Implementation of a Cluster-based Heterogeneous Edge Computing System for Resource Monitoring and Performance Evaluation

YU-WEI CHAN¹, HALIM FATHONI², HAO-YI YEN³, and CHAO-TUNG YANG⁴,⁵.

¹Department of Information Management, Providence University, Taichung City, Taiwan (e-mail: ywchan@gm.pu.edu.tw)
²Department of Industrial Engineering and Enterprise Information, Tunghai University, Taichung City, Taiwan (e-mail: halim.fathoni@gmail.com)
³Department of Computer Science, Tunghai University, Taichung City, Taiwan (e-mail: g07350009@thu.edu.tw)
⁴Department of Computer Science, Tunghai University, Taichung City, Taiwan (e-mail: ctyang@thu.edu.tw)
⁵Research Center for Smart Sustainable Circular Economy, Tunghai University, Taichung, Taiwan

Corresponding author: Chao-Tung Yang (e-mail: ctyang@thu.edu.tw).

ABSTRACT In the past decade, Internet of Things (IoT) technology has been widely used in various applications in people’s daily life. Currently, IoT applications mainly depend on the powerful cloud datacenters as the computing and storage centers. However, with such cloud-centric frameworks, a large amount of data will be transferred between the end devices and the remote cloud datacenters via a long wide-area network, which may potentially result in intolerable latency and a lot of energy consumption. To alleviate this problem, the edge computing (EC) paradigm is exploited to sink the cloud computing capability from the network core to network edges in proximity to end devices, so as to enable computation-intensive and latency-critical edge intelligence applications to be executed in a real-time manner. With the increasing amount of edge devices, it is essential to obtain the status of devices in real-time for realizing the overall resources of heterogeneous edge devices. Thus, it is important to construct a system which can be used to monitor each device’s status and obtain the performance of each device. In this study, a cluster-based heterogeneous edge computing environment is implemented for resource monitoring and performance evaluation. In the experiment, three deep learning models of object detection are used to evaluate the performance of the implemented system. Through the experimental results, we can easily realize the resource usage, including the high range dynamics of electricity and power consumption of heterogeneous edge devices via the visualization results.

INDEX TERMS Edge computing, resource monitoring, Kubernetes, Prometheus, Grafana.

I. INTRODUCTION

In recent years, IoT technology has been widely used in many fields such as smart cities, smart industries, smart medical care, and smart surveillance. In order to successfully deploy these services, a large number of IoT devices are required. According to statistics, the number of IoT devices will reach 125 billion by 2030 [1]. These IoT devices will generate large amounts of data. Coupled with the rise of Artificial Intelligence (AI), the massive amounts of data collected by IoT devices have opened up countless possibilities for AI applications.

In the traditional cloud-centric approach, centralized machine learning schemes are centrally executed in cloud-based servers or data centers [2]. However, the cloud-centric approach suffers from long propagation delay and can cause unacceptable latency in real-time application [3]. In addition, the massive data transmit to the cloud will burden the backbone network, consume a large amount of energy and cause privacy issues of users. To mitigate these challenges, a new computation paradigm has been proposed, which is called Edge Computing [4], [5]. The Edge Computing has naturally been proposed as a solution, in which the computing and storage capabilities of edge devices are used to make model training closer to where data is generated. Due to data processing at the edge of the network, such new computing paradigm improved data analysis and response time significantly [6].

Edge devices are commonly suitable for efficiently accel-
erating deep learning algorithms with low cost. To fulfill this requirement, some low-power edge devices with hardware accelerators for AI are developed. Among these products, a series of products of the NVIDIA Jetson are widely popular edge devices. These products contain the Graphics Processing Unit (GPU) hardware component and a parallel computing platform and application programming interface model, which is called CUDA (Compute Unified Device Architecture). With the platform, software developers can use a CUDA-enabled GPU for complex computations in AI. Intel and Google also develop a GPU-based accelerator for edge devices, such as Google coral and Intel Movidius Neural Compute Stick (NCS) that can integrate with Raspberry Pi or other devices. Thus, more heterogeneous edge devices are deployed to build an Edge Computing environment for deep learning applications [7], [8]. However, to build and maintain a cluster edge environment of having a large number of heterogeneous edge devices is a challenge due to the heterogeneity of these devices [9]. For instance, the Raspberry Pi 3B+ and Raspberry Pi 4 require ARMv6 and ARMv7 architectures, respectively while Jetson Nano and TX2 require ARM64 architectures. The packages and environments used in different architectures are also different.

Furthermore, it is very important to monitor the resource usage, power consumption of edge devices. In addition, the performance of inference model on edge devices is also needed to be monitored to let users easily realize the overall system and dynamic changes of resources in the operational environment. Thus, if we want to monitor the status of resource usage of various edge devices and evaluate the overall system performance, how to build an efficient, reliable and easy-to-use visualized environment is also a challenge. Since the data of resource and performance indicators for monitoring collected come from heterogeneous edge devices, how to integrate these devices and the collected data by considering the heterogeneity of devices for effectively building an easy-to-use resource monitoring platform is the main goal of this work. In our previous work [10], we have successfully proposed a container-based resource usage monitoring system for edge devices. However, only the basic architecture and functions were considered and implemented.

In this work, we implement a cluster-based Edge Computing system by integrating Docker [11], Kubernetes [12], Prometheus [13], Grafa na [14] and Node Exporter [15] software for monitoring the overall resource usage, energy consumption with the visualization techniques and evaluating the performance of deploying the proposed system on three object detection algorithms of AI, which are the Single Shot MultiBox Detector (SSD) [16], YOLO and Faster R-CNN [17], respectively. With Kubernetes, the cluster system can be easily established and deployed. With Node Exporter, the relevant performance indicators in various edge devices are efficiently collected. In addition, Prometheus integrates device backhaul performance indicators and stored in the database as the source information. Then, Grafa na is deployed as a visualized platform to show the operational status of each device in real-time and to run the deep learning module on the edge devices by observing changes in performance on the panel to verify the results of this visualized platform.

The main contribution of this work is that we successfully implement a cluster-based Edge Computing environment by integrating the following technologies, which are the Docker, Kubernetes, Prometheus, Grafa na and Node Exporter, to build an easy-to-use and friendly resource monitoring platform for effectively monitoring the status of resource usage of various edge devices and evaluating the overall system performance.

The rest of this paper is organized as follows. Section II introduces the literature review and related works. In Section III, the system architecture and implementations are presented. Experimental results are shown in Section IV. Finally, concluding remarks are given in Section V.

II. BACKGROUND REVIEW AND RELATED WORKS

In this section, background knowledge and related works are introduced, including Docker, Prometheus, Node Exporter, Grafa na, object detection, and related works.

A. DOCKER

Docker is an open-source software for developing, deploying and executing applications [11]. Docker allows users to separate applications in the system environment to form smaller containers, thereby increasing the speed of software deployment. It is designed to simplify and standardize deployment methods in a variety of environments, and it contributes significantly to adopt this service design and management style. The software has the following advantages:

1) Lightweight resource utilization: Containers isolate and use the host’s kernel at the process level, rather than virtualizing the entire operating system.
2) Portability: All the environment parameters of the containerized application are bundled into a container, allowing it to run on any Docker host.
3) Predictability: Hosts don’t care what’s running inside the container, and the container doesn’t care which host it runs on. All interfaces are standardized and the interactions is predictable.

Docker containers are similar to virtual machines, but in principle, containers virtualize operating systems while virtual machines virtualize hardware. Thus, containers are more portable and consume less system resources than virtual machines [6]. A core operating system is run independently, inside the container so that it can be deployed in different system environments regardless of the differences in various system environments.

B. PROMETHEOUS

Prometheus is an open-source software for environmental monitoring and alerting [13]. It records real-time indicators in a Time-Series DataBase (TSDB) built using the http pull model and has flexible query and instant alerting functions.
The Prometheus ecosystem consists of multiple components, which are mainly presented as (1) the Prometheus server which scrapes and stores time series data. (2) the client libraries are used to detect application codes. (3) an alert manager is used to handle alerts.

C. NODE EXPORTER

Node Exporter is a small index collection program, which is mainly used as a data source for Prometheus [15]. It captures different index data in real-time and constructs an http pull model to provide Prometheus for data collection and storage.

D. GRAFANA

Grafana is an open-source system for multi-platform analysis and interactive visualization [14]. Grafana supports a considerable number of data sources to facilitate the visualization of data. When it is connected to data sources, it will provide charts, graphs, and alarms for the Web. It also provides the conversion of the TSDB data into exquisite graphics and visual effects tools and can be expanded through the plug-in system. Users can use the interactive query builder to create sophisticated monitoring dashboards.

E. OBJECT DETECTION

1) YOLO

YOLO stands for “You Only Look Once” is a real-time object detection system [18] that applies a single deep neural network to the full image. The neural network divides an image into regions and predicts bounding boxes and probabilities for a region in the image. For object classification and detection, YOLO provides pretrained models for implementing the image recognition algorithms. In this experiment of this study, we use the COCO dataset which has 80 categories to evaluate the implementation of our proposed system.

2) Faster R-CNN

The architecture of Faster R-CNN has several moving parts that make it become a complex algorithm [17]. For object detection of an image, the following steps are performed: (1) A list of bounding boxes are searched and marked. (2) A label is assigned to each bounding box. (3) The probability values are obtained with respect to each label and bounding box. Each image which is represented by height x width x depth is processed through a pretrained CNN through the intermediate layers. Finally, a convolutional feature map is obtained. In addition, to find a predefined number of regions (also called bounding boxes) which may contain objects, Faster R-CNN method uses a feature which was processed by CNN algorithms, that is commonly called Region Proposal Network (RPN). Generating a variable-length list of bounding boxes is one of the issues in Object Detection. In addition, using anchors in RPN is one of the solutions to solve these issues. Anchors is a fixed-sized reference bounding box, which is placed uniformly throughout the original image. This anchor is to answer the question: “Inside the anchor any relevant object” and “how to adjust the anchor better to fit the relevant object” [17]. Then, region of interest will apply bounding boxes with relevant objects and also extract feature which would correspond to the relevant object into new tensors.

3) Single Shoot Multi-box Detector

Feed-Forward Convolutional Network produces a fixed-size collection of bounding boxes. Also, scores for the presence of object class instance in the boxes, followed by a non-maximum suppression step to produce the final detection results, become a base of Single Shoot Multibox Detector (SSD) [16]. The important feature of SSD is applied to multi-scale convolutional bounding box output attached to multiple feature maps at the top of the network. The SSD creates an efficient space of box shapes and also improves the performance.

4) DeepStream

DeepStream SDK provides AI-powered intelligent video analytics applications and services, which are suitable as streaming analytics toolkits [19]. The DeepStream SDK used the Gstreamer framework, designed to write an audio and video application easier to linked and arranged in a pipeline. Further, this pipeline will then defined as the flow of data. The core function of Gstreamer is to provide a framework for plugins, data flow, and media type handling. Finally, the Deepstream SDK can be optimized to build end-to-end AI-powered applications for analyzing video and sensor data.

F. RELATED WORKS

In the following, some related works are given to let readers realize recent related studies, which are about the research of performance evaluation and resource monitoring. In [20], Nikhil Marathe et al. analyze Docker and other containers, which help this study to specifically realize the Docker Swarm and Kubernetes technology, and show how to access the cluster node service through the help of Docker swarm and Kubernetes, and explain the differences between them.

Nitin Sukhija et al. [21] proposed an active monitoring and management data center operation architecture that can scale to accommodate the heterogeneity and complexity of the new generation of systems. The proposed architecture of this work enabled large-scale active monitoring and management by integrating the latest technologies such as Kubernetes, Prometheus, Grafana, and other predictive platforms with data. This comprehensive infrastructure helped centralized services and deployment coordination, automatically analyze streaming data, correlate data from multiple sources, and set thresholds for alarms to determine core issues from a single visualized graph.

With regard to the related works of resource monitoring. In [22], Wenyan Chen et al. proposed two popular monitoring tools, which are the Perf and Prometheus, to explore the various features of the micro-architecture and the application level of parallel workloads running in the containers of the
same server. They believe that workloads play an essential role in resource allocation and performance optimization. Their research focus is to quantify the interference caused by workloads by analyzing the characteristics of workloads that are run separately at the same location. In [23], authors formulated a scheduling problem to optimize the framework and proposed an efficient heuristics algorithm based on the simulated annealing strategy. The results show that their frameworks increase the monitoring frame rate up to 10 times and reduce the detection delay up to 85% comparing to the cloud monitoring solution.

In addition, for Edge Computing, one of the critical issues is the power consumption. Since most of the edge devices are often resource-constrained, it is difficult for edge devices to run deep learning applications, which often need a large amount of computation and energy. Some related works of power consumption on edge devices are introduced in the following.

In [24], authors present a novel time-energy-cost analysis of wimpy Edge Computing in comparison with traditional brawny cloud computing. The researcher uses brawny heterogeneous Amazon EC2 and Jetson TK1 and TX1 as a wimpy heterogeneous system. The results show that Jetson TX1 has worse time-cost performance compared to Jetson TK1 systems, this is because that Jetson TX1 has lower operating core clock frequency and lower instructions-per-cycle of Jetson TX1’s GPU on some compute-intensive applications compared with Jetson TK1. This work [25] also studied the performance analysis with considering energy efficiency. In this study, two two matrices are used to measure energy consumption. To illustrate the use of total energy, a medium scale of HPC system was set up in the experiment. The experimental results show that the underlying architecture and the programming model are the crucial factors to the performance and energy efficiency.

In [26], authors investigated the power consumption of deep learning applications on Embedded GPU systems. They proposed to use the YOLO methods to perform a real-time object detection algorithm on Jetson TX1 and TX2. They also implemented low-power image recognition challenges on evaluating the system. The experimental results showed that Jetson TX2 with Max-N mode had the highest throughput as well as the most efficiency. The results also indicated the trade-off between the throughput and power efficiency which could be adjusted from observing the edge devices TX2. This work [27] proposed a deep learning-based method for detecting traffic flow on the edge node. First, the authors provided a vehicle detection algorithm based on the YOLO v3 model trained with a significant amount of traffic data. This model was pruned to ensure that it was effective on edge devices. Then, A real-time vehicle tracking counter that combines vehicle detection and tracking algorithms to detect traffic flow is proposed. Finally, the author migrated and deployed both the vehicle detection network and multiple object tracking network to the Jetson TX2 platform. On the edge devices, the test results showed that the model could detect traffic flow with an average processing speed of 37.9 FPS and an average accuracy of 92.0%.

In [28], heterogeneous edge devices, which were the Jetson Xavier, Jetson TX2 and Jetson Nano, were used to evaluate the power consumption and processing frame rates. The experimental results showed that a high performance was required while the power consumption was low on a mobile robot. In addition, the results showed that Jetson TX2 had the best power efficiency with comparing to Xavier and Nano. In [29], authors proposed a method to reduce energy consumption without compromising accuracy and frame rate. In this work, a Google coral USB accelerator and a Raspberry Pi 4 devices are used. The experimental results showed that the accuracy can reach 62.3%. In addition, running the CNN in this work is faster and more efficient than the one in a tiny-YOLO network. In [30], a power-efficient layer mapping technique for CNNs deployed on integrated CPUs and GPUs. The experiments on NVIDIA Jetson TX2 demonstrated that layer mapping of YOLO V3-Tiny has an impact on power consumption. From the experimental results, we found that (1) almost all of the convolution layers were not suitable for mapping to CPUs. (2) the pooling layer could be mapped to CPU to reduce power consumption, but a larger output tensor could result in decreased inference speed. (3) the detection layer could be mapped to CPU as long as its floating-point operations were not too large, and (4) the channel and upsampling layers were both suitable for mapping to CPUs. This study provided information that can be used to develop power-efficient layer mapping strategies for integrating the CPU and GPU platforms.

III. SYSTEM ARCHITECTURE AND IMPLEMENTATION

In the section, we will present the system architecture and its implementation. First, the software architecture and its components of the system are introduced. Then, the main components which are the master node and the worker node are also introduced, respectively. Finally, the system implementations of the proposed system are presented specifically.

A. SYSTEM ARCHITECTURE

The overall software architecture of the proposed system is shown in Fig. 1. The proposed system is divided into two parts, the first one is the master node, which consists of four components that will be introduced in detail next. The second one is the worker node, which consists of three components that will also be elaborated in the following section.

In this work, a heterogeneous EC system is implemented, in which the hardware equipment of Raspberry Pi 3B+, Raspberry Pi 4, NVIDIA Jetson Nano and NVIDIA Jetson TX2 are deployed. In the system, the open-source software is employed to establish a containerized system environment. The construction steps of the proposed system, as shown in Fig. 2, are listed as follows:

- Step1: The Kubernetes and Docker software is used to establish a containerized system environment and man-
age the running status and updates between containers to achieve a high-quality service environment.

• Step2: Prometheus and Grafana software is deployed on the server host to serve as a monitoring tool for system index collection and visual presentation.

• Step3: Prometheus Node Exporter software deploys the containerization to Edge Computing devices to collect the internal system indicators of each device, and then monitor the resource status of each device through Prometheus for returning the indicator data and storing them in the Time Series Database (TSDB).

• Step4: The resource usage status of the devices in the Edge Computing environment are presented visually through the Grafana software.

• Step5: The models trained on the server host are finally deployed on edge devices to verify the monitored performance changes.

B. THE MASTER AND WORKER NODES

In this section, we will first introduce the main construction components of the system environment. Then, we will introduce the operations and service deployments with respect to the master and worker nodes, respectively.

1) The System Environment Construction and Service Deployment

With regard to the system environment construction, the open-source software is used to establish a containerized system environment. First, the Kubernetes and the Docker software is used to establish a cluster environment, in which the server host serves as the master node in the cluster while the edge devices serve as the cluster’s worker nodes to provide the schedule of service allocation. The master node acts as the primary control plane for the Kubernetes cluster. It also acts as the primary contact point for administrators and users. In addition, it also provides a number of cluster-wide systems for relatively simple working nodes. In general, components on the primary server can work together to accept user requests, determine the best way to schedule workload containers, authenticate clients and nodes, adjust cluster-wide networks and manage scaling and health check responsibilities. These components can be installed on a single computer or distributed across multiple servers. In the following, components and operations of the master and the worker nodes are elaborated in detail, respectively.

2) The Master Node

The components of the master node are introduced as follows:

1) Application Programming Interface (API) server: The API server component is one of the most important primary services. This is the primary management point for the entire cluster since it allows the users to configure the workload and organization units of Kubernetes. It is also responsible for ensuring that the service details of the etcd store and the deployed containers which are consistent. In addition, it acts as a bridge between various components to maintain health of the cluster and to propagate information and commands. The API server implements the RESTful API interface, which indicates that many different tools and libraries can easily communicate with it. As the default method for local computers to interact with the sets of Kubernetes, you can use a client called kubectl.

2) Etcd: Etcd is an open source, distributed, consistent key-value store, used for shared configuration, service discovery, and scheduler coordination of distributed systems. It is the primary datastore of Kubernetes. As the primary datastore of Kubernetes, etcd stores and replicates the state of all Kubernetes clusters. Since it is a key component of a Kubernetes cluster, it is very important for etcd to have reliable configuration and management methods. Thus, the component of etcd is used to store configuration data, which is accessible to each node in the cluster. This can be used for configuring/re-configuring components based on the latest information and for service discovery. In addition, the component is also used to get features, like leader elections and distributed locking for maintaining...
the status of cluster. Etcd can be configured on a single primary server or between multiple machines in a production scenario. To provide a simple http/json API, the interface of setting or retrieving values of the component is also designed to be simple.

3) Controller-Manager: The component of controller manager is responsible for processing and managing many tasks. It manages different controllers which regulate the state of the cluster, manage the life cycle of workloads and perform routine tasks. For instance, the replication controller ensures that the number of copies defined for the Pod corresponds to the number of current deployments on the cluster. Details of these operations are written to etcd, where the controller manager monitors changes through the API server.

4) Scheduler: The component of scheduler is responsible for assigning workloads to specific nodes in the cluster. It reads the operational requirements of the workloads, analyzes the current infrastructure environment and places works on one or more acceptable nodes. Scheduler is also responsible for tracking the available capacity of each host to ensure that the workloads are not larger than the available resources. In addition, the component of scheduler has to know the total capacity and the resources that have been allocated to the existing workloads of each server.

3) The Worker Node

In the system, the worker node consists of three components, which are stated as follows:

1) Kubelet: The kubelet component in the worker node provides services of the primary contacting point for each node in a cluster group. The component is responsible for transmitting information forward and back with the control plane services. Also, it is used to interact with the etcd store to read configuration values or to write new values in detail. In addition, the Kubelet service communicates with the primary component for authentication in the cluster and receives the commands and work. The received work acts as a manifest which defines the amount of work and operational parameters. Then, the Kubelet component is responsible for maintaining the working state on the server node. It controls the container’s operations of start or destroy that the container needed.

2) Proxy: To manage separate segmentation of host’s subnet and to make the services available to other components, the proxy component is needed to run the proxy services on each server node. The proxy component forwards requests to the correct container, performs load balancing. In addition, the component is usually responsible for ensuring that the network environment is predictable and accessible, but isolated where appropriate.

3) The Container Runtime: The first component that each node must have is the container runtime. Typically, this requirement can be met by installing and running Docker, but alternatives such as rkt and runc can also be used. Containers are responsible for starting and managing containers when they run, and these applications are encapsulated in a relatively isolated but lightweighted operating environment. Each unit of work in the cluster is implemented at its base level as one or more containers that must be deployed. The container runtime on each node is the component that eventually runs the container defined in the workload submitted to the cluster.

The main components of the master and worker nodes are described above. Next, the processes of deploying services throughout the cluster system are presented. First, when the cluster wants to schedule the services to worker node, the user inputs the instructions to establish the Pods through kubectl. The instructions are then authenticated by the users and passed to the API server in the master node, which backup the instructions to etcd. Second, the controller-manager then receives a message from the API server which needs to create a new Pod and checks that a new Pod will be built if the resources are allowed. Finally, when the scheduler visits the API server on a regular basis, it asks the controller-manager if a new Pod is built or found, the scheduler will be responsible for delivering the pod to the most suitable node. Although the processes in practice seems to be complex, the Kubernetes will automatically complete the subsequent deployment actions. The cluster architecture of Kubernetes is shown in Fig 3.

![Figure 3: The Kubernetes cluster architecture.](image-url)
C. SYSTEM IMPLEMENTATION

In the system, a monitoring system is built on the edge server to collect all the data of resource indicators of edge devices. We use Prometheus to collect and store all the data of resource indicators of edge devices. In addition, Grafana is used to visualize the data of resource indicators as well as the power consumption data.

1) Acquisition of Resource Indicators

In the system, to provide Prometheus the index data of standard format for storage, the system uses the node exporter as a service to collect and provide a standard format index. The heterogeneous equipment need to deploy node exporter with different environmental architectures, such as Raspberry Pi3 and 4, which use the ARMv6 and ARMv7 architectures, respectively. The NVIDIA Jetson Nano and Jetson TX2 use the ARM64 architecture. Each device node uses the deployed node exporter to collect internal resource indicators, and it exports each resource indicator retrieved according to internal settings to the corresponding port (its default value is 9100). The derived resource indicators are mainly the CPU usage, memory usage, system load and so on, which are shown in Figure 4.

![FIGURE 4. The resource index from the implementations of node exporter.](image)

2) The Resource Indicators of Unified Storage

After obtaining the data of resource indicators of various equipment, the data are then be stored to databases for the monitoring and the usages of other services. This work uses Prometheus as the database for data integration and storage, which is deployed on the listening device to consolidate and store the data. Prometheus stores the physical returned indicators in the time series database. In addition to effectively controlling the data storage, Prometheus can easily control the storage time through settings. Fig. 6 shows the implementations of deploying Prometheus on the listening device for collecting the resource indicators from the node exporter.

![FIGURE 6. An example of deploying Prometheus for collecting the resource indicators from the node exporter.](image)

HTTP requests. In general, the indicator name is composed of ASCII characters, numbers, underscores, and colons. The definition of indicator name has to meet the rules of regular expression [a-z,A-Z;][a-z,A-Z,0-9;]*.

- Label: The label makes the same time series samples have different dimensions of identity. For instance, the label http_requests_totalmethod="Get" represents the get transmission method of user’s requests. The other transmission method is the post method. The keys in the label are composed of ASCII characters, numbers and underscores. The definition of label has to meet the rules of regular expression [a-z,A-Z;][a-z,A-Z,0-9;]*.

- Samples: The samples represent the actual time series, each series includes a floating-point values of float64 and a millisecond timestamp.

Prometheus will use the data of indicators as the time-series data which will be stored. According to the need of users, the data of indicators can be classified based on the index mining of entities. The example of setting index mining of entities by using Prometheus is shown in Fig. 5.

![FIGURE 5. An illustrative example of setting index mining of entities by using Prometheus.](image)

After setting up the index mining of entities, the resource index is then collected via the node exporter which is deployed on the listening device to consolidate and store the data. Prometheus stores the physical returned indicators in the time series database. In addition to effectively controlling the data storage, Prometheus can easily control the storage time through settings. Fig. 6 shows the implementations of deploying Prometheus on the listening device for collecting the resource indicators from the node exporter.

Prometheus also provides a simple data search interface. A query needs the information of the required resource indicators by inputting the corresponding indicator grid. For
instance, by inputting `go_info` to view the information of the entity name and task classification of each monitoring device, version and other information.

3) Data Visualization of Resource Indicators
After collecting and storing data by Prometheus, the Grafana open-source software is deployed on the server to provide the visualization environment with respect to different resource indicators of edge devices. In this work, Prometheus is responsible for collecting data and Grafana is then used to set up a database to store the data of resource indicators. The setting interface of data sources with Prometheus is shown in Fig. 7.

To make the data of resource indicators visualize, a panel is needed to perform the indexed-based syntax query with a simple counter to get the number of CPU corresponding to the physical devices. In addition, the panel is not limited to the visualization of one indicator. The data come from the multiple different types of indicators can be visualized on the same panel to show the changes of these indicators. The visualization with respect to multiple different types of indicators on the same panel is shown in Fig. 8.

In this study, Grafana is used to integrate different types of panels into a dashboard interface, in which different resource indicators of physical devices are visualized with appropriate panels which are integrated into an interface to provide the monitoring and evaluation with respect to the usage of resources of edge devices. Fig. 9 shows the dashboard of the system implementations with respect to different types of resource indicators.

D. THE EXPERIMENTAL WORKFLOW
In the following, Fig. 10 shows the experimental workflow of this system. The experiment is run by launching various object detection algorithms on NVIDIA Jetson devices, and by classifying the images on the Raspberry Pi 4 devices with NCS. Three different object detection algorithms, that are the Faster-CNN, SSD, and YOLO v3 use the DeepStream pipeline. In addition, the Raspberry Pi 4 device also uses the NCS to run the Inception-V3, VGG16, and MobileNet algorithms. In the following experiment, we will show the experimental results with the integration of Kubernetes, Prometheus, Grafana and Node Exporter software.

IV. EXPERIMENTAL RESULTS
In this section, experimental results are presented to illustrate the analysis results of the implemented system.

A. THE SETTINGS OF EXPERIMENTAL ENVIRONMENT
In this work, one server host and four edge devices are integrated, in which the server host acts as the master node in the Kubernetes cluster architecture that deploys Prometheus and Grafana software to manage container services of edge devices. The experimental environment of Kubernetes cluster architecture is shown in Fig. 11. The hardware specifications of the master and worker nodes are presented in Table 1. The deployed software specifications are given in Table 2.

To build a Kubernetes cluster, the Kubernetes master node is first built, which is added as a working node by initializing the cluster token to the edge devices generated by initializing the basic Kubernetes suite. Then, four edge devices, which are the Raspberry Pi 3, Raspberry Pi 4, Jetson Nano and Jetson TX2, respectively, installed the Kubernetes suite as the working nodes but did not initialize, and then the individual token generated by the master was entered to join the cluster to complete the initial setup of the entire system cluster.

B. DATA COLLECTION AND VISUALIZATION
In this study, the data collected from four heterogeneous edge devices, which are the NVIDIA Jetson Nano, NVIDIA Jetson TX2, Raspberry Pi 3, and Raspberry Pi 4 are visualized through Prometheus. In addition, the resources usage of these devices are also monitored with Prometheus. Among the devices, the Raspberry Pi 4 device is used as the verification object to observe the status of resources usage after executing the deep learning methods. The resources include CPU, memory and the average system load. In the implemented system, the usage of resources of devices are monitored with the proposed three dashboards, which are (1) the resource monitoring panel. (2) the device’s health status panel. (3) the parallel monitoring panel. An illustrative example is that we...
The monitoring of CPU resource usage

The monitoring of disk, memory and network traffic resources usage.

FIGURE 9. An Illustrative example of the dashboard with respect to different types of resource indicators of edge devices.

FIGURE 10. Workflow for Experiment.

FIGURE 11. The Kubernetes cluster architecture in the experimental environment.

In this work, the performance metrics we use are mainly referred to the study [25] which uses two metrics to measure the energy consumption, they are the idle power, denoted...
**TABLE 1.** The hardware specification of the proposed system.

| Item                  | Core                                      | Disk  | Ram  |
|-----------------------|-------------------------------------------|-------|------|
| Web Server            | Intel(R) Core(TM) i7-4790                  | 1TB   | 16G  |
| Raspberry Pi 3 B+     | Broadcom BCM2837/B0                        | 32GB  | 1GB  |
| Raspberry Pi 4        | Broadcom BCM2711 Quad core Cortex-A72 (ARM v8) 64-bit | 32GB  | 4GB  |
| NVIDIA Jetson Nano    | Quad-core ARM Cortex-A57 MPCore processor | 32GB  | 4G   |
| NVIDIA Jetson TX2     | Dual-Core NVIDIA Denver 2 64-Bit CPU      | 32GB  | 8G   |

**FIGURE 12.** The status monitoring of the resource usage of Jetson Nano while running the SSD object detection methods.

**TABLE 2.** The software specifications of the proposed system.

| Software version         |
|--------------------------|
| Ubuntu 18.04             |
| Kubernetes 1.14.3        |
| Docker CE 19.03.8        |
| Node Exporter 18.1       |
| Prometheus 2.17.0-rc.3   |
| Grafana 6.7.1            |
| Python 3.6.8             |
| Tensorflow 1.12          |
| NVIDIA DeepStream 4.0    |

as $P_{idle}$, and the processing power, denoted as $P_{processing}$, respectively. The idle power indicates the consumed power when the edge device runs all the system functions in the background. Thus, it does not run any object detection applications. The processing power $P_{processing}$ indicates that the consumed power when the edge device runs the object detection applications. The total power, denoted as $P_{total}$, is defined as the sum of the idle power and the processing power. All the processes will be recorded in the database and also will be visualized using Grafana, as shown in Fig. 14. In addition, to calculate the energy consumed run in the object detection applications, which is denoted as $E_{processing}$ is calculated by Eq. (1).

$$E_{processing} = \int_{t_{start}}^{t_{end}} P_{processing}(t) \, dt,$$

(1)

where $t_{start}$ and $t_{end}$ represent the time when the application starts and finishes.

The energy consumed for system idle, denoted as $E_{idle}$, is calculated as presented in Eq. (2).

$$E_{idle} = \int_{t_{start}}^{t_{end}} P_{idle}(t) \, dt.$$  

(2)

The total energy consumed, denoted as $E_{total}$, is calculated as $E_{total} = E_{idle} + E_{processing}$. The energy-performance efficiency is calculated and evaluated by using the product of the total consumed energy and the time amount of delay, which represents the performance. In this work, we use the energy-performance efficiency metric to prevent from choosing configurations that achieve faster execution time while much more energy is consumed.
C. MODEL DEPLOYMENT AND VALIDATION

In this study, we deploy three deep learning models, which are the Inception-v3, VGG16 and MobileNet on the Raspberry Pi 4 to validate the performance of the implemented system. The three models are firstly performed for vehicle type identification. The data set used in the study is based on the hottest wheels of stolen cars in the United States in 2017 [33]. Thus, we only select the first ten classes to shorten the training time.

1) Honda Civic (1998): 45,062
2) Honda Accord (1997): 43,764
3) Ford F 150 (2006): 35,105
4) Chevrolet Silverado (2004): 30,056
5) Toyota Camry (2017): 17,276
6) Nissan Altima (2016): 13,358
7) Toyota Corolla (2016): 12,337
8) Dodge/Ram Pickup (2001): 12,004
9) GMC Sierra (2017): 10,865
10) Chevrolet Impala (2008): 9,487

The data represent the number of stolen cars per model in 2017. Automotive images are extracted from the Vehicle Manufacturing and Model Recognition Data Set (VMMRdb). Multi-year vehicles are then mapped to the stolen car category (based on external similarity) to provide more samples. These images consist of 6877 files that contain 10 classes based on the selected category. For training purposes, the data set is divided into three types for training, validation and testing with 70%, 10% and 20% percentages. The training dataset consists of 5098 images from 10 classes. The validation data set consists of 586 images from 10 classes. The testing data set consists of 1,193 images that belong to 10 classes. Then, the models are performed with FP16 and FP32 to observe the changes of resources when running the optimization models before and after.

Finally, the method of data set optimization is presented. With regard to optimization of data set, the work uses open AI software to accelerate the performance of training and inferring. The OpenVINO toolkit developed by Intel facilitates the deployment of inference processing models by converting and optimizing training models for any downstream hardware target. The toolkit supports CPU trained TensorFlow, Caffe and MXNet, integrated GPUs, VPU (Movidius Meriad 2/Nerve Computing Stick) and FPGA-trained models. The system optimizes the deep learning library using the TensorFlow framework. The data analytics acceleration Library and Intel Python Distribution are the basic building blocks of machine learning. The deep neural network open source library contains CPU optimization capabilities.

In the implemented system, the resource changes of CPU and memory when running the Inception-v3, VGG16, and MobileNet three unoptimized models with FP 16 and FP32 are shown in Fig. 15 and Fig. 16, respectively. On the other hand, the resource changes of CPU and memory when running the three optimized models with FP 16 and FP32 are then shown in Figs. 17 and 18. From these figures, we see that the status of resources change can be monitored. In terms of CPU usage, there was no significant difference between FP16 and FP32 before and after optimization, with total CPU usage: VGG16 (15%), InceptionV3 (20%), and MobileNet (33%). After optimization, the more obvious difference for the overall resource change curve becomes more flat. The use
of resources in memory is not too obvious curve, it can be seen that the demand for memory is small.

FIGURE 15. The resource changes of CPU when running the three un-optimized models with FP16 and FP32.

FIGURE 16. The resource changes of memory when running the three unoptimized models with FP16 and FP32.

FIGURE 17. The resource changes of CPU when running the three optimized models with FP16 and FP32.

FIGURE 18. The resource changes of memory when running the three optimized models with FP16 and FP32.

The experimental results show that the internal resource usage information of heterogeneous devices collected through Prometheus can be presented in real-time with Grafana. With the developed monitoring environment, the time of using CPU and memory when running three training models in an edge device can be measured. The status of resources usage can also be monitored and analyzed. In addition, the differences in resource consumption and accuracy between models can also be evaluated in this study. The experimental results verify the availability and effectiveness of the proposed system.

D. DISCUSSION

To the best of our knowledge, there is no other similar research works like the work we proposed currently. Thus, we only compared to the state-of-the-art works focusing on the performance monitoring of using tools to obtain parameters when edge devices run the deep learning algorithms. In [34], authors proposed Perf4sight which was a method that allowed the combinations of various networks, devices and frameworks to analyze the memory consumption and latency of training. In the work, models were built to predict memory consumption and latency of training for the Jetson TX2 devices and PyTorch frameworks using decision trees, respectively, with a mean error of 5.53% and 9.37%. These error rates significantly improved those obtained when modeling the same attributes by using CNN training technique.

In addition, this study [35] aimed to learn about the inference workflow and performance of the YOLO network by using the edge devices of Jetson Nano, Jetson Xavier NX, and Raspberry Pi 4B with NCS. In this benchmark, two versions of the YOLO network were used to predict different video contents across the three edge devices mentioned above. In the work, the performance of the three edge devices were compared, and their respective resource characteristics were discussed and evaluated. Furthermore, practical recommendations were given to indicate how to deploy AI applications on each of these intelligence edges. In our proposed work, we have successfully implemented a cluster-based edge computing environment by integrating the open source software, such as Docker, Kubernetes, Prometheus, Grafana and Node Exporter to build an easy-to-use resource monitoring platform for effectively monitoring the usage status of memory and CPU resources in various edge devices and evaluating the overall system performance. Currently, there is no other related works like the work we proposed that successfully integrate the open source software to build the resource monitoring and evaluation platform for edge computing environments.

V. CONCLUSION

In this paper, we have implemented an integrated performance evaluation and resource monitoring system for heterogeneous edge devices. In the system, we have successfully deployed the Docker software to build a containerized environment and deployed the Kubernetes software to build a Edge Computing-based cluster environment. With Docker’s lightweight and fast-deploying container services, tedious manual environment construction is no longer required. With Kubernetes, the system could restart automatically the system before exception, so as to solve the hassle of having to reinstall the environment. In addition, we have also suc-
uccessfully deployed Prometheus, Grafana and Node Exporter software to build a visualized resource usage monitoring and performance evaluation system with respect to edge devices. In terms of monitoring of resources usage, the TSDB of Prometheus could effectively control the length of stored data and provided connections. The resource indicators that Node Exporter could collect different types of data. Grafana had a good effect on visualization presentation. It was not only a variety of visualization tool, but also a suitable tool for presenting the various data sources. The high dynamic range of electricity could be used to optimize the power consumption of edge devices, and those fluctuation could also be monitored using Grafana.

In the absence of Docker and Kubernetes, Prometheus, Node Exporter, Grafana all had to issue multiple installation instructions and parameter sets that could be used on the server and device side. Regardless of the installer’s complexity, in management and maintenance, if an environmental anomaly occurred or the need of new devices, either the environment had to be reinstalled, or the corresponding hardware architecture had to be used to find a matching software version. With the features of Kubernetes and Docker, the entire monitoring environment could be managed and quickly deployed. With the implemented system, it was easy to install the suitable software if a new edge device joined the cluster environment.

In summary, this work provided a high-quality service deployment and resource monitoring solution. In this work, the pre-models were used in the experiments to verify the effectiveness of the implemented system. The monitoring system was very important, not just only for deep learning models, such as big data analysis, distributed system, and parallel computing, all were needed to monitor and evaluate various devices’ internal resources. Thus, the proposed monitoring platform in this work was to provide developers more easy and convenient development and visualization system.

The future works will include the following studies. First, the cloud-based architecture will be included in the system to build a cloud-edge collaboration architecture, in which the cloud-based architecture has powerful computing, storage and network resources while the edge-based architecture has the capability of real-time analysis and responses. Thus, the cloud-edge collaboration architecture will be more beneficial for edge intelligence applications. Second, by running simulation on edge devices in this work, deep learning modules are performed on only one device. Thus, deep learning applications are performed under the same architecture. In the future, model transformations are performed on a cross-device architecture. Finally, with the Kubeflow platform, which is a deep learning platform launched by Kubernetes, we will integrate Kubeflow to establish environmental components that are compatible with Kubernetes as the deep learning platform of our proposed system. Therefore, the whole system management and deployment will be more convenient and efficient in the deep learning applications.

ACKNOWLEDGMENT

This work was supported in part by the Ministry of Science and Technology, Taiwan, under Grant MOST 110-2621-M-029-003-, 110-2221-E-029-002-MY3, 110-2221-E-126-004- and 110-2622-E-029-003-.

REFERENCES

[1] H. J., “Number of connected iot devices will surge to 125 billion by 2030,” IHS Markit Technology, 2017.
[2] S. Banjumali, O. P. Pratap, P. Kuvaja, and M. Oivo, “Software architectures of the convergence of cloud computing and the internet of things: A systematic literature review,” Information and Software Technology, vol. 122, p. 106271, 2020.
[3] X. Xu, Q. Liu, Y. Luo, K. Peng, X. Zhang, S. Meng, and L. Qi, “A computation offloading method over big data for iot-enabled cloud-edge computing,” Future Generation Computer Systems, vol. 95, pp. 522–533, 2019.
[4] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, “Edge computing: A survey,” Future Generation Computer Systems, vol. 97, pp. 219–235, 2019.
[5] H. Zahmatkesh and F. Al-Turjman, “Fog computing for sustainable smart cities in the iot era: Caching techniques and enabling technologies—an overview,” Sustainable Cities and Society, vol. 59, p. 102139, 2020.
[6] L. U. Khan, I. Yaqoob, N. H. Tran, S. A. Kazmi, T. N. Dang, and C. S. Hong, “Edge-computing-enabled smart cities: A comprehensive survey,” IEEE Internet of Things Journal, vol. 7, no. 10, pp. 10 200–10 232, 2020.
[7] M. Zhang, F. Zhang, N. D. Lane, Y. Shu, X. Zeng, B. Fang, S. Yan, and H. Xu, “Deep learning in the era of edge computing: Challenges and opportunities,” Fog Computing: Theory and Practice, pp. 67–78, 2020.
[8] L. N. Huynh, Y. Lee, and R. K. Balan, “Deepemn: Mobile gpu-based deep learning framework for continuous vision applications,” in Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, 2017, pp. 82–95.
[9] H. Ning, Y. Li, F. Shi, and L. T. Yang, “Heterogeneous edge computing open platforms and tools for internet of things,” Future Generation Computer Systems, vol. 106, pp. 67–76, 2020.
[10] H. Fathoni, H. Y. Yen, C. T. Yang, C. Y. Huang, and E. Kristiani, “A container-based edge device monitoring on kubernetes,” in Proc. of 10th International Conference on Frontier Computing (FC 2020), 2020, pp. 340–345.
[11] “Docker,” (Accessed: 16-Jan-2020). [Online]. Available: http://www.docker.com
[12] E. Kristiani, C.-T. Yang, C.-Y. Huang, Y.-T. Wang, and P.-C. Ko, “The implementation of a cloud-edge computing architecture using opensnack and kubernetes for air quality monitoring application,” Mobile Networks and Applications, vol. 26, no. 3, pp. 1070–1092, 2021.
[13] “Prometheus monitoring system,” (Accessed: 08-Mar-2020). [Online]. Available: https://prometheus.io
[14] “Grafana,” (Accessed: 20-Mar-2020). [Online]. Available: https://grafana.com/
[15] “Prometheus node exporter,” (Accessed: 15-Mar-2020). [Online]. Available: https://prometheus.io/docs/guides/node-exporter/
[16] W. Liu, D. Anguelov, D. Ershan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “SSD: Single shot multibox detector,” in European conference on computer vision. Springer, 2016, pp. 21–37.
[17] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” IEEE transactions on pattern analysis and machine intelligence, vol. 39, no. 6, pp. 1137–1149, 2016.
[18] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 779–788.
[19] “NVIDIA DeepStream SDK,” (Accessed: 08-Mar-2020). [Online]. Available: https://developer.nvidia.com/deepstream-sdk
[20] N. Marathe, A. Gandhi, and J. M. Shah, “Docker swarm and kubernetes in cloud computing environment,” in 2019 13rd International Conference on Trends in Electronics and Informatics (ICEEI). IEEE, 2019, pp. 179–184.
[21] N. Sukhija and E. Bautista, “Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus,” in 2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Commu.
nimations, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), IEEE, 2019, pp. 257–262.

W. Chen, K. Ye, and C.-Z. Xu, “Co-locating online workload and offline workload in the cloud: An interference analysis,” in 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2019, pp. 2278–2283.

Y. Huang, Y. Liu, F. Wang, X. Fan, J. Liu, and V. C. Leung, “An edge computing framework for real-time monitoring in smart grid,” in 2018 IEEE International Conference on Industrial Internet (ICI). IEEE, 2018, pp. 99–108.

D. Loghin, L. Ramapantulu, and Y. M. Teo, “On understanding time, energy and cost performance of wimpy heterogeneous systems for edge computing,” in 2017 IEEE International Conference on Edge Computing (EDGE). IEEE, 2017, pp. 1–8.

V. Klöhn, D. Yokoyama, A. Yokoyama, G. Silva, M. Ferro, and B. Schulze, “Performance and energy efficiency evaluation for hpc applications in heterogeneous architectures,” in 2018 Symposium on High Performance Computing Systems (WSCAD). IEEE, 2018, pp. 162–169.

K. Rungsuptaweekoon, V. Vissontiviseth, and R. Takano, “Evaluating the power efficiency of deep learning inference on embedded gpus,” in 2017 2nd International Conference on Information Technology (ICIT). IEEE, 2017, pp. 1–5.

C. Chen, B. Liu, S. Wan, P. Qiao, and Q. Pei, “An edge traffic flow detection scheme based on deep learning in an intelligent transportation system,” IEEE Transactions on Intelligent Transportation Systems, vol. 22, no. 3, pp. 1840–1852, 2020.

T. Peng, D. Zhang, R. Liu, W. K. Asari, and J. S. Loomis, “Evaluating the power efficiency of visual slam on embedded gpu systems,” in 2019 IEEE National Aerospace and Electronics Conference (NAECON). IEEE, 2019, pp. 117–121.

V. Czymmek, C. Möller, L. O. Harders, and S. Hussmann, “Deep learning approach for high energy efficient real-time detection of weeds in organic farming,” in 2021 IEEE International Instrumentation and Measurement Technology Conference (I2MTC). IEEE, 2021, pp. 1–6.

T. Wang, K. Cao, J. Zhou, G. Zhang, and X. Wang, “Power-efficient layer mapping for cnns on integrated cpu and gpu platforms: A case study,” in Proceedings of the 26th Asia and South Pacific Design Automation Conference, 2021, pp. 627–632.

K. Tamilselvan and P. Thangaraj, “Pods—a novel intelligent energy efficient and dynamic frequency scalings for multi-core embedded architectures in an iot environment,” Microprocessors and Microsystems, vol. 72, p. 102907, 2020.

M. S. Aslanpour, S. S. Gill, and A. N. Toosi, “Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research,” Internet of Things, p. 100273, 2020.

YU-WEI CHAN received the B.S. and the M.S. degree in Information Engineering from Tamkang University, Taiwan, in 1997 and 2001, respectively, and the Ph.D. degree in Computer Science from National Tsing Hua University (NTHU), Hsinchu, Taiwan, in 2013. He joined the College of Computing and Informatics of Providence University (PU), Taiwan, in February 2016, as an Associate Professor. His current research interests include cloud computing, edge computing, deep learning applications, and game theoretic wireless networking.

HALIM FATHONI received a master’s degree from Computer Science and Information Enterprise, Asia University, Taichung, Taiwan, in 2009. Since 2010, he joined at Politeknik Negeri Lampung, Bandar Lampung, Indonesia. Currently pursuing a Ph.D. degree at the Department of Industrial Engineering and Enterprise Information, and collaborate with High-Performance Computing Laboratory of Computer Science, Tunghai University, Taichung.

HAO-YI YEN received the B.S. degree in Computer Science from Tunghai University, Taichung in 2017 and M.S. degree in Computer Science from Tunghai University, Taichung in 2020.

CHAO-TUNG YANG is Distinguished Professor of Computer Science at Tunghai University in Taiwan. He received his Ph.D. in computer science from National Chiao Tung University in July 1996. In August 2001, he joined the Faculty of the Department of Computer Science at Tunghai University. He is serving in a number of journal editorial boards, including Future Generation Computer Systems, International Journal of Communication Systems, KSII Transactions on Internet and Information Systems, Journal of Cloud Computing. He has published more than 300 papers in journals, book sections and conference proceedings. His present research interests are in cloud computing, big data, parallel computing, and deep learning. He is a member of the IEEE Computer Society and ACM.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/