Computers and Electrical Engineering 93 (2021) 107227

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

A 3.5-tier container-based edge computing architecture

Ching-Han Chen, Chao-Tsu Liu *

Department of Computer Science and Information Engineering, National Central University, Taoyuan 32001, Taiwan, ROC

ARTICLE INFO

Editor: Dr. M. Malek

Keywords:
AIoT
Artificial intelligence
Container
Internet of Things
Kubernetes
Microservices

ABSTRACT

Concomitant with the explosive growth in the number of artificial intelligence Internet of Things (AIoT) devices, a large amount of data is being constantly generated. Further, cloud computing has become increasingly popular for AIoT edge devices. However, challenges such as bandwidth limitations and connection environment constraints exist. To overcome these challenges, distributing computing resources on AIoT gateways or small cloud servers is necessary. In this study, the fog edge computing IoT (FECIoT) architecture was expanded by adding a new hardware layer. Specifically, a 3.5-tier edge computing AIoT (ECAIoT) architecture was developed based on microservices, containers, hardware artificial intelligence engine technology, and an IoT protocol. Experimental results indicate that the request-based load balancing architecture of ECAIoT results in better performance in terms of response time and processing speed. Furthermore. the architecture allows the system to scale flexibly to support different scenarios and demand loads.

1. Introduction

Since its introduction in 2005 and over 10 years of development, cloud technology has been adopted across various sectors. Residential cloud backup surveillance cameras and different types of cloud technology are becoming integrated into our daily life. In particular, current Internet services offer various applications based on the cloud environment. Furthermore, cloud services, such as Facebook, Google, Netflix, and virtual private networks, are affecting and changing our current lifestyles.

The concept of the Internet of Things (IoT) was introduced into the supply chain management system in 1999. Since then, it has continued to develop in various fields, such as home care, security surveillance, and automotive electronics, and has become one of the driving forces for the growth of science and technology.

Numerous sensors are interconnected via the IoT or pass large amounts of data to servers [1]. Data generated by the large-scale IoT are sent to the cloud for processing, which requires time and a stable network topology. For time-sensitive IoT applications or IoT devices that are often offline, cloud computing cannot or has limited ability to provide relevant services. Edge computing (EC) is a new technology developed to overcome this limitation. In addition to processing data at the network edge, EC can send limited traffic to the cloud center to save bandwidth and reduce network latency. For example, the recent COVID-19 pandemic has forced people to stay home, increasing network traffic, latency, and cloud server loading. By implementing EC technology, cloud server loads can be offloaded to local servers. EC technology can also provide more computing resources for IoT devices. Fig. 1 illustrates the three-tier architecture of EC [1, 2], which consists of IoT devices, EC servers, and cloud servers, each performing different functions.

Artificial intelligence IoT (AIoT) [3, 4] is a novel technology that integrates artificial intelligence (AI) with existing IoT

This paper is for special section VSI-lbdl. Reviews processed and recommended for publication by Guest Editor Dr. Raman Singh.

* Corresponding author

https://doi.org/10.1016/j.compeleceng.2021.107227
Received 1 August 2020; Received in revised form 13 April 2021; Accepted 17 May 2021
Available online 3 June 2021
0045-7906/© 2021 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

architectures, allowing the IoT network to use AI techniques to process data from IoT devices. If AI is deployed on EC servers, its analysis capabilities can be utilized to convert massive IoT data into manageable units that can be sent to the cloud servers. Additionally, it can make preliminary decisions on EC to reduce the latency time and bandwidth required for transfer to the cloud servers. Research indicates that the market size of IoT embedded AI software will reach 700 million U.S. dollars by 2025. Moreover, the combination of AI and IoT will accelerate the digitalization of various fields because the decision-making power of AI considerably affects IoT, and IoT in turn provides the cornerstones of AI through IoT device-generated large data. However, AI involves high computing resource consumption, and EC is a resource-constrained environment. Gathering all computing resources at the edge enables the network to provide flexible on-demand services or additional computing resources to cover urgent computing requirements.

To overcome the challenges outlined above, we designed a new AIoT architecture based on an embedded hardware architecture and explored the potential of this AIoT architecture. The AIoT architecture can meet the demanding performance requirements of IoT to provide, calculate, and analyze sensor data. However, the inherent problem of IoT is that it is an embedded architecture, and it is often faced with the need to increase and strengthen computing capabilities through the special architecture of an IoT gateway. Among the methods proposed to alleviate these problems, Chen et al. [5] developed a multi-embedded microcontroller (MCU) to provide Industrial IoT (IIoT) efficiency for data transmission and increase gateway expansibility. However, under the AIoT architecture, scalability will be a major issue, especially in an EC environment. AIoT involves resource-intensive applications such as AI or image processing applications, which will compound the problem if the architecture lacks AI accelerator hardware. An example scenario is an application that utilizes a cluster of face recognition surveillance cameras, each equipped with artificial intelligence hardware. If many objects suddenly enter the frame of one of the surveillance cameras, that camera could utilize the computing resources of the other cameras in the cluster to speed up image processing and face recognition [6]. To solve these problems, an architecture that incorporates lightweight virtualization and container technologies is necessary.

Clearly, it is crucial to select flexible, efficient, and automated EC tools to deploy applications in the ECAIoT environment effectively. Although various technologies can perform each of the EC functions, these technologies still need to be integrated together to meet the special requirements of AIoT applications. Supported AI algorithms must be scalable and flexible as well as lightweight because they need to be executed separately by the embedded hardware. Based on the above architectural requirements, this study proposes a 3.5-tier edge computing AIoT (ECAIoT) framework, in which the edge computing layer of the traditional three-layer architecture is coupled with AI hardware to meet the extended requirements of AIoT edge computing.

The contributions of this study are as follows:

- An AI accelerated microservices layer is added to the three-layer FECIoT architecture. This enhances it to accommodate effectively ECAIoT applications.
- A new software and hardware ECAIoT architecture is developed based on embedded hardware and integrated microservices technology to make it scalable and flexible.
- A software communication architecture is designed with a request-based load balancing mechanism. The mechanism enables IoT devices to send multiple different jobs simultaneously over the same connection, which can be delivered to different deep learning accelerator hardware. Consequently, hardware resources are utilized more efficiently in the ECAIoT environment and less computing resource is required.

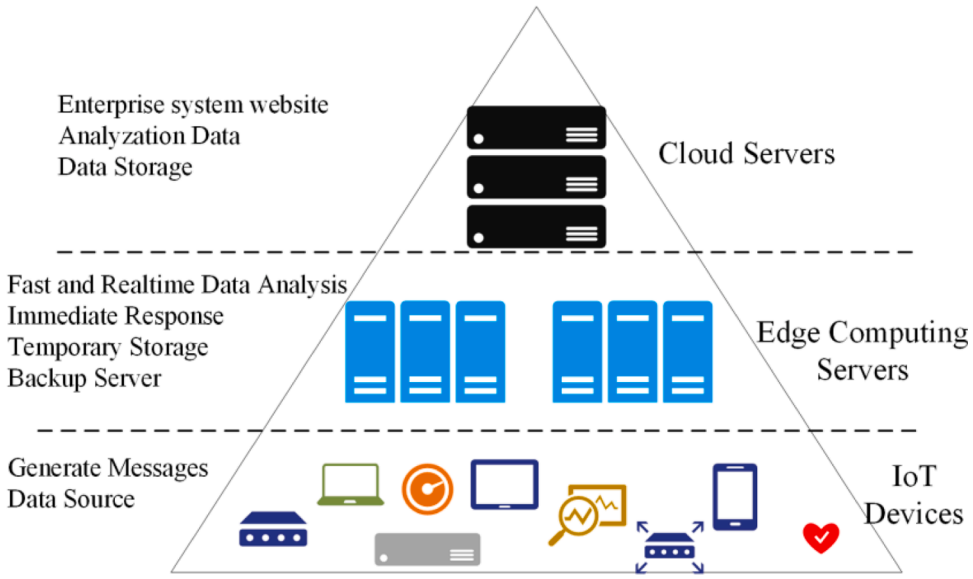


Fig. 1. Schematic representation of the three-tier EC architecture

- A face recognition system was designed and implemented to evaluate the feasibility of the proposed software and hardware architecture. The results verified that architecture can satisfy the original design goal.
- The architecture also satisfies the requirements of an AIoT Gateway. It solves some of the problems often encountered in AIoT environments. For example, it can create embedded network topologies independently and can support the cloud architecture. The same AIoT environment can be scaled with low latency as it employs a highly effective load balancing system, and can quickly develop and deploy system architectures in an embedded environment.

The remainder of this paper is organized as follows. Section II presents the basic concepts of ECAIoT. Section III describes the architecture in detail. Section IV outlines the experiments conducted and analyzes the results obtained. Finally, Section V presents concluding remarks.

2. AIoT and edge computing concepts

2.1. Edge computing

The concept of EC appeared in the late 1990s when Akamai introduced the content delivery network (CDN) to improve website performance and response speed. Akamai deployed server nodes globally that stored locally cached versions of websites. Consequently, CDNs improved user experience, reduced the bandwidth between users and cloud servers, and saved computing resources [7]. Recently, the telecommunications industry has become interested in EC. In 2014, the European Telecommunications Standards Institute (ETSI) established an industry specification group to standardize multi-access edge computing. Since then, telecommunications and academic industries have begun to study the future possibilities of EC technologies [8].

When EC is coupled with the IoT environment, it enables the distribution of computing resources between IoT devices and cloud servers [9]. Listed below are examples of how the current IoT could benefit from EC.

- The amount of data generated may exceed the cloud computing and transmission capacities. For example, airplanes generate large amounts of data when flying, but the available satellite bandwidth is limited. EC can be used to process and analyze the data quickly to provide real-time flight information and help control the flight computer. Subsequently, the analyzed data required can be uploaded via the satellite uplink when resources become available.
- It may not be possible to connect to a cloud server at any time. IoT devices have limited connection capabilities and are located in environments with different conditions, e.g., networks or areas with limited power, bandwidth, and resources. Therefore, an IoT gateway close to the IoT devices is required to assist in storing and processing data.
- EC can be utilized for data pre-processing. Large amounts of device data can be processed before being uploaded at the edge of the network. For example, a surveillance camera can firstly analyze an image via EC and backup a video or perform a face recognition function when an object is detected.

Overall, EC can save cloud computing bandwidth, reduce response time, and, in some cases, limit energy consumption. Additionally, it is often called the last mile of cloud network topology because it can extend cloud services from a central office to a remote location [1].

EC technology differs from client–server architectures. The applicable scenarios for EC are limited to the edge of the cloud computing server. If the system is not connected to the cloud server, the EC topology cannot be compared with that of a local computing scenario. EC offers a specific micro-cloud computing environment, but the computing energy and power consumption differ from those of cloud computing. The potential application scenarios for EC are as follows:

1 Rural territories: Many remote small islands and villages rely on expensive satellite network connections for sufficient communication. Therefore, rural territories are suitable for the deployment of EC technologies.
2 Mine pits: Deep underground mines have limited external connection capabilities because communication cables are often disconnected and cable instability or routing produces high network latency. EC can significantly increase the reliability of IoT networks in mine pits.
3 Factories, seaports, and airports: These areas require high network reliability and proximity to prevent hacker attacks. An independent network can ensure high security and low network latency.

Because EC can meet the needs of the IoT device environment, it can 1) provide local computing resources, 2) allow IoT devices to offload computing jobs at the edge, and 3) offer new niches for small cloud service providers [10].

2.2. Fusion of IoT, AI, and EC

Mining reliable data is a central topic within the IoT research field. AI is considered to be a new research direction as regards data collection methods [11].

AI is increasingly being introduced into IoT solutions for IoT applications and systems. Many IoT applications generate large amounts of data, and AI may achieve good data processing results. In specific fields, such as image processing, AI often performs better than traditional algorithms and takes less time to develop. Therefore, because of the limited energy and computing capabilities of IoT

devices, research is still being conducted on the execution of AI on IoT devices [4].

EC can be employed to move computing resources from a centralized cloud server to edge nodes near the user end, bringing major improvements to the existing cloud computing [12]:

- Edge nodes can pre-process large amounts of data and then send them to the cloud server.
- The resources of the entire cloud can be optimized through the computing power of the edge nodes.

EC has the potential to improve the shortcomings of cloud computing architectures in IoT significantly. Moving computer resources from a centralized cloud architecture to the edge can reduce the number of routers and transmission latency [13]. For example, Zhou et al. used EC and AI to handle imaging problems [14]. They first processed raw data in the form of images and video clips at edge nodes. Then, after preliminary filtering, data authentication, and correlation identification, only the required data were transmitted to the sensing platform. This approach significantly reduced the total data volume transmitted to the sensor server platform.

There are many applications in which EC can be combined with AI, such as hydroponic greenhouses, electronic microgrids, and multi-camera surveillance systems in manufacturing. This technique can reduce the data generated by cameras by 75%, thereby reducing the pressure of data processing in the cloud and increasing the timeliness. Zhou et al. also implemented automatic machine learning (AutoML) on Kubernetes [15], and Chang et al. realized an RSA encryption algorithm for offloading digital signature generation to a GPU-accelerated IoT gateway [16].

2.3. Virtualization and container technology

In hardware-level virtualization, a virtual environment, called the guest machine, is simulated on a physical computer, called the host machine. The virtual system is allowed to create several operating systems (OSs) on the guest machine, which are independent and, by default, cannot easily access each other. The communication and hardware sharing between OSs is realized through the top-level computer. Container technology is based on OS-level virtualization and refers to an OS paradigm in which the kernel allows the existence of multiple isolated user space instances. For instance, FreeBSD Jail, Unix Chroot, and Linux Docker are based on container technology. Specifically, they run under the same core and OS, but the processes and directories are independent.

Many researchers have compared the performance of virtual machines and containers. In general, the execution time, CPU, memory usage, and power consumption of containers are better than those of virtual machines. However, the security and versatility of virtual machines are better than those of containers. For example, a virtual machine can run Linux and FreeBSD simultaneously, but a container can only execute one of them because containers can only run different OSs in the same kernel (e.g., Debian, Red Hat, Ubuntu). Table 1 compares containers and virtual machines. Both technologies have advantages and disadvantages and are suitable for different applications.

2.4. Microservice architecture

Microservices are small applications that can be independently deployed, extended, and tested. The advantage of the microservices architecture is that each small service is designed and developed using different technologies and plans. This approach facilitates increased flexibility during the development stage because the malfunctioning of one microservice does not affect the others.

Recently, microservices have become popular because they are easy to scale up or down and even discard when deploying applications to the cloud. Additionally, deploying multiple small programs is much simpler than deploying a single large application (e.g., monolithic architecture). Each microservice is connected with an independent database and uses an application programming interface (API) for communication. Such convenience is necessary for large websites on the cloud.

2.5. Microservice architecture

The lightweight characteristics of microservices are suitable for application to embedded IoT environments, lightweight and scalable applications on the cloud, or in EC [17]. The features and advantages of microservices after adding them to the IoT environment are as follows [18]:

Table 1
Virtualization versus containers.

Technology	Containers	Virtualization
OS	Shared with host OS	Independent of host OS
CPU architecture	Same as host OS	Can run on different CPU architectures
Portability	Can only run on the same OS/architecture	Can run on different OSs/architectures
Security	All applications run on the same kernel; security depends on the host OS	High security; each OS runs individually; difficult to access guest OS from host OS
Hardware access	Can access hardware via host driver	Needs VM; OS and hardware support input/output virtualization
Memory requirement	Low; containers can access host devices via device files	High; each guest OS has its own memory requirement
File sharing	Low; different containers can share files	High; each virtual machine has its own file system

1 Provision of high-availability and system backup services: The relatively harsh IoT environment requires such services, and researchers can focus on important jobs like systems monitoring or analysis or designing retransmission architectures.
2 Fast update deployment/zero downtime: A microservice can exclusively update important application components in an unstable network environment. It can allow the system to wait until the application is downloaded, after which it is updated. It can also update part of an application or the entire application all at once, while keeping the service online.
3 Container: IoT environments have different applications that may need different libraries and setup. Containers can separate them and run applications with different environments and libraries. They also provide the ability to deploy applications independently.
4 Suitable for technology diversity: IoT solutions come from different hardware and software vendors. Microservices can use container technology to deploy applications on different libraries for different technologies.
5 Machine to Machine Communication: Both microservices and IoT systems need machine to machine communication. Thus, IoT applications can usually be easily deployed to the microservices architecture without redesigning the communication software.

Microservices are suitable for all computing resources and run heavy loading programs. Studies have been conducted on performing deep computing learning on microservices. Tsai et al. integrated distributed application analysis on Kubernetes [19]. Boltunov et al. used embedded AI hardware and Kubernetes to build a MicroGrid managing system, which used GlusterFS to save distributed data and implement power load forecasting algorithm on it [20].

3. 3.5-Tier ECAIoT architecture

Lin et al. proposed a three-tier architecture that integrates fog and edge computing into an IoT network, called FECIoT [1]. Their three-tier architecture consists of the following:

1. Perception Layer: The lowest layer of the IoT architecture is the sensor layer.
2. Network Layer: The middle layer of the IoT architecture collects data from the perception layer and sends them to the IoT center for processing.
3. Application Layer: The highest layer of the IoT architecture receives data from the network center and provides the services needed to process the data.

Lin et al. added fourth and fifth layers to the FECIoT architecture, where the service layer is located between the application and network layers, and introduced several concepts. Their architecture involves 1) a sensing layer, 2) a network layer, 3) a service layer, 4) an application layer, and optionally 5) a business layer above the application layer. As the name implies, the service layer provides a variety of services. The business layer extracts from the application to provide more complex services and provides full management, operation, and deployment for all IoT systems. However, as these layers cannot describe how to implement AIoT using EC, we redesigned the three-layer FECIoT and added hardware support to meet the requirements for AIoT implementation using EC based on the original three-layer design (see Fig. 2).

After adding the AI accelerated microservices layer to the original three layers, we created a 3.5-tier framework, considering that the hardware layer does not entirely belong to but assists the application layer.

The layers and their corresponding functions in our extended 3.5-tier architecture are as follows:

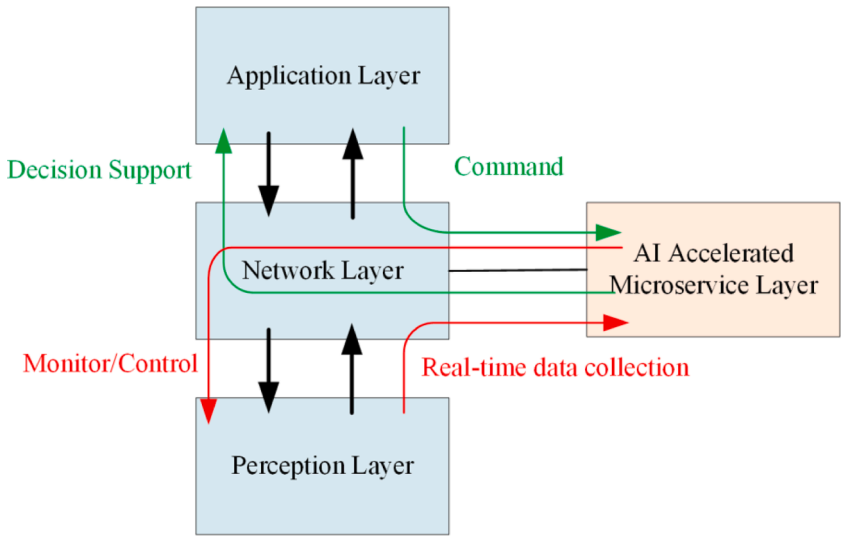


Fig. 2. Schematic representation of the 3.5-tier ECAIoT architecture.

- Application Layer:

  Responsible for human–machine interaction, control, AI model generation, and AI application deployment to AI accelerated microservices layer.

- Network Layer:

This is the IoT gateway, responsible for bridging different IoT protocols, receiving/processing data from the perception layer and communication with the application layer and AI accelerated microservices layer.

l Perception Layer

The function of this layer is the same as in the three-tier perception layer, which is the sensor layer.

l AI Accelerated Microservices Layer

This layer preprocesses the intermediary data and responds to the perception layer in time. The role of this layer is established by the application layer as an intermediary layer between the application and perception layers. Part of the operation and control belonging to the application layer is placed on this layer. This is done to enhance the edge computing capabilities of EC, so that it can have low latency and the ability to respond quickly to the perception layer.

From these four layers, we also added four processing activities, which can more clearly show how the 3.5 tier ECAIoT abstraction layer architecture processes data:

l Command:

The application layer deploys AIoT applications and the AI model.

l Real-time Data Collection:

Data are collected from the perception layer and processed. It uses AI accelerator hardware to speed up the processing of data.

l Monitor/Control:

After the AI accelerated layer processes data, depending on the requirement, the sensor is notified whether to continue monitoring or controlling the device to change parameters (for example, modify the sensor's monitoring angle).

l Decision Support:

The sorted data are sent to the application layer for final decisions, such as variation in the data collected from the sensors, and predicting it by the AI model. If the AI accelerated microservices layer cannot handle it or special data, then those data will be sent to the application layer to for a final decision or analysis.

We believe that the 3.5 tier ECAIoT architecture with the addition of AI accelerated microservices layer can better show the ability of the ECAIoT architecture and how it processes data. It also shows that this architecture is closer to the perception layer than the application layer with faster response times and low latency.

To verify the feasibility of the proposed architecture, we developed a series of software and hardware combinations that can implement the ECAIoT framework. These combinations are introduced in the following sub-sections.

3.1. System design purpose

The proposed architecture must meet the following minimum requirements:

- The EC must be able to build networks and cloud-service-like systems independently without relying on the central cloud services.
- The AI hardware must be flexible, i.e., the AI hardware must be able to be added, removed, and support a redundant system at any time.
- The AIoT communication protocols need to be lightweight, efficient, and capable of transmitting large amounts of data and supporting a wide range of programming languages.
- Load balancing for the AIoT communication protocols must be implemented to maximize the use of EC resources, i.e., support-protocol-based load balancing.
- Applications must be built using a microservices architecture, which is highly scalable and flexible, and current mainstream container technologies must be implemented.
- The system architecture must use embedded hardware to meet the requirements of EC and AIoT.

3.2. Load Balancing

At the start of our new ECAIoT architecture research, we tried to use Docker Swarm (a special mode of the Docker service that allows containers to be scaled across multiple Docker daemons) as our ECAIoT architecture. However, Docker Swarm uses DNS as the connection load balancer, which will concentrate connections on one or more machines. This results in some servers not getting any jobs from the IoT devices. Thus, we encountered a system architecture problem. In a connection-based load balancing mechanism, multiple IoT devices may be connected to the same hardware, while others may be kept on standby, resulting in inefficient use of hardware resources.

The concentrating of processing resources on special machines led us to rethink our architecture. Specifically, we felt that we needed a different load balancer architecture to avoid wasting computing resources. The following are possible solutions to this problem:

- Let IoT devices connect to the application directly

This can achieve our purpose, and it is very easy to implement. However, when clients and servers are added in this system, server information must be re-acquired and connections will grow exponentially. Direct connection also has a security issue.

- One connection per request

This will incur numerous connections and it will increase latency and consume significant amounts of resources. Furthermore, it has a security issue.

- Protocol-based load balancer

This system can analyze the protocol, extract every request it contains, and send them to each hardware equally. This architecture requires an extra proxy, on which it can run a load balancer. The proxy can also be a good firewall. This is a more secure solution. However, this solution will consume more computing resources to achieve these functions.

Obviously, the best idea choice is a protocol-based load balancer. Our design uses an HTTP/2-based IoT protocol, in which multiple requests can be made to the same connection. The protocol-based load balancer can be used to distribute connections to different hardware, implying that the load balancing mechanism must be analyzed and processed by the protocol. However, protocol-based load balancing is difficult to implement. Fig. 3 illustrates the difference between connection-based and protocol-based load balancing. In connection-based load balancing, different requests are sent to the same host. In contrast, requests are referred to the protocol layer and sent to different hosts in protocol-based load balancing.

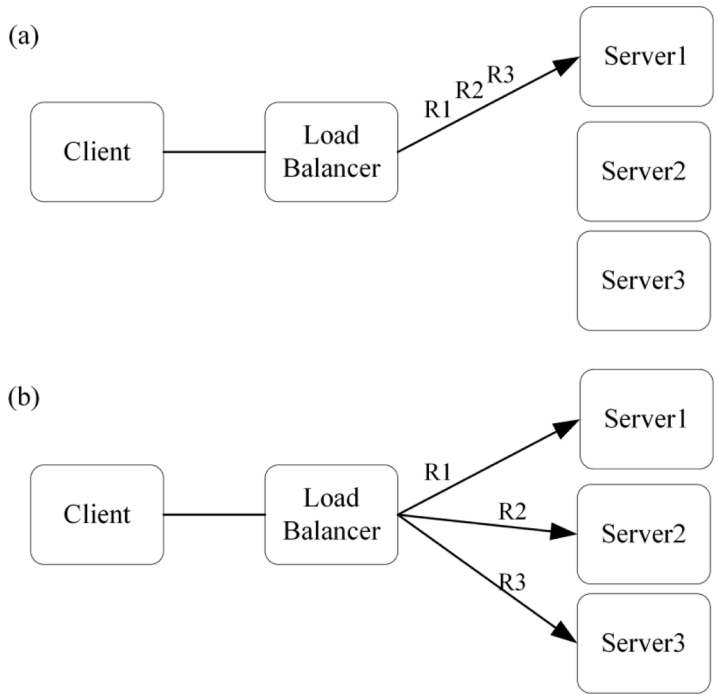


Fig. 3. Schematic representations of (a) connection-based load balancing and (b) protocol-based load balancing.

3.3. System design

The actual design of ECAIoT is a three-layer architecture [21, 22]. System pre-planning is necessary for easy implementation on the actual architecture. Selecting an appropriate software system is challenging owing to the wide variety of available technologies. However, before development, it is still necessary to hierarchize and modularize the system characteristics. For the proposed system architecture, we adopted the Machine Intelligence and Automation Technology (MIAT) Lab system design methodology, which is derived from the integrated computer-aided manufacturing definition (IDEF)[5]. The system uses a high-level language to hierarchize and modularize the system function planning. Then, system design is decomposed into independent modules from the top down. Fig. 4 presents the preliminary hierarchical structure obtained from using the IDEF0 design concept.

3.4. Kubernetes

Kubernetes is an open architecture for automatically deploying, scaling, and managing containers. The concept is derived from the Google Borg project, which can support many container tools, such as Docker or LXC/LXD (Linux containers). Although similar tools exist (e.g., Docker Swarm), Kubernetes is the most widely used because it enables easy expandability and the use of various components and documents. The largest augmented reality game in the world, Pokémon GO, uses Kubernetes technology, and the largest Kubernetes cluster on Google Compute Engine is associated with this game.

Kubernetes is a decentralized system consisting of master and worker nodes. The following components are on the master node:

- etcd: a distributed key value storage system called etcd
- API server: controller manager
- System scheduler: coordinates the current data and places the newly created pod on the node that can execute the pod.

The following components are on the worker node:

- Kubelet: the main system that communicates with the master node
- kube-proxy: used to connect with the outside world
- Pod: a set of containers

Fig. 5 shows the basic architecture and internal functionality of the Kubernetes cluster.

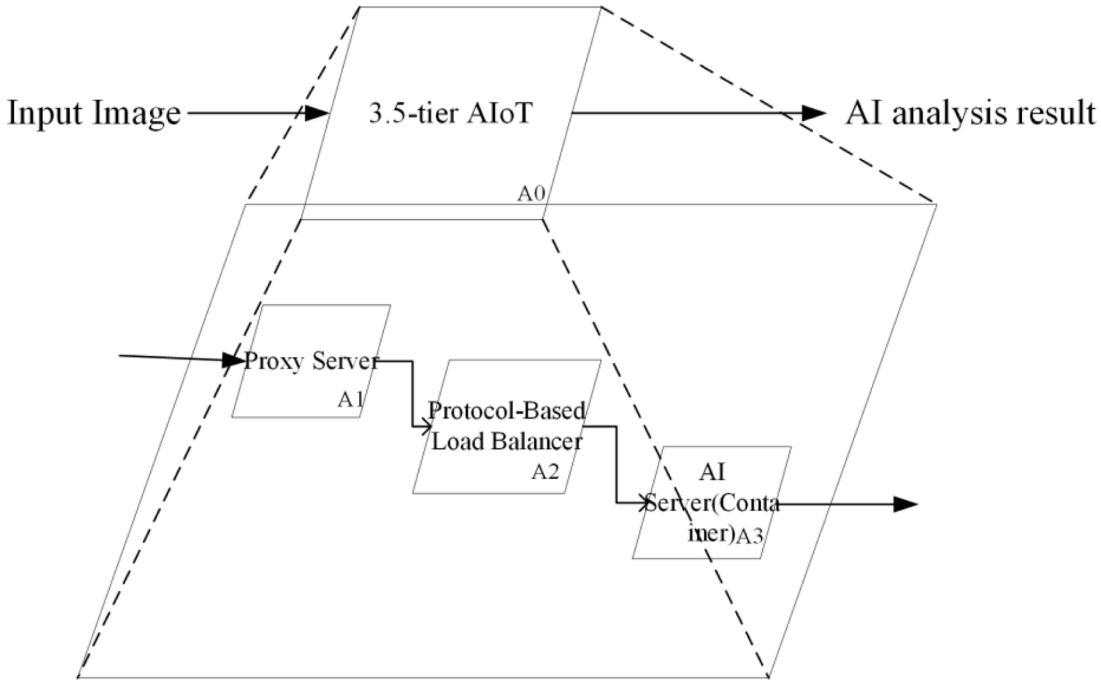


Fig. 4. Schematic representation of the IDEF0 hierarchical structure.

3.5. gRPC and ProtocolBuf

gRPC is a high-performance, open-source universal remote process call (RPC) framework. Although IoT has many communication protocols (e.g., MQTT, XMPP, CoAP, and AMQP), few protocols are suitable for ECAIoT, considering the original requirements, i.e., high-performance, low protocol consumption (uses binary code as the data-transfer format, not a text-based/readable language such as xml), and protocol-based load balancing supported by the software. After researching the available protocols, we selected gRPC because it met all of the requirements. gRPC is an open-source RPC framework and library developed by Google. It supports many programming languages, such as Python, Golang, C++, and C#. gRPC uses ProtocolBuf as the RPC interface description language, which enables users to generate interface code templates in multiple programming languages. These codes can exist alone or be mixed with other protocols or frameworks. They are generally used to replace data exchange languages, such as XML or JSON.

The gRPC library provides a gRPC server, which is called a gRPC stub on the client. It supports multiplexed data transmission and is an efficient and lightweight RPC protocol. gRPC supports authentication, bidirectional streaming, stream control, and super features by default and provides an efficient client with many possible applications, e.g., communication between services in a microservices architecture and connection of an application or browser in a mobile phone to the back end. gRPC can also be applied to high-performance IoT environments.

Further, gRPC uses HTTP/2 as a low-level communication protocol. HTTP/2 is suitable and compatible with most current environments. One major difference between HTTP/2 and 1.1 is the multiple request ability of the former, i.e., in the case of multiple requests, HTTP/2 will combine multiple transmission control protocol (TCP) streams into a single TCP connection without requiring a new connection to be built for each request. If data are requested continuously, HTTP/2 can save valuable connection establishment time [23]. Fig. 6 illustrates the gRPC protocol stack.

3.6. Linkerd2

Linkerd is a service mesh framework on Kubernetes. It mainly solves network and security problems caused by the microservices architecture or containerization. It provides runtime debugging, observability, reliability, and security without requiring the code or pod to be changed.

Linkerd2 is implemented by injecting a reverse-proxy program into Kubernetes pods. All network traffic into and out of the pods passes through this reverse-proxy program to control traffic. Linkerd2 has a control plane and a data plane. The control plane is responsible for collecting data, providing an API, and controlling the data plane. The data plane is a lightweight agent written in RUST.

We used Linkerd2 as the main component of the HTTP/2 and gRPC load balancer because, as mentioned earlier, a protocol-based load balancing architecture was necessary. Linkerd2 can be employed to analyze the gRPC protocol and run load balancing on it. However, to become integrated into the three-tier ECAIoT architecture, it requires a proxy as an intermediary layer, because pods are scattered on different machines, and when worker nodes receive a connection, they cannot redirect it to other worker nodes, but can only forward it. Given the scope of this study, we did not consider connection forwarding because it is not considered load balancing.

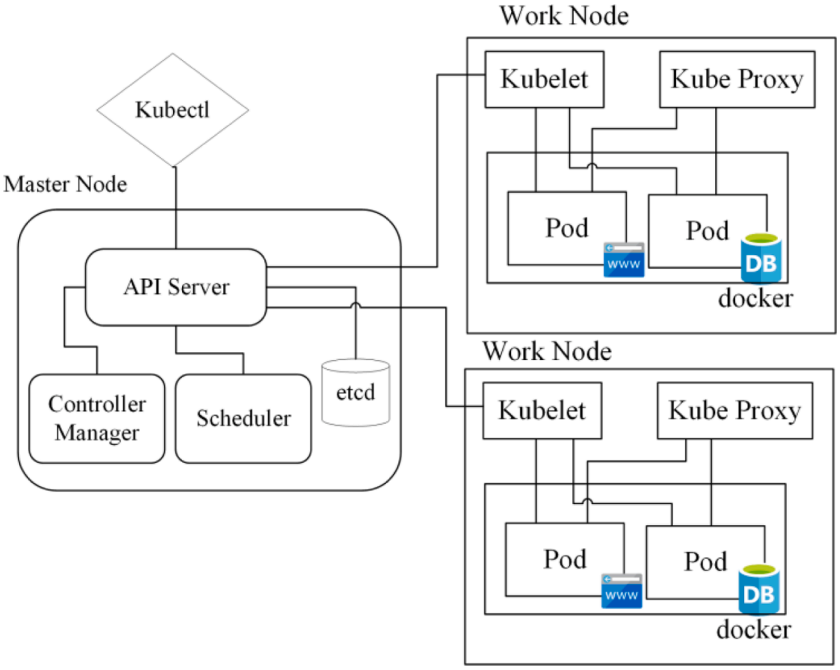


Fig. 5. Schematic representation of the Kubernetes architecture.

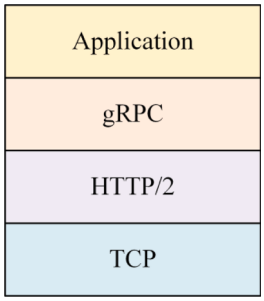


Fig. 6. gRPC protocol stack.

One option would be to perform load balancing on the client side, but the system cannot know beforehand how clients use the hardware resources. Therefore, the best strategy is to add a proxy between the client and the protocol-based load balancer. This proxy increases latency but also assists in transferring requests to the cloud when necessary. Fig. 7 is a schematic representation of the proxy–cloud connection process.

Linkerd2 offers another advantage in that it uses an exponentially weighted moving average (EWMA) as its load-balancing algorithm. As such, it will select a server offering a high level of performance according to recently acquired data, unlike the traditional round-robin (RR) algorithm whereby requests are simply sent to the next server. The EWMA algorithm thus improves server performance and usage. Since latency often occurs in AI applications and failure may occur in an AIoT environment, EWMA has a higher success rate and can also realize a higher level of performance than other algorithms like RR or last load. It can greatly improve overall system performance in an ECAoT environment.

4. System validation

4.1. 3.5-Tier ECAIoT hardware architecture

As a last step, we designed the upper-level system architecture by selecting a solution and investigating and developing the selected system. The following introduces the specifications and technologies used in this architecture from both the hardware and software perspectives.

Hardware:

In ECAIoT, embedded hardware is a priority. The selected hardware platforms included the following:

1. Raspberry Pi 4:

Raspberry Pi 4 allows for Kubernetes node provisioning, is powered by an ARM64 architecture with a quad-core Cortex-A72 (ARMv8) 64-bit SoC@1.5 GHz, and has 4 GB of RAM. It has enough computing resources to support EC microservice nodes.

2. Nvidia Jetson TX2:

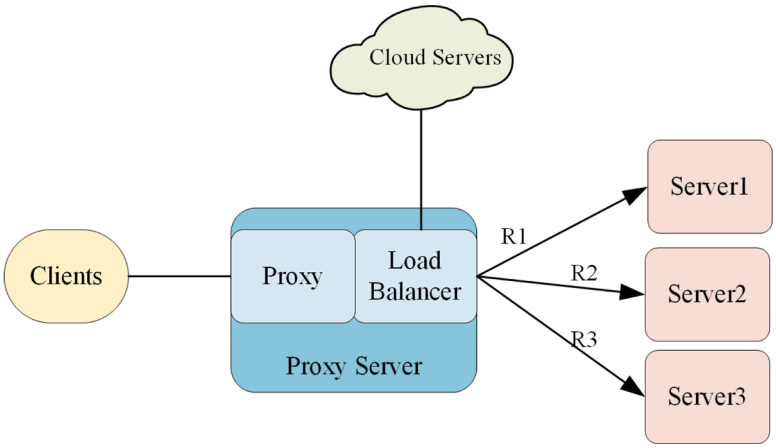


Fig. 7. Proxy server connecting to the cloud.

As an AI hardware worker node, we used Nvidia Jetson TX2. It is powered by a quad-core 2.0 GHz 64-bit ARMv8 A57 processor, a dual-core 2.0 GHz superscalar ARMv8 Denver processor, and an integrated Pascal GPU 1.3 GHz with 256 cores and 8 GB RAM. The GPU can provide up to 1.33 TFLOps AI engine performance, and a Huawei P20 Pro can provide 1.92 TFLOPs AI peak performance. Further, the P20 Pro has been used in AI for photo scene recognition [24]. As a cellphone is a small embedded AI environment, the CPU/GPU hardware can provide sufficient computing power for AI and running the Kubernetes work node. In addition, the Jetson TX2 only consumes 7.5 W of power (maximum 15 W), which enables it to be used as an embedded AI platform or edge IoT scene applications by changing its power-efficiency modes [25].

Both types of hardware are embedded software architectures and can execute the AIoT/EC architecture, which can be used to verify the ECAIoT architecture.

Software:

1 Container infrastructure:

Kubernetes, the most popular container technology, is mainly used to help manage containers and build services, such as application allocators or AI algorithms.

1 Communication protocol:

gRPC is a high-efficiency and low-bandwidth protocol that supports various programming languages. It is based on protocol buffers and a universal remote procedure call (RPC) framework.

1 Proxy server:

The NGINX server is used for the gRPC proxy, and the NGINX HTTP server is one of most popular HTTP and reverse proxies in the world. It is designed to be a high-performance and low resource consumption HTTP server. In a Netcraft worldwide survey, it took first place with a percentage of 32.69%.

1 Protocol-based load balancer:

Linkerd2 is a service mesh for Kubernetes that can also perform gRPC load balancing.

1 AI container:

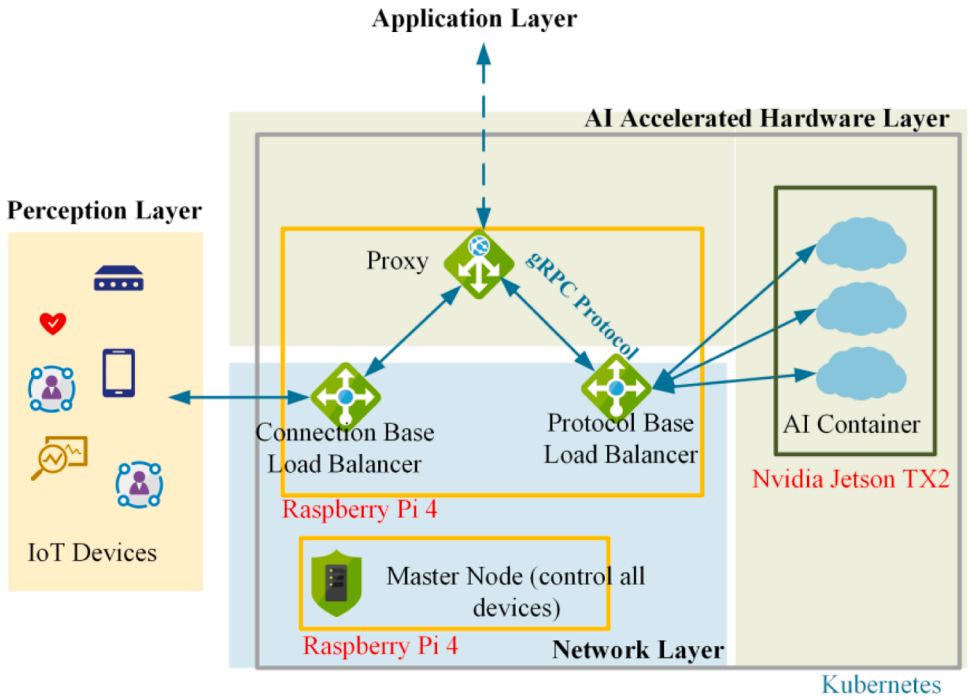


Fig. 8. Schematic representation of the 3.5-tier container-based ECAIoT architecture.

Keras and Python were used to develop the AI containers. Python is the main development language in current AI operations. Keras is a popular AI framework that can connect various AI computing engines. TensorFlow was used as the underlying AI layer.

The abovementioned hardware and software were coupled with the system architecture design presented in Section III.C. Fig. 8 presents the proposed 3.5-tier container-based ECAIoT architecture with real implementation and planning.

The following sub-sections introduce the main techniques for implementing this architecture.

4.2. Hardware environment

For this study, we used the following hardware.

1 Two Raspberry Pi 4s: one for the Kubernetes master node and one for the proxy server
2 Three Nvidia Jetson TX2s: AI hardware, Kubernetes worker node, Max-P ARM power-consumption mode (default)
3 An ASUS VC65 i5×86_64: IoT devices client that can send simultaneous connections/requests

4.3. Experimental evaluation

Firstly, we measured the number of clients and the response speed. The AI model used for testing was the Keras VGG16 [26] . VGG16 is a popular model, but the Keras pre-training model is too large to run on a Jetson TX; therefore, the model was slightly modified to fit the embedded environment. Kaggle Dogs vs Cats supplied the testing images. The parameters were as follows: loss value: 0.0017, accuracy: 0.9995, verification loss value: 0.191, and verification accuracy: 0.9640. One to three hardware devices were tested. The server returned the analysis results and timestamp to the client.

Remote login was used to execute the test program and calculate the number of predicted images per second and prediction time per image in milliseconds. When collecting data, the data for the first 20s is discarded, and an average will be obtained by testing for $\geq$ 40s, to attain a stable state earlier and thus obtain more accurate data.

We also measured the approximate power consumption of the system for reference: this system has two Raspberry pi 4 units and three Nvidia Jetson TX2 units, and the test results are shown in Table 2.

System standby means that no test is running yet, just the system with the default applications. System on full-operation means that 8 clients are run on the x86 platform (the client's power consumption isn't included), and 3 servers on 3 Nvidia Jetson TX2s. Power meters were used to measure the power consumption of the whole system, the Nvidia Jetson TX2, the Raspberry Pi 4 on 110V side, and USB power of the Raspberry Pi 4 on the USB side. The measurements were made several times and an average result was calculated. We observed an approximate deviation of 3% from the power meters when on system standby and an approximate deviation of 15% when in full-operation.

When the system is under full-operation, the power meter shows that the power usage varies greatly, and one cannot simply summarize the Raspberry Pi 4 and Nvidia Jetson TX2 power consumption as whole system power consumption due to the deviations during full-operation (Fig. 9)

4.4. Protocol-based load balancing

Firstly, we confirmed that the program could correctly allocate each analysis request made by the client to each AI hardware. Fig. 10(a) shows the status of a single client request. RPS denotes the number of requests per second, and in P99 and P95, P denotes "percentile" and the subsequent number is the relevant percentile value, e.g., P99 means 99th percentile, corresponding to the slowest 1% of average request delay times. In the center box of Fig. 10(a), RPS stands for requests per second. The proxy processes 7.88 requests/s. The three RPS numbers shown in the boxes on the right of Fig. 10(a) indicate how many requests that server has processed; It can be seen that each piece of AI hardware has allocated resources to assist with predicting the image requirements. The third server tends to have fewer resources, but the figure is not always low, and is always changing. Although the average loading is not balanced, each piece of AI hardware received requests, which means that the purpose of the original system design has been met and the connection-based load balancing problem is solved such that there is no AI job request that concentrates on one or several servers so that the system resources can be better utilized. Fig. 10(b) shows the simulation of multiple IoT clients. At this point in time, requests from clients exceed the maximum capacity of three servers. At this time, it can be seen that the servers are at the limits of their capacity. As such, there are no server resources available anymore, which also solves the problem of connection-based load balancing.

Table 2
System power consumption.

Measurement Item	System standby	System on full-operation
Raspberry Pi 4	3.5W (110V side)3W(USB side)	3.7W (110v side)3.2W(USB side)
Nvidia Jetson TX2	3W	12W
Whole system	16W	45W

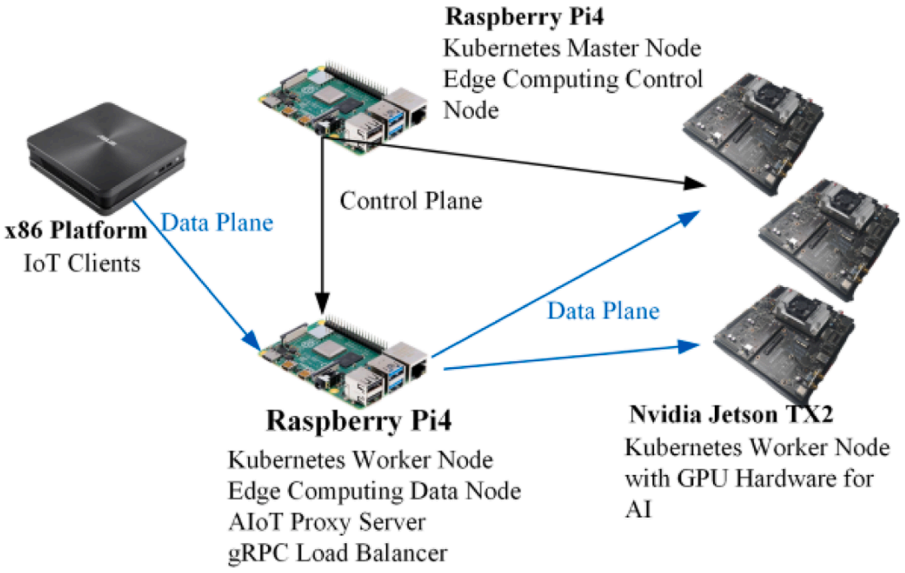


Fig. 9. depicts the system validation environment.



Fig. 10. Screenshots of the client request status for (a) a single client and (b) multiple clients.

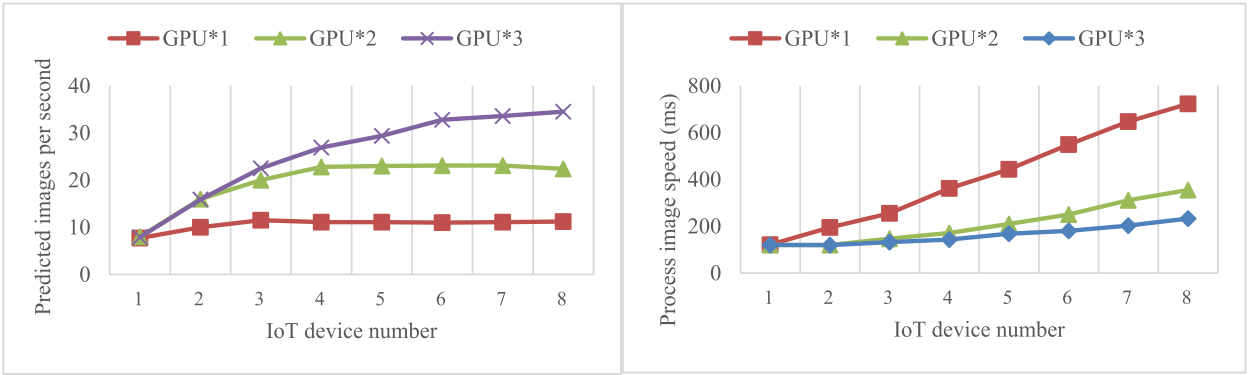


Fig. 11. Performance test results: (a) analyzed images per second and (b) process image speed in milliseconds

4.5. Architecture performance test

Performance tests were conducted for one to eight IoT device client(s) and one to three piece(s) of AI hardware. The purpose of this experiment was to determine whether by expanding the AI hardware to reduce the operation delay at heavy loading it could provide high-efficiency calculations in the ECAIoT environment. Fig. 11 shows the test results, where the pieces of AI hardware are denoted GPU*1(one GPU device), GPU*2(two GPU devices), and GPU*3(three GPU devices). The method used to calculate the predicted number of images/s involves having the computing machine process several images. If there are several machines, the processing results are totaled. The image processing speed is calculated by calculating the average time that elapses from the sending of an image to the receiving of that image within a given block of time. The method used to calculate the image processing speed involves determining the difference between the send data time to the receive data time, then dividing by the elapsed time to get the expected image processing speed.

Fig. 11(a) indicates that each piece of AI hardware could process approximately 8 AI requests per second. As the load on the AI hardware side increases, the number of requests that can be processed also increases. Fig. 11(b) shows the response time from when the client sends the image to when the client receives the image. When one piece of AI hardware receives many client requests, the image processing time increases significantly. In the same situation where there are eight AIoT clients transmitting data simultaneously, when three AI hardware provide computing power at the same time, the process image speed can be maintained at 240 ms. If the image-processing speed is calculated by determining the average time required to obtain the test result shown in Fig. 11(b), we can see that the image processing time for one client is the same, but among the eight clients, the image processing time of one server is very high, so we measure image prediction time from start to the end on the server to enable an analysis of the system.

The test results are shown in Fig. 12. It can be seen that, if only one client sends a request, the image prediction time is around 99 ms, regardless of the number of servers, and will always be the same. This means that, in this system, even it still has a margin, the fastest processing speed for a single thread is 99 ms. This behavior, caused by the python gRPC library, only supports synchronous transfer. It cannot send multiple requests simultaneously. Thus, multiple clients can also send more requests to the server at any one time. When the number of clients is increased to 2, the best prediction performance is obtained with a single server. Given this result, we can conclude that, with an increase in the number of clients with a similar prediction time, the image processing time will increase because the request queue is longer, such that more time is needed for dequeuing. When there are multiple servers, the prediction time curve is better, and thus the response time is also better. This implies that hardware with multiple GPUs will produce better results, in that this load-balancer algorithm uses EWMA as its load balancing algorithm. It will thus automatically send a request to the fastest server, so that this system processes/predicts images more effectively.

These test results were in line with our expectations; by increasing the number of servers, coupled with the characteristics of EWMA load balancing, priority is given to the sending of requests to faster servers, such that the AI hardware can provide more computing power. With an increase in the computing power, the system's latency is reduced.

This result is a good reference for ECAIoT environments that need to maintain low latency. Furthermore, it indicates that we successfully extended this architecture to more severe IoT clients, while maintaining a certain processing latency, which is in line with our goal.

4.6. Pre-training model performance test

Because this is an ECAIoT environment, not a common PC environment, for embedded systems we need a reference value for future model selection. Sometimes a more powerful or less memory-efficient model is selected in the embedded system. Herein, we used Keras pre-trained models, which can be used for future reference on the same performance comparison base. Considering the computing power and memory capacity of this ECAIoT environment, we selected some models that can connect 12 AIoT devices concurrently without problems. The following models were selected: InceptionV3, MobileNetV2, and NASNetMobile. ResNet152V2 was selected as the maximum size model that can run on Jetson TX2. The image files were preset to dimensions of 224×224.

Fig. 13 presents the performance speeds of various pre-training models when used on the proposed architecture. InceptionV3 and MobileNetV2 have good performances and small response times. They are followed by NASNetMobile, whereas ResNet152v2 is the slowest. These test results can be used as a reference for future research; it is not suitable to choose ResNet152V2 as model if processing time or speed is essential in a sensitive environment.

5. Conclusion

AI hardware has not been scaled to the EC AIoT environment previously. We designed a new concept of 3.5-tier ECAIoT architecture that can clearly show the EC needs for more proximity to the data production side. On the basis of this concept, a new ECAIoT architecture based on microservices was developed. The architecture has protocol-based load balancing that can be highly efficient in sending requests to multiple pieces of AI hardware evenly and to fulfill the AI hardware resource requirement. Further, we designed a cat and dog recognition AI application to prove the feasibility and performance of the ECAIoT architecture.

The experimental results indicate that implementing more clients improves the performance in terms of the response time and processing speed and that the ECAIoT architecture can execute various AIoT/EC architecture applications. In addition, it can expand and flexibly scale on the proposed architecture. Compared with the conventional cloud computing architecture, the ECAIoT architecture is more lightweight and lower cost. Further, embedded hardware can be used to build the system. The AIoT/EC environment can meet various needs through changes in the hardware architecture. Finally, we demonstrated that an increase in the amount of AI

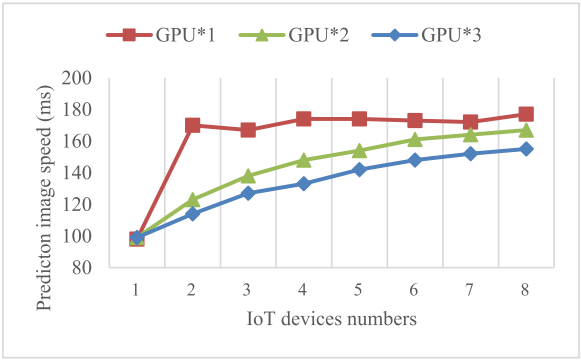


Fig. 12. Image prediction time on server.

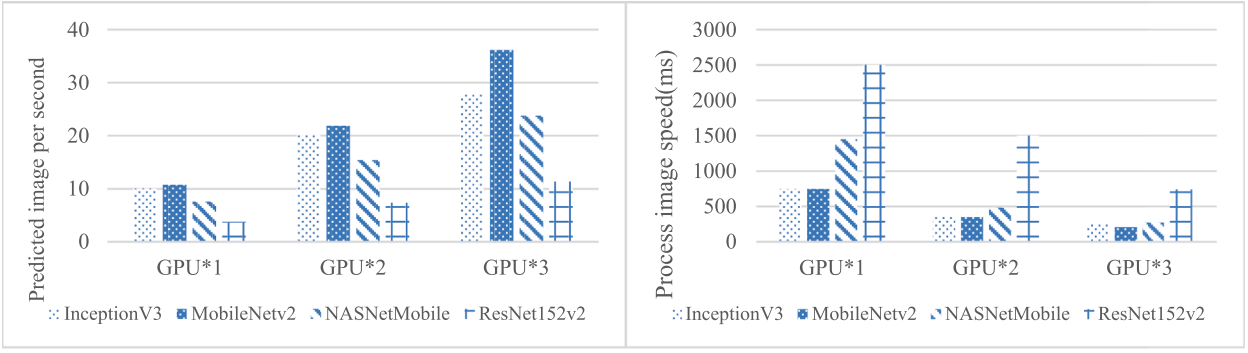


Fig. 13. Keras pre-training model performance test results: (a) number of images analyzed per second and (b) image processing time in milliseconds

hardware enables more IoT clients to be served and keeps the EC latency low.

In this study, we developed a feasible container ECAIoT architecture that could be implemented in other studies. In future research, we intend to conduct experiments using the ECAIoT architecture, e.g., in the field of agriculture. We intend to deploy an ECAIoT system for animal behavior recognition, which can cope with situations such as those presented in this paper, e.g., limited connection and resource-constrained environments.

Declaration of Competing Interest

The authors declare no conflict of interest.

References

[1] Omoniwa B, Hussain R, Javed MA, Bouk SH, Malik SA. Fog/edge computing-based IoT (FECIoT): architecture, applications, and research issues. IEEE Internet Things J 2018;6:4118–49.
[2] Lin J, Yu W, Zhang N, Yang X, Zhang H, Zhao W. A survey on internet of things: architecture, enabling technologies, security and privacy, and applications. IEEE Internet Things J 2017;4:1125–42.
[3] Lai Y-H, Chen S-Y, Lai C-F, Chang Y-C, Su Y-S. Study on enhancing AIoT computational thinking skills by plot image-based VR. Interactive Learn Environ 2019:1–14.
[4] Pham H-T, Nguyen M-A, Sun C-C. AIoT Solution survey and comparison in machine learning on low-cost microcontroller. In: 2019 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS). IEEE; 2019. p. 1–2.
[5] Chen C-H, Lin M-Y, Liu C-C. Edge computing gateway of the industrial internet of things using multiple collaborative microcontrollers. IEEE Network 2018;32:24–32.
[6] Morabito R, Cozzolino V, Ding AY, Beijar N, Ott J. Consolidate IoT edge computing with lightweight virtualization. IEEE Network 2018;32:102–11.
[7] Satyanarayanan M. The emergence of edge computing. Computer 2017;50:30–9.
[8] Porambage P, Okwuibe J, Liyanage M, Ylianttila M, Taleb T. Survey on multi-access edge computing for internet of things realization. IEEE Commun Surv Tut 2018;20:2961–91.
[9] Shi W, Cao J, Zhang Q, Li Y, Xu L. Edge computing: Vision and challenges. IEEE Internet Things J 2016;3:637–46.
[10] Pan J, McElhannon J. Future edge cloud and edge computing for internet of things applications. IEEE Internet Things J 2017;5:439–49.
[11] Li H, Ota K, Dong M. Learning IoT in edge: Deep learning for the Internet of Things with edge computing. IEEE Network 2018;32:96–101.
[12] Tran TX, 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.
[13] 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.
[14] Zhou Z, Liao H, Gu B, Huq KMS, Mumtaz S, Rodriguez J. Robust mobile crowd sensing: when deep learning meets edge computing. IEEE Network 2018;32:54–60.

[15] Zhou J, Velichkevich A, Prosvirov K, Garg A, Oshima Y, Dutta D. Katib: a distributed general AutoML platform on Kubernetes. In: 2019 {USENIX} Conference on Operational Machine Learning (OpML 19); 2019. p. 55–7.
[16] Chang C-C, Lee W-K, Liu Y, Goi B-M, Phan RC-W. Signature gateway: offloading signature generation to IoT gateway accelerated by GPU. IEEE Internet Things J 2018;6:4448–61.
[17] 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.
[18] Chen L. Microservices: architecting for continuous delivery and devops. In: 2018 IEEE International Conference on Software Architecture (ICSA). IEEE; 2018. p. 39–397.
[19] Tsai P-H, Hong H-J, Cheng A-C, Hsu C-H. In: Distributed analytics in fog computing platforms using TensorFlow and Kubernetes. 2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS). IEEE; 2017. p. 145–50.
[20] Boltunov A, Vasilev S, Karpenko V, Voloshin A, Voloshin E. Modelling a distributed computing system for BigData processing and managing algorithms implementation in MicroGrids. In: 2019 2nd International Youth Scientific and Technical Conference on Relay Protection and Automation (RPA). IEEE; 2019. p. 1–15.
[21] Kulik V, Kirichek R. The heterogeneous gateways in the industrial internet of things. In: 2018 10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT). IEEE; 2018. p. 1–5.
[22] Dolui K, Datta SK. Comparison of edge computing implementations: fog computing, cloudlet and mobile edge computing. In: 2017 Global Internet of Things Summit (GIoTS). IEEE; 2017. p. 1–6.
[23] Williams M, Benfield C, Warner B, Zadka M, Mitchell D, Samuel K, Tardy P. Twisted and HTTP/2. Expert Twisted. Springer; 2019. p. 339–63.
[24] Ignatov A, Timofte R, Chou W, Wang K, Wu M, Hartley T, Van Gool L. AI benchmark: running deep neural networks on android smartphones. In: Proceedings of the European Conference on Computer Vision (ECCV); 2018. p. 0-.
[25] Blanco-Filgueira B, García-Lesta D, Fernández-Sanjurjo M, Brea VM, López P. Deep learning-based multiple object visual tracking on embedded system for IoT and mobile edge computing applications. IEEE Internet Things J 2019;6:5423–31.
[26] Qi X, Liu C. Enabling deep learning on IoT edge: Approaches and evaluation. In: 2018 IEEE/ACM Symposium on Edge Computing (SEC). IEEE; 2018. p. 367–72.

Ching-Han Chen received D.E.A in informatique from Automatique et Productique in 1992 and Ph.D. from Franche-Comte University in 1995. He has been an Associate Professor in the Department of Electrical Engineering, I-Shou University, and is currently a Professor of Computer Science and Information Engineering at National Central University, Taiwan. His research interests include embedded system design, machine vision, and robotics.

Chao-Tsu Liu received B.Sc. in electronic engineering from National Yunlin University of Science and Technology in 2000 and M.S. in computer science and information engineering from National Central University, Taiwan, in 2012. He is currently pursuing a Ph.D. at National Central University. He has expertise in network domains, SoCs, embedded systems, cloud computing, LTE base stations, and IoT devices.