# Elastic Resource Management for Deep Learning Applications in a Container Cluster

Ying Mao, *Member, IEEE*, Vaishali Sharma, *Student Member, IEEE*, Wenjia Zheng, *Student Member, IEEE*, Long Cheng, *Senior Member, IEEE*, Qiang Guan, and Ang Li, *Member, IEEE*

**Abstract**—The increasing demand for learning from massive datasets is restructuring our economy. Effective learning, however, involves nontrivial computing resources. Most businesses utilize commercial infrastructure providers (e.g., AWS) to host their computing clusters in the cloud, where various jobs compete for available resources. While cloud resource management is a fruitful research field that has made many advances in production, such as Kubernetes and YARN, few efforts have been invested to further optimize the system performance, especially for Deep Learning (DL) training jobs in a container cluster. This work introduces FlowCon, a system that is able to monitor the individual evaluation functions of DL jobs at runtime, and thus to make placement decisions and resource allocations elastically. We present a detailed design and implementation of FlowCon and conduct intensive experiments over various DL models. The results demonstrate that FlowCon significantly improves DL job completion time and resource utilization efficiency when compared to default systems. According to the results, FlowCon can improve the completion time by up to 68.8% and meanwhile, reduce the makespan by 18.0%, in the presence of various DL job workloads.

**Index Terms**—Cloud computing, deep learning, containerized applications, resource management, high performance analytics

## 1 Introduction

In the past decades, increasing usage of the Internet has caused a spectacular information explosion, and this trend of increasing usage is predicted to continue in the coming years. Billions of users are consuming the Internet through various products and services such as websites, mobile applications, online games, and IoT devices. Back-end service providers are supported by state-of-the-art cloud infrastructures such as Microsoft Azure [1]. Cloud infrastructure providers employ large data centers to provide services to their customers. Virtualization, which focuses on decoupling hardware from software services (1), is one of the emerging technologies used in data centers and cloud environments to improve efficiency. Businesses are increasingly using machine learning algorithms to gain new insights from massive Internet data sets. Amazon [2] uses data collected on product performance and customer reviews to make more informed decisions about future product launches and inventory control. Morgan Stanley [3] and Standard Poor's [4] invest their resources to learn from proprietary data assets and public market historical data to promote their financial services.

Machine learning methods generally aim to fit a function $\hat{f}(\mathbf{x}, \theta)$ to a dataset. The function can be used as a model to make predictions based on the data or discover patterns in the data. In fitting the model error is usually defined by a loss function $J(\theta)$, and the target is to optimize the model parameters to minimize the loss. Usually, extremely large datasets and a large amount of computational resources are required to train machine learning algorithms. Therefore, efficient approaches which can handle data at scale have become desirable as well. For example, the learning systems frequently train various models over distributed data centers. Learning systems also often utilize services such as distributed data collection, processing, and storage offered in today's big cloud data centers. Deep learning architectures in particular often have extremely large parameter sets, sometimes running into the hundreds of millions and thus require extremely large volumes of data to train. Training a deep learning model on such a large volume of data is not only time-consuming but also extremely resource-intensive. Particularly when multiple models are being optimized on a shared computing infrastructure, there is a large potential for significant contention between models for CPU and memory.

This work aims to improve the resource contention problem for concurrent deep learning tasks in a cloud environment and consequently to accelerate their training through efficient cluster-wide resource management. The learning tasks considered are implemented as *containerized* learning applications in *containerized cloud architectures*, where training


- *Ying Mao, Vaishali Sharma, and Wenjia Zheng are with the Department of Computer and Information Science, Fordham University, New York, NY 10458 USA. E-mail: {ymao41, vsharma20, wzheng33}@fordham.edu.*
- *Long Cheng is with the School of Control and Computer Engineering, North China Electric Power University, Beijing 102206, China. E-mail: paracheng@gmail.com.*
- *Qiang Guan is with the Department of Computer Science, Kent State University, Kent, OH 44240 USA. E-mail: qguan@kent.edu.*
- *Ang Li is with the High-Performance-Computing (HPC) Group, Pacific Northwest National Laboratory (PNNL), Richland, WA 99354 USA. E-mail: ang.li@pnnl.gov.*


*Manuscript received 12 November 2021; revised 7 July 2022; accepted 17 July 2022. Date of publication 27 July 2022; date of current version 7 June 2023. (Corresponding author: Ying Mao.)*
*Recommended for acceptance by H. Wang.*
*Digital Object Identifier no. 10.1109/TCC.2022.3194128*

tasks are completed within a *container:* a lightweight, service-level virtualized environment in which processes can be run. Containers such as those enabled by Docker [5] and Kubernetes [6] provide a sandboxed, portable environment for the execution of arbitrary code on arbitrary machinery. More importantly, the resource usage of a container can be easily manipulated, which makes the technique quite popular in cloud computing.

While traditional multi-task training assumes a certain basis exists among learning tasks and can be accelerated through cross-task collaboration, the system proposed in this work focuses on the multiple machine learning tasks submitted to the shared distributed computing facilities. These tasks are independent and based on differing data schemes for arbitrary applications, where collaborative learning is not feasible. Currently popular opensource deep learning frameworks and libraries such as Pytorch [7] and TensorFlow [8] provide their images for deep learning services. Although various approaches have been proposed to optimize the efficiency of resource sharing across containers on cloud infrastructures in general, few of these approaches are designed specifically for deep learning. Existing approaches fail to take into account that DL training jobs typically have a nonlinear rate of convergence. Convergence rates generally decrease with time, which implies that the resource efficiency in terms of training gains per unit resource will decrease with time for a given deep learning job. This information is very valuable and current cloud platforms and methods do not consider this information, and thus they are liable to waste resources by allocating them to jobs where the gains in loss with respect to time are small or perhaps not appreciable (more details see Section 2).

In this paper, a novel container workflow management scheme is proposed, `FlowCon` [9], which aims to accelerate the overall system performance of multiple learning tasks running on a containerized cloud cluster through real time resource allocation. As opposed to a static or fixed configuration, `FlowCon` monitors the progress of learning jobs and dynamically configures the resource limit for each of them based on a novel metric called *growth efficiency*. The main contributions of this paper are summarized as follows:

- We introduce the concept of growth efficiency, a measure of the magnitude of the change in the loss function per unit of compute resource and propose `FlowCon` along with a suite of algorithms to monitor growth efficiency of deep learning jobs.
- `FlowCon` is designed to elastically allocate and/or withdraw the resources to/from each learning job at run time, allowing jobs to converge more quickly without significant scheduling overhead. It mainly includes a dynamic resource configuration and a growth efficiency based container placement strategy.
- We implement `FlowCon` in the popular container platform, Docker, with intensive experiments using various DL frameworks demonstrating its effectiveness. Specifically, compared to the default system, it reduces completion time of individual jobs by up to 68.8% and 18.0% on makespan.

## 2 Related Work

Due to the massive uses of deep learning, developing new models is an extremely active research area with numerous new approaches being proposed continually [10], [11]. However, very few of the models are developed explicitly to be efficient, rather most attention is given to their accuracy and quality [12], [13] and model size [14], [15], [16], [17]. From resource efficiency perspective, recent advances, include TRADL [18], SpeCon [19], DQoEs [20] and Chronus [21], consider deep learning trainings and inferences in a multi-tenant architecture. They attempt to share the resource more efficiently in order to satisfy specific requirements, such as user-specified targets, response time and deadlines. However, all of them focus on the cluster-wide scheduling and fail to take local resource management into account. Attentions have been given to developing explicitly resource and cost efficient learning applications. For example, MArk [22], a general-purpose inference serving system built in Amazon Web Services, aims to reduce response-time from client's side and service cost from developer's side. While it achieves significant improvement in performance-cost mode, MArk only applies to inferences applications.

As a resource-intensive process, most frequently, researchers and engineers train their models in a large cloud or cluster environment [23]. Generally, when more resources are given to a specific job, the training time will decrease [24], [25]. However, since resources are limited, providers must have a well-defined methodology for deciding which training jobs to prioritize. A limited amount of work has been done on this topic to-date. Running deep learning tasks inside virtual containers, ProCon [26] optimizes the cluster by selecting a best-fit host based resource contention rates for incoming tasks. While it achieves performance improvement, the scheduler only focuses on the cluster-wide container placement and fails to consider resource management on individual workers. Liquid [27] tackles this management problem by estimating individual task's resource requirement. The network-efficient scheduling methods are proposed to accelerate the training jobs. Authors in [28] propose a programming library that enables communication efficient top-k sparsification for distributed training. Gandiva$_{fair}$ [29] focused on the fairness problem of multi-sized training in a cluster. However, its automated trading mechanism requires a much more fine-grained checkpoints than the existing ones on Pytorch and Tensorflow.

Traditionally, GPUs that equip with thousands of computing cores are thought to perform way better than CPUs in training workloads. However, recent researches show that it might not always true. Focused on a CPU-GPU hybrid computing environment, AlloX [30] attempts to reduce the average job completion time by using a min-cost bipartite matching mechanism. It achieves surprisingly better performance when compared with GPU-only training. However, AlloX estimator requires the access to jobs' metadata, such as training iterations, which is not commonly available to the service providers. Additionally, to deploy Allox, the system needs to execute sample jobs to collect required data for its algorithms. Moreover, solely based on

a CPU, SLIDE [31] drastically outperform on the best available GPU with an optimized implementation of Tensorflow. It carefully tailors randomized hashing algorithms with the right data structures that allow asynchronous parallelism.

This work proposes FlowCon [9], [32], a resource management toolkit for containerized deep learning applications in cloud computing. With respect to nature of deep learning, FlowCon monitors the growth efficiency for the running task and dynamically adjust the resource assignment to improve the overall performance.

## 3 Background and Motivation

In this section, we introduce the containerization and utilize an intuitive example to motivate the proposed system.

### 3.1 Containerization

Containerized applications take advantages of a virtualization platform that is independent of operating systems. A sandboxed runtime environment is created by the platform to enable targeted services without launching the entire virtual machine and reducing the overhead. Docker and Kubernetes are the representative containerization platforms. In a Docker-enabled server, clients can send commands to the local docker daemon to launch a new container on a physical node.

To initialize a containerized application, we can use `docker run -d <App_Image>` command. It utilizes the image that packages all the required dependencies of this job. When the container is running, a rich set of tools is provided to manage and control it. For instance, with `docker exec <command/program_name>`, users can run a command or program in the container and `docker commit <container_id>` command is used to create a new image from recent changes. Executing `docker update <container_id> -memory 5 G`, the Docker daemon will set the upper limit of memory to 5 G for the specified container.

Regarding containerized deep learning applications, developers, researchers, and data scientists can use existing libraries and frameworks to develop learning models. In this field, TensorFlow [8], Pytorch [7] are the dominant players. The deep learning frameworks provided by them facilitate the ease of development learning models, such as LSTM-CNN and LSTM-RNN.

### 3.2 Motivation of FlowCon

Utilizing a single computing node in a production environment is challenging and lacks resilience as well as scalability. Usually, a group of servers on the cloud is built to provide the infrastructure to achieve reliable and scalable performance. Various toolkits have been developed aiming container orchestration in cluster environments, e.g., Docker Swarm and Kubernetes.

By default, the containerized system enables a free competition environment. It maintains fairness among all containers. The amount of resource that a container can obtain depends on its demands, implementation (e.g., data size and multi-threading) and fair share policy (e.g., 1/n, where n is the total number of containers). Alternatively, when launching a container, clients can explicitly specify an upper limit, such as 2 GB memory and 1.5 CPU cores.

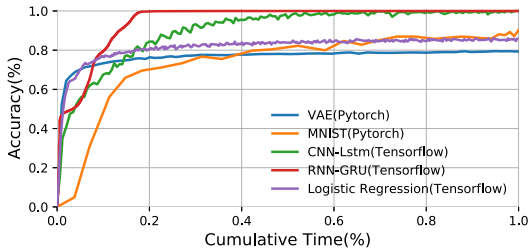



Fig. 1. Training progress of five models.

However, these mechanisms are not optimal for deep learning tasks. There are two main reasons. (1) It is very challenging for ordinary users understand how much resources should be assigned to each of their containers. In order to optimize the overall system performance, various factors have to be taken into consideration. (2) In a production environment, most models don't need to be perfect in a distant future. In some settings such as real-time data analytics, a model would be frequently requested by applications (e.g., prediction) even *before* convergence is reached. In this case, bringing the model to an acceptable (rather than perfect) level of accuracy is the most important. (3) Normally, training tasks have their own learning progresses. For those fast learning models, they can reach an acceptable state with less time. A straightforward fair share among all tasks may results in a resource waste. Since the on-going models that are already in an acceptable state will continue to utilize as much resource as those with much optimization left to do, even though the nearly-converged jobs only make small gains in optimizing their evaluation function per unit computing resource.

As an intuitive example, Fig. 1 presents the learning process of 5 built-in models in Pytorch and Tensorflow. In this experiment, models are running inside their own containers and sharing the resources on the same physical machine. As we can see from the figure, the RNN-GRU model on Tensorflow reaches 90.0% accuracy with only 14.5% of the cumulative time. Upon the task finishing, this value increases to 93.2%. This observation suggests that in the first 14.5% of the total time, the model achieves 96.8% of its maximum. Later, it takes 85.5% of the total time for another 3.2% of the accuracy. Usually, there are many other tasks running concurrently and seeking computational resources. It is reasonable to shift parts of the computing resource occupied by RNN-GRU training to the other learning tasks. In this work, we propose FlowCon to accomplish this goal.

## 4 The FlowCon System

The design of FlowCon is discussed in this section. We introduce the proposed architecture, main system functionalities, and growth efficiency.

### 4.1 FlowCon Architecture

A typical cluster of containers has multiple managers and workers in the system to achieve reliability and resilience.

Clients send commands to the manager, who is responsible for finding best hosts for the new containers. Fig. 2 presents the system architecture of FlowCon.

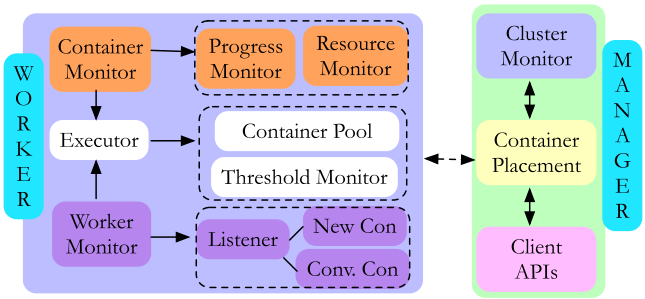



Fig. 2. FlowCon system architecture.

The components of FlowCon reside in both the manager and workers. While the manager has a global view of the whole system, FlowCon executes most of the algorithms and data collections on the workers in order to prevent overwhelming it.

Fig. 2 draws the architecture of FlowCon, which consists of three modules on the manager side and three modules on the worker side.

Container Monitor: The container monitor maintains a list of running jobs inside each container. The values of evaluation functions are collected iteratively. In addition, it monitors the resource usage, such as CPU and memory, per each active container.

Worker Monitor: A worker monitor measures the container pool on the worker. There are two listeners, New Cons and Converged Cons. Unlike the container monitor that focuses on the jobs inside a particular container, these listeners target the status of container pools. The New Cons listener tracks the newly submitted containers from the manager and prioritize them by assigning more resources. The Converged Cons listener tracks the containers that have finished the whole training and releases the occupied resource to others.

Executor: The key module on the worker is the Executor. It analyzes the data such as values of evaluation functions and resource usages of each containers. Periodically, required parameters are calculated and algorithms are executed (details in Section 4) to update the resource configuration for each container and threshold for categorization. When a listener sends an alert message due to worker status changes (e.g., a new container assigned), the Executor will interrupt the current interval and start running the necessary algorithms.

On the manager side, FlowCon attempts to reduce the computation to protect it from overload. The *Client APIs* interact with users and accept the commands, like launching a new container. *Container Placement* module accepts these launching commands and utilizes the information from *Cluster Monitor* component to select the best host for incoming containers. The *Cluster Monitor* is responsible for required information, such as number of containers and values of evaluation functions, which is collected from workers.

### 4.2 System Optimization Problem

FlowCon targets on efficiently utilize the constrained resources inside a system. Traditionally, if the resource is occupied by a task, the system considers it is "in use". Given a deep learning training job, however, the term "in use" fails to accurately indicate efficiency, as described in Fig. 1.

**Algorithm 1.** Dynamic Resource Management on Worker $W_i$

Initialization: $c_{id} \in \{c_1, c_2, \ldots, c_n\}$, $W_i$, Time $t$, Watching List $WL$, Completing List $CL$, New List $NL$, $\alpha$ and $itval$.
**for** $c_{id} \in W_i$ & $\alpha \neq -1$ **do**
  Calculate $G_{W_i,c_{id}}(t)$
  **if** $G_{W_i,c_{id}}(t) < \alpha$ & $c_{id} \in NL$ **then**
    $NL$.remove($c_{id}$)
    $WL$.insert($c_{id}$)
  **else if** $G_{W_i,c_{id}}(t) < \alpha$ & $c_{id} \in WL$ **then**
    $WL$.remove($c_{id}$)
    $CL$.insert($c_{id}$)
  **else if** $G_{W_i,c_{id}}(t) \geq \alpha$ **then**
    $NL$.insert($c_{id}$)
    $WL$.remove($c_{id}$)
    $CL$.remove($c_{id}$)
**if** $\forall c_{id} \in W_i, c_{id} \in CL$ **then**
  **for** $c_{id} \in W_i, r_i \in R$ **do**
    $L_{c_{id},r_i} = 1$
  $itval = itval \times 2$
**else**
  **for** $c_{id} \in W_i, r_i \in R$ **do**
    **if** $c_{id} \in CL$ **then**
      $L_{c_{id},i} = \frac{G_{W_i,c_{id}}}{\sum_{c_{id}} G_{W_i,c_{id}}}$
      $L_{c_{id},i} = Max\{L_{c_{id},i}, \frac{1}{|c_{id}|}\}$
    **else if** $c_{id} \in WL$ **then**
      $L_{c_{id},r_i} = L_{c_{id},i}$
    **else**
      $L_{c_{id},i} = \frac{G_{W_i,c_{id}}}{\sum_{c_{id}} G_{W_i,c_{id}}}$

With respect to the characteristics of deep learning training process, a new concept of efficiency that reflects an application's gains on its evaluation function. Given a cluster with a set of active containers, $\{c_{id}\}$, each container uses its own evaluation function to assess its type of machine learning model (e.g., loss reduction and inception score) $E_{c_{id}}(t)$. For each model, based on its $E(t)$, we define the progress score for the container $c_{id}$ to be Eq. (1), where $t_i - t_{i-1}$ is the measurement interval. The value of $P_{c_{id}}(t_i)$ is the per-second progress within the interval.

$$P_{c_{id}}(t_i) = \frac{|E_{c_{id}}(t_i) - E_{c_{id}}(t_{i-1})|}{t_i - t_{i-1}}. \quad (1)$$

Here, $P_{c_{id}}(t_i)$ reflects the progress over a given time interval, but it does not account for the resources used towards that progress. Therefore, we propose the *growth efficiency* for each container $c_{id}$ with an active deep learning job. Eq. (2) presents the growth efficiency with respect to different types of resources (e.g., CPU, memory, network I/O and block I/O), denoted by $r_i$, and $R_{c_{id},r_i}(t_i)$ is a function that returns the average resource usage of $c_{id}$ within the interval $t_i - t_{i-1}$ among each $r_i$.

$$G_{c_{id},r_i}(t_i) = \frac{P_{c_{id}}(t_i)}{R_{c_{id},r_i}(t_i)}. \quad (2)$$

FlowCon aims to maximize the sum of growth efficiency for the whole system in each interval, where each learning

model has its own evaluation function and can be calculated in real-time. Assume that there are $n$ containers, each runs one job in the system, and $R_{i_{max}}$ denotes the overall resource capacity for $r_i$. Then, our performance optimization problem can be formalized as $\mathcal{P}$ below, where $G_{c_{id},r_i}$ can be computed from measurements of $E_{c_{id}}(t_i)$ and $R_{c_{id},r_i}(t_i)$.

$$\mathcal{P}: \ \text{Max} \sum_{i}^{n} G_{c_{id},r_i}$$
$$\textbf{s.t.} \ \sum_{i}^{n} r_i \leqslant R_{imax}. \tag{3}$$

Growth efficiency provides a new metric to evaluate the model training process that reflects the real-time resource usage. Considering a shared computing environment, resource distribution dynamically affects the value of each individual job's growth efficiency. From a system perspective, it has to maximize the overall growth efficiency. Thus, there is a trade-off between system-wide performance and individual jobs. Additionally, fairness among all the jobs should be considered.

Due to the page limit, we omit the theoretical analysis and provable guarantees. Instead, `FlowCon` focuses on system design, algorithm implementation, and numeric performance analysis.

## 5 Solution of FlowCon

In this section, we discuss the algorithm design of the dynamic resource configuration, the listeners to reduce the overhead on workers, the adaptive threshold for container categorization and container placement for incoming workloads.

### 5.1 Dynamic Resource Configuration

In a virtual machine (VM) based cluster, each of VM is allocated with a fixed amount of resource, such as 10 GB memory and 4 CPU cores. This scheme maintains a better isolation, however, it fails to update the configuration to efficiently utilize the resources. Given a container based cluster, the resource distribution can be updated on-the-fly with specified resource limits. When the containers are initialized without any resource limits, the free competition environment is enabled just like processes in an operating system. A resource limit can be assigned to any container at runtime to update the resource allocation.

For example, the command `docker update <options> container_id` can reset the resource limit as desired. The sample options include `–cpus` for the number of cores, `–cpu-rt-runtime` for CPU real-time runtime in microseconds, `–memory` for memory usage in MB, `–blkio-weight` for a relative weight of block I/O and etc. Finally, values of the limit set by the `docker update` commands are soft limits, which means that the even if the container cannot maximize its own resource, the unused option will be utilized by others.

### 5.2 Resource Assignment in FlowCon

In a container cluster, the manager receives commands from clients and selects a worker to launch the container. This newly launched container will compete for resources with existing workloads on the same computing node. With the default scheduler, each container is assigned the same priority that leads to an uniform resource distribution on the worker. This fair sharing scheme provides a general-purpose solution. As we have discussed, however, it fails to take characteristics of deep learning training jobs into consideration. In comparison, a growth-efficiency based method is utilized in `FlowCon` to update resource assignments of each running container in a dynamical way.

As shown in Line 1 of Algorithm 1, each $W_i$ first receives the following parameters from its manager: the time $t$, the threshold $\alpha$ and the algorithm interval $itval$. Moreover, it initializes three lists as below to categorize each container:

- New List ($NL$): Young and quickly growing
- Watching List ($WL$): Near convergence
- Completing List ($CL$): Converging and growing slowly

When the threshold is -1, it deactivates the Algorithm 1 (Line 2). The details in this scenario is discussed in Algorithm 3. Based on a positive threshold and the growth efficiency, the algorithm places each active container into the proper list (Lines 2 - 13). If all containers are in the $CL$, then each container's resource limit is set to 1 allowing them to compete freely for resources (Lines 14 - 16). While `FlowCon` is permitting free competition, it is no longer necessary for the system to run the algorithm at the initial interval. Instead, `FlowCon` utilizes an exponential back-off scheme to double the $itval$ in order to reduce the overhead of running the algorithm (Line 17). Once the growth efficiency is less than the preset threshold, `FlowCon` applies the following rules:

- Each container in the $CL$ has its resource limit set based on its growth during the time interval $\frac{G_{W_i,c_{id}}}{\sum_{c_{id}} G_{W_i,c_{id}}}$ (Lines 18 - 21)
  - If growth is exceedingly small, which is common after convergence, the resource limit is set to a lower bound to prevent abnormal behavior caused by limited resources (Line 22).
- The resource limits of containers in the $WL$ remain unchanged (Line 24).
- Allocate more resources to containers in the $NL$ (Line 26).

### 5.3 Listeners in FlowCon

The container monitor provides information that allows Algorithm 1 to dynamically allocate resources based on the growth-efficiency in each container and to reduce the scheduling overhead with an exponential back-off scheme. However, there is latency between the time that a worker's state changes (e.g., a new container is initiated) and the point that it can reallocate resources. To improve this issue, `FlowCon` deploys lightweight background-listeners to track the container states in real-time.

With the same set of parameters, Algorithm 2 presents the workflow of listeners on $W_i$. First, it initializes the $CL$, $WL$, $NL$ and $itval$, and it uses $i$ to record the number of iteration of the listener (Line 1). When the $i^{th}$ iteration is running, it uses the function $T(i)$ to fetch the total number of containers on the $W_i$ (Line 2). In all runs after the first run,

the listener calculates the difference $c$, between the most recent two iterations (Lines 3 - 4). If $c > 0$, it means that there are $c$ new containers now active in the system, so the listener will stop and the algorithm finds out the $c_{id}$ of the new containers and add them to the $NL$ (Lines 5 - 7). In the meantime, it resets the $itval$ to the original value in order to break the exponential back-off scheme, and then starts to run Algorithm 1 to update the resource allocation as well as increases the iteration number i. (Lines 8 - 9). The case when $c < 0$ indicates that some containers have completed their jobs. The algorithm will then find the relevant containers by their $c_{id}$, remove them from their associated category ($NL$, $CL$ or $WL$) and release their resources (Lines 10 - 15). Finally, we reset the $itval$, start running Algorithm 1 and increment the iteration number $i$.

**Algorithm 2.** Listener on Worker $W_i$

1: Parameter Initialization: $CL$, $WL$, $NL$, $itval$, $i = 0$
2: $T(i) =$ total number of container at iteration $i$
3: **if** $i \neq 0$ **then**
4: $c = T(i) - T(i-1)$
5: **if** $c > 0$ **then**
6: **for** $c_{id} \in W_i$ & $c_{id} \notin CL$ & $\notin WL$ & $\notin NL$ **do**
7: $NL$.insert($c_{id}$)
8: $itval =$ initial_value
9: Run Algorithm 1 and $i++$
10: **else if** $c < 0$ **then**
11: **for** $c_{id} \in CL \mid \in WL \mid \in NL$ and $c_{id} \notin W_i$ **do**
12: $NL$.remove($c_{id}$)
13: $WL$.remove($c_{id}$)
14: $CL$.remove($c_{id}$)
15: Release_resource $c_{id}$
16: $itval =$ initial_value
17: Run Algorithm 1 and $i$++

### 5.4 Adaptive Threshold

The dynamic resource configuration is designed by categorizing jobs into three different stages, $NL, WL$ and $CL$. The key to determine the stages is a threshold, $\alpha$. On one hand, a small threshold results in many containers staying in $NL$ and competes for resource with new jobs. A large threshold, on the other hand, leads to jobs quickly evolve to $CL$`FlowCon` uses an adaptive threshold scheme to dynamically update the threshold value.

Algorithm 3 presents the details. When a new container joins this worker, $w_i$, it first initializes the parameters, e.g., active containers, $NL, WL, CL$ lists (Line 1). When all of the training jobs are in $CL$, `FlowCon` sets $\alpha$ to -1, which guides Algorithm 1 to deactivate itself and enable free competition among all containers (Lines 2-4). If there is only one active container on $w_i$ and it is in $NL$ or $WL$, the system set $\alpha$ to the half to its previous per-second progress (Lines 5-6). In this case, $\alpha$ does not have any effect on its limit since Algorithm 1 would assign all the available resources to it. With multiple jobs running across different categories, `FlowCon` calculates the sum of per-second progress gains for $NL$ and $CL$, respectively (Lines 8-12). Finally, we set the threshold to the half of the average gain of $NL$ and $CL$ (Line 13).

**Algorithm 3.** Adaptive Threshold on Worker $w_i$

1: Parameters initialization:
$c_1, \ldots c_{id} \ldots, c_n \in = w_i$
$NL, WL, CL$ for $w_i$
$|w_i| = |NL| + |WL| + |CL|$
Time $t$
2: **if** $|CL| = |w_i|$ **then**
3: $\alpha = -1$
4: Return;
5: **else if** $|W_i| = 1$ **then**
6: $\alpha = \frac{|E_{c_{id}}(t_2) - E_{c_{id}}(t_1)|}{t_2 - t_1} \div 2$
7: Return;
8: **for** $c_{id} \in w_i$ **do**
9: **if** $c_{id} \in NL$ **then**
10: $\alpha_{nl} = \alpha_{nl} + P_{c_{id}}(t_i)$
11: **else if** $c_{id} \in WL$ **then**
12: $\alpha_{wl} = \alpha_{wl} + P_{c_{id}}(t_i)$
13: $\alpha = (\alpha_{nl}/|NL| + \alpha_{wl}/|WL|) \div 2$

### 5.5 Container Placement in `FlowCon`

When an incoming job arrives at the manager, the first task it needs to complete is to select a worker to host the new container. In Docker Swarm and Kubernetes clusters, users can specify requirements for their hosts, such as memory space and network bandwidth. The system utilizes specified constraints to filter out unqualified workers. When there are multiple qualified workers, different placement preferences [33], [34] are in effect to select best workers.

**Algorithm 4.** Container Placement for Incoming $c_j$ on Manager

1: Parameters initialization:
$w_i \in \{w_1, w_2, \ldots, w_n\} = W$,
$NL, WL, CL$ for $w_i$
$c_{id} \in w_i$, Time $t$
2: **for** $w_i \in W$ **do**
3: **if** $\forall c_{id} \in w_i, c_{id} \in CL$ **then**
4: Select $w_i$ with minimum $|CL|$
5: $w_i$.host($c_j$)
6: **else**
7: $D_{NL}(t) = \sum_{c_{id} \in w_i \ \& \ c_{id} \in NL} G_{w_i, c_{id}}(t)$
8: $D_{WL}(t) = \sum_{c_{id} \in w_i \ \& \ c_{id} \in WL} G_{w_i, c_{id}}(t)$
9: $D_{CL}(t) = \sum_{c_{id} \in w_i \ \& \ c_{id} \in CL} G_{w_i, c_{id}}(t)$
10: $S_{w_i} = |NL + 1| \times D_{NL}(t)$
$|WL| \times D_{WL}(t)$
$|CL| \times D_{CL}(t)$
11: **for** $w_i \in W$ **do**
12: Find $w_i$ with minimum $S_{w_j}$
13: $w_i$.host($c_j$)

With respect of containerized deep learning applications, `FlowCon` ranks the workers based on the growth efficiency of running jobs on them. Algorithm 4 details the container placement strategy in `FlowCon`. First, the algorithm initializes the parameters, such as system time, active workers, containers on the workers and associated $NL, WL, CL$ (Line 1). Next, the manager tracks if there are any workers that all its running jobs are in the completing list. A full completing list indicates that the jobs are in their final stage of the

TABLE 1
Tested Deep Learning Models

| Model [36] [37] | Eval. Function | Plat. |
|---|---|---|
| Variational Autoencoders (VAE) [38] | Reconstruction Loss | P/T |
| Modified-NIST (MNIST) [39] | Cross Entropy | P/T |
| Long Short-Term Memory (CFC) [40] | Softmax | T |
| Long Short-Term Memory (CRF) [41] | Squared Loss | P |
| Bidirectional-RNN [42] | Softmax | T |
| Gated Recurrent Unit (GRU) [43] | Quadratic Loss | T |

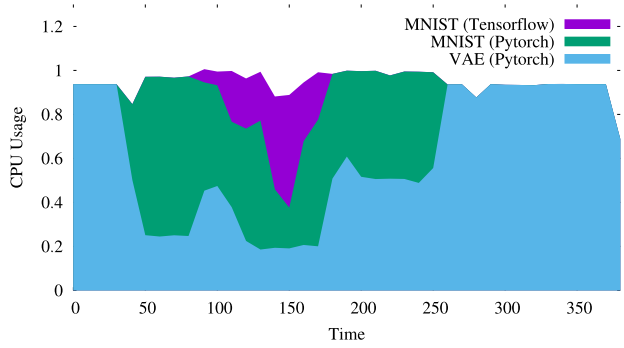



Fig. 3. FlowCon.

training. In this case, the algorithm selects $w_i$ with minimum number of active containers (Lines 2-5). When this new container joins $w_i$, the Algorithm 1 would prioritize it by limit the resource usage of those in $CL$.

If all the workers have at least one container in $NL$ or $WL$, it suggests some of the existing jobs still move very fast. In this case, the manager estimates the gross resource demands by summing up of growth efficiency in each category (Line 6-9). Then, the values are weighted with the number of containers except $NL$. For $NL$, FlowCon add 1 to assume that the incoming container is assigned to this worker (Line 10). Finally, the algorithm selects the worker with minimum score to place the new container.

## 6 Evaluation

In this section, we first discuss the implementation, workload and experiment settings. Then, we conduct intensive experiments on the cloud with different settings to evaluate the proposed system from various prospective.

### 6.1 Experimental Framework

Based on Docker 20.10.7, we implement FlowCon as a middleware that residents on workers and managers. On the manager, it interacts with clients for the incoming containers and then execute the algorithms to calculate its best host. The required data for calculation is collected from the workers. On the worker side, it monitors the running containers and dynamically configure their upper limits to prioritize fast growing ones.

We conduct our evaluation on two cloud platforms for comprehensive experiments. The 1-Manager-1-Worker cluster is built on the NSF Cloudlab [35] in the Downtown Data Center from the University of Utah. Specifically, it uses the R320 physical node, which contains a Xeon E5-2450 processor and 16 GB Memory. The Multiple-Worker clusters are built on Google Cloud Platform with 4 vCPUs and 16 GB memory in a Ubuntu 20.04 virtual machine.

To ensure an easy-access evaluation, we test FlowCon with various deep learning models. Both models (e.g., training code, parameter settings) and datasets (training data) are build-in and provided by the Pytorch and Tensorflow platforms. The following Table 1 lists the models that involved in our experiments.

### 6.2 Experiment Setup and Evaluation Metrics

FlowCon is designed to dynamic manage containerized deep learning applications. Since the targeted workloads are computation-intensive, in the experiments, we focus on the resource management of CPU cores. However, the FlowCon algorithms support CPU cores, Memory, Network I/O and Disk I/O. It can be easily extended to other types of resources.

In our experiments, we compare FlowCon with the default scheduler Docker Swarm, denoted as $DS$. The jobs are randomly picked up from the pool and randomly submitted to the clusters within an given interval. To summary, we have conducted the following experiments.

- Single-Worker Cluster: We evaluate FlowCon with a 1-Manager-1-worker cluster. In this case, the container placement algorithm is not in use as there is only one option for assignment. It validates the effectiveness and performance of Algorithms 1, 2, and 3.
- Multi-Worker Cluster: We further evaluate FlowCon with 4-worker and 8-worker cluster on Google Cloud Platform. In this case, all algorithms are active. Different numbers of jobs are submitted to the system to evaluate the scalability of the system. Additionally, we considered both homogeneous and heterogeneous clusters.

With the collected data, we mainly consider the three metrics to evaluate the system performance.

- Overall makespan: It indicates the total length of the schedule for all the jobs in the system (across all workers);
- Individual job completion time: It is the completion time of each individual jobs in the system;
- CPU usage: It provides the computation power for the training jobs.

### 6.3 Single-Worker Cluster

In this subsection, we evaluate FlowCon performance with a 1 Manager 1 Worker cluster. In these experiment, the container placement algorithm is not in effect.

First, we conduct a experiment with three jobs as a proof-of-concept experiment. The first job, VAE on Pytorch, starts at 0 s, then, MNIST on Pytorch begins at 40 s, and the last job, MNIST on Tensorflow launches at 80 s. It demonstrates the effectiveness of FlowCon on resource configuration at runtime. The detailed CPU resource usage is illustrated on Figs. 3 and 4. Fig. 4 clearly shows that the Docker Swarm system does not prioritize any jobs at any stages. Given a specific time, the running containers received a equal distribution of CPU resources without any runtime

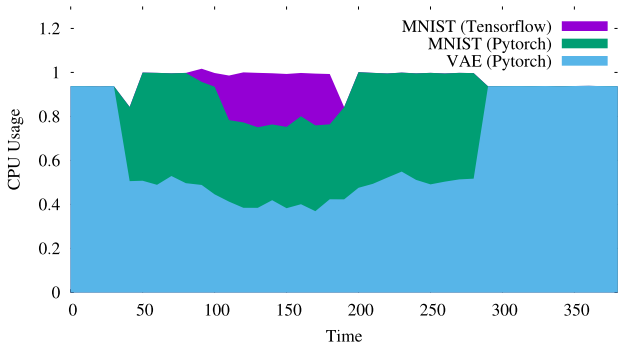



Fig. 4. *DS*.

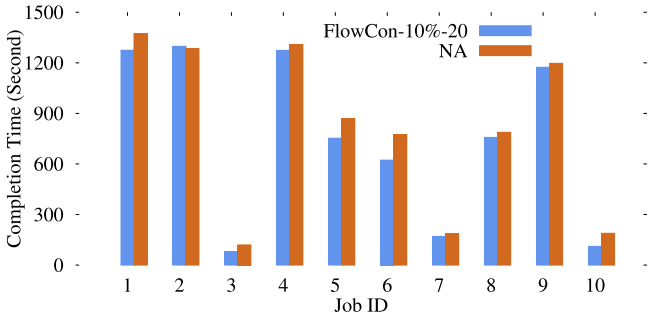



Fig. 5. 10 jobs experiment.

configuration. For example, from 40 s to 80 s and 180 s to 280 s, the CPU usages of VAE (Pytorch) and MNIST (Pytorch) are approximately equivalent. This is a common approach for a general-purpose computing platform. With `FlowCon`, however, the system can dynamically configure resource limits for each job with the respect of their growth efficiency at runtime. Demonstrated on Fig. 3, when MNIST (Pytorch) is launched at time 40 s, `FlowCon` takes two actions: (1) assigns VAE's (Pytorch) a resource limit since it is growing slowly, and (2) categorizes MNIST (Pytorch) to $NL$ and resource limit to 1, allowing for the maximum resource. As shown ont the figure, VAE (Pytorch) receive 25% while MNIST (Pytorch) will use 75% of the total resources.

*10-Job Experiments*. Next, the system is evaluated with more workloads that, naturally, generates more challenges.

With the same single-worker cluster, we randomly select the models from the pool and randomly submit them to the system within 0 s - 200 s interval. The comparison of completion time is presented in Fig. 5. Overall, `FlowCon` is able to reduce the completion time of 9 out of 10 jobs. For Job-2, however, its completion time records an increases of 1.1%. This is due to the fact that the submission interval between Job-2 and Job-3 is very small (13 s), and this quickly leads to resource reduction on Job-2. In this case, `FlowCon` would benefit more on the new one. For the improved 9 jobs, it achieves completion time reductions from 1.8% to 41.2%, with the largest one occurring in Job-10: from 188.3 s to 110.8 s. In this experiment, the makespans are 1350.7 s and 1384.9 s for `FlowCon` and *DS*, respectively.

In `FlowCon`, the resource allocation is updated based on the associated value of growth efficiency such that the fast-growth containers would get more resources. With the 10-job experiment, we pick up two representative

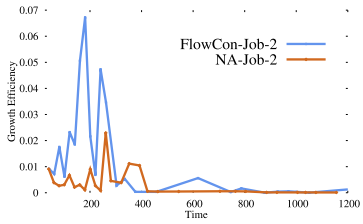



Fig. 6. Growth Efficiency(J-2).

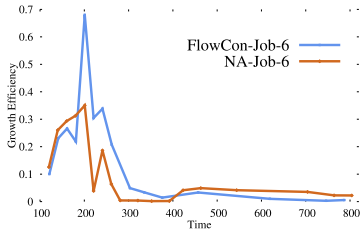



Fig. 7. Growth Efficiency(J-6).

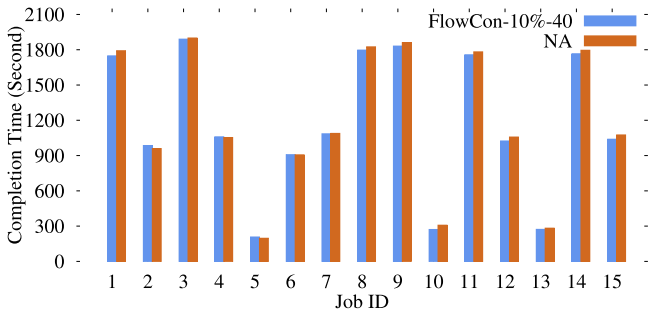



Fig. 8. 15 jobs experiment.

jobs, Job-2 and Job-6, where Job-6 outperformed and Job-2 underperformed.

The values of growth efficiency are illustrated on Figs. 6 and 7. Comparing the results of Job-2, obviously, `FlowCon` records higher growth efficiency at the very beginning. This is due to the fact that with `FlowCon`, jobs are under a controlled competition such that Job-2 has a higher limit than Job-1, which means the system will assign more resources to it with the expectation that Job-2 would grow faster. Even when Job-3 joins the system, resources for Job-2 will not reduce too much since it is still in the $WL$, not $CL$. With $DS$, however, the system fails to prioritize any active container and all jobs are in a free competition environment. The difference in resource assignment only control by the training code, e.g., neural network design, multi-threading.

When converged, Job-2 enters $CL$ in `FlowCon`. In this case, the system lowers its priority by assigning a smaller upper to it. The released resources will be moved to newer jobs due to a smaller value of the limit, which results in a loss when compares to $DS$ at the time 320 s.

A different trend is discovered in Fig. 7. In the first 2 iterations, Job-6 obtains a sightly lower values of growth efficiency in `FlowCon` when compared with $DS$. The reason lies in the fact that, after launching a container, it requires time to collect data in `FlowCon`for the calculation and updating the resource configuration for the other active 5 jobs in the system.

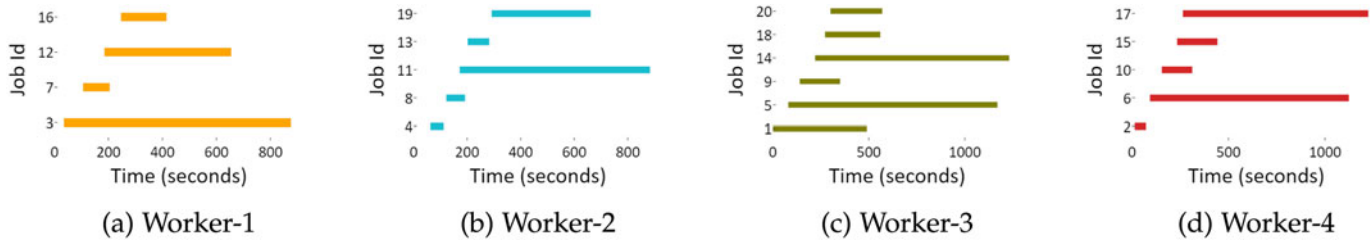



Fig. 9. Container placement for 20 jobs experiment with FlowCon.

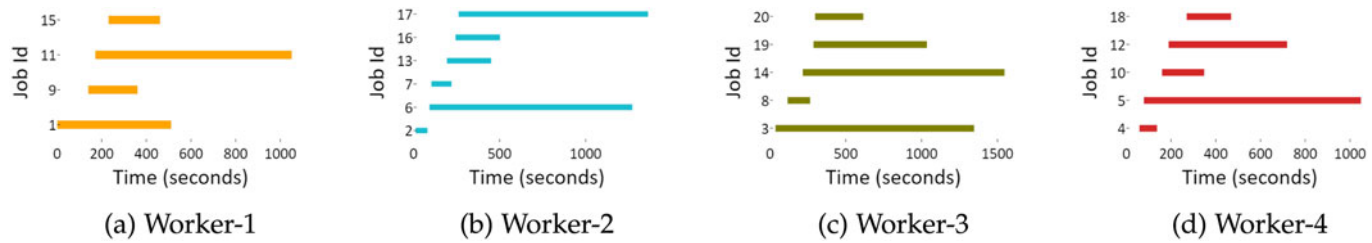



Fig. 10. Container placement for 20 jobs experiment with $DS$.

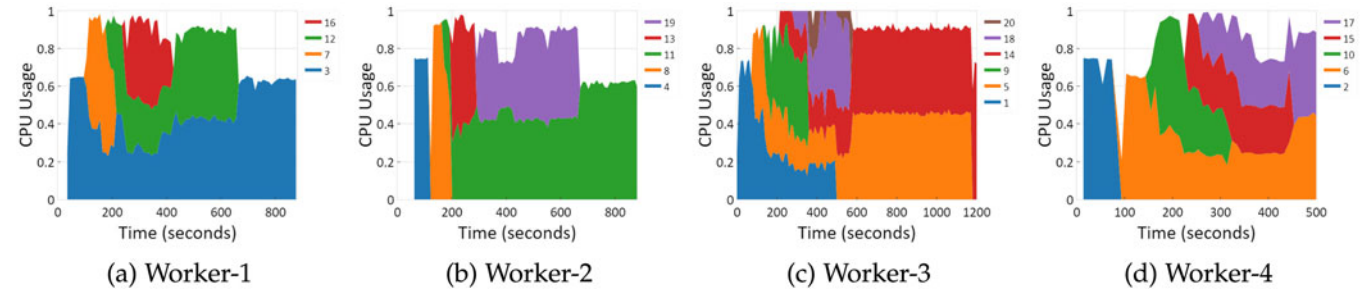



Fig. 11. CPU usage for 20 jobs experiment with FlowCon.

*15-Job Experiments*. To create more challenges to the proposed system, we further increase the number of jobs to 15. Similar to the previous experiment, all jobs are randomly submitted to the system during the interval of 0 s to 200 s. The degree of resources competition increases along with the number of concurrent containers.

Fig. 8 plots the completion time. With the data, the same trend is found as 10-Job experiment: FlowCon slightly improves the makespan from 1980.1 s to 1950.9 s (1.5%). The more competition in the system, the greater the challenge for elastic container management at runtime. Taking look at the individual training jobs in Fig. 8, there are 11 out of 15 jobs complete faster. For those underperformed ones, completion time increases: (1) Job-2, from 959.4 to 985.1; (2) Job-4: from 1059.6 to 1053.3; (3) Job-5: from 196.2 to 207.5; and (4) Job-6, from 906.4 to 907.9. However, we can see these increments are quite small, e.g., Job-5's completion time increases the most, only by 5.7%. For the other 11 jobs, the reduction percentages range from 1.2% to 11.9% and the largest degree of reduction occurs in Job-10, from 308.1 s to 271.4 s.

## 6.4 Multi-Worker Cluster

In this subsection, we present the results from clusters with multiple workers, which is a most common case on the cloud. Specifically, we build two clusters, 1-Manager 4-Workers and 1-Manager 8-Workers. The clusters are built in the us-central1-a data center from Google Cloud Platform. Both workers and the manager equip with 4 vCPU cores and 16 GB memory.

*Container Placement*. In a multiple-worker cluster, selecting a host for an incoming container is the first task that the system needs to complete. Figs. 10 and 9 presents the container distribution of a 20-job experiment in a 1-Manager-4-Worker cluster. Different container placement algorithms reflect different design principles. As a general container management platform, Docker Swarm is built to accommodate various applications. It selects the host based on the reports (through heartbeat messages) from the workers. These reports usually contain, number of running containers, their resource consumptions. To prevent overwhelming the manager node, these reports cannot be sent too frequently. The delay in between two reports, however, may lead to inappropriate decisions from the manager side. For example, the submission intervals of Job-4 and Job-5, Job-6 and Job-7 are 13 s and 18 s. In the Docker Swarm system (Fig. 10), when assigning Job-5, the manager still reads the Worker-4's report from the previous round that fails to take Job-5 into consideration.

With FlowCon (Fig. 9), however, the new host is selected based on growth efficiency, which prioritizes the fast growing jobs. For Job-5 and Job-7, even though the latest report from workers fails to reflect the resources that occupied by Job-4 and Job-6 due to short submission intervals, FlowCon is able to make a right decision on host selection. This is due to the fact that FlowCon would automatically categorize the

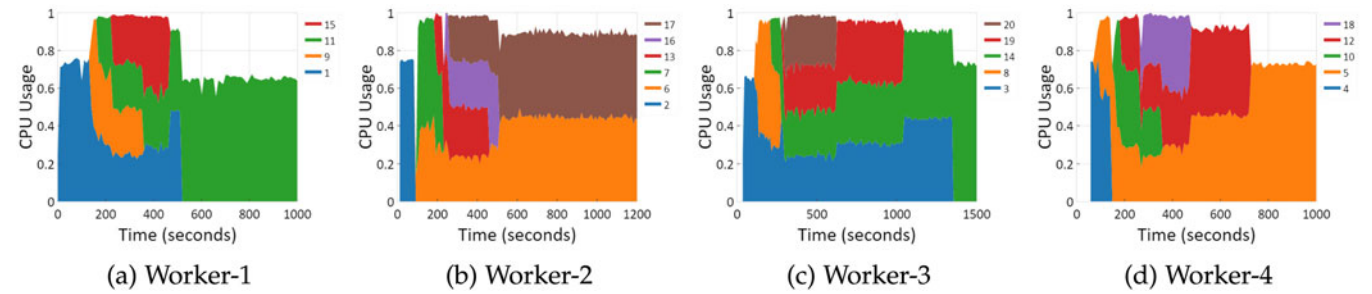



Fig. 12. CPU usage for 20 jobs experiment with $DS$.

incoming container into $NL$ and assuming that it would be a fast growing one (Line 7 in Algorithm 3). This design prevents the manager from misleading by outdated reports without using frequent messages from workers. Please note that the $NL$ is temperately added 1 and when the new report arrives, the manager will update the value of $NL$ for the worker.

*CPU Usage*. On every worker, the dynamic resource management is in execution to carefully monitor and allocate resource. Figs. 11 and 12 plot the CPU usage for the above-mentioned 20-job experiment. The same trend is discovered as we haven seen in the 3-job experiment. For example, on Fig. 11a, we can clearly see that, at time 200 s, the resource distribution is 73.2% for Job-7 (the second job on Worker-1) and 22.1% for Job-3 (the first job on Worker-1). The uneven allocation different stages those two jobs. At time 200 s, Job-7 still grows fast and is categorized in $NL$. Job-3, however, moves very slow according to its growth efficiency.

Without an indicator to differentiate jobs, Docker Swarm treats different jobs equally on all workers. At any given time, the resources are evenly distributed to all the active jobs on this Worker-3. In Fig. 12c, for instance, at time 500 s, the distribution is roughly 25% for each running job.

Another finding that we can discover in Figs. 12 and 11 is that some jobs fail to consume all available resources without any competition. For example, Job-1 and Job-3 occupy around 75% and 65% even running alone on the worker. The behavior is due to the implementation, e.g., multithreading and concurrent computation. Since the limit utilized in the system is a soft limit, the unused resource will not waste if there are other jobs running concurrently. Furthermore, the figures present the resource usage data from the actual workloads and exclude the overhead generated by our algorithms and the operating system. This is also the reason that figures rarely show 100% usage. As Fig. 12a presents, at time 180 s, the resource is allocated as 65.2% and 30.1% for Job-1 and Job-9 with Docker Swarm, respectively.

*Completion Time*. Based on the container placement on the manager and the resource management on workers, the Fig. 13 illustrates the completion time of the 20-job experiment. Obviously, `FlowCon` outperforms Docker Swarm by reducing 18 out of 20 jobs. In the 18 improved jobs, Job-13 wins the most and its completion time is reduced from 258.4 s to 80.7 s, 68.8%. The reason lies in the fact that, with `FlowCon`, Job-13 is hosted by Worker-2. At beginning of the training, it is competing resource with Job-8 and Job-11, where Job-8 is classified as $CL$ and Job-11 is in $WL$. In this case, the `FlowCon` gives Job-13 more priority, where it gets 60.8% and Job-11 is given 36.0% at time 220 s. In terms of average completion time, `FlowCon` achieves 18.9% reduction, from 532.8 s to 431.9 s. The reason lies in the fact that when assigning more resources to fast moving jobs, they are able to complete faster, when those jobs finish, the release resources would benefit the previous constrained jobs. When considering the makespan, `FlowCon` achieves a reduction of 18.0% from 1330.0 s to 1089.9 s. This indicates the system is more efficient with `FlowCon`.

*Scalability - More Workloads*. Next, we conduct a 40-job experiment on the same 4-worker cluster and jobs are randomly submitted to the cluster within [0, 600s] interval. Fig. 14 plots the result of completion time. Comparing with the previous experiment, we doubled the workload. The increased number of jobs produces more challenges to the system. Overall, `FlowCon` is able to improve 24 out of 40, 60%, jobs. The largest gain is found on Job-19, which reduced 60.6% from 329.3 s to 129.6 s. With `FlowCon`, Job-19 is hosted by Worker-3 and during its lifetime, it is competing resources with two others, Job-12 and Job-21, where Job-12 is in $CL$ when Job-19 joins the system. With $DS$, however, Job-19 is assigned to Worker-2. It is competing with

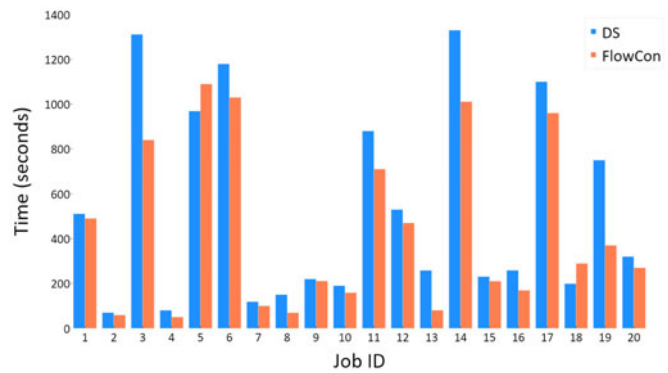



Fig. 13. 4-worker cluster with 20 jobs.

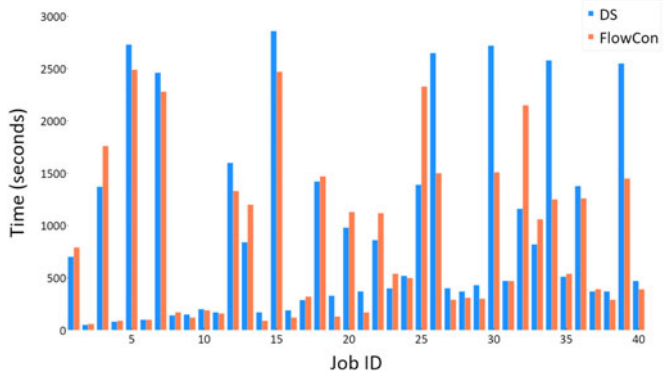



Fig. 14. 4-worker cluster with 40 jobs.

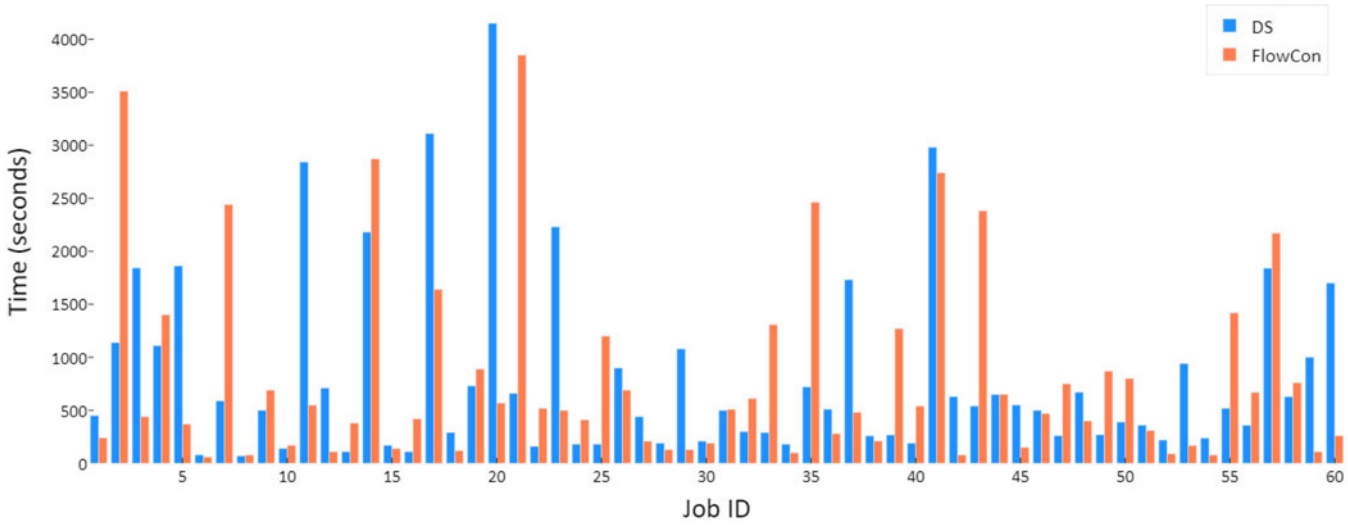



Fig. 15. 8-worker homogeneous cluster with 60 jobs.

four others, Job-3, -13, -21 and -27. *DS* fails to distinguish any jobs and therefore, each of the job gets a fair share of the resource. The biggest loss is discovered on Job-32 and it uses 85.4% more time to finish. In `FlowCon`, Job-32 is assigned to Worker-1 that takes 6 long-running jobs and with *DS*, it only has 5. When all containers are in *CL*, `FlowCon` enables free competition among them. The more jobs running concurrently, the less resource everyone gets.

Considering the whole system, `FlowCon` records an average reduction of 8.9% in completion time and the makespan is reduced 12.9%, from 2860.2 s to 2490.8 s.

*Scalability - More Workers (Homogeneous and Heterogeneous).* Besides the number of jobs, we increase the number of workers. In this experiment, we construct two clusters consists of 1 Manager and 8 Workers. The first one is a homogeneous cluster that all 9 servers have the same configuration, 4 vCPUs and 16 GB memory. The second one is a heterogeneous cluster that the manager has 4 vCPUs and 16 GB memory, the 8 workers are consists of 2 of the each of following 4 settings, (1) 2 vCPUs/8 GB, (2) 4 vCPUs/16 GB, (3) 8 vCPUs/32 GB and (4) 16 vCPUs/64 GB. The 60 jobs are randomly submitted to both clusters within a 0 to 1000 s submission interval. The increased number of workers provides more options to the manager when selecting hosts.

Fig. 15 presents the completion time from the homogeneous cluster. Overall, `FlowCon` is able to improve 51.7% of the jobs (31 out of 60). Individually, the performance varies significantly. For example, Job-59 reduced 89.0% in `FlowCon`, but Job-39 increased 3.7 times. The reason for the variations is that different container distribution plans results in imbalance at certain points. For example, For example, Job-39 is assigned to Worker-3 in *DS* and in the majority of its lifetime, there are 4 jobs running concurrently. With `FlowCon`, it is hosted by Worker-7 and there are 5 active jobs competing for resources. From the system point of view, the average completion time improves 1.2% and the makespan records a 7.2% reduction, from 4151.7 s to 3851.5 s.

Fig. 16 plots the results from the heterogeneous cluster. Overall, there are 32 out of 60 jobs (53.3%) get improvement with `FlowCon`. In average, the completion time reduces 0.8%. A similar trend is discovered in this experiment. The individual performance varies quite large. On one hand,

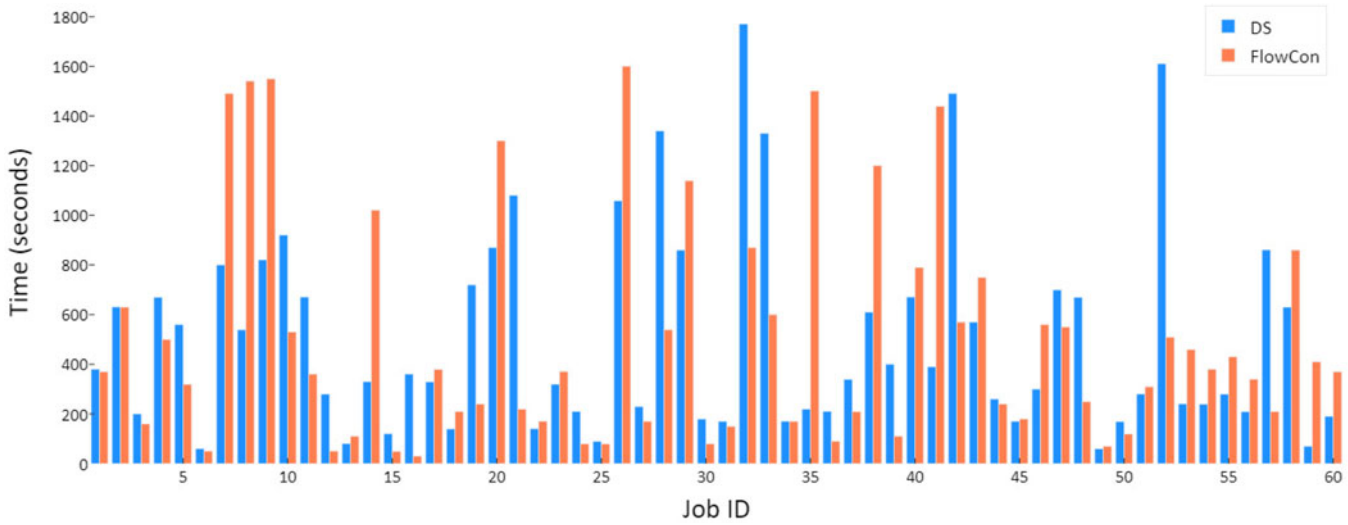



Fig. 16. 8-worker heterogeneous cluster with 60 jobs.

FlowCon is able to improve Job-16 by 91.2%. On the other hand, Job-35 records a 5.8x times increase. The reason lies in the fact that, with Docker Swarm, some workers get less workload than others and naturally, the jobs that runs on these workers complete faster. However, when considering the makespan, FlowCon is able to reduce 9.6% from 1770.4 s to 1600.3 s, which makes the system more efficient.

Compared with smaller-scale experiments, the degree of improvement reduced in the 60-job experiments. The reason lies in the fact that when the number of jobs increases, the resource contention increases, which naturally results in longer execution times. Consequently, the duration of all containers in Completing List is longer than in the previous experiments. When all the running jobs are categorized as converged, FlowCon utilizes a fair resource distribution mechanism such that on a specific worker, each job obtains an equal share of resources. At this converged stage, FlowCon works the same as *DS*. Therefore, the longer this stage is, the smaller the improvement. In the experiments presented by Figs. 15 and 16, the system entered converged stage at time 1304 s and 1097 s, which indicates that 62.0% and 31.4% of the whole experiments are in this stage.

## 7 Conclusion and Discussion

In this paper, we proposed FlowCon with a suite of algorithms to dynamically manage the cluster. With the respect to the proposed metric, growth efficiency, FlowCon aims to facilitate dynamic configuration and placement for containerized deep learning applications at runtime. FlowCon is built on top of Docker Swarm and evaluated on both academic infrastructures (NSF Cloudlab) and the commercial cloud platform (Google Cloud). We conducted intensive experiments with different deep learning models on two frameworks, Pytorch and Tensorflow. The experimental results demonstrate the effectiveness of FlowCon. Specifically, compared to a Docker Swarm, it has achieved significant performance improvement by up to 68.8% reduction in completion time for individual jobs and 18.0% in makespan.

Currently, the growth efficiency is based on the evaluation function, which is associated with deep learning models. As the future work, we plan to extend this design to other applications, such as quality of services (QoS) based workloads that other metrics can be utilized to measure the efficiency of resources. Additionally, besides computational resources, other types of resources, e.g., memory spaces and network bandwidth, should be evaluated. We plan to explore a fine-grained management on each container and implement a per-job threshold to customize the categorization process of the individual tasks. Furthermore, based on the growth efficiencies of each individual job, the theoretical trade-off analysis, provable performance guarantees, and boundaries as well as resource fairness should be studied.

## References

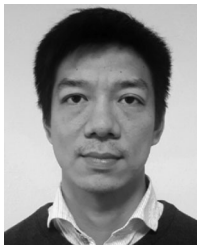

[1] Microsoft azure, 2022. [Online]. Available: https://azure.microsoft.com/en-us/
[2] Amazon, 2022. [Online]. Available: https://www.amazon.com/
[3] Morgan stanley, 2022. [Online]. Available: https://www.morganstanley.com/
[4] Standard poors, 2022. [Online]. Available: https://www.standardandpoors.com
[5] Docker, 2022. [Online]. Available: https://www.docker.com/
[6] Kubernetes, 2022. [Online]. Available: https://kubernetes.io/
[7] Pytorch, 2022. [Online]. Available: https://pytorch.org/
[8] Tensorflow, 2022. [Online]. Available: https://www.tensorflow.org/
[9] V. Sharma, "Dynamic resource management schemes for containerized deep learning applications," Master's thesis, Dept. Comput. Inform. Sci., Fordham Univ., New York, NY, USA, 2022.
[10] S. Minaee, Y. Y. Boykov, F. Porikli, A. J. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 7, pp. 3523–3542, Jul. 2022.
[11] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun, "Deep learning for 3D point clouds: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, no. 12, pp. 4338–4364, Dec. 2021.
[12] Q. Xie, M.-T. Luong, E. Hovy, and Q. V. Le, "Self-training with noisy student improves ImageNet classification," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 10687–10698.
[13] H. Touvron, M. Cord, A. Sablayrolles, G. Synnaeve, and H. Jégou, "Going deeper with image transformers," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2021, pp. 32–42.
[14] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Trans. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.
[15] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 7686–7695.
[16] B. Reddy, Y.-H. Kim, S. Yun, C. Seo, and J. Jang, "Real-time driver drowsiness detection for embedded system using model compression of deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2017, pp. 121–128.
[17] S. A. Stein *et al.*, "QuClassi: A hybrid deep neural network architecture based on quantum state fidelity," *Proc. Mach. Learn. Syst.*, vol. 4, pp. 251–264, 2022.
[18] W. Zheng *et al.*, "Target-based resource allocation for deep learning applications in a multi-tenancy system," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2019, pp. 1–7.
[19] Y. Mao, Y. Fu, W. Zheng, L. Cheng, Q. Liu, and D. Tao, "Speculative container scheduling for deep learning applications in a kubernetes cluster," *IEEE Syst. J.*, to be published, doi: 10.1109/JSYST.2021.3129974.
[20] Y. Mao *et al.*, "Differentiate quality of experience scheduling for deep learning inferences with docker containers in the cloud," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2022.3154117.
[21] W. Gao, Z. Ye, P. Sun, Y. Wen, and T. Zhang, "Chronus: A novel deadline-aware scheduler for deep learning training jobs," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 609–623.
[22] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 1049–1062.
[23] N. Krichevsky, R. St Louis, and T. Guo, "Quantifying and improving performance of distributed deep learning with cloud storage," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2021, pp. 99–109.
[24] K. Hazelwood *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2018, pp. 620–629.
[25] Y. Zhang, J. Yao, and H. Guan, "Intelligent cloud resource management with deep reinforcement learning," *IEEE Cloud Comput.*, vol. 4, no. 6, pp. 60–69, Nov./Dec. 2017.
[26] Y. Fu *et al.*, "Progress-based container scheduling for short-lived applications in a kubernetes cluster," in *Proc. IEEE Int. Conf. Big Data*, 2019, pp. 278–287.
[27] R. Gu *et al.*, "Liquid: Intelligent resource estimation and network-efficient scheduling for deep learning jobs on distributed GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2808–2820, Nov. 2022.
[28] S. Shi *et al.*, "Towards scalable distributed training of deep learning on public cloud clusters," *Proc. Mach. Learn. Syst.*, vol. 3, pp. 401–412, 2021.
[29] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.
[30] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu, "AlloX: Compute allocation in hybrid clusters," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.

[31] B. Chen *et al.*, "SLIDE: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems," *Proc. Mach. Learn. Syst.*, vol. 2, pp. 291–306, 2020.
[32] W. Zheng, M. Tynes, H. Gorelick, Y. Mao, L. Cheng, and Y. Hou, "FlowCon: Elastic flow configuration for containerized deep learning applications," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, Art. no. 87. [Online]. Available: https://doi.org/10.1145/3337821.3337868
[33] Deploy services to a swarm, 2022. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/
[34] Pods to nodes, 2022. [Online]. Available: https://docs.docker.com/engine/swarm/services/#placement-preferences
[35] Nsf cloudlab, 2022. [Online]. Available: https://cloudlab.us/
[36] Pytorch examples/dataset, 2022. [Online]. Available: https://github.com/pytorch/pytorch
[37] Tensorflow examples and datasets, 2022. [Online]. Available: https://github.com/floydhub/dl-docker
[38] Vae, 2022. [Online]. Available: https://jaan.io/what-is-variational-autoencoder-vae-tutorial/
[39] Mnist, 2022. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[40] Lstm, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Long_short-term_memory
[41] S. Zheng *et al.*, "Conditional random fields as recurrent neural networks," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 1529–1537.
[42] M. Berglund, T. Raiko, M. Honkala, L. Kärkkäinen, A. Vetek, and J. T. Karhunen, "Bidirectional recurrent neural networks as generative models," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 856–864.
[43] Gru. [Online]. Available: https://en.wikipedia.org/wiki/Gated_recurrent_unit

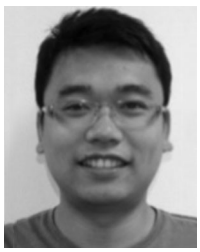
**Ying Mao** (Member, IEEE) received the PhD degree in computer science from the University of Massachusetts Boston, in 2016. He is an Assistant Professor with the Department of Computer and Information Science, Fordham University, New York City. He was a Fordham-IBM research fellow, in 2019. His research interests mainly focus on the fields of cloud computing, virtualization, resource management, data-intensive platforms and containerized applications.

**Vaishali Sharma** (Student Member, IEEE) received the bachelor's, master's, and PhD degrees in engineering from India, and 2nd master's degree in data science from Fordham University, in 2022. She was a data science research assistant with the Department of Computer and Information Science, Fordham University and was researching in the areas of distributed computing and cloud resource management. Her research work was focused on building/implementing the cluster management algorithms on virtualized computing platforms and accelerating the performance of deep learning and machine learning jobs on containerized cloud clusters. She is a technical reviewer of Scientific Reports, Elsevier and Springer Journals, an active IEEE student member, and 2020 IBM Data Science Foundations specialist.

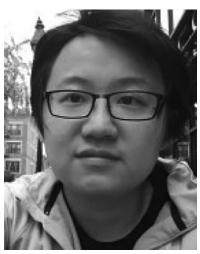
**Wenjia Zheng** (Student Member, IEEE) received the master of science degree in data science from Fordham University, in May 2020. She was a research assistant with the Department of Computer and Information Science, Fordham University. Her research mainly interests include deep learning, cloud computing and resource management for virtualized computing platforms.



**Long Cheng** (Senior Member, IEEE) received the BE degree from the Harbin Institute of Technology, China, in 2007, the MSc degree from the University of Duisburg-Essen, Germany, in 2010, and the PhD degree from the National University of Ireland Maynooth, in 2014. He is a full professor with the School of Control and Computer Engineering, North China Electric Power University, Beijing. He was an Assistant Professor with Dublin City University, and a Marie Curie fellow with University College Dublin. He also has worked with organizations such as Huawei Technologies Germany, IBM Research Dublin, TU Dresden and TU Eindhoven. He has published more than 70 papers in journals and conferences like *IEEE Transactions on Parallel and Distributed Systems*, *IEEE/ACM Transactions on Networking*, *IEEE Transactions on Computers*, *IEEE Transactions on Services Computing*, *IEEE Transactions on Automation Science and Engineering*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Big Data*, *IEEE Transactions on Intelligent Transportation Systems*, *IEEE Transactions on Very Large Scale Integration*, CIKM, ICPP, CCGrid and Euro-Par etc. His research focuses on distributed systems, deep learning, cloud computing and process mining. He is an associate editor of *Journal of Cloud Computing*.



**Qiang Guan** is an assistant professor with the Department of Computer Science, Kent State University, Kent, Ohio. He is the direct of Green Ubiquitous Autonomous Networking System Lab (GUANS). He is also a member of Brain Health Research Institute (BHRI) with Kent State University. He was a computer scientist in data science with Scale team at Los Alamos National Laboratory before joining KSU. His current research interests include fault tolerance design for HPC applications; HPC-Cloud hybrid system; virtual reality; quantum computing systems and applications.



**Ang Li** (Member, IEEE) received the bachelor's degree from the CS Department, Zhejiang University, China, in 2010, and the two PhD degrees from the Electrical and Computer Engineering (ECE) Department, National University of Singapore (NUS), Singapore, and the Electrical Engineering (EE) Department of Eindhoven University of Technology (TU/e), The Netherlands, in 2016. He is a senior computer scientist with the High-Performance-Computing (HPC) Group of Pacific Northwest National Laboratory (PNNL) since November 2016. His research has been focusing on software-hardware co-design for scalable heterogeneous HPC, particularly GPUs, since 2009. His research covers full-stack design from circuit level up to architecture, system, library, and applications.