electronics

MDPI

Article

Person Re-Identification Microservice over Artificial Intelligence Internet of Things Edge Computing Gateway

Ching-Han Chen and Chao-Tsu Liu *

Department of Computer Science and Information Engineering, National Central University, Taoyuan 32001, Taiwan; pierre@g.ncu.edu.tw
* Correspondence: richliu@g.ncu.edu.tw

Abstract: With the increase in the number of surveillance cameras being deployed globally, an important topic is person re-identification (Re-ID), which identifies the same person from multiple different angles and different directions across multiple cameras. However, because of the privacy issues involved in the identification of individuals, Re-ID systems cannot send the image data to cloud, and these data must be processed on edge servers. However, there has been a significant increase in computing resources owing to the processing of artificial intelligence (AI) algorithms through edge computing (EC). Consequently, the traditional AI internet of things (AIoT) architecture is no longer sufficient. In this study, we designed a Re-ID system at the AIoT EC gateway, which utilizes a microservice to perform Re-ID calculations on EC and balances efficiency with privacy protection. Experimental results indicate that this architecture can provide sufficient Re-ID computing resources to allow the system to scale up or down flexibly to support different scenarios and demand loads.

Keywords: AIoT; artificial intelligence; container; edge computing; Internet of Things; Kubernetes; microservices; person re-identification

check for updates

Citation: Chen, C.-H.; Liu, C.-T. Person Re-Identification Microservice over Artificial Intelligence Internet of Things Edge Computing Gateway. *Electronics* **2021**, *10*, 2264. https://doi.org/10.3390/electronics10182264

Academic Editors: Abdellah Touhafi and Gianluca Cornetta

Received: 30 July 2021
Accepted: 10 September 2021
Published: 15 September 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Based on public safety requirements, an increasing number of surveillance cameras are being deployed globally, which usually include wide-area and nonoverlapping fields of view to provide an enhanced coverage area. Such camera networks are often accompanied by large amounts of image data. The image data do not rely on manual supervision; in fact, several computing resources are required to process these images and obtain key data. The utilization of more computing resources can provide more high-quality and advanced analysis. Security issues based on surveillance cameras, such as the face, gait, and license plate recognition, have always been significantly important research topics, which have several mature solutions; however, in recent years, AI has been involved in various fields; particularly, research in the field of imaging is diverse and popular. Consequently, the traditional internet of things (IoT) architecture has also been challenged based on the computing requirements of artificial intelligence (AI) processes. Image data generated by the large-scale IoT are sent to the cloud for processing, which requires time and stable network topology. Large-scale IoT data require high bandwidth to transfer data; however, the IoT gateway might not have high bandwidth or may have limited-access internet; consequently, cloud computing cannot or has limited ability to provide relevant services. Edge computing (EC) is a new technology developed to overcome this limitation [1]. In addition to processing data at the network edge, EC can transmit limited traffic to the cloud center to save bandwidth and reduce network latency. One of the development directions is to move from the traditional multi-core AI gateway architecture to fog/EC architecture to provide distributed computing with multiple CPU/hardware cores to provide more computing resources [2,3].

AI IoT (AIoT) is a novel technology that integrates AI with existing IoT architectures, allowing the IoT network to use AI techniques to process data from IoT devices [1]. The

AIoT EC gateway can solve the aforementioned problems; however, the AIoT EC gateway also demonstrates the following challenges: 1. A method must be developed to provide decision-making capabilities to AIoT systems through EC in the AIoT gateway. 2. Several AI data include personal information; therefore, privacy, security, and protection are crucial issues. 3. AIoT applications must be run on low-cost hardware, and it must also satisfy real-time responsibility [4]. 4. Limited by low-cost hardware, the AIoT gateway is usually designed for a single purpose [5]; consequently, it does not have features, such as flexibility, scalability, fast deployment, and ease of maintenance, and can only supports single deep learning applications. The solution is to adopt a microservice as an EC software architecture. Microservices can be used as AI application solutions for embedded platforms, such as the AIoT gateway, taking advantage of both AIoT and EC, as well as flexible and scalable hardware computing resources [6].

In the research pertaining to surveillance cameras, in addition to biometrics, such as face and gait recognition, understanding the surveillance scene through computer vision requires the ability to track people through multiple cameras and perform crowd motion analysis and activity detection. Tracking people across multiple cameras is essential for wide-area scene analysis, and person re-identification (Re-ID) is an important function of multi-camera tracking. Person Re-ID is defined as the process of establishing the correspondence between the images of a person captured using different cameras. The technique is used to judge whether the instances captured by different cameras belong to the same person [7]. Re-ID technology can increase the success rate of multi-lens human body recognition; however, many factors affect the recognition success rate of Re-ID. Re-ID depends on the camera and includes features, such as viewpoints, low image resolution, illumination changes, unconstrained poses, occlusions, heterogeneous modalities, lens distortion, background cutter, and unconstrained color on different cameras, which can result in various changes and uncertainties. The changes in clothes and the open-world model further increase the difficulty of Re-ID, resulting in variations and uncertainty. Large-scale galleries are a big challenge to system performance and implementation [8]. Figure 1 shows a scene with multiple perspectives, and Re-ID can be performed in this scene.

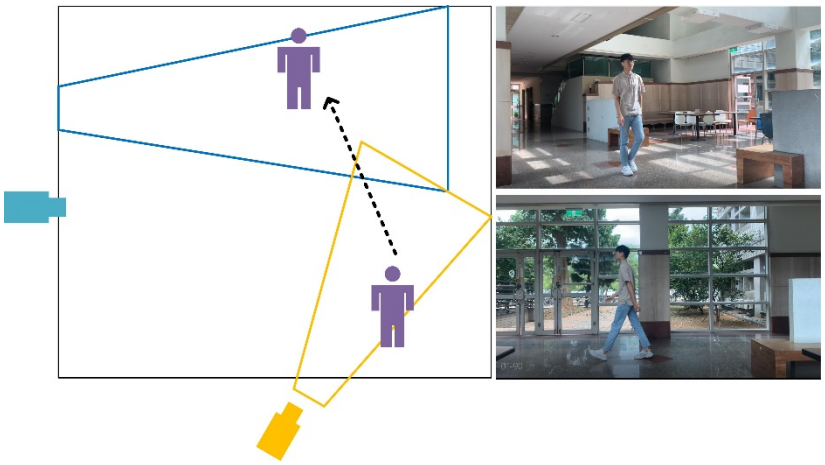


Figure 1. Multi-camera re-identification (Re-ID) scenario.

To overcome the challenges outlined above, we designed and implemented a person Re-ID AIoT EC gateway on an embedded hardware. More computing resources are required to run the Re-ID application using a deep learning model. Consequently, we implemented our system based on a microservice and private cloud EC system. The AIoT EC gateway is an efficient way to provide high-performance, scalable, and flexible applications for Re-ID. Moreover, it satisfies the demanding performance requirements for Re-ID data analysis.

The contribution of this paper is to design an architecture for Re-ID application and use microservices characteristics to deploy Re-ID inference programs run on a scalable

and flexible system; this system can flexibly run with different numbers of embedded AI hardware to provide as high of a performance as the personal computer system can provide. Additionally, user service mesh to solve connection's load balancing problem.

The remainder of this paper is organized as follows. Section 2 presents the basic concepts of personal Re-ID microservice over the AIoT EC gateway architecture. Section 3 describes the architecture in detail. Section 4 outlines the experiments conducted and analyzes the results obtained. Finally, Section 5 presents concluding remarks.

2. Personal Re-Identification Microservice over Edge Computing (EC) Concepts

2.1. Personal Re-Identification

In public places, such as department stores, stations, or schools, customers must be tracked on different cameras. The goal of Re-ID is to identify a person appearing at different times on different cameras. In addition to the different cameras varying viewing angles, various other factors, such as light and shadow, variations in capture time, and color temperature, affect the accuracy of recognition. Even in a particular camera with the same angle, the image captured by the camera may change because of marginal deviations in the walking direction or position [9].

Early research efforts primarily focused on the construction of handcrafted features with body structures, such as context modeling [10], color calibration [11], and distance metric learning [12]. Recent research has relied on the advancement of deep learning [13,14].

Generally, building a person Re-ID system for a specific scenario requires the following five steps:

1. Raw Data Collection: In addition to obtaining raw video data from different surveillance cameras, calibration or preliminary background filtration must be performed.
2. Bounding Box Generation: This process pertains to the detection of a person and extraction of bounding boxes that contain the images of the person. Moreover, person pose detection technology can be employed [15].
3. Training Data Annotation: Annotating the cross-camera label is indispensable for discriminative Re-ID model learning, owing to variations in cross-camera images.
4. Model Training: This is the most extensively studied Re-ID paradigm in the literature.
5. Pedestrian Retrieval: The image gallery is used to test the learned results in the Re-ID model. Certain previous studies have also investigated a ranking optimization to improve the retrieval performance.

Based on these five steps, two predominant Re-ID research domains were extended: closed-world and real-world domains. In the closed-world domain, data are collected, and then the aforementioned five steps are performed. Conversely, the real-world domain must consider factors other than those considered in the aforementioned five steps, as follows:

(1). Heterogeneous Data: The data may be obtained from different sources, such as infrared images, sketches, depth images, or text description. The system must consider the input of multiple formats of data in the model and engine.
(2). Bounding Box Generation from Videos or Images: Extra work is required to mark the bounding boxes from videos when compared to marking from images.
(3). Unavailable/Limited Labels: There are insufficient label annotations for real-world data labels. Moreover, noisy annotations (i.e., incorrect labels) represent another problem.
(4). The real-world data might not have a dataset for training.

Table 1 presents a comparison between the closed- and open-world person Re-ID processes. Although the closed-world process is not required to consider several issues, such as the real-world process, it faces other challenges, such as identifying a method to obtain the feature or size of the dataset. The most common learning strategy is a feature learning strategy, which comprises four primary categories:

(1). Global feature representation learning: The learning process extracts a global feature vector for the image of each person [16].

(2). Local feature representation learning: It learns part/region-aggregated features. It is primarily used to cut the characteristics of the body into several small parts, which are then individually recognized to obtain a better recognition rate. It is sometimes combined with different local feature learning methods, such as human posture recognition or horizontal-divided region features [17].
(3). Auxiliary feature representation learning: This method considers additional annotated information; for example, semantic attributes are also adopted by Re-ID feature representation learning or generative adversarial network generation [18].
(4). Video feature representation learning: It uses multiple video frames and time information; the primary challenge is to accurately capture the time information [19,20].

Table 1. Closed-world versus open-world person re-identification.

Closed-World	Open-World
Single-modality Data	Heterogenous Data
Bounding Boxes Generation	Raw Images/Videos
Sufficient Annotated Data	Unavailable/ Limited Labels
Correct Annotation	Noisy Annotation
Query Exist in Gallery	Open-Set
Less Computing/AI Resources	More Computing/AI Resources

Besides feature learning, a major research area in Re-ID is the convolutional neural network (CNN). A common CNN architecture for Re-ID is the 50-layer residual network (ResNet50) [21]. Y. Wang et al. proposed a deep architecture called BraidNet with a specially designed WConv and Channel Scaling layers [22]. A novel network architecture named the "Multi-Level Factorization Network" [23], which contains multiple stacked blocks to model various latent factors at a specific level, was proposed. An efficient small-scale network named the Omni-scale Network (OSNet) [24] was designed by incorporating point- and depth-wise convolutions.

One of the optimizations of Re-ID is to rely on a ranking of optimization or re-ranking process, as the ranking of optimization performs a crucial role in improving the retrieval performance in the testing stage. Re-ranking through metric fusion is another popular approach for improving the ranking performance, which is based on multiple metrics (or similarities) [25]. The basic idea of re-ranking is to utilize the gallery-to-gallery similarity to optimize the initial ranking list. Additionally, Rank fusion exploits the multiple ranking lists that are obtained with different methods to improve the retrieval performance [26].

Evaluation Metrics: To evaluate the performance of the Re-ID system, the parameters of cumulative matching characteristics (CMC) and the mean average precision (mAP), which measures the average retrieval performance, were considered. CMC-k (i.e., rank-k matching accuracy) represents the probability that a correct match appears in the top k-ranked retrieved results [27].

2.2. Omni-Scale Network

From the various Re-ID model technologies, in this paper, we introduce a closed-world deep learning Re-ID model, i.e., OSNet [24]. Deep CNNs have been widely used in person Re-ID, and most CNNs were originally designed for object category-level recognition. However, none of the existing Re-ID models address omni-scale feature learning and learn features at both homogeneous and heterogeneous scales. OSNet is an architecture designed for learning omni-scale feature representations.

Figure 2 shows the OSNet building block, which is a lightweight network design. The building block consists of multiple convolutional feature streams with different receptive fields. To capture various scales, different feature scales are determined by varying the exponent. Critically, the resulting multiscale feature maps are dynamically fused by channel-wise weights, which are generated by a unified aggregation gate (AG). The AG is

a subnetwork parameter that is shared across all streams with several desirable properties for effective model training.

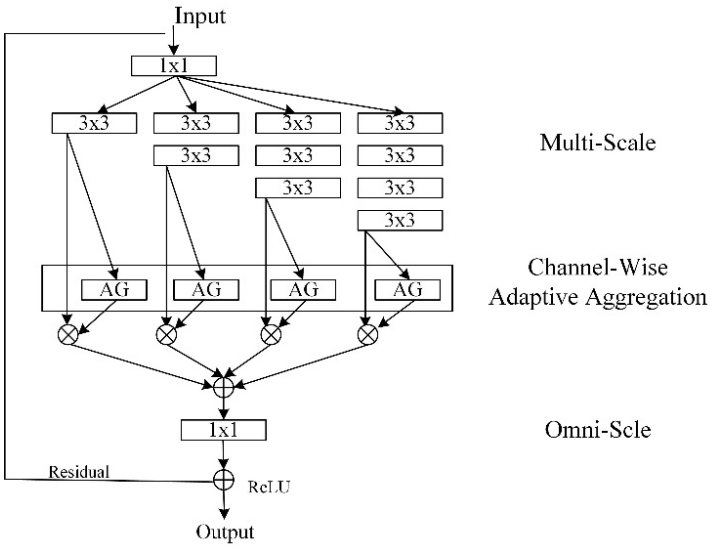


Figure 2. Schematic of the building block for the Omni-scale Network.

Moreover, OSNet adopts a design that employs factorized convolutions using MobileNet, which provides the following benefits: (1) A lightweight network with a small number of model parameters, which is less prone to overfitting; (2) In a large-scale surveillance network, Re-ID performs feature extraction at the camera end instead of sending the raw videos to a central server. For on-device processing, small Re-ID networks are preferred.

In OSNet implementation, the following improvements are incorporated: (1) Depth-wise separable convolutions: it improves the standard 3×3 convolution and uses a point-wise $\rightarrow$ depth-wise stream, which results in more effective omni-scale feature learning. (2) Multiscale feature learning: the largest receptive field is 9×9, as shown in Figure 2. The shortcut connection allows features at smaller scales that are learned in the current layer to be preserved effectively in the subsequent layers, thus enabling the final features to capture an entire range of spatial scales. (3) Unified AG: a dynamic-scale fusion is achieved by a novel AG, which uses channel-wise weights that are dynamically computed by being conditioned on the input data. Adaptive/input-dependent feature-scale fusion strategies are more desirable for Re-ID.

The results of various tests indicate that the combination of the above technologies makes OSNet better than other types of model backbones (i.e., ResNet50, MobileNetV2, or Inception).

2.3. EC and the Fusion of Internet of Things (IoT), Artificial Intelligence (AI), and EC

The concept of EC appeared in the late 1990s when Akamai introduced a content delivery network (CDN), a distrusted network of deployed proxy servers and a high-performance server, which improves the website's response speed and access performance. Akamai deployed server nodes globally, which provide clients data from locally cached website-contained data. Consequently, CDNs improve the user experience when visiting the website, reduce the response time and bandwidth between cloud servers and users, and improve computing resources usage [28].

When EC is coupled with the AIoT environment, it enables the distribution of access latency and computing resources between AIoT devices and cloud servers [3]. AIoT applications and clients will generate large amounts of data, such as images or video data, and AI can achieve good data processing results. In specific fields, such as Re-ID, AI often performs better than traditional algorithms, such as Global and local feature representation learning, and requires less time for research and development. Therefore, the limited

computing capabilities and resources of the AIoT gateway must be considered, and research is still being conducted on the execution of AI on AIoT devices [29,30]. Deploying IoT with EC technology is a way to apply AI computing resources to the IoT topology. EC can be employed to move computing resources from a centralized cloud server to edge nodes, resulting in significant improvements in the processing of IoT data [31]. The different ways in which the current IoT technology can benefit from EC are presented below:

(1). Save bandwidth and cloud computing resources: If a large amount of data is concentrated in the cloud, it is necessary to ensure a large bandwidth or build a large data center in a certain place, especially if the bandwidth is a limited resource; it is reasonable to provide this close to the edge side. For example, placing a video stream server close to a cellphone base station will reduce the backbone bandwidth.
(2). Areas with no internet or limited access to internet, such as rural villages, only have satellite connections, which can be disconnected in bad weather.
(3). Utilized for data preprocessing: The data are processed first before uploading to the cloud; for example, if the Re-ID features cannot be found in the database, then only the feature data, without the image data, can be uploaded to the cloud, which can save time and bandwidth.
(4). Maintain privacy without transferring any data to the cloud: In certain situations, data can only be processed locally because of privacy or security reasons, such as facial images or features of security personnel.
(5). The resources of the entire cloud can be optimized by the edge computing nodes.

Overall, EC has the potential to significantly improve the AIoT gateway framework; moreover, EC can reduce the response time, save cloud computing bandwidth, and, in certain cases, limit energy consumption and also can maintain privacy [32].

EC technology can be incorporated at the edge of the cloud computing server. If the system is not connected to the cloud server, the EC topology can be regarded as a specific micro-cloud system according to the software planning situation, and it differs according to the local computing scenario. EC can be deployed in the AIoT environment more flexibly according to demand and can also move computer resources from a centralized cloud server to the edge to reduce the bandwidth and transmission latency [33]. There are many applications in which EC can be combined with AI. Gong et al. presented an intelligent cooperative EC in IoT networks to achieve a complementary integration of AI and EC [34]; moreover, Chiu et al. considered a novel AIoT service platform that collects video data from the edge devices of individuals [35].

2.4. Microservice Architecture and Container Technology

Microservices are small applications that can be independently deployed, extended, and tested. The advantage of the microservice architecture is that each small service is designed and developed individually using different technologies and plans. Each small service uses an application programming interface (API) for communication and interaction. This approach facilitates increased flexibility in all stages of the software development life cycle, which includes development, testing, and deployment. Because each small service can be upgraded individually, if a microservice malfunctions, only this service can be reverted back to the original version, and it does not affect the other services [36,37].

Microservices are based on a technology called container technology. Containers overcome the primary limitations of virtual machines (VMs). First, container technology is an operation system (OS)-level virtualization and refers to an OS paradigm in which the kernel allows the existence of multiple isolated user space instances. For instance, Docker is a popular container technology that is run on different OS and CPU architecture [38]. Specifically, they are operated under the same CPU and OS, but the processes, data in memory, and file system are independent. When compared to VMs, containers present more efficacy in terms of the CPU execution time, memory consumption, and disk space usage. The power consumption of containers is better than that of VMs [39–42].

Although VMs are considered to be more secure than containers, there are certain important aspects relating to container security: (1) protecting a container from applications inside it, (2) inter-container protection, (3) protecting the host from containers, and (4) protecting containers from a malicious or semi-honest hosts. On Linux, there are certain security implementations for containers, such as namespace, control group (CGroup), capability, or secure computing mode (seccomp) [43]. These technologies are briefly introduced subsequently.

6. Namespaces: Namespace performs the job of isolation and virtualization of system resources for a collection of processes. It can isolate processes, users, file systems, and other components, and can control the range of the components, which can enhance the confidence of a user in the container [44].
7. CGroups: CGroups can control the accountability and limitation of resource usage, such as CPU runtime, system memory, input/output (I/O), and network bandwidth. Moreover, limited resource usage can prevent denial-of-service attacks between containers.
8. Capabilities: Linux systems implement the binary option of root and non-root dichotomy. It can allow a non-root application (e.g., nginx) to bind to a specific port (e.g., Transmission Control Protocol (TCP) port 80, which requires root authority to operate it).
9. Seccomp: Seccomp is a Linux kernel feature that filters system calls to the kernel. It can reduce attacks from system calls.

These container technology-based security implementations also increase the acceptability of microservices for users and enterprises. Because containers can use the above solutions to achieve similar reliability and trustworthiness as VMs, they can compete with VMs and also lower the overall execution cost.

Microservices are the specialization of an implementation approach for service-oriented architectures that are used to build flexible, independently deployable software systems for cloud-native applications, serverless computing, and applications using lightweight container deployment. Therefore, microservices are already becoming mainstream in cloud services [45]. This is because they are easy to scale up or down and even discard when deploying applications to the cloud. Additionally, deploying multiple small programs is significantly simpler than deploying a single large application (e.g., monolithic architecture). Large websites require this characteristic to deploy their applications on the cloud to cope with sudden traffic and resource demand, such as during an online shopping festival.

The lightweight and scalable characteristics of microservices are also suitable for application to embedded AIoT environments [46]. The features and advantages of microservices after adding them to the AIoT environment are as follows [47]:

1. Provision of high-availability and system backup services: The relatively harsh AIoT environment requires such services, and different amounts of hardware or functions can be used for different environments and applications, e.g., to build a high-availability system on systems that require multiple backups, and researchers can focus on important jobs, such as system monitoring or analysis.
2. Fast update deployment/zero downtime: The design of the microservice allows the system to update the application without stopping the service, and the system update can also occur after the background update is completed.
3. Container: AIoT environments or applications require different libraries and setups. Containers can provide the ability to run these applications in individual environments.
4. Suitable for technology diversity: AIoT solutions are obtained from different hardware and software vendors. Solution vendors can implement their software solution in containers and simply open the API for their customers; this can avoid certain library integration issues.
5. Machine-to-machine communication: Both microservices and AIoT systems require low-level machine-to-machine protocol for communication. Protocols between AIoT objects are not usually different; they require an API or other ways to exchange data. Microservices implement several APIs for communication, especially for a fast

API framework. Thus, AIoT applications can rely on fast API frameworks for easy deployment to the microservice architecture.

Microservices are suitable for computing resources and operating programs with heavy loads. Studies have been conducted on the application of deep learning in microservices. Olivier et al. operated their micro-cluster on ODROID-N2 and NVIDIA Jetson hardware to run an inference task with the PlantVillage dataset on Kubernetes [48]. Boltunov et al. demonstrated appropriate end-to-end coordination of resources to support practical IoT deployments. A design proposal for an IoT slice orchestrator was presented, where Edge X Foundry and Mainflux were deployed on the IoT gateway and IoT server, respectively. These functions are used in the validation environment and are implemented using microservice-based architectures [49].

3. Personal Re-Identification AI IoT (AIoT) EC Gateway

Certain other studies have also proposed EC IoT architectures. For example, Junxia et al. proposed an EC IoT secured framework that uses a three-layer architecture, i.e., core, edge, and IoT architecture layers. The core layer is responsible for hosting various applications and managing the end-to-end architecture of the IoT. The EC layer consists of edge services such as data exchange, storage, processing, and job migration between different edge servers. The infrastructure layer is composed of low-powered embedded sensors and IoT devices [50]. Lin et al. proposed a three-tier architecture that integrates fog and EC into an IoT network, called Fog Edge Computing IoT [32]. The three-tier architecture consists of the following layers. The perception layer is the sensor layer and includes the camera or motivation sensors. The network layer collects data from the perception layer and transmits them to the IoT center for processing. The application layer receives data from the network center and provides the services required to process the data.

In this section, we discuss the system architecture of the proposed framework. The proposed framework has four layers, as shown in Figure 3. These layers are the camera communication, tracking, application, and AI accelerated layers. The details of the four layers are presented subsequently.

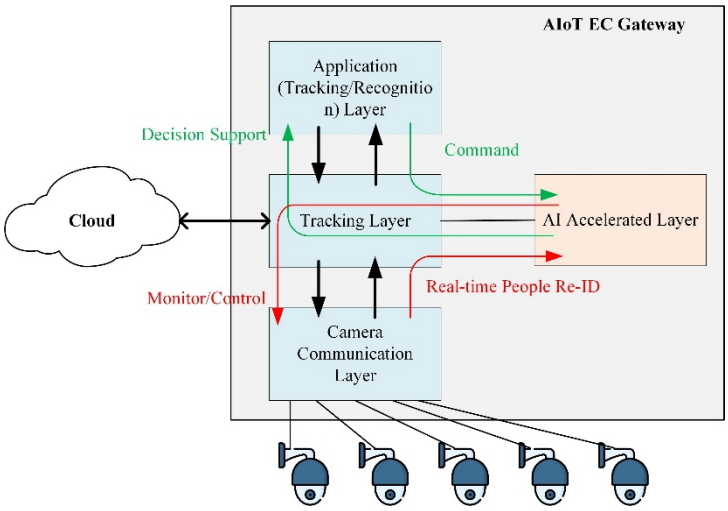


Figure 3. Schematic representation of the person Re-ID microservice over artificial intelligence internet of things (AIoT) edge computing (EC) gateway.

The layers and their corresponding functions in our person Re-ID microservice over the AIoT EC gateway are as follows:

- AI Accelerated Layer

This layer preprocesses the intermediary image data and responds to the tracking layer in time. The AI accelerated layer functions as an inference. This layer scales up and down with the tracking layer.

- Application (Tracking Recognition) Layer:

This layer is responsible for human–machine interaction, control, and AI application deployment to the AI accelerated microservices layer. If necessary, an AI model can be generated.

- Tracking Layer:

A major Re-ID image process layer, the AIoT EC gateway core, is responsible for receiving data from the camera communication layer, processing image data, and collecting inference results from the AI accelerated layer; if it cannot identify the Re-ID feature in the database, it can transmit the Re-ID feature data to the cloud and request the cloud to analyze the Re-ID feature data. The inferred result is transmitted back to the application layer for future decision-making purposes. This layer scales up and down and also influences the AI accelerated layer to provide different computing or AI hardware resources.

- Camera Communication Layer

This layer communicates with the camera sensor, which is responsible for transferring sensor data from the cameras. Moreover, it provides a load-balancing or high-availability function for cameras.

Figure 3 shows a four-layered schematic of a person Re-ID microservice over the AIoT EC gateway. There are four processing activities for these four layers, which can clearly show how the AIoT EC gateway abstraction layer architecture processes data. The four activities are as follows:

- Command:

The application layer deploys AIoT applications and the AI model.

- Real-time People Re-ID Collection:

Image data are collected from the camera through the camera communication layer and processed in time. It uses AI-accelerated hardware to accelerate the processing of data.

- Monitor/Control:

After the AI accelerated layer processes the data, the sensor is notified if the parameters must be changed in the devices (for example, changing the angle or brightness of the camera).

- Decision Support:

The sorted data are transmitted to the application layer for final decisions, such as the results of Re-ID obtained by the AI model. If the AI accelerated layer cannot handle this or the special data, then the inferred feature data will be sent to the cloud for a final decision or analysis.

The person Re-ID microservice over the AIoT EC gateway architecture can provide the person Re-ID function on a microservice's architecture with an AI accelerator function. This architecture offers scalability and elasticity to the system, requiring smaller and more efficient hardware (using only what is really required). The user can scale up/down the AI accelerated hardware arbitrarily according to the situation with different computing resources to execute different scales of Re-ID applications.

To verify the feasibility of the proposed architecture with the Re-ID architecture, we developed a series of software and hardware combinations that can implement Re-ID on the AIoT EC gateway. These combinations are discussed in the following subsections. The following goals are defined to complete such an architecture.

- This architecture is run on embedded hardware, such as the ARM64 system.
- This architecture must have sufficient AI hardware resources to perform Re-ID functions on EC; if requirements change, it should be able to scale up and down easily.
- Additional flexible AI hardware and software are permitted for improved support for redundant systems.

- AIoT protocol that can transmit images efficiently and has a high-efficacy load-balancing function is implemented.

3.1. Torchreid

Torchreid [51] is a Re-ID software framework based on PyTorch with functions to support model development, end-to-end training, and evaluation of Re-ID models such as OSNet, ResNet50, Inception, and MobileNV2. It also has a general data loader to support multiple datasets such as Market-1501 or CUHK03. It provides a general framework and API for implementing the function and training. In addition, it also supports re-rank and cross-dataset comparisons and visualization tools. These functions are easy to use for the development of a Re-ID model.

Torchreid framework has several components, and the descriptions are presented below:

- Data: The DataManager class includes the training and test data loaders that are responsible for loading and classifying data, sampling, and data augmentation. DataManager can also support multiple datasets as the training data.
- Engine: It is designed as a streamlined pipeline for training and evaluation of deep Re-ID models. Based on this engine, the author implemented softmax loss and triplet loss learning paradigms. A Visualization toolkit is implemented in this class for visualizing the ranking result of a Re-ID CNN and represents the visualization of activation maps.
- Model: The model class provides multipurpose CNN architectures, which include common Re-ID models such as ResNet50.

This architecture can not only rapidly develop the Re-ID model but can also help us to improve the deployment of the Re-ID architecture on the AIoT EC gateway architecture. A good framework can save a lot of repetitive work and focus on important topics.

3.2. Kubernetes

Kubernetes is an open architecture for automatically deploying, scaling, and managing containers. The concept is derived from the Borg project developed by Google and already donates to the Cloud Native Computing Foundation (Linux Foundation), which can support several container tools, such as Docker, contained, CRI-O. Although similar tools exist, e.g., Docker Swarm or Apache Mesos, Kubernetes is the most widely used in the world because it has a big and rapidly growing ecosystem and a community that develops a lot of features and provides various components and documents.

Kubernetes is a decentralized system consisting of master and worker nodes and is based on the concept of "pod." Each pod groups one or more containers; therefore, they are run on the same group of physical machines or VMs and use the same context with the co-scheduling of programs. It can be executed on a premise or in a public cloud infrastructure. It is a cluster composed of a master–worker architecture [52]. The following components are present in the master node:

- etcd cluster: a simple, distributed key value storage, which is used to store cluster data;
- kube-apiserver: central management server that receives requests for pods, services, and other modifications;
- kube-scheduler: helps schedule the pods in the various nodes for resources utilization;

The following components are present in the worker node:

- kubelet: the main service in the worker node, regular, new, or modified pods specifications and ensures pods are healthy and running in the desired state;
- kube-proxy: a proxy service that runs on each worker node to expose services to the outside world;
- pod: a set of containers that should be controlled as a single application.

Figure 4 shows the Kubernetes architecture diagram and internal functionality of the Kubernetes cluster with master node and worker node.

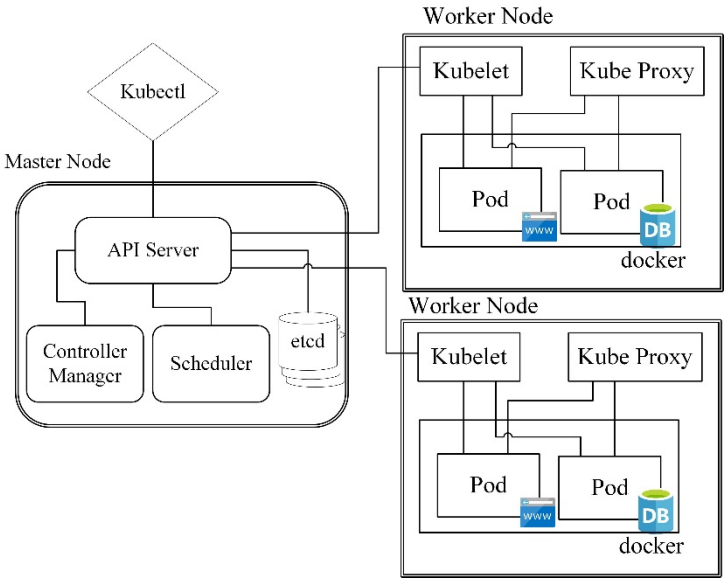


Figure 4. Schematic representation of the Kubernetes architecture.

3.3. gRPC and Protobuf

gRPC is a high-performance, open-source, next-generation universal remote process call (RPC) framework. Although IoT has several communication protocols (e.g., Message Queuing Telemetry Transport (MQTT), Extensible Messaging and Presence Protocol (XMPP), Constrained Application Protocol (CoAP), and Advanced Message Queuing Protocol), when high performance and low-protocol consumption (uses binary code as the data-transfer formats, not a text-based/readable language such as Extensible Markup Language (XML)) are required, few protocols are suitable for the AIoT EC gateway; moreover, protocol-based load-balancing support by the software must also be considered. Although the common IoT protocols such as MQTT, XMPP, and CoAP have their own advantages [53], after researching the available protocols and considering the challenges of TCP/user datagram protocol compatibility, low latency, multiplexing, and other characteristics, we selected gRPC because it satisfied all requirements.

gRPC uses protocol buffer (Protobuf) as the RPC interface description language, and it supports many programming languages, such as Android Java, Python, Go, Node.js, PHP, Rudy, C++, and C#. Moreover, it generates cross-platform client and server bindings for multiple programming languages. They are generally used to provide high-performance communication, such as bi-direction streaming, and replace data exchange languages, such as XML or JavaScript Object Notation.

gRPC also supports multiplexed data transmission and is an efficient and lightweight RPC protocol. gRPC supports flow control, authentication, bidirectional streaming, and several other advantageous features by default and provides an efficient client with multiple possible applications; for example, this framework also implements a small gRPC server, which can provide a communication function between services in a microservice architecture.

Further, gRPC uses Hypertext Transfer Protocol Version 2 (HTTP/2) as a communication protocol. HTTP/2 began as the SPDY protocol, which was developed primarily at Google LLC with the intention of reducing transfer latency by using compression, multiplexing, and prioritization techniques [54]. HTTP/2 is suitable and compatible with most current environments, which have easy-to-use firewalls to control communication and do not require a change in the current network environment setting; it is also suitable for the AIoT EC gateway. One major difference between HTTP/2 and HTTP/1.1 is multiplexing, which allows multiple streams or elements in parallel using a single TCP connection. If data are requested continuously, HTTP/2 can save a valuable connection establishment time [55]. Therefore, gRPC can also be applied to high-performance AIoT environments.

Figure 5 illustrates the gRPC protocol stack.

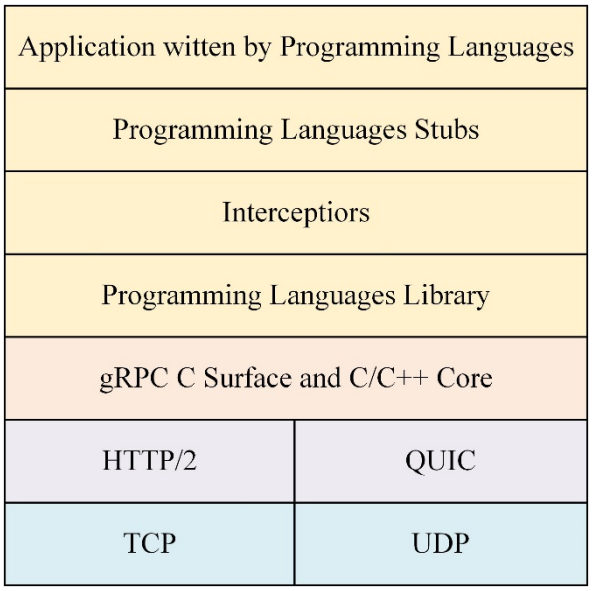


Figure 5. gRPC protocol stack.

3.4. Linkerd2 and Load Balancing

Although microservices can significantly improve the speed and agility of the software service delivery, it also increases the complexity between applications and the network topology; the service mesh is a dedicated infrastructure layer over microservices without imposing modifications to the service implementations. The common contenders for the service mesh in Kubernetes are Istio and Linkerd. Linkerd is implemented by injecting an ultralightweight transparent proxy next to each Kubernetes pod. All network traffic to and from the pods passes through these transparent proxies to control the traffic; moreover, the transparent proxies act as highly instrumented out-of-process network stacks and send and receive telemetry and control signals, respectively, from the controller, which can provide important functions, such as load balancing.

Linkerd2 can support gRPC request-based load balancing, which achieves load balancing with every request within the connection; it allows requests to be sent to different servers by balancing the workloads in a single connection. This allows multiple AIoT client requests to be connected to different servers efficiently, which is significantly important for effectively utilizing the AI hardware resources. For simple load balancing such as least-connection-based load balancing, it can achieve a situation-based fast response environment, such as a CDN. Here, the server can quickly return the requests of the client, such as images or small files; the server is generally not required to wait and can query requests and send files to the client immediately; however, the AI application is slower and requires a longer time. In the AIoT EC gateway, the embedded hardware must be consumed as much as possible. Applications such as AI inference are suitable for request-based load balancing. However, request-based load balancing cannot be performed on network ingress; it can only be performed on network egress because a packet has already arrived at the ingress port; therefore, a proxy server for egress load balancing must be designed. Figure 6 shows a schematic representation of the proxy–cloud connection process. Let us assume that an AIoT client sends three different requests R1, R2, and R3. When traffic passes the egress port of the proxy server, the requests-based load balancing will separate these three requests to different servers. Without load balancing, the requests are sent to the dedicated server. The low-level load-balancing algorithm in Linkerd uses an exponentially weighted moving average (EWMA). It selects a server that offers a high level of performance according to recently acquired data. Thus, the EWMA algorithm improves the server's performance and usage. EWMA has a higher success rate and can achieve a higher level of performance. Unlike round-robin, which always chooses the next server to

send a request, EWMA can significantly improve the overall system performance in an EC AIoT environment.

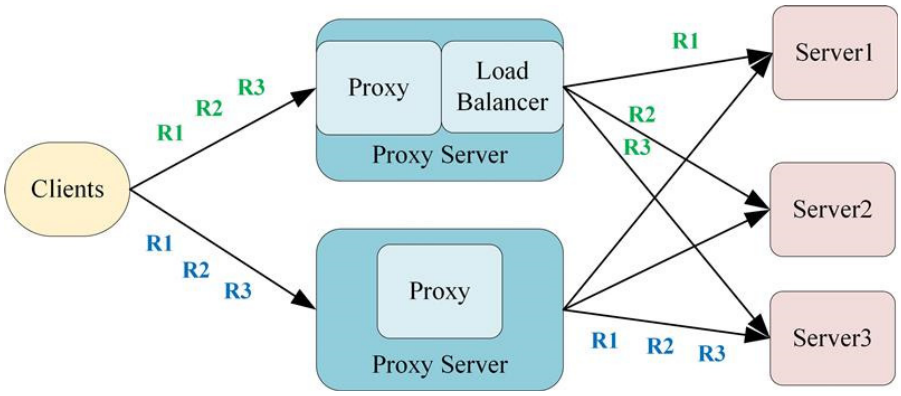


Figure 6. Proxy server connecting to the cloud with/without request-based load balancing.

The introduction of Kubernetes and linked architecture has demonstrated an advantage. It can make distributed processing problems easier. Users need not consider distributed systems when writing applications; they can just send requests to the server; then, the architecture will automatically manage the computing resources, and the system will automatically and efficiently distribute the requests to various GPU accelerated hardware with faster processing speed. This reduces the software complexity and system planning difficulty; if hardware performance cannot satisfy the requirements, it can increase or decrease the number of AI hardware to satisfy performance requirements at any time.

4. System Validation

Figure 7 shows the experiment and development environment. Although the embedded system can build AI model, but it is too slow and complex most time, so we use the x86 platform to build AI model. This is also the most popular embedded system development model. The following describes the development steps for each step:

1. Run model training and operate feature data and development inference software on the x86 hardware; it will create the model and feature data in this stage.
2. Build and test personal Re-ID and gRPC proxy(nginx)'s container images on the ARM64 platform. For personal Re-ID containers, it will include step 1′s model and feature data. In this step, we can make sure the personal Re-ID container works well.
3. Upload container images to cloud container storage. Generally, it will use a docker hub as cloud container storage.
4. Prepare the Kubernetes configuration file and load the configuration file; then, the Kubernetes system will download and deploy containers to where it should be placed in the system automatically.
5. Multiple cameras should be set up in a real environment, but for the performance, it will not be easy to control it. Therefore, we use one x86 PC to simulate multiple cameras and measure the whole system's performance. When tested, it will send multiple images to the gPRC proxy, and the gRPC proxy will forward requests to Re-ID inference servers.

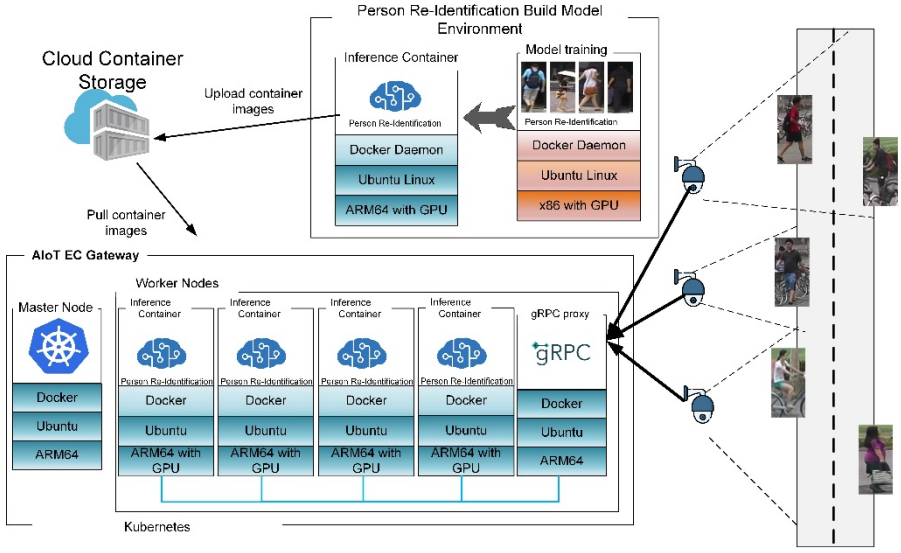


Figure 7. AIoT EC gateway development and experiment environment.

4.1. AIoT EC Gateway Hardware Architecture

This chapter describes the method used to run this experiment and introduces the experimental system architecture used for the investigation and further development. The specifications and technologies used in this architecture, from both hardware and software perspectives, are as specified subsequently.

Hardware:

We propose an AIoT EC gateway. The selected hardware platform includes the following:

1. Raspberry Pi 4(RPi4): RPi4 is the most popular embedded system that can run Kubernetes and is powered by ARM64 architecture with a quad-core 64-bit Cortex-A72 (ARMv8) and 4 GB of RAM.
2. NVIDIA Jetson TX2: We used four Jetson TX2 devices as the Kubernetes worker node with an AI hardware engine. They are powered by a quad-core ARMv8 A57 processor, a dual-core ARMv8 Denver processor, and an integrated 8 GB RAM and 256 Pascal GPU 1.3 GHz processor cores. For power consumption, the Jetson TX2 consumes a maximum of 15 W of power, which enables it to be used as an embedded AI platform or for AIoT EC gateway applications by different power-efficiency modes [4].

Both types of hardware can operate embedded software and have sufficient computing power to run the Re-ID experiment.

Software:

- Container infrastructure: Kubernetes is primarily used to help manage containers and Re-ID services.
- AIoT Communication protocol: uses gRPC as the low-bandwidth protocol with high efficiency, which is based on Protobuf.
- Proxy server: The nginx server is a high-performance HTTP and reverse proxy server, which is used for the gRPC proxy.
- Protocol-based load balancer: Linkerd2 is the service mesh application, which also performs HTTP/2 load balancing.
- AI container software: PyTorch was used to develop the software run in AI containers. PyTorch is a popular machine learning framework based on the Torch library and was used as the underlying AI Accelerated layer.

The abovementioned software and hardware were coupled with the AIoT EC gateway system architecture design presented in Section 3.3. The following subsections introduce the primary techniques for implementing this architecture.

4.2. Hardware Environment

We used the following hardware in this study.

- Two RPi4 devices: One for the Kubernetes master node and another for proxy server.
- Four Jetson TX2 devices: These were used as the Kubernetes worker node with an AI-accelerated engine; the default uses Max-N operation mode.
- ASUS VC65 i5 x86_64: This was used to simulate AIoT device clients that can send simultaneous connections/requests.

Figure 8 depicts the system validation environment.

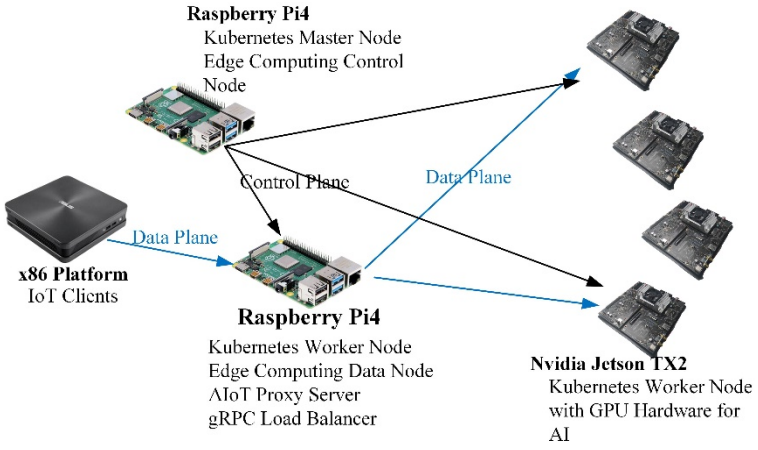


Figure 8. Person Re-ID microservice over AIoT EC gateway system validation environment.

- Intel Core i7-8700 CPU with NVIDIA 1080 Ti GPU and 32 GB RAM: These were used to simulate the EC server for training the Re-ID model and to create a database for AIoT EC gateway inference. Moreover, this machine was used to run an inference test to compare the performance with that of the AIoT EC gateway.

4.3. Experimental Evaluation and Power Consumption

To verify whether the AIoT EC gateway achieves our design goal, we performed the following experiments: (1) performance comparison between the AIoT EC gateway and the x86 system, (2) AIoT EC gateway power consumption test, (3) performance test with multiple AIoT clients and the AIoT EC gateway, and (4) different model performance tests on AIoT EC gateway (Re-ID models such as ResNet50, InceptionV4, MobileNetV2, and SqueezeNet were used). We used the Re-ID image dataset, Market-1501 [27], to evaluate the inference performance, which is a significantly popular dataset on the Re-ID benchmark.

First, we measured the Re-ID inference image number in seconds on the Jetson TX2 and x86 system and the power consumption on the Jetson TX2, the result is shown as Table 3. This experiment has two purposes: (1) to determine the power consumption between the various performance modes of the Jetson and (2) to compare the performance of a single Jetson TX2 with that of the edge server. Let us consider the performance of a single Jetson TX2. To avoid latency issues with a massive client, eight threads (clients) were transmitted simultaneously to measure the system performance. From there, it can be noted that different Jetson TX2 models have different power consumptions. In the Max-Q mode, the device used 62% power and achieved 82% performance when compared to the Max-N mode. Although the x86 system has powerful computing resources and stronger AI hardware, the Jetson TX2 AI hardware can provide relatively good performance for Re-ID inference, which implies that the AIoT gateway architecture will be more energy efficient for EC inference operation.

Table 2 lists the results of the power consumption in the AIoT EC gateway system, which has two RPi4 units and four Jetson TX2 units. This includes the power consumption during the full-load, idle, and without power states in the system for power consumption

measurement baseline. For measuring the power consumption in the idle and full-load states, the power consumption when the system runs 16 AIoT test clients was determined, and the average of the determined power consumption value was considered as the power consumption result. During data collection, the data for the first 20 s were discarded, and the average was obtained by testing for ≥100 s to first attain a stable state; this aided in obtaining more accurate data. When the board power was completely switched off, the power consumption occurred owing to the power meter and power adapter; the power consumption of both the RPi4 and Jetson TX2 units was approximately 3 W, while the full-load power consumption was approximately 47 W. In the idle state, the power consumption for the Jetson TX2 was approximately 3 W, which increased to approximately 9.87 W during full-load operation.

4.4. x86 and AIoT EC Gateway Performance Comparision

In the experiment, we compare the performance of x86 and AIoT EC gateways. Here we use three indicators, processing image speed: this indicator refers to the processing time of each image, AI interference image speed: GPU hardware execution Inference speed, the unit of the above two indicators is ms, and the total number of processes refers to the number of images predicted per second. The x86 machine we use is an Intel Core i7-8700 CPU with NVIDIA 1080 Ti GPU and 32 GB RAM. The 1080 Ti has 3584 cuda cores. For the AIoT EC gateway, we run four Jetson TX2s as AI hardware servers. Jetson TX2 only has 256 cuda cores, while 1080 Ti has 14 times more cuda cores than Jetson tx2.

Figure 9 is a comparison between x86 and the AIoT EC gateway. In the figure, colors are used to classify the x86 and AIoT EC gateways. There are six indicators in the figure for easy classification of the test result: a dark color means an x86 test result, and a light color means a AIoT EC gateway test result.

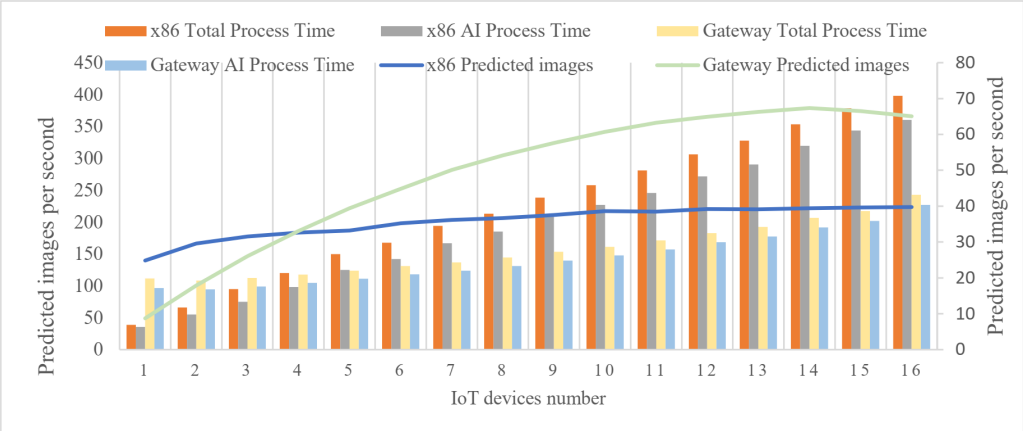
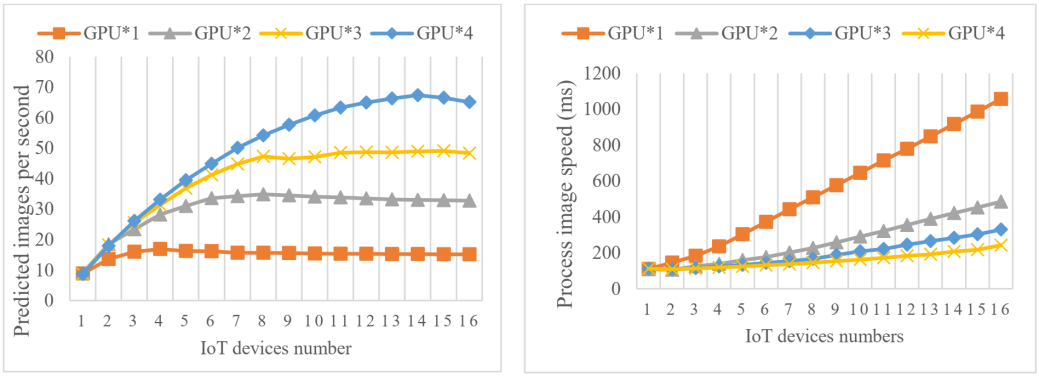


Figure 9. Performance comparison between x86 and the AIoT EC gateway.

First, we can check one IoT device's result. Obviously, x86 has a much better total process time and AI process time than the AIoT gateway; x86 only takes around 35% of the time that the AIoT EC gateway takes. Additionally, x86 can provide much higher predicted images than the AIoT EC gateway when it has only one IoT device, and due to there being no proxy between the client and server, network latency is very good. However, when there are four IoT devices, both the x86 system and the AIoT EC gateway show similar results. When the IoT device number is more than four, we can see that the AIoT system has a better response time and higher predicted image numbers. From these test results, we also know that although the performance of AIoT is not as good as x86 when the number of IoT devices is low, when the client number of IoT devices increases, this architecture has great advantages in executing Re-ID inference applications.

If we do not need such high performance, it can be flexible to reduce the GPU number. From the test results shown in Figure 10, we know that if there are only three GPUs, the

maximum predicted image number is 48. It is still higher than 39 in the x86 platform, and if we consider power consumption or other reasons, e.g. the system only needs a processing efficiency of 30 images per second. Therefore, it only needs two GPUs, and it can also provide the ability to process around 33 images per second.



(**a**) (**b**)

Figure 10. Performance test results: (**a**) analyzed images per second and (**b**) image processing speed in milliseconds.

Based on the above test results and considering the power consumption test in Tables 2 and 3, we can see that this architecture has the following advantages:

1. Power consumption: According to Tables 2 and 3, the power consumption of the x86 GPU is approximately equal to that of the entire four-GPU system. If only three GPUs are used for comparison, x86 versus the AIoT EC gateway is 48 vs. 36.5 W, while the number of x86 only counted the GPU power consumption and excluded other parts of the system (e.g., CPU/DISK/DRAM).
2. Performance and flexibility: Compared with x86, only a specific model of GPU can be used. The performance is fixed on the x86 system, the excess performance will be wasted, and the insufficient performance needs to be replaced by the hardware. The AIoT EC gateway can flexibly control the number of server hardware to meet the demand. It can use the same architecture to meet the embedded system's requirement, so there is no need to consider using personal hardware, thereby saving the cost of the embedded system.
3. By comparing the AIoT EC gateway with the x86 system, we can know its flexibility and performance advantages. In addition, the AIoT EC gateway has lower power consumption and lower thermal, and because of this, the system design can reduce more space used than the x86 system. These are additional benefits of migrating the system to the AIoT EC gateway.

Table 2. System Power Consumption.

Measurement Item	Raspberry Pi 4	Jetson TX2	Entire System
Power Off on All Boards	-	-	2.28 W
4*Jetson TX2 Standby	3 W	3 W	19.64 W
4*Jetson TX2 Full-load Operation	3.3 W	9.87 W	47.07 W

Table 3. Single-server power consumption.

Measurement Item	Inference Image Number (S)	Power Consumption
Jetson TX Max-N Mode Denver*2 2.0G/ARMA57*4 2.0G GPU 1.3G	15.06	10.42
Jetson TX Max-P Mode ARMA57*4 1.12G GPU 1.12G	14.43	9.77
Jetson TX Max-P Core All Mode Denver*2 1.4G/ARMA57*4 1.4G GPU 1.12G	13.76	7.78
Jetson TX Max-Q Mode ARMA57*4 1.2G GPU 0.85G	12.38	6.46
X86 System with 1080 Ti	36.70	48 (Only GPU)

4.5. Architecture Performance Test

Performance tests with multiple AIoT clients and the AIoT EC gateway were conducted for one to sixteen AIoT devices(client(s) and one to four AI hardware servers. The purpose of this experiment was to determine whether this system can provide high efficiency or suitable Re-ID inference computing resources in the AIoT EC gateway and to determine whether the expansion of the AI hardware would reduce the operation delay at system heavy loading.

Figure 10 shows the test results. The method used to calculate the predicted number of images involves the processing of several images by the computing machine. If there are several AI hardware servers, the image processing results are totaled. The image processing speed was calculated by calculating the average time that elapsed from the transmission to the reception of an image within a given block of time. To calculate the image processing speed, the period of difference between the sent data time and the received data time was determined, which was then divided by the period of elapsed time to obtain the expected image processing speed.

Figure 9 indicates that each piece of AI hardware can process approximately 16 AI requests from AIoT client(s) per second. As the load on the AI hardware side increases, the number of requests that can be processed also increases; it can be noted that GPU*4 can process a maximum of 66 requests per second. It was also observed that the number of images processed did not increase after 13 IoT clients on GPU*4, and this trend did not increase after eight AIoT clients on GPU*3. The best prediction performance was obtained when the number of clients was four on GPU*1. This test result can provide a baseline about the maximum system capacity; moreover, the number of images to be processed and the number of AI hardware required can be simultaneously determined. From the test results in Figure 9, it can be noted that irrespective of the number of servers, identical results were obtained for a single AIoT client. This behavior may have been caused by the Python gRPC library, where the library and GPU combination in the embedded hardware was unable to support simultaneous asynchronous I/O during the implementation. This system can use multiple clients to send more requests to obtain a higher performance at one time to determine the maximum computing process capability.

Figure 10b shows the response time from when the AIoT client(s) sends the image to when the AIoT client(s) receives the image. The test result for 11 AIoT clients was 171.29 ms on GPU*4 and was 167 ms for eight clients on GPU*3; moreover, the approximate processing time was approximately the same as GPU*3, which is also the maximum process capacity of the system. For more clients, the system process time increases, but it cannot increase the process number. As shown in Figure 9, the image processing time for one client is almost the same for all the cases with a different number of devices; however, with the increase in the number of clients, the image processing time of a server obviously

increases. This test result shows the advantage of a low-latency system when the number of AI hardware is increased.

We also measured the network latency of the system, and the test results are shown in Figure 11. The test algorithm deducts the hardware prediction time from the image processing time, and it can be observed that if only one server receives a request, the image prediction time is evidently increased, even if each AI hardware server listens to 16 instances. However, it can be noted that the network latency for 16 instances on GPU*2 is better than that of 8 instances on GPU*1. In the test result for multiple servers, there was no obvious network latency, which is better than expected; when there are multiple AIoT EC gateway servers, the prediction time curve was improved; thus, the response time was also improved. This implies that the hardware with multiple GPUs will produce better results where EWMA is used as the load-balancing algorithm. Thus, it automatically sends a request to the fastest server so that the multiple servers or AI hardware system can processes/predict images more effectively than one server/AI hardware.

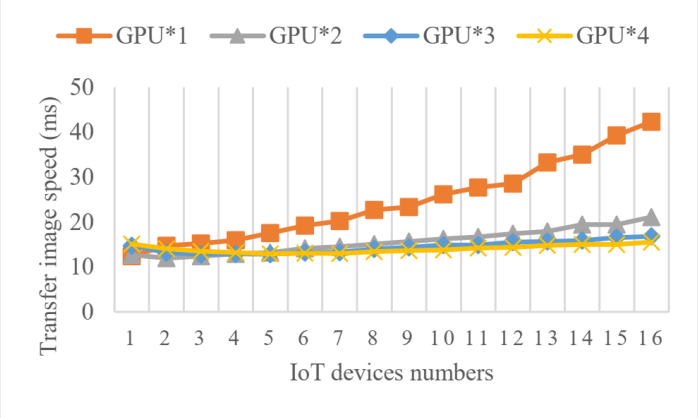


Figure 11. Image prediction time on server.

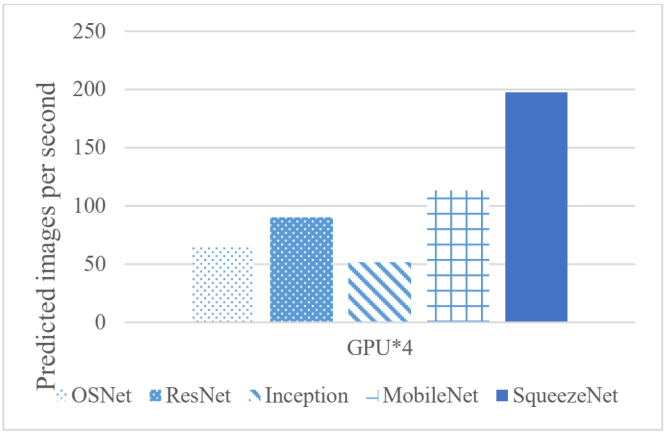
These test results were in line with our expectations; thus, by increasing the number of AIoT EC gateway servers, coupled with the characteristics of EWMA-based load balancing, the transmission of requests to faster servers is prioritized. Multiple servers are effective in reducing latency, and with an increase in computing power, the network latency of the entire system is reduced.

4.6. Alternative Model for Re-ID Performance Test

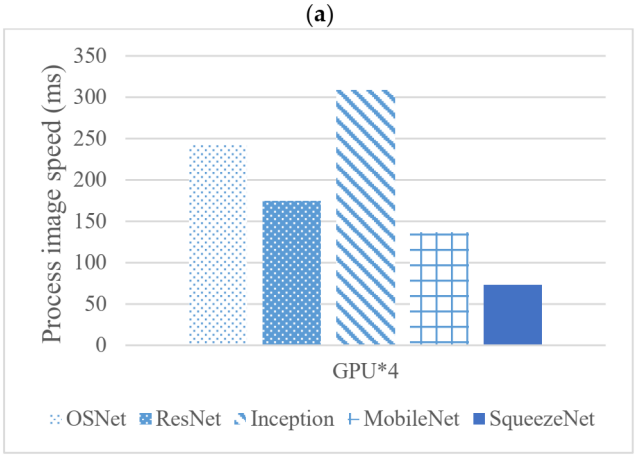
In this section, the performance when using other models on this architecture was determined for future reference. The AIoT EC gateway environment requires a baseline value for future Re-ID model selection. Sometimes, a less powerful and less memory- or resource-efficient model is selected in the embedded system. In our study, we used the selected person Re-ID default support models, which can be used for future reference on the basis of this performance comparison. Considering the computing power and memory capacity of this AIoT EC gateway environment, we selected two popular Re-ID models and two lightweight models that can connect 16 AIoT devices concurrently without problems. The following models were selected: ResNet50, InceptionV4, MobileNetV2, and SqueezeNet, in addition to the OSNet, which was tested in the previous section. The mAP values obtained for the different models were as follows: OSNet—85.6%, ResNet50—78.4%, InceptionV4—66%, MobileNetV2—48.6%, and SqueezeNet—38.2%.

Figure 12 illustrates the performance speeds of various pretraining models when used in the proposed architecture. SqueezeNet and ResNet50 exhibited better performances and shorter response times. OSNet and MobileNetV2 demonstrated comparatively inferior results, while InceptionV4 was the slowest. These test results can be used as a reference for future research. InceptionV4 is unsuitable if processing time or speed is essential

in a sensitive environment. Moreover, lightweight models are unsuitable for the Re-ID test because the model performance and expected results were inadequate. OSNet could process 65 images, and ResNet50 could process 89.7 images. Thus, OSNet has a superior mAP test result, but ResNet50 has a relatively better performance.



(**a**)



(**b**)

Figure 12. Re-ID default model performance test results: (**a**) number of images analyzed per second and (**b**) image processing time in milliseconds.

5. Conclusions

We designed a new architecture for Re-ID applications on the AIoT EC gateway. This architecture can efficiently perform Re-ID-related inference work on the EC side. This architecture expands the AI hardware through microservices to achieve better low-latency performance. This is based on protocol-based load balancing, which can be highly efficient in evenly processing multiple requests with multiple AI hardware and fulfilling the AI hardware resource requirements.

The experimental results indicate that implementing more servers significantly improves the system performance in terms of the processing speed and response time when compared to only one server. In addition, it can expand and flexibly scale the proposed architecture. When compared to a single PC or traditional server, the AIoT EC gateway is more flexible, lightweight, less expensive, and has less power consumption. The experiment also proved that the AIoT EC gateway can flexibly scale up or down to satisfy different Re-ID performance requirements. Finally, we demonstrated that an increase in the amount of AI hardware enables more AIoT devices to be served and maintains less latency in the AIoT EC gateway.

In this study, we developed a feasible AIoT EC gateway that can be implemented in other studies. In future research, other microservices with several practical functions that are also suitable for AIoT EC gateways, such as rolling update, network management,

resource monitoring and logging, and self-healing, will be analyzed. We intend to extend these experiments to include gait recognition on the AIoT EC gateway to improve precision and efficiency.

Author Contributions: Conceptualization and methodology, C.-H.C.; software, validation, and writing, C.-T.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Katz, J.S. AIoT: Thoughts on Artificial Intelligence and the Internet of Things. Available online: https://iot.ieee.org/conferences-events/wf-iot-2014-videos/56-newsletter/july-2019.html (accessed on 15 June 2021).
2. Chen, C.-H.; Lin, M.-Y.; Liu, C.-C. Edge computing gateway of the industrial internet of things using multiple collaborative microcontrollers. *IEEE Netw.* **2018**, *32*, 24–32. [CrossRef]
3. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge computing: Vision and challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [CrossRef]
4. Blanco-Filgueira, B.; García-Lesta, D.; Fernández-Sanjurjo, M.; Brea, V.M.; López, P. Deep learning-based multiple object visual tracking on embedded system for IoT and mobile edge computing applications. *IEEE Internet Things J.* **2019**, *6*, 5423–5431. [CrossRef]
5. Kristiani, E.; Yang, C.-T.; Huang, C.-Y.; Ko, P.-C.; Fathoni, H. On construction of sensors, edge, and cloud (ISEC) framework for smart system integration and applications. *IEEE Internet Things J.* **2020**, *8*, 309–319. [CrossRef]
6. Chen, C.-H.; Liu, C.-T. A 3.5-tier container-based edge computing architecture. *Comput. Electr. Eng.* **2021**, *93*, 107227. [CrossRef]
7. Gheissari, N.; Sebastian, T.B.; Hartley, R. Person Reidentification Using Spatiotemporal Appearance. In Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), New York, NY, USA, 17–22 June 2006; IEEE: Manhattan, NY, USA, 2006; pp. 1528–1535.
8. Ye, M.; Shen, J.; Lin, G.; Xiang, T.; Shao, L.; Hoi, S.C. Deep learning for person re-identification: A survey and outlook. *IEEE Trans. Pattern Anal. Mach. Intell.* **2021**. [CrossRef]
9. Bedagkar-Gala, A.; Shah, S.K. A survey of approaches and trends in person re-identification. *Image Vis. Comput.* **2014**, *32*, 270–286. [CrossRef]
10. Wang, X.; Doretto, G.; Sebastian, T.; Rittscher, J.; Tu, P. Shape and Appearance Context Modeling. In Proceedings of the 2007 IEEE 11th International Conference on Computer Vision, Rio de Janeiro, Brazil, 14–21 October 2007; IEEE: Manhattan, NY, USA, 2007; pp. 1–8.
11. Javed, O.; Shafique, K.; Shah, M. Appearance Modeling for Tracking in Multiple Non-Overlapping Cameras. In Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), San Diego, CA, USA, 20–25 June 2005; IEEE: Manhattan, NY, USA, 2005; pp. 26–33.
12. Yi, D.; Lei, Z.; Liao, S.; Li, S.Z. Deep Metric Learning for Person Re-Identification. In Proceedings of the 2014 22nd International Conference on Pattern Recognition, Stockholm, Sweden, 24–28 August 2014; IEEE: Manhattan, NY, USA, 2014; pp. 34–39.
13. Zheng, Z.; Zheng, L.; Yang, Y. A discriminatively learned cnn embedding for person reidentification. *ACM Trans. Multimed. Comput. Commun. Appl. (TOMM)* **2017**, *14*, 1–20. [CrossRef]
14. Cheng, D.; Gong, Y.; Zhou, S.; Wang, J.; Zheng, N. Person Re-Identification by Multi-Channel Parts-Based Cnn with Improved Triplet LossFunction. In Proceedings of the IEEE conference on computer vision and pattern recognition, Las Vegas, NV, USA, 27–30 June 2016; IEEE: Manhattan, NY, USA, 2016; pp. 1335–1344.
15. Cao, Z.; Hidalgo, G.; Simon, T.; Wei, S.-E.; Sheikh, Y. OpenPose: Realtime multi-person 2D pose estimation using Part Affinity Fields. *IEEE Trans. Pattern Anal. Mach. Intell.* **2019**, *43*, 172–186. [CrossRef] [PubMed]
16. Zheng, L.; Zhang, H.; Sun, S.; Chandraker, M.; Yang, Y.; Tian, Q. Person Re-Identification in the Wild. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017; IEEE: Manhattan, NY, USA, 2017; pp. 1367–1376.
17. Yao, H.; Zhang, S.; Hong, R.; Zhang, Y.; Xu, C.; Tian, Q. Deep representation learning with part loss for person re-identification. *IEEE Trans. Image Process.* **2019**, *28*, 2860–2871. [CrossRef]
18. Lin, Y.; Zheng, L.; Zheng, Z.; Wu, Y.; Hu, Z.; Yan, C.; Yang, Y. Improving person re-identification by attribute and identity learning. *Pattern Recognit.* **2019**, *95*, 151–161. [CrossRef]
19. Dai, J.; Zhang, P.; Wang, D.; Lu, H.; Wang, H. Video person re-identification by temporal residual learning. *IEEE Trans. Image Process.* **2018**, *28*, 1366–1377. [CrossRef] [PubMed]
20. Wang, C.-Y.; Chen, P.-Y.; Chen, M.-C.; Hsieh, J.-W.; Liao, H.-Y.M. Real-Time Video-Based Person Re-Identification Surveillance with Light-Weight Deep Convolutional Networks. In Proceedings of the 2019 16th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS), Taipei, Taiwan, 18–21 September 2019; IEEE: Manhattan, NY, USA, 2019; pp. 1–8.
21. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; IEEE: Manhattan, NY, USA, 2016; pp. 770–778.

22. Wang, Y.; Chen, Z.; Wu, F.; Wang, G. Person Re-Identification with Cascaded Pairwise Convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; IEEE: Manhattan, NY, USA, 2018; pp. 1470–1478.
23. Chang, X.; Hospedales, T.M.; Xiang, T. Multi-Level Factorisation Net for Person Re-Identification. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; IEEE: Manhattan, NY, USA, 2018; pp. 2109–2118.
24. Zhou, K.; Yang, Y.; Cavallaro, A.; Xiang, T. Omni-Scale Feature Learning for Person Re-Identification. In Proceedings of the IEEE/CVF International Conference on Computer Vision, Seoul, Korea, 27–28 October 2019; IEEE: Manhattan, NY, USA, 2019; pp. 3702–3712.
25. Bai, S.; Tang, P.; Torr, P.H.; Latecki, L.J. Re-Ranking Via Metric Fusion for Object Retrieval and Person Re-Identification. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Long Beach, CA, USA, 15–20 June 2019; IEEE: Manhattan, NY, USA, 2019; pp. 740–749.
26. Ye, M.; Liang, C.; Yu, Y.; Wang, Z.; Leng, Q.; Xiao, C.; Chen, J.; Hu, R. Person reidentification via ranking aggregation of similarity pulling and dissimilarity pushing. *IEEE Trans. Multimed.* **2016**, *18*, 2553–2566. [CrossRef]
27. Zheng, L.; Shen, L.; Tian, L.; Wang, S.; Wang, J.; Tian, Q. Scalable Person Re-Identification: A benchmark. In Proceedings of the IEEE International Conference on Computer Vision, Santiago, Chile, 7–13 December 2015; IEEE: Manhattan, NY, USA, 2019; pp. 1116–1124.
28. Satyanarayanan, M. The emergence of edge computing. *Computer* **2017**, *50*, 30–39. [CrossRef]
29. Li, H.; Ota, K.; Dong, M. Learning IoT in edge: Deep learning for the Internet of Things with edge computing. *IEEE Netw.* **2018**, *32*, 96–101. [CrossRef]
30. Pham, H.-T.; Nguyen, M.-A.; Sun, C.-C. AIoT Solution Survey and Comparison in Machine Learning on Low-cost Microcontroller. In Proceedings of the 2019 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS), Taipei, Taiwan, 3–6 December 2019; IEEE: Manhattan, NY, USA, 2019; pp. 1–2.
31. Tran, T.X.; Hajisami, A.; Pandey, P.; Pompili, D. Collaborative mobile edge computing in 5G networks: New paradigms, scenarios, and challenges. *IEEE Commun. Mag.* **2017**, *55*, 54–61. [CrossRef]
32. Omoniwa, B.; Hussain, R.; Javed, M.A.; Bouk, S.H.; Malik, S.A. Fog/edge computing-based IoT (FECIoT): Architecture, applications, and research issues. *IEEE Internet Things J.* **2018**, *6*, 4118–4149. [CrossRef]
33. Chen, X.; Pu, L.; Gao, L.; Wu, W.; Wu, D. Exploiting massive D2D collaboration for energy-efficient mobile edge computing. *IEEE Wirel. Commun.* **2017**, *24*, 64–71. [CrossRef]
34. Gong, C.; Lin, F.; Gong, X.; Lu, Y. Intelligent cooperative edge computing in internet of things. *IEEE Internet Things J.* **2020**, *7*, 9372–9382. [CrossRef]
35. Chiu, T.-C.; Shih, Y.-Y.; Pang, A.-C.; Wang, C.-S.; Weng, W.; Chou, C.-T. Semisupervised Distributed Learning With Non-IID Data for AIoT Service Platform. *IEEE Internet Things J.* **2020**, *7*, 9266–9277. [CrossRef]
36. Thönes, J. Microservices. *IEEE Softw.* **2015**, *32*, 116. [CrossRef]
37. Zimmermann, O. Microservices tenets. *Comput. Sci.-Res. Dev.* **2017**, *32*, 301–310. [CrossRef]
38. Anderson, C. Docker [Software Engineering]. Available online: https://www.computer.org/csdl/magazine/so/2015/03/mso2015030102/13rRUy2YLWr (accessed on 14 May 2021).
39. Morabito, R.; Kjällman, J.; Komu, M. Hypervisors Vs. Lightweight Virtualization: A Performance Comparison. In Proceedings of the 2015 IEEE International Conference on Cloud Engineering, Tempe, AZ, USA, 9–13 March 2015; IEEE: Manhattan, NY, USA, 2015; pp. 386–393.
40. Joy, A.M. Performance Comparison between Linux Containers and Virtual Machines. In Proceedings of the 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, India, 19–20 March 2015; IEEE: Manhattan, NY, USA, 2015; pp. 342–346.
41. Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An Updated Performance Comparison of Virtual Machines and Linux Containers. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015; IEEE: Manhattan, NY, USA, 2015; pp. 171–172.
42. Salah, T.; Zemerly, M.J.; Yeun, C.Y.; Al-Qutayri, M.; Al-Hammadi, Y. Performance Comparison between Container-Based and VM-Based Services. In Proceedings of the 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), Paris, France, 7–9 March 2017; IEEE: Manhattan, NY, USA, 2017; pp. 185–190.
43. Sultan, S.; Ahmad, I.; Dimitriou, T. Container security: Issues, challenges, and the road ahead. *IEEE Access* **2019**, *7*, 52976–52996. [CrossRef]
44. Chandramouli, R.; Chandramouli, R. *Security Assurance Requirements for Linux Application Container Deployments*; US Department of Commerce, National Institute of Standards and Technology: Washington, DC, USA, 2017.
45. Yu, G.; Chen, P.; Zheng, Z. Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. *IEEE Trans. Cloud Comput.* **2020**. [CrossRef]
46. Taherizadeh, S.; Stankovski, V.; Grobelnik, M. A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers. *Sensors* **2018**, *18*, 2938. [CrossRef] [PubMed]

47. Chen, L. Microservices: Architecting for Continuous Delivery and Devops. In Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 30 April–4 May 2018; IEEE: Manhattan, NY, USA, 2018; pp. 39–397.
48. Debauche, O.; Mahmoudi, S.; Mahmoudi, S.A.; Manneback, P.; Lebeau, F. A new edge architecture for ai-iot services deployment. *Procedia Comput. Sci.* **2020**, *175*, 10–19. [CrossRef]
49. Fernandez, J.-M.; Vidal, I.; Valera, F. Enabling the orchestration of IoT slices through edge and cloud microservice platforms. *Sensors* **2019**, *19*, 2980. [CrossRef]
50. Li, J.; Cai, J.; Khan, F.; Rehman, A.U.; Balasubramaniam, V.; Sun, J.; Venu, P. A secured framework for sdn-based edge computing in IOT-enabled healthcare system. *IEEE Access* **2020**, *8*, 135479–135490. [CrossRef]
51. Zhou, K.; Xiang, T. Torchreid: A library for deep learning person re-identification in pytorch. *arXiv* **2019**, arXiv:1910.10093.
52. Menouer, T. KCSS: Kubernetes container scheduling strategy. *J. Supercomput.* **2021**, *77*, 4267–4293. [CrossRef]
53. Dizdarević, J.; Carpio, F.; Jukan, A.; Masip-Bruin, X. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Comput. Surv. (CSUR)* **2019**, *51*, 1–29. [CrossRef]
54. de Saxcé, H.; Oprescu, I.; Chen, Y. Is HTTP/2 Really Faster than HTTP/1.1? In Proceedings of the 2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Hong Kong, China, 26 April–1 May 2015; IEEE: Manhattan, NY, USA, 2015; pp. 293–299.
55. Williams, M.; Benfield, C.; Warner, B.; Zadka, M.; Mitchell, D.; Samuel, K.; Tardy, P. Twisted and HTTP/2. In *Expert Twisted*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 339–363.