IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 13, 2022, accepted April 2, 2022, date of publication April 11, 2022, date of current version April 14, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3166154

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

YU-WEI CHAN[1], HALIM FATHONI[2,3], HAO-YI YEN[4], AND CHAO-TUNG YANG[4,5]

[1]Department of Information Management, Providence University, Taichung 43301, Taiwan
[2]Department of Industrial Engineering and Enterprise Information, Tunghai University, Taichung 407224, Taiwan
[3]Departement of Ekonomi dan Bisnis, Politeknik Negeri Lampung, Bandar Lampung 35141, Indonesia
[4]Department of Computer Science, Tunghai University, Taichung 407224, Taiwan
[5]Research Center for Smart Sustainable Circular Economy, Tunghai University, Taichung 407224, Taiwan

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

This work was supported in part by the Ministry of Science and Technology (MOST), Taiwan, under Grant 110-2621-M-029-003, Grant 110-2221-E-029-002-MY3, Grant 110-2221-E-126-004, and Grant 110-2622-E-029-003; and in part by the National Applied Research Laboratories (NARLabs), Taiwan, under Grant 03108F1106 and Grant 03109F1106.

ABSTRACT In the past decade, Internet of Things (IoT) technology has been widely used in various applications in daily life. Currently, IoT applications primarily depend on powerful cloud data centers as computing and storage centers. However, with such cloud-centric frameworks, numerous data are transferred between end devices and remote cloud data centers via a long wide-area network, which will result in intolerable latency and a lot of energy consumption. The edge computing paradigm is exploited to sink the cloud computing capability from the network core to network edges in proximity to end devices to enable computation-intensive and latency-critical edge intelligence applications to be executed in a real-time manner to alleviate this problem. With the increasing number of edge devices, it is essential to obtain the status of devices in real time to realize the overall resources of heterogeneous edge devices. Thus, constructing a system that can monitor each device's status and performance is important. This study implements a cluster-based heterogeneous edge computing system by integrating the Docker, Kubernetes, Prometheus, Grafana and Node Exporter technologies for resource monitoring and performance evaluation. In the experiment, three deep learning models for object detection evaluate the performance of the implemented system. Through the constructed resource monitoring platform, the resource usage status of various edge devices can be monitored easily. In addition, the overall system performance can also be evaluated effectively.

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. Numerous IoT devices are required to successfully deploy these services. According to statistics, the number of IoT devices will reach 125 billion by 2030 [1]. These IoT devices generate massive volumes of data, and coupled with the rise of artificial intelligence (AI), the data collected by IoT devices offer countless possibilities for AI applications.

The associate editor coordinating the review of this manuscript and approving it for publication was Haris Pervaiz.

In the traditional cloud-centric approaches, centralized machine learning schemes are centrally executed in cloud-based servers or data centers [2]. However, a cloud-centric approach suffers from long propagation delay and can cause unacceptable latency in real-time applications [3]. In addition, the massive data transmitted to the cloud burdens the backbone network, consumes vast energy, and causes privacy issues for users. A new computation paradigm, called Edge Computing (EC), has been proposed to mitigate these challenges [4], [5]. Edge computing has been proposed as a solution, in which the computing and storage capabilities of edge devices are used to take model training closer to where data are generated. Due to data processing at the network

edge, such a new computing paradigm significantly improves data analysis and response time [6].

Edge devices are commonly suitable for efficiently accelerating deep learning algorithms with low cost. Some low-power edge devices with hardware accelerators for AI are developed to fulfill this requirement. Among these products, a series of products from Nvidia Jetson are widely popular edge devices. These products contain a graphics processing unit (GPU) hardware component, parallel computing platform, and application programming interface (API) model called the compute unified device architecture (CUDA). With the platform, software developers can use a CUDA-enabled GPU for complex computations in AI.

Intel and Google also developed a GPU-based accelerator for edge devices, Google Coral and Intel Movidius Neural Compute Stick (NCS), which 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, building and maintaining a cluster edge environment with numerous heterogeneous edge devices is a challenge due to the heterogeneity of these devices [9]. For instance, Raspberry Pi 3B+ and Raspberry Pi 4 require ARMv6 and ARMv7 architectures, respectively, whereas Jetson Nano and TX2 require ARM64 architecture. The packages and environments in different architectures are also different.

Furthermore, it is very important to monitor the resource usage and power consumption of edge devices. In addition, the performance of the inference model on edge devices must also be monitored to let users easily realize the overall system and dynamic changes in resources in the operational environment. Thus, if we aim to monitor the status of resource usage of various edge devices and evaluate the overall system performance, building an efficient, reliable, and easy-to-use visualized environment is also a challenge. The data for resource and performance indicators for monitoring collected data come from heterogeneous edge devices; thus, integrating these devices and the collected data by considering the heterogeneity of devices to effectively build an easy-to-use resource monitoring platform is the main goal of this work. In our previous work [10], we 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], Grafana [14], and Node Exporter [15] software to monitor the overall resource usage and energy consumption with visualization techniques and evaluate the performance of deploying the proposed system on three object detection algorithms of AI: the single-shot multibox detector (SSD) [16], YOLO, and Faster region-based convolutional neural network (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 stored in the database as source information. Then, Grafana is deployed as a visualized platform to show the operational status of each device in real time and 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, Grafana, and Node Exporter to build an easy-to-use and friendly resource monitoring platform to effectively monitor the status of resource usage of various edge devices and evaluate 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, Prometheous, Node Exporter, Grafana, object detection, and related works.

A. DOCKER

Docker is open-source software for developing, deploying, and executing applications [11]. Docker allows users to separate applications in the system environment to form smaller containers, increasing the speed of software deployment. It is designed to simplify and standardize deployment methods in various environments and contributes significantly to adopting 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 environment parameters of the containerized application are bundled into a container, allowing it to run on any Docker host.
3) Predictability: Hosts do not care what is running inside the container, and the container does not care on which host it runs. All interfaces are standardized, and interactions are predictable.

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

B. PROMETHEOUS

Prometheus is 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 data visualization. When connected to data sources, it provides charts, graphs, and alarms for the web. It also converts the TSDB data into exquisite graphics and visual effect tools and can be expanded through the plugin system. Users can employ 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 Microsoft Common Objects in Context (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 determine 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.

The anchor is a fixed-sized reference bounding box placed uniformly throughout the original image. This anchor determines any relevant object inside the anchor and ''how to adjust the anchor better to fit the relevant object'' [17]. Then, the region of interest applies bounding boxes with relevant objects and extracts features corresponding to the relevant object into new tensors.

3) SINGLE SHOOT MULTI-BOX DETECTOR

The feed-forward convolutional network produces a fixed-size collection of bounding boxes. In addition, 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 the SSD [16]. The important feature of the 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 improves 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], 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.

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 centralize services, coordinate deployment, automatically analyze streaming data, correlate data from multiple sources, and set alarm thresholds to determine core issues from a single visualized graph.

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

For edge computing, power consumption is a critical issue. Most edge devices are often resource-constrained; thus, it is difficult for edge devices to run deep learning applications, which often need substantial computations and energy. Some related works on power consumption on edge devices are introduced in the following.

In [24], the authors presented a novel time-energy-cost analysis of wimpy edge computing compared to traditional brawny cloud computing. The researcher used a brawny heterogeneous Amazon EC2 and Jetson TK1 and TX1 as a wimpy heterogeneous system. The results indicate that Jetson TX1 has worse time-cost performance than Jetson TK1 systems because Jetson TX1 has a lower operating core clock frequency and lower instructions per cycle for the Jetson TX1 GPU on some computationally intensive applications compared with Jetson TK1. In [25], the authors also studied the performance analysis considering energy efficiency. In this study, two matrices were used to measure energy consumption. A medium-scale HPC system was set up in the experiment to illustrate total energy use. The experimental results reveal that the underlying architecture and programming model are crucial factors for performance and energy efficiency.

In [26], the authors investigated the power consumption of deep learning applications on embedded GPU systems. They proposed using YOLO methods to perform a real-time object detection algorithm on Jetson TX1 and TX2. They also implemented low-power image recognition challenges to evaluate the system. The experimental results demonstrated that Jetson TX2 with Max-N mode had the highest throughput and efficiency. The results also indicated a trade-off between the throughput and power efficiency, which could be adjusted by observing edge devices in TX2.

In [27], the authors proposed a deep learning-based method to detect the traffic flow on the edge node. First, the authors provided a vehicle detection algorithm based on the YOLO v3 model trained with a significant volume of traffic data. This model was pruned to ensure that it was effective on edge devices. Then, a real-time vehicle tracking counter was proposed combining vehicle detection and tracking algorithms to detect traffic flow. Finally, the author migrated and deployed the vehicle detection network and multiple object tracking network to the Jetson TX2 platform. On edge devices, the test results indicated that the model could detect the traffic flow with an average processing speed of 37.9 FPS and an average accuracy of 92.0%.

In [28], heterogeneous edge devices, Jetson Xavier, Jetson TX2, and Jetson Nano, were used to evaluate the power consumption and processing frame rates. The experimental results demonstrated that high performance was required, whereas the power consumption was low on a mobile robot. In addition, the results revealed that Jetson TX2 had the best power efficiency compared to Xavier and Nano. In [29], the authors proposed a method to reduce energy consumption without compromising the accuracy and frame rate. A Google Coral USB accelerator and Raspberry Pi 4 devices were used in this work. The experimental results indicated that the accuracy could reach 62.3%. In addition, running the CNN in this work is faster and more efficient than that in a tiny YOLO network.

In [30], a power-efficient layer mapping technique for CNNs was deployed on integrated CPUs and GPUs. The experiments on Nvidia Jetson TX2 demonstrated that layer mapping YOLO v3-Tiny influences power consumption. From the experimental results, (1) almost all convolutional layers were unsuitable for mapping to CPUs. (2) the pooling layer could be mapped to the CPU to reduce power consumption, but a larger output tensor could decrease the inference speed. (3) the detection layer could be mapped to the CPU as long as its floating-point operations were not too large. Finally, (4) the channel and upsampling layers were suitable for mapping to CPUs. This study provided information that can be used to develop power-efficient layer mapping strategies to integrate the CPU and GPU platforms.

III. SYSTEM ARCHITECTURE AND IMPLEMENTATION

In this section, we present the system architecture and its implementation. First, the software architecture and its components are introduced. Then, the main components, the master and worker nodes, are also provided. 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 edge computing system is implemented, in which the hardware equipment of Raspberry Pi 3B+, Raspberry Pi 4, Nvidia Jetson Nano, and Nvidia Jetson TX2 was deployed. Open-source software is employed to establish a containerized environment in the system. The

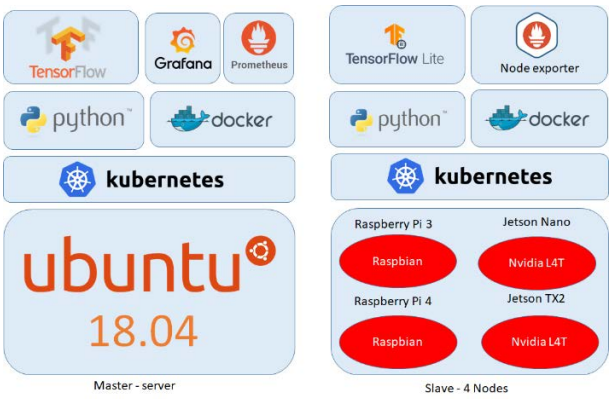


FIGURE 1. The software architecture of the proposed system.

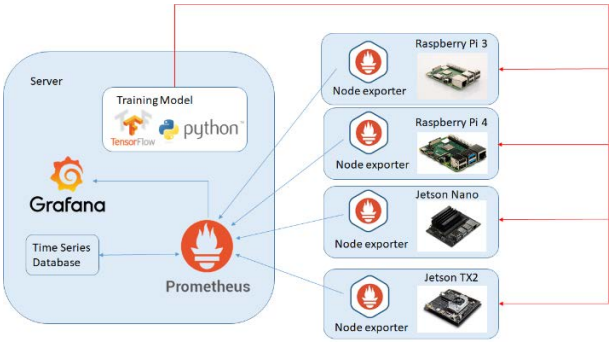


FIGURE 2. The processing flow of the proposed system.

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 manage 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 for each device. It monitors the resource status for each device through Prometheus to return the indicator data and store them in a TSDB.
- Step4: The resource usage status of the devices in the edge computing environment is presented visually through the Grafana software.
- Stpe5: The models trained on the server host are finally deployed on edge devices to verify the monitored performance changes.

B. MASTER AND WORKER NODES

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

1) SYSTEM ENVIRONMENT CONSTRUCTION AND SERVICE DEPLOYMENT

Concerning the system environment construction, open-source software establishes a containerized system environment. First, the Kubernetes software and Docker software establish a cluster environment, where the server host serves as the master node in the cluster, whereas the edge devices serve as the cluster worker nodes to provide a 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 several cluster-wide systems for relatively simple working nodes. In general, components on the master 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 section, the components and operations of the primary and worker nodes are elaborated on in detail, respectively.

2) MASTER NODE

The components of the master node are introduced as follows:

1) API server: The API server component is one of the most important primary services. This component is the primary management point for the entire cluster because it allows users to configure the workload and organization units for Kubernetes. This server is also responsible for ensuring that the service details of the etcd storage and deployed containers are consistent. In addition, it acts as a bridge between various components to maintain the health of the cluster and propagate information and commands. The API server implements the RESTful API interface, indicating 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, a client called kubectl can be used.
2) Etcd: Etcd is an open-source, distributed, consistent key-value storage 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. As it is a key component of a Kubernetes cluster, etcd must 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 component can be used for service discovery and configuring/re-configuring components based on the latest information. In addition, the component is also used to obtain features, such as leader elections and distributed locking to maintain the cluster status. Etcd can be configured on a single primary server or between multiple machines

in a production scenario. The interface of setting or retrieving values of the component is also designed to provide a simple HTTP/JSON API.

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) WORKER NODE

In the system, the worker node consists of three components:

1) Kubelet: The Kubelet component in the worker node provides services for the primary contact point for each node in a cluster group. The component is responsible for transmitting information forward and backward with the control plane services. It interacts with the etcd storage to read configuration values or write new values in detail. In addition, the Kubelet service communicates with the primary component for authentication in the cluster and receives commands and work. The received work acts as a manifest that 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 operations of starting or destroying as the container needs.
2) Proxy: The proxy component must run the proxy services on each server node to manage the separate segmentation of the host's subnet and make the services available to other components. The proxy component forwards requests to the correct container and 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) Container runtime: The first component 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 lightweight 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.

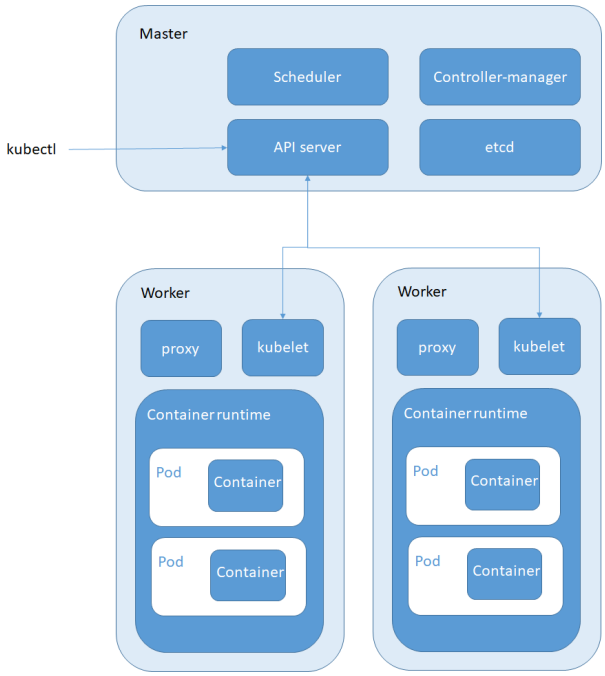


FIGURE 3. The Kubernetes cluster architecture.

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 aims to schedule services on the worker node, the user inputs the instructions to establish the pods [31] through kubectl. The users authenticate the instructions and pass them to the API server in the master node, which backs up the instructions to etcd. Second, the controller manager receives a message from the API server, which must create a new pod and check that the new pod will be built if the resources are allowed. Finally, when the scheduler visits the API server regularly, it asks the controller manager whether a new pod has been built or found. The scheduler is responsible for delivering the pod to the most suitable node. Although the processes in practice seem to be complex, Kubernetes automatically completes the subsequent deployment actions. The cluster architecture of Kubernetes is shown in Fig 3.

In the following, the system implementations of the system are introduced.

C. SYSTEM IMPLEMENTATION

A monitoring system is built on the edge server to collect all data on resource indicators for edge devices, using

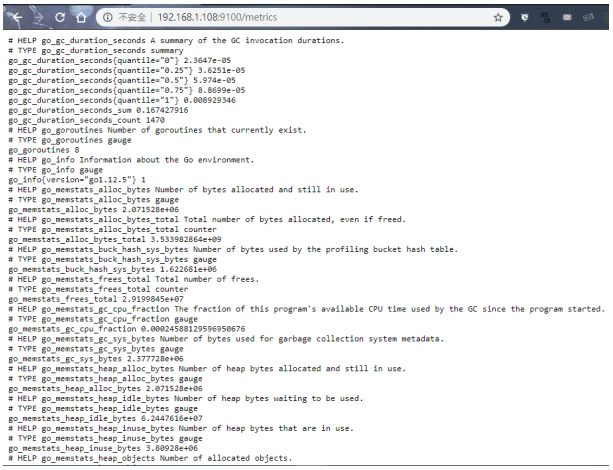


FIGURE 4. The resource index from the implementations of node exporter.

```
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
    - targets: ['localhost:9090']
  # 192.168.1.107(Jetson_TX2)

  - job_name: 'Jetson_TX2'
    scrape_interval: 5s
    static_configs:
    - targets: ['192.168.1.107:9100']
    relabel_configs:
      - source_labels: [__address__]
        regex: '.*'
        target_label: instance
        replacement: 'Jetson_TX2'
```

FIGURE 5. An example of setting the index mining of entities using Prometheus.

Prometheus to collect and store them. In addition, Grafana is used to visualize the data of resource indicators as well as the power consumption data.

1) ACQUISITION OF RESOURCE INDICATORS

The system uses Node Exporter as a service to collect and provide a standard format index to Prometheus for storage. The heterogeneous equipment must deploy Node Exporter with different environmental architectures, such as Raspberry Pi 3 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, which exports each resource indicator retrieved according to the internal settings to the corresponding port (its default value is 9100). The derived resource indicators are primarily CPU usage, memory usage, and system load, as presented in Figure 4.

2) RESOURCE INDICATORS FOR UNIFIED STORAGE

After obtaining the resource indicator data for various equipment, they are stored in databases for monitoring and use by other services. This work uses Prometheus as the

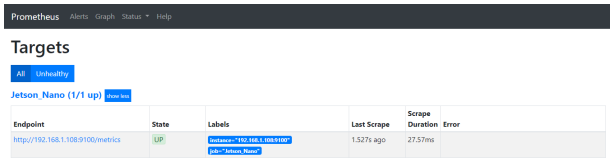


FIGURE 6. An example of deploying Prometheus to collect resource indicators from Node Exporter.

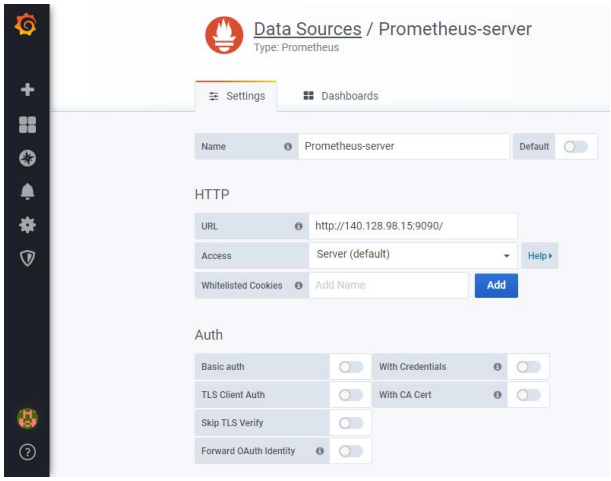
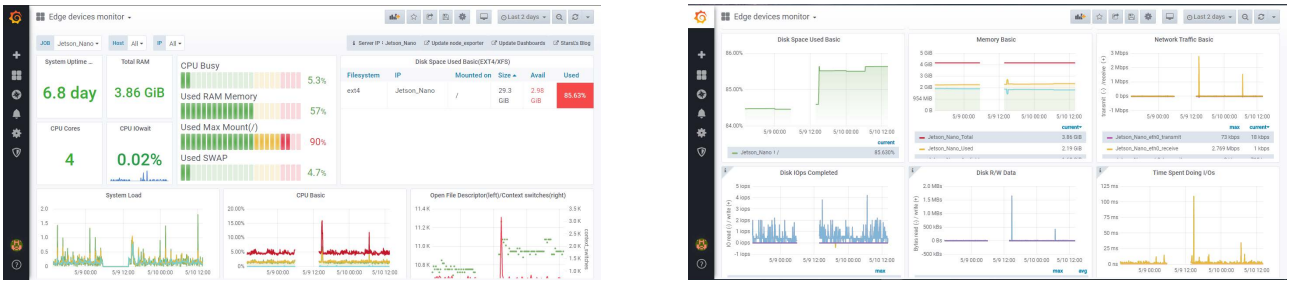


FIGURE 7. Setting interface for data sources with Prometheus.

FIGURE 8. Visualization for multiple indicator types on the same panel.

database for data integration and storage, which is deployed on the server side. The data stored in Prometheus are time-series data, which are uniquely identified by the name of the metric and a series of tags (also called labels). Different tags represent different time series. The time-series data consist of the following formats:

- Indicator name: The indicator name should have a semantic meaning and is generally used to indicate the function of the indicator. For instance, the indicator *http_requests_total* represents the total number of HTTP requests. Generally, the indicator name comprises ASCII characters, numbers, underscores, and colons. The definition of the indicator name must meet the rules of regular expressions [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



(a) The monitoring of CPU resource usage

(b) The monitoring of disk, memory and network traffic resource usage

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

transmission method is the *post* method. The keys in the label comprise ASCII characters, numbers, and underscores. The definition of label must meet the rules of regular expressions [a-z, A-Z:][a-z, A-Z, 0-9:]*.

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

Prometheus uses indicator data as the stored time-series data. According to the user needs, indicator data can be classified based on the index mining of entities. An example of setting the index mining of entities using Prometheus is shown in Fig. 5.

After setting up the index mining of entities, the resource index is collected via Node Exporter deployed on the listening device to consolidate and store data. Prometheus stores the physical returned indicators in the time-series database. In addition to effectively controlling data storage, Prometheus can easily control storage time through the settings. Moreover, Fig. 6 depicts the implementations of deploying Prometheus on a listening device to collect resource indicators from Node Exporter.

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

3) DATA VISUALIZATION OF RESOURCE INDICATORS

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

A panel is needed to perform an indexed-based syntax query with a simple counter to obtain the number of CPUs corresponding to the physical devices to visualize the resource indicator data. In addition, the panel is not limited to the visualization of one indicator. The data from multiple indicator types can be visualized on the same panel to display the changes in these indicators. The visualization for multiple indicator types on the same panel is shown in Fig. 8.

In this study, Grafana was used to integrate panel types into a dashboard interface, where resource indicators for physical devices are visualized with appropriate panels integrated into an interface to provide monitoring and evaluation concerning resource usage for edge devices. In addition, Fig. 9 shows the dashboard of the system implementations for different resource indicator types.

D. EXPERIMENTAL WORKFLOW

In this paper, Fig. 10 presents the experimental workflow of this system. The experiment was run by launching various object detection algorithms on Nvidia Jetson devices and classifying the images on Raspberry Pi 4 devices with the NCS. Three object detection algorithms, Faster-RCNN, 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 for integrating Kubernetes, Prometheus, Grafana, and Node Exporter software.

IV. EXPERIMENTAL RESULTS

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

A. SETTINGS FOR THE EXPERIMENTAL ENVIRONMENT

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

The Kubernetes master node is built to create a Kubernetes cluster, which is added as a working node by initializing the cluster token to the edge devices generated by initializing the

TABLE 1. 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 BCM2837B0 Cortex-A53 (ARMv8) 64-bit	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 Quad-Core ARM® Cortex®-A57 MPCore	32GB	8G

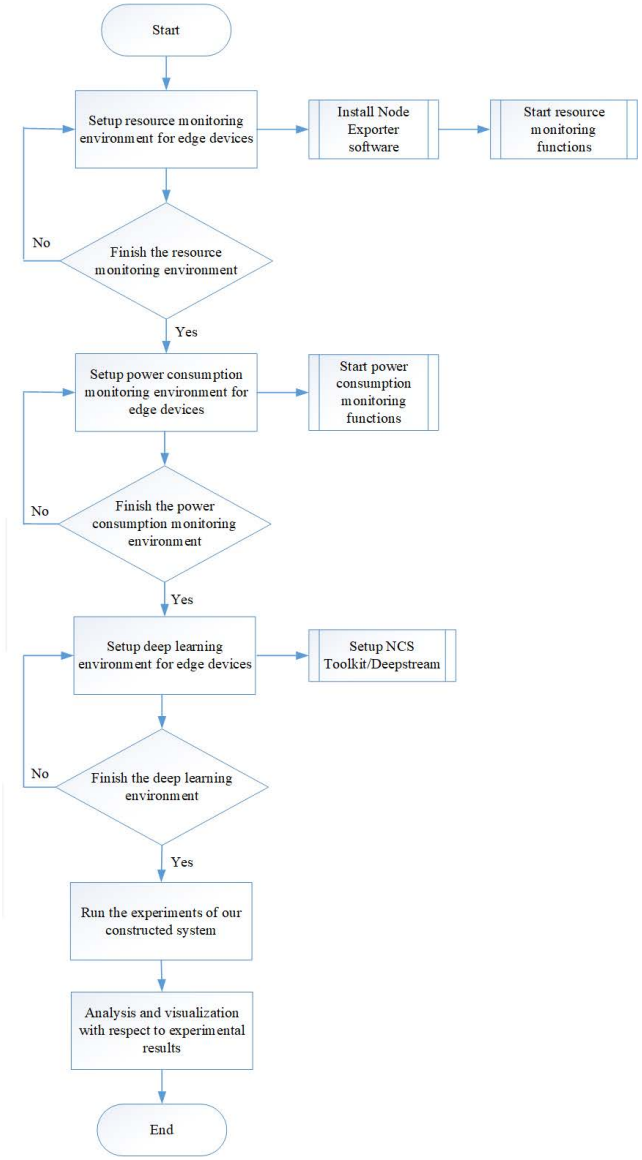


FIGURE 10. Workflow for the experiment.

basic Kubernetes suite. Then, four edge devices, Raspberry Pi 3 and 4 and Jetson Nano and TX2, were installed in the Kubernetes suite as working nodes but were not initialized. Then, the individual token generated by the master node was added to join the cluster to complete the initial setup of the entire system cluster.

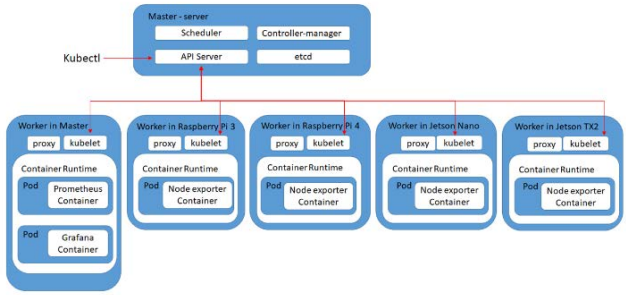


FIGURE 11. Kubernetes cluster architecture in the experimental environment.

TABLE 2. 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

B. DATA COLLECTION AND VISUALIZATION

In this study, the data collected from four heterogeneous edge devices (Nvidia Jetson Nano, Nvidia Jetson TX2, Raspberry Pi 3, and Raspberry Pi 4) are visualized through Prometheus. In addition, the resource usage of these devices is also monitored using Prometheus. Among the devices, the Raspberry Pi 4 device was used as a verification object to observe resource usage status after executing the deep learning methods. The resources include CPU, memory, and the average system load. In the implemented system, the device resource usage was monitored with the three proposed dashboards: (1) the resource monitoring panel, (2) device health status panel, and (3) parallel monitoring panel. An illustrative example is that we use the Jetson Nano device to run the Single Shot MultiBox Detector (SSD) object detection methods. The status of resource usage is shown in Fig. 12.

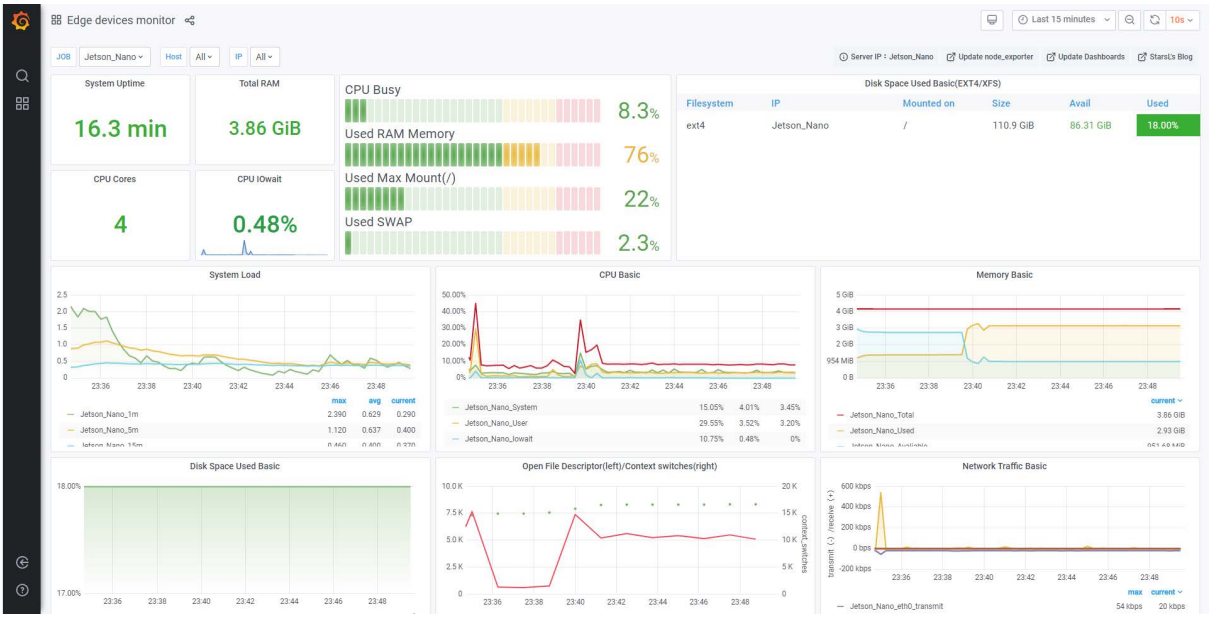


FIGURE 12. Status monitoring of resource usage by Jetson Nano while running the SSD object detection methods.

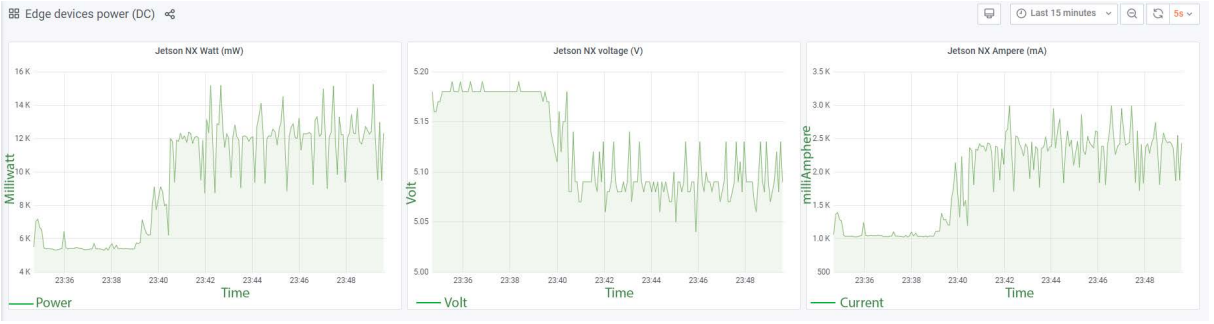


FIGURE 13. Status monitoring of the power consumption of Jetson Nano while running the SSD object detection methods.

FIGURE 14. Comparison of power consumption using Jetson Nano for different object detection methods: (a) SSD, (b) YOLO, and (c) Faster R-CNN.

In addition, the power consumption of Jetson Nano while running the SSD object detection methods is illustrated in Fig. 13. We realize the power fluctuation by integrating with the INA219 wattmeter and Grafana. The fluctuation is defined as the high dynamic range of electricity [32], captured by the INA219 wattmeter and visualized in Grafana. When the Jetson Nano runs the object detection algorithm, the power consumption and processing load are high (as shown in Fig. 13). In addition to the SSD object detection algorithm, we also employed the YOLO and Faster R-CNN object detection algorithms. The comparison results of the power consumption using Jetson Nano concerning these different object detection methods are obtained, as depicted in Fig. 14.

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 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).

$$E_{processing} = \int_{t_{start}}^{t_{end}} P_{processing}(t)\, dt, \tag{1}$$

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).

$$E_{idle} = \int_{t_{start}}^{t_{end}} P_{idle}(t)\, dt. \tag{2}$$

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, the Inception-v3, VGG16, and MobileNet on Raspberry Pi 4, to validate the performance of the implemented system. The three models are first performed for vehicle type identification. The data set in this study is based on the most commonly stolen wheels of cars in the United States in 2017 [33]. Thus, we only selected the first 10 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). Multiyear vehicles were mapped to the stolen car category 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 1193 images that belong to 10 classes. Then, the models were performed with FP16 and FP32 to observe the changes in resources before and after running the optimization models.

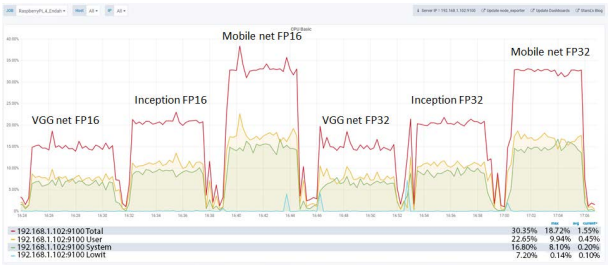


FIGURE 15. Resource changes of CPU when running the three unoptimized models with FP16 and FP32.

Finally, the method of data set optimization is presented. Concerning the optimization of the data set, the work uses open AI software to accelerate the performance of training and inference. 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 the CPU trained TensorFlow, Caffe and MXNet, integrated GPUs, VPUs (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 16. Resource changes of memory when running the three unoptimized models with FP16 and FP32.

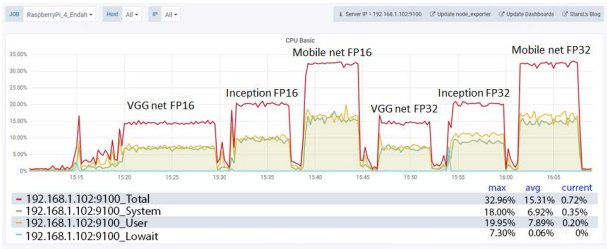


FIGURE 17. Resource changes of CPU when running the three optimized models with FP16 and FP32.

FIGURE 18. 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 the proposed work with state-of-the-art works focusing on the performance monitoring of using tools to obtain parameters when edge devices run deep learning algorithms. In [34], the authors proposed Perf4sight, which allows combinations of various networks, devices, and frameworks to analyze the memory consumption and latency for training. In their work, the models were built to predict memory consumption and the latency of training for the Jetson TX2 devices and PyTorch frameworks using decision trees with a mean error of 5.53% and 9.37%, respectively. These error rates significantly improved those obtained when modeling the same attributes using the CNN training technique.

In addition, this study [35] aimed to learn about the inference workflow and performance of the YOLO network 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 detect different video content across the three edge devices mentioned above. The performance of the three edge devices was compared, and their respective resource characteristics were discussed and evaluated. Furthermore, practical recommendations were provided to indicate how to deploy AI applications on these intelligence edges. In the proposed work, we have successfully implemented a cluster-based edge computing environment by integrating open-source software, such as Docker, Kubernetes, Prometheus, Grafana, and Node Exporter. We built an easy-to-use resource monitoring platform, effectively monitored the usage status of memory and CPU resources in various edge devices, and evaluated the overall system performance.

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 also successfully 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 resource usage, the TSDB of Prometheus could effectively control the length of the stored data and provide connections. The resource indicators that Node Exporter could different types of data. Grafana had a good effect on the visualization presentation. It was not only a variety of visualization tool but was 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 fluctuations could also be monitored using Grafana.

In the absence of Docker and Kubernetes, Prometheus, Node Exporter, and Grafana had to issue multiple installation instructions and parameter sets that could be used on the server and device sides. Regardless of the installer's complexity in management and maintenance, if an environmental

anomaly or the need for new devices occurred, either the environment had to be reinstalled or the corresponding hardware architecture had to be used to determine a matching software version. With the features of Kubernetes and Docker, the entire monitoring environment could be managed and quickly deployed. It was easy to install suitable software with the implemented system if a new edge device joined the cluster environment.

In summary, this work provided a high-quality service deployment and resource monitoring solution. 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 systems, and parallel computing; all were needed to monitor and evaluate various devices' internal resources.

Future work will include the following studies. First, cloud-based architecture will be included in the system to build a cloud-edge collaboration architecture, where 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 would be more beneficial for edge intelligence applications. Second, by running simulations on edge devices in this work, deep learning modules would be performed on only one device. Thus, deep learning applications would be performed under the same architecture. In the future, model transformations will be performed on a cross-device architecture. Finally, with the Kubeflow platform, a deep learning platform launched by Kubernetes, we aim to integrate Kubeflow to establish environmental components compatible with Kubernetes as the deep learning platform of the proposed system.

ACKNOWLEDGMENT

The authors are grateful to the National Center for High-performance Computing for computer resources and facilities.

REFERENCES

[1] *Number of Connected IoT Devices Will Surge to 125 Billion by 2030*, IHS Markit Technol., London, U.K., 2017.

[2] A. Banijamali, O.-P. Pakanen, P. Kuvaja, and M. Oivo, "Software architectures of the convergence of cloud computing and the Internet of Things: A systematic literature review," *Inf. Softw. Technol.*, vol. 122, Jun. 2020, Art. no. 106271.

[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 Gener. Comput. Syst.* , vol. 95, pp. 522–533, Jun. 2019.

[4] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Gener. Comput. Syst.*, vol. 97, pp. 219–235, Aug. 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," *Sustain. Cities Soc.*, vol. 59, Aug. 2020, Art. no. 102139.

[6] L. U. Khan, I. Yaqoob, N. H. Tran, S. M. A. Kazmi, T. N. Dang, and C. S. Hong, "Edge-computing-enabled smart cities: A comprehensive survey," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 10200–10232, Oct. 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," in *Fog Computing: Theory and Practice*. Hoboken, NJ, USA: Wiley, 2020, pp. 67–78.

[8] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 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 Gener. Comput. Syst.*, vol. 106, pp. 67–76, May 2020.

[10] H. Fathoni, H.-Y. Yen, C.-T. Yang, C.-Y. Huang, and E. Kristiani, "A container-based of edge device monitoring on Kubernetes," in *Proc. 10th Int. Conf. Frontier Comput. (FC)*, 2020, pp. 340–345.

[11] *Docker*. Accessed: Jan. 16, 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 OpenStack and Kubernetes for air quality monitoring application," *Mobile Netw. Appl.*, vol. 26, no. 3, pp. 1070–1092, Jun. 2021.

[13] *Prometheus Monitoring System*. Accessed: Mar. 8, 2020. [Online]. Available: https://prometheus.io

[14] *Grafana*. Accessed: Mar. 20, 2020. [Online]. Available: https://grafana.com/

[15] *Prometheus Node Exporter*. Accessed: Mar. 15, 2020. [Online]. Available: https://prometheus.io/docs/guides/node-exporter/

[16] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.* New York, NY, USA, 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 Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017.

[18] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.

[19] *NVIDIA DeepStream SDK*. Accessed: Mar. 8, 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 *Proc. 3rd Int. Conf. Trends Electron. Informat. (ICOEI)*, Apr. 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 *Proc. IEEE SmartWorld, Ubiquitous Intell., Comput., Adv. Trusted Comput., Scalable Comput., Commun., Cloud, Big Data Comput., Internet People Smart City Innov. (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Aug. 2019, pp. 257–262.

[22] W. Chen, K. Ye, and C.-Z. Xu, "Co-locating online workload and offline workload in the cloud: An interference analysis," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun., IEEE 17th Int. Conf. Smart City, IEEE 5th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Aug. 2019, pp. 2278–2283.

[23] Y. Huang, Y. Lu, F. Wang, X. Fan, J. Liu, and V. C. M. Leung, "An edge computing framework for real-time monitoring in smart grid," in *Proc. IEEE Int. Conf. Ind. Internet (ICII)*, Oct. 2018, pp. 99–108.

[24] D. Loghin, L. Ramapantulu, and Y. M. Teo, "On understanding time, energy and cost performance of Wimpy heterogeneous systems for edge computing," in *Proc. IEEE Int. Conf. Edge Comput. (EDGE)*, Jun. 2017, pp. 1–8.

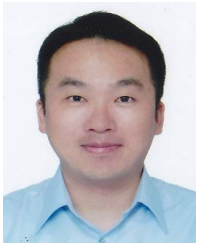
[25] V. Klôh, D. Yokoyama, A. Yokoyama, G. Silva, M. Ferro, and B. Schulze, "Performance and energy efficiency evaluation for HPC applications in heterogeneous architectures," in *Proc. Symp. High Perform. Comput. Syst. (WSCAD)*, Oct. 2018, pp. 162–169.

[26] K. Rungsuptaweekoon, V. Visoottiviseth, and R. Takano, "Evaluating the power efficiency of deep learning inference on embedded GPU systems," in *Proc. 2nd Int. Conf. Inf. Technol. (INCIT)*, Nov. 2017, pp. 1–5.

[27] 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 Trans. Intell. Transp. Syst.*, vol. 22, no. 3, pp. 1840–1852, Mar. 2021.

[28] T. Peng, D. Zhang, R. Liu, V. K. Asari, and J. S. Loomis, "Evaluating the power efficiency of visual SLAM on embedded GPU systems," in *Proc. IEEE Nat. Aerosp. Electron. Conf. (NAECON)*, Jul. 2019, pp. 117–121.

[29] 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 *Proc. IEEE Int. Instrum. Meas. Technol. Conf. (I2MTC)*, May 2021, pp. 1–6.
[30] 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 *Proc. 26th Asia South Pacific Design Autom. Conf.*, Jan. 2021, pp. 627–632.
[31] 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 Microsyst.*, vol. 72, Feb. 2020, Art. no. 102907.
[32] 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 Things*, vol. 12, Dec. 2020, Art. no. 100273.
[33] *NICB: Top 10 Most Stolen Vehicles in 2017*. Accessed: Sep. 19, 2018. [Online]. Available: https://www.claimsjournal.com/news/national/2018/09/19/286821.htm
[34] A. Rajagopal and C.-S. Bouganis, "perf4sight: A toolflow to model CNN training performance on edge GPUs," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. Workshops (ICCVW)*, Oct. 2021, pp. 963–971.
[35] H. Feng, G. Mu, S. Zhong, P. Zhang, and T. Yuan, "Benchmark analysis of Yolo performance on edge intelligence devices," in *Proc. Cross Strait Radio Sci. Wireless Technol. Conf. (CSRSWTC)*, Oct. 2021, pp. 319–321.



YU-WEI CHAN received the B.S. and M.S. degrees 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 as an Associate Professor with the College of Computing and Informatics, Providence University (PU), Taiwan, in February 2016. His current research interests include cloud computing, edge computing, deep learning applications, and game theoretic wireless networking.

HALIM FATHONI received the master's degree from the Computer Science and Information Enterprise, Asia University, Taichung, Taiwan, in 2009. Currently, he is pursuing the Ph.D. degree with the Department of Industrial Engineering and Enterprise Information, and collaborate with High Performance Computing Laboratorium of Computer Science, Tunghai University, Taichung. Since 2010, he has been with Politeknik Negeri Lampung, Bandar Lampung, Indonesia. His research interests include cloud computing, edge computing, and AIoT.

HAO-YI YEN received the B.S. and M.S. degrees in computer science from Tunghai University, Taichung, in 2017 and 2020, respectively.

CHAO-TUNG YANG received the Ph.D. degree in computer science from National Chiao Tung University, in July 1996. In August 2001, he joined the Faculty of the Department of Computer Science, Tunghai University. He is a Lifetime Distinguished Professor of computer science with Tunghai University, Taiwan. 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*, and *Journal of Cloud Computing*. He has published more than 300 papers in journals, book sections, and conference proceedings. His research interests include cloud computing, big data, parallel computing, and deep learning. He is a member of the IEEE Computer Society and ACM.