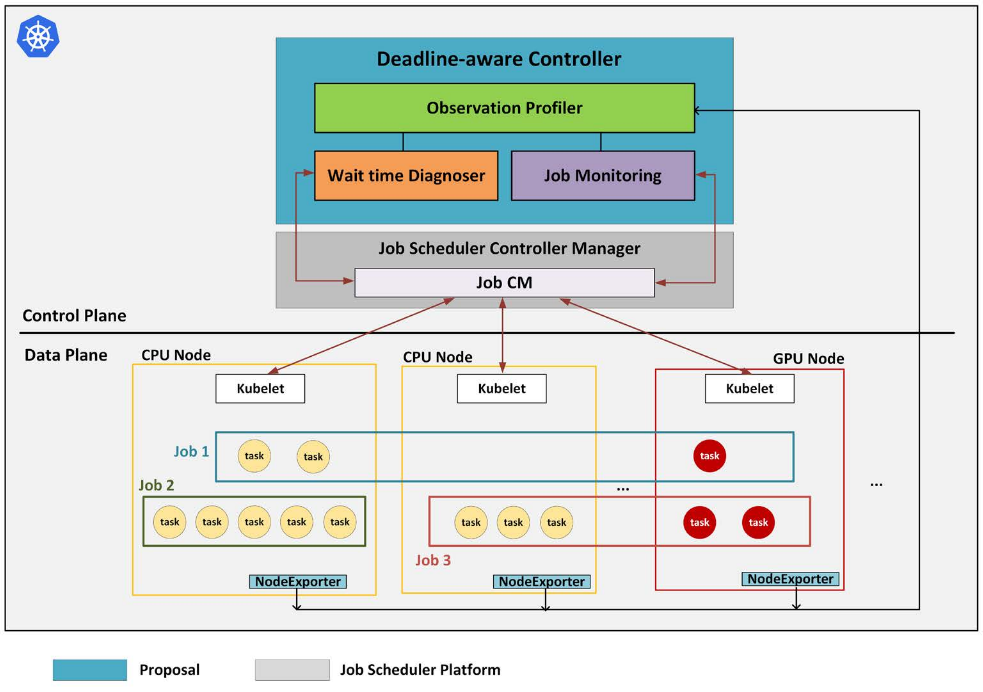

Fig. 2: Deadline-aware Job Scheduling Architecture.

Algorithm 1 Job Deadline Endowment Algorithm

```
1:  Cluster $K$, Queue wait time $T_\omega$, Execution time $T_\epsilon$
2:  $P \leftarrow Profile\_table(CPU, RAM, execution)$
3:  $P_{GPU} \leftarrow Profile\_table(CPU, GPU, RAM, execution)$

4:  $\tau_i \leftarrow task_i(CPU_i, RAM_i)$
5:  $J \leftarrow [\tau_i, ..., \tau_n]$
6:  $S_{CPU} = \sum_{i=1}^{n} \tau_i.getCPU()$
7:  $S_{RAM} = \sum_{i=1}^{n} \tau_i.getRAM()$
8:  **if** $S_{CPU} \div K.getCPU() > S_{RAM} \div K.getRAM()$ **then**
9:      $T_\epsilon = search\{P, DRF(CPU)\}$
10:     $T_\omega = SLA_{target} - T_\epsilon + \beta$
11: **else if** $S_{CPU} \div K.getCPU() < S_{RAM} \div K.getRAM()$ **then**
12:     $T_\epsilon = search\{P, DRF(RAM)\}$
13:     $T_\omega = SLA_{target} - T_\epsilon + \beta$
14: **else**
15:     $T_\epsilon = search\{P_{GPU}, DRF(GPU)\}$
16:     $T_\omega = SLA_{target} - T_\epsilon + \beta$
17: **end if**
```

matched one (or the nearest higher one) is also selected. This makes sure that the prior resource provided is enough for the job. When T_ω is a negative value, the controller automatically alerts the user to change to a suitable SLA_{target}.

B. Deadline-aware Job Scheduling Architecture

The proposal not only aims to obtain the target SLA but it is also developed to holistically provide a deadline-aware machine learning jobs scheduling and management system. Figure

Algorithm 2 Job Execution Deadline Management Algorithm

```
1:  Heaviest_task $H$, Ready_tasks $\Phi$
2:  $\Gamma \leftarrow running\_tasks[task_i, ..., task_n]$
3:  $Q \leftarrow queueing\_tasks[task_j, ..., task_m]$
4:  **while** $Q$ not empty **do**
5:      $H = search\{\Gamma\}$
6:      **for** $task_j$ in $Q$ **do**
7:          **if** $(\sum \Phi \oplus task_j.getT_\omega) \leq H.getT_\omega$ **then**
8:              $\Phi.append(task_j)$
9:          **end if**
10:     **end for**
11:     $status\_update\{H, RUN \rightarrow STOP \rightarrow QUEUE\}$
12:     **for** $task_j$ in $\Phi$ **do**
13:         $status\_update\{task_j, QUEUE \rightarrow RUN\}$
14:     **end for**
15: **end while**
```

2 describes the Deadline-aware Job Scheduling architecture, the Deadline-aware Controller consists of three components, namely, Observation Profiler, Wait time Diagnoser, and Job Monitoring.

The Controller receives requests and constructs job profiles using diagnoses from the Diagnoser. It also supports collecting job observations from the cluster and monitors the scheduling status. The Profiler acquires observation collections to process and store that information as a resource for the Diagnoser. The Job Monitoring uses Algorithm 2 to monitor the running jobs at the task level which puts the deadline on them. There is a trade-off of dropping SLA-excessed jobs to keep the SLA

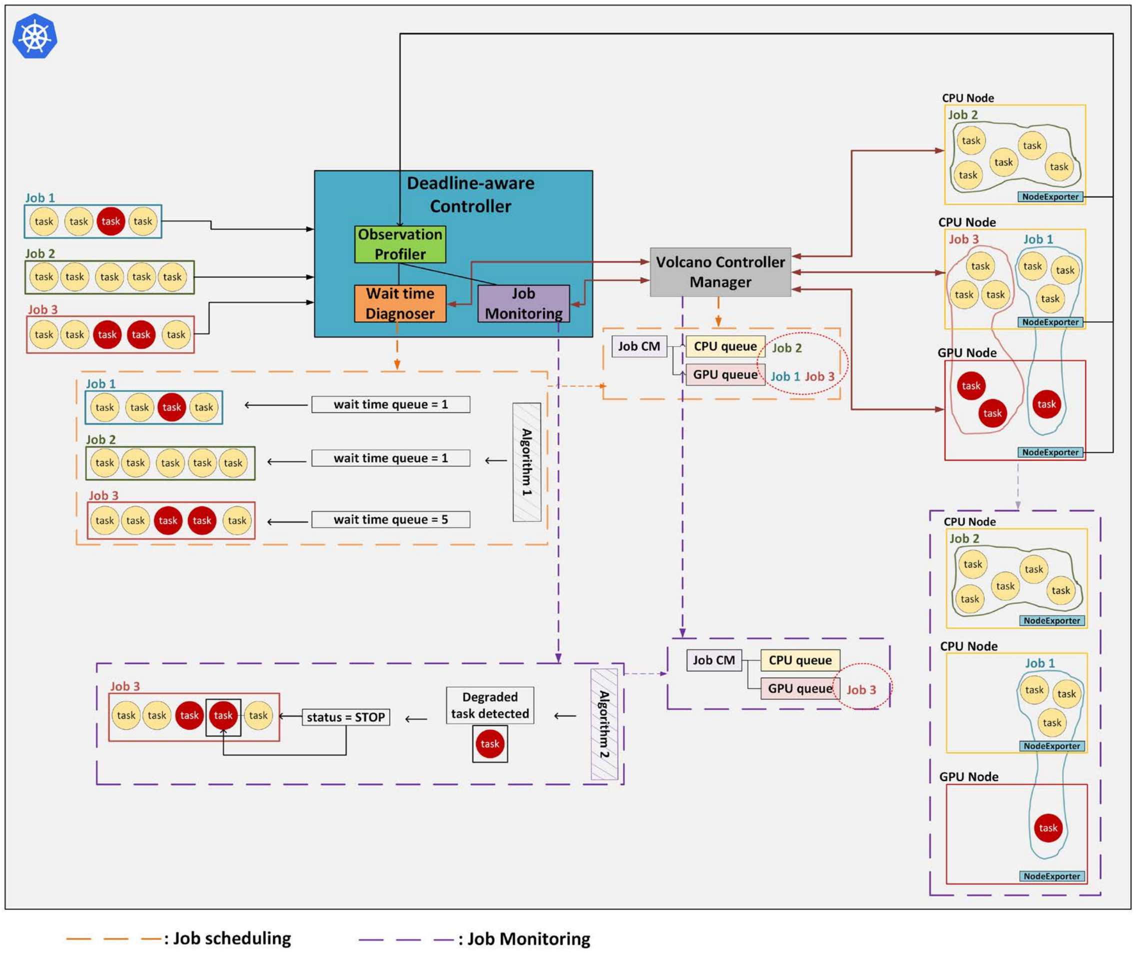

Fig. 3: An example of Job scheduling and monitoring management workflow where Volcano is used as a job scheduler.

of other jobs. The Algorithm finds the running heaviest task H that is over the assigned execution time and then replaces it with other lighter tasks and reschedules the related job of the task H. With this mechanism, the proposal maximizes the number of SLA-satisfied jobs.

In comparison with using prediction models, as in Figure 1, they used different queue states and job details to predict a hundred wait time predictions using a trained model and then took the average wait time predictions to obtain the final result. Because queue state data is mostly affected by resource utilization, it could remain unchanged for a long time if resources are being occupied by a degraded task/job. This leads to biased results from the trained model that are not able to reflect the actual status of resource utilization. Meanwhile, in Figure 3, the Wait time Diagnoser uses historical and real-time job observations from resource nodes with Algorithm 1 to quickly assign/modify a wait time queue for a job which is faster than waiting for a combined result from a hundred predictions. Simultaneously, the Job Monitoring monitors the status of running jobs using real-time job observations with Algorithm 2 to detect degraded tasks, which reduces the interference in machine learning jobs. As a result, the proposal could integrate with different cloud schedulers to accelerate them in aspects of resource utilization, jobs' runtime, and jobs' lifecycle management.

IV. Performance Evaluation

A. Experimental Environment

In terms of implementation, the Operator pattern (software extensions to Kubernetes) [10] approach is used to implement the proposed design. The Controller is built using Custom Resource Definitions (CRDs) [11] to apply algorithms to Kubernetes clusters. After the job is scheduled, the Controller continues to observe the state of the pods in which the job's tasks are scheduled to maintain the job life cycle. Meanwhile, the Profiler is programmed in Python language to effectively process data collected from the NodeExporter. The NodeExporter and Scheduler are built based on Prometheus [12] and Volcano [13] (which is a Kubernetes native batch scheduling system), respectively. The Controller passes submitted jobs with deadline information to Volcano to accelerate deploying jobs in the cluster.

The proposal was implemented on one Kubernetes cluster. The cluster was composed of 3 nodes (1 master and 2 workers).

```
apiVersion: sla-operator.dcn.com/v1alpha1
kind: Slaml
metadata:
  name: slaml-scheduling
  namespace: machine
spec:
  isSla: true
  slaTarget: 50
  name: pytorch-ml-job
  tasks: [
  {
    taskName: "ml-task-1",
    type: data,
    containerImage: "data-preprocessing",
    containerRegistry: "ddocker122",
    containerTag: "latest",
    containerReplicas: 1,
    CPU: "2m",
    memory: "2Mi"
  },
  {
    taskName: "ml-task-2",
    type: feature,
    containerImage: "feature-extraction",
    containerRegistry: "ddocker122",
    containerTag: "latest",
    containerReplicas: 3,
    CPU: "3m",
    memory: "2Mi"
  },
  {
    taskName: "ml-task-3",
    type: model,
    containerImage: "model-training",
    containerRegistry: "ddocker122",
    containerTag: "latest",
    containerReplicas: 3,
    modelParams: [
    {"batch-size": 64, "epochs": 14, "lr": 0.0001},
    {"batch-size": 128, "epochs": 56, "lr": 0.0001},
    {"batch-size": 256, "epochs": 14, "lr": 0.0001}]
    GPU: "4",
    memory: "4Mi"
  }]
```

Listing 1: An example of job request to Deadline-aware Cluster.

Each node was hosted by a server with Ubuntu 20.04.5 LTS OS and 5.4.0-153-generic kernel. The master node was characterized by 8 vCPUs and 33 Gb memory. The worker nodes, on which experimental jobs were scheduled, had the configuration of 4 vCPUs and 16 Gb memory each. The cluster was configured with Kubernetes version v1.23.6, Volcano version v1.8.0, Python version 3.8, Prometheus version v0.12.0, and Kubebuilder [14] (i.e, CRD development tool) version v3.11.1. The Profiler was set up with the Google Cluster Workload Traces [15] data set as historical observations. The data set

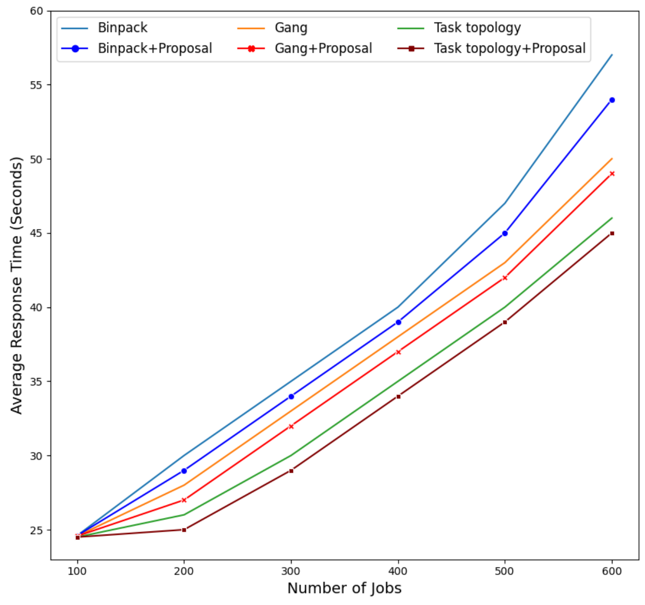

Fig. 4: Deadline-aware jobs scheduled by different algorithms.

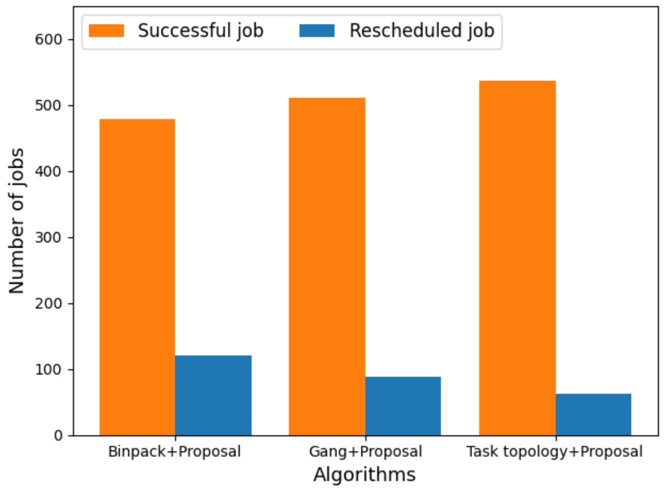

Fig. 5: Job schedulings.

was processed to get the required fields for experiments as follows: the required *RAM* and *CPU* values from the task were extracted, and the execution time of each task was calculated by subtracting the completion time from the starting time of each task.

B. Scenarios Description

The setup experiment was to show the guaranteed user SLA of a job in an environment with a shortage of resources. A number of dummy jobs were kept running on the cluster to consume around 90% resources. The scheduler algorithms such as Gang, Binpack, and Task-topology were tested in the cluster to assess how the Deadline-aware Operator accelerates scheduling algorithms in Volcano.

The MNIST [16] data set was used to build up a machine-learning problem using Pytorch. The job flow was split into different tasks, namely, data preprocessing, feature extraction, and model training. Each task had different resource requirements that resulted in different SLAs. A job request can be submitted using a *yaml* file as shown in Listing 1.

C. Experimental Results Analysis

The Average Response Time (ART) was used as the performance metric which is the average time between the request from end-users for the job execution to the result response as follows.

$$ART = \frac{\sum_{i=1}^{N} response_i}{N} \tag{3}$$

Figure 4 shows that the proposal reduces the ART of scheduling algorithms in Volcano. It queues incoming jobs in a manageable manner with the Wait time Controller. In Figure 5, the number of failed jobs that need to be rescheduled is different among algorithms. Because the goal of the Binpack scheduling algorithm is to fill as many existing nodes as possible that could increase the degradation of tasks when too many tasks utilize the same node's resources. It results in the highest ART. The Deadline-aware Operator helps reschedule degraded jobs to reduce a noticed ART of Binpack algorithm.

V. Conclusion

In this research, a proposal for machine learning batch job scheduling in cloud-native clusters is presented, which proactively offers the queue wait time for each job based on jobs' features and historical observations to accelerate job scheduling algorithms. Machine learning jobs in a cloud-native cluster are automatically monitored, provisioned, and maintained by the Deadline-aware Job Scheduling using the Operator pattern approach. According to the performance evaluation, the proposal significantly enhanced the average response time of machine learning job flows under the pressure of a high load of requests and limited resources.

Acknowledgment

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (No. 2022-0-01015, Development of Candidate Element Technology for Intelligent 6G Mobile Core Network) and partly supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2023-RS-2023-00258649) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

References

[1] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. "Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021, pp. 493–506.

[2] Yali Zhao et al. "SLA-Aware and Deadline Constrained Profit Optimization for Cloud Resource Management in Big Data Analytics-as-a-Service Platforms". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 146–155.

[3] Yixin Bao, Yanghua Peng, and Chuan Wu. "Deep Learning-based Job Placement in Distributed Machine Learning Clusters". In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 2019, pp. 505–513.

[4] Salah Zrigui et al. "Improving the performance of batch schedulers using online job runtime classification". In: *Journal of Parallel and Distributed Computing* 164 (2022), pp. 83–95.

[5] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. "Backfilling using system-generated predictions rather than user runtime estimates". In: *IEEE Transactions on Parallel and Distributed Systems* 18.6 (2007), pp. 789–803.

[6] Rajath Kumar and Sathish Vadhiyar. "Prediction of queue waiting times for metascheduling on parallel batch systems". In: *Job Scheduling Strategies for Parallel Processing: 18th International Workshop, JSSPP 2014, Phoenix, AZ, USA, May 23, 2014. Revised Selected Papers 18*. Springer. 2015, pp. 108–128.

[7] Ju-Won Park, Min-Woo Kwon, and Taeyoung Hong. "Queue congestion prediction for large-scale high performance computing systems using a hidden Markov model". In: *The Journal of Supercomputing* 78.10 (2022), pp. 12202–12223.

[8] Nick Brown et al. "Predicting batch queue job wait times for informed scheduling of urgent hpc workloads". In: *arXiv preprint arXiv:2204.13543* (2022).

[9] Ali Ghodsi et al. "Dominant resource fairness: Fair allocation of multiple resource types". In: *8th USENIX symposium on networked systems design and implementation (NSDI 11)*. 2011.

[10] *Operator pattern*. Last accessed, 18 October 2023. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[11] *Custom Resources Definitions*. Last accessed, 18 October 2023. URL: kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[12] *Prometheus*. Last accessed, 18 October 2023. URL: https://prometheus.io/.

[13] *Volcano*. Last accessed, 18 October 2023. URL: https://volcano.sh/en/docs/.

[14] *Kubebuilder*. Last accessed, 18 October 2023. URL: https://book.kubebuilder.io/introduction.

[15] *Google cluster-usage traces v3*. Last accessed, 18 October 2023. URL: https://github.com/google/cluster-data/blob/master/ClusterData2019.md.

[16] Han Xiao, Kashif Rasul, and Roland Vollgraf. "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms". In: *arXiv preprint arXiv:1708.07747* (2017).