Journal of Technology Informatics and Engineering
Vol.1, No.1 April 2022
e-ISSN: 2961-8215; p-ISSN: 2961-9068, Hal 01-29

BLACK BOX APPROACH TO MONITORING CONTAINER MICROSERVICES IN FOG COMPUTING

Danang[1], Nuris dwi setiawan[2], Indra Ava Adianta[3]
Universita Sains dan Teknologi Komputer

Abstract

In recent years IoT has developed very rapidly. IoT devices are used to monitor and control physical objects to transform the physical world into intelligent spaces with computing and communication capabilities. Compared to cloud computing, fog computing is used to support latency-sensitive applications at the edge of the network which allows client requests to be processed faster. This study aims to propose a monitoring framework for containerized black box microservices in a fog computing environment to evaluate CPU overhead, as well as to determine the operating status, service characteristics, and dependencies of each container.

This study proposes a monitoring framework to integrate computing resource usage and run-time information from service interactions using a black box approach that seeks to integrate service-level information and computing resource information into the same framework. The proposed framework is limited to observing information monitoring after the server receives a request. This study uses JMeter to simulate user actions, which send requests to the server, and this research assumes the user knows the IP address of the server. For container monitoring methods in fog computing, all are indirect monitoring methods.

The results of this study indicate that the proposed framework can provide operational data for visualization that can help system administrators evaluate the status of running containers using a black box approach. System administrators do not need to understand and modify target microservices to gather service characteristics from containerized microservices. Regarding future research, it is suggested to expand the exploration of modified system information, and that part of the container management tool code can be pre-tried so that the framework proposed in this study can provide real-time quantitative indexes for the load balancing algorithm to help optimize the load balancing algorithm.

Keywords*: Internet of Things, Black Box, Micro Container, Fog Computing, Cloud Computing*

Received Maret 05, 2022; Revised Maret 20, 2022; April 13, 2022

INTRODUCTION

Until now the Internet of Things (IoT) has experienced rapid development. IoT devices consist of sensors and/or actuators, connectivity, and computing power. These devices are used to monitor and control physical objects. IoT devices are responsible for collecting data through sensors. Since IoT devices have limited computing power, cloud computing resources are used to analyze sensor data (Doukas & Maglogiannis, (2012)). In this design model, there is a clear division of tasks between the cloud and the IoT devices placed at the edge of the network. That is, the edge is responsible for data collection whereas the cloud is responsible for data processing and management of IoT devices. Cloud systems can combine data from various devices for IoT services. The cloud can collect data from temperature, humidity, anemometer sensors, sensitivity gauges, and air quality detectors to analyze weather forecasting information (Kaur & Maheshwari, (2016)). However, with the increasing number of IoT devices and data, as well as the physical distance between the network edge and the cloud data center, the volume of data to be transferred is expected to increase drastically resulting in a large network load. Even the promise of 5G isn't enough for some apps as 5G is for mobile networks. In addition to network congestion, cloud computing capabilities can also reach their limits during peak hours. Due to excess communication and computing resources, high latency will occur. For some services, high latency is unacceptable.

To overcome the challenges posed by network latency, the concept of fog computing proposed by Cisco (2015) has received widespread attention. Fog computing refers to the deployment of servers with low computing performance distributed near the edge of the network closer to the data source. These resources are used to provide data analysis, aggregation data processing, and other services. In this way, not all data needs to be sent to the cloud-based system, and most of the data processing can be done in the fog nodes. The communication load on the network core is greatly reduced. In the scenario of cloud computing and Internet of Things applications, another important technology is the containerization of microservices. The use of microservices has been proposed for use in software development as opposed to monolithic software. Traditional monolithic software design means all software functions are packaged in one program whereas microservices design can separate various functions in software into different programs. This design pattern provides more flexibility Sun et al, (2017). Programs that provide some of these software functions are called microservices. Microservices can be packaged in containers. IoT applications can consist of several microservices. Some of these microservices can be shared by various IoT applications. Microservices can be encapsulated in containers and can be deployed on different compute nodes. Microservices may need to be replicated due to high demand and this requires appropriate nodes to deploy services. There is also a need to be able to determine whether an IoT application is capable of meeting run-time requirements. This requires monitoring interactions among microservices. Furthermore, determining the nodes to host microservices must consider the potential for overload to avoid it.

In the use of fog computing to expand cloud computing, the orchestration of computing resources plays an important role. Most fog computing resource orchestration work assumes that monitoring of resources and service interactions is in place. The data required by resource orchestration frameworks are diverse and can be categorized into two categories. One is the consumption of hardware resources as represented by CPU, RAM, and network bandwidth in addition to service layer information, such as the number of requests received by microservices,

Journal of Technology Informatics and Engineering
Vol.1, No.1 April 2022
e-ISSN: 2961-8215; p-ISSN: 2961-9068, Hal 01-29

the error rate of requests processed by microservices, and dependencies between microservices (Mayer). & Weinreich, (2017)).

BACKGROUNDS

The development of the internet has brought more and more users and over time more complex needs. The advent of cloud computing has greatly reduced the cost of computing resources for developers and allowed clients to rent computing resources from cloud computing vendors who have data centers with strong computing resources. Developers can deploy their applications and services in data centers and pay according to resource usage. With cloud computing, developers can better control costs by deploying computing resources as needed. Cloud computing can provide dynamic and flexible computing and resources, scalability, stable backup, and simple deployment.

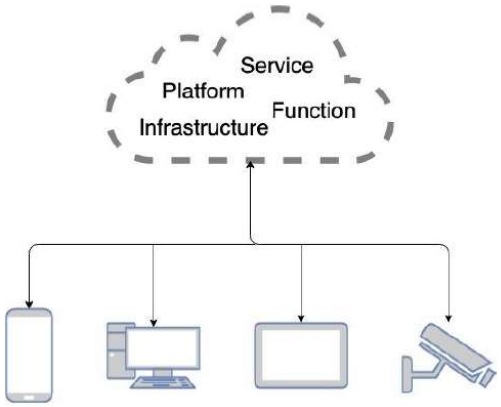

Figure 1: Cloud computing

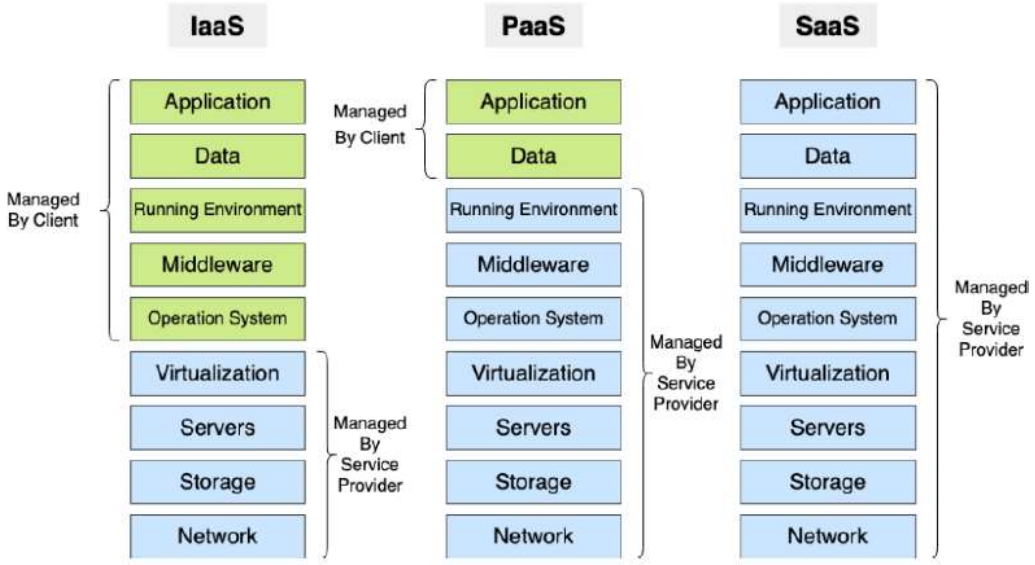

Figure 2: Cloud Service Types and Layers

Cloud computing provides various service models as presented in figure 2. The most common modes are IaaS, PaaS, and SaaS. IaaS can provide the hardware tools needed by applications, including servers, storage, and networking. Developers can lease these resources to deploy their operating systems, computing environments, and applications. IaaS service providers include Rackspace, Amazon Web Service, Microservice Azure, etc. IaaS and PaaS service providers also provide and manage operating systems, middleware, and software. The client only needs to focus on developing and maintaining the application. AWS Elastic Beanstalk and Google

App Engine are typical PaaS providers. SaaS is a way for service providers to directly deliver services to end users. Users can access data stored in the cloud via any device at any location. A representative example of such a service is Google Drive Storage, a Google application.

RELATED WORK

IoT application scenarios typically include sensors that generate data that are often periodic and often generate large amounts of data. Reliance on cloud computing means the transmission of data over long distances resulting in high latency for processing and the potential for sending large amounts of data which can lead to network congestion. This led Cisco to propose the fog computing concept Bonomi et al, (2017) which refers to the deployment of computing resources at the edge of the network closer to the data source. The computing resources/capacity of a fog node server is less than a cloud data center Yi et al, (2015).

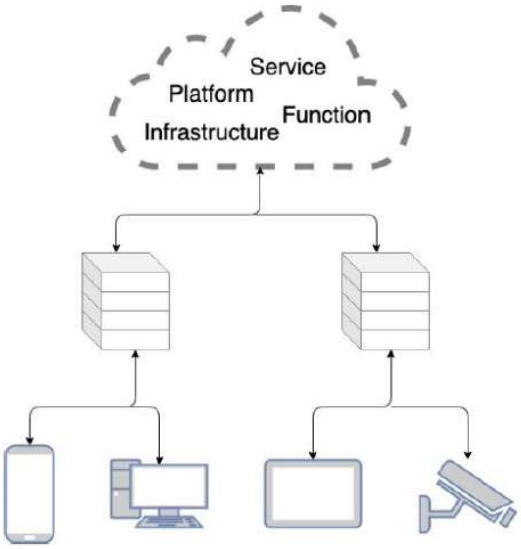

Figure 3: Fog computing

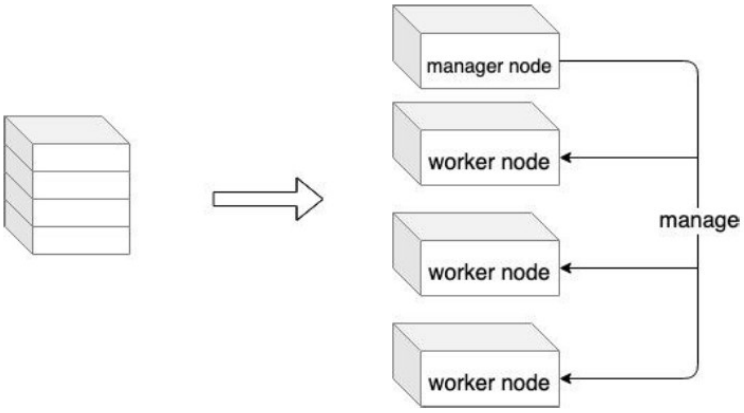

Figure 4: Mist Nodes

Compared to cloud computing, where all computing tasks are centralized and delivered in the cloud, fog computing is used to support latency-sensitive applications at the edge of the network

enabling client requests to be processed faster. This feature makes it possible to have latency-sensitive application scenarios such as autonomous driving Huang et al, (2017), healthcare Kraemer et al, (2017) smart factories Brito et al, (2016). If the computing performance of fog computing cannot support the client's demand, fog computing can be used as a gateway to aggregate and filter the original data, thereby reducing the communication burden from the client to the cloud. The application of fog computing varies to Yi et al, (2015), but its main feature is combining several computing devices with low computational performance into a cluster which is often referred to as a fog node. Among the various fog node implementation methods, the Raspberry Pi is considered a suitable and promising implementation method. Bellavista et al, (2017) demonstrated the feasibility of Raspberry Pi as a fog node computing device and claimed that fog nodes using Raspberry Pis had good scalability, flexibility, appropriate cost, and easy deployment features. Other research shows that the use of Raspberry Pi for fog computing to implement the Raspberry Pi fog node to process real-time data can be used on the fog node. Fog computing is an extension of cloud computing. However, extensions of this kind do more than just increase deployment volume and server density. Compared to cloud computing, fog computing is closer to the user, and relatively more dynamic in responding to usage scenarios, such as providing automated services for autonomous driving Huang et al, (2017), providing instant data processing and equipment management for smart factories Brito et al, (2016) and providing computing for smart city infrastructure, etc. These different application scenarios mean that fog computing has different characteristics, such as geographic awareness of equipment, real-time service migration according to user requirements, diversity of server hardware, and limited computing performance of hardware equipment.

In fog computing container monitoring and microservices are usually done separately with different designs and tools. Container monitoring is a type of virtual resource monitoring and its focus is to provide visibility of virtual machine resources and performance. Therefore, the indicators that are the focus of container monitoring solutions are Health (On/Off), Performance (CPU, RAM, Bandwidth, Storage), capacity, security, and power (Chandran & Walvekar, 2014). The focus of monitoring microservices is to ensure the stable operation and optimization of service applications. Therefore, indicators of concern are at the service layer, such as request tracking, service-specific error rates, and service interactions and dependencies (Mayer & Weinreich, 2017). These two monitoring methods have different logic and design goals, so they are often not integrated into the same framework. In a fog computing environment, the combination of these two technologies can be used well for various service scenarios.

LITERATURE REVIEW

To represent the information obtained in the study of the literature, this research takes the ideas of Yigitoglu et al, (2017), namely a review survey (Bonomi et al, 2014)), where the fog computing orchestration analysis must be circled by Probe (Monitoring), Analysis, Plan, and Execute, and the monitoring objective must provide the data required for orchestration and analysis. Then Yigitoglu et al, (2017) represent different algorithms in three different scenarios that may require different monitoring data. Yigitoglu et al, (2017) proposed compiling a task specification file that describes the characteristics of microservices to load the appropriate task on the appropriate server. Their research focus includes fog computing topologies, inter-node communication, and support for IoT devices. However, these works do not consider the use of containers.

Bonomi et al (2014) analyzed the characteristics of fog computing from the perspective of application scenarios and compositional structures. Cloud computing resources are managed centrally, and the composition of server resources is relatively more homogeneous. As an extension

of the cloud computing layer, the fog computing layer consists of heterogeneous server devices. This heterogeneous coverage includes high-end servers, edge routers, single-board computers, set-top boxes, and end devices such as vehicles or mobile phones. In a fog computing environment, the network infrastructure may also be heterogeneous (eg: LTE, and WiFi). To standardize the management of fog node devices, Bonomi et al (2014) defined a fog abstraction layer as shown in Figure 5. This fog abstraction layer hides device heterogeneity by defining devices from a computing resource perspective. Compute, storage, and network resources can be virtualized. Monitoring data will be used for service delivery and orchestration. Although the container monitoring solution is to provide monitoring data for container management/orchestration algorithms, under different requirements, there are different requirements for monitoring data. Yigitoglu et al. (2017) propose Foggy, which is an orchestration framework for containerized microservices in a fog computing environment. User Preferences and desired container behavior are defined using a JSON file. Foggy monitors resource usage of containers and fog nodes through CAdvisor. Foggy uses a self-matching algorithm to match each microservice with the most suitable fog node to support maximizing service quality. Placements can be adapted to changing resource requirements.

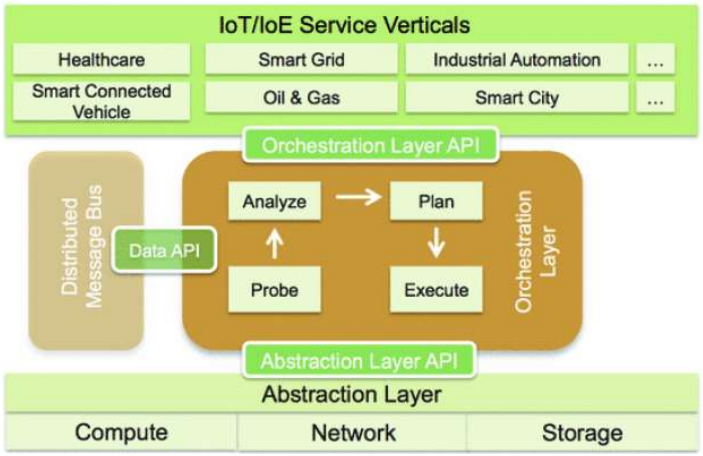

Figure 5: Abstract Mist Layers and Orchestration Circles (Bonomi et al, 2014)

Großmann & Klug (2017) developed PyMon which is a container monitoring framework for fog nodes. Großmann & Klug (2017) observed that a cloud monitoring solution transplanted into a fog computing environment is ineffective because this solution does not take into account that the fog computing nodes are not as powerful as cloud servers. Großmann & Klug (2017) introduced PyMon a lightweight monitoring solution for single-chip computers. At PyMon, the collection of monitored data is done via Monit. Monit is an open-source Unix/Linux system monitoring tool. Firdhous et al., (2014) explained that Monit can request system information and send monitoring data via HTTP. Monit is installed on every worker node. Monit can periodically collect host information and generate XML files. However, Monit doesn't support monitoring container information, so the authors modified Monit with additional information that includes the container's CPU and RAM usage, image name, and state. Host information and additional container information are included in the XML file. XML files are periodically sent to the manager's server via HTTP. On the server (master node), PyMon provides support for data aggregation and filter

processing. The pre-processed data is entered into a Postgres SQL database on the management server for long-term storage.

After completing PyMon, Großmann et al., (2017) conducted a PyMon evaluation study. The purpose of the evaluation study is to verify whether PyMon and monitoring tools in the cloud computing field can adapt to the fog computing environment and whether future research directions should continue to focus on reducing hardware overhead. Therefore, they selected two tools commonly used in the field of cloud computing monitoring and deployed them in the fog computing environment. The tools are Prometheus and CAdvisor. Prometheus, as the data aggregation server used on the manager node, corresponds to the MonitCollector role in PyMon. CAdvisor in the Prometheus stack is responsible for gathering hardware information on worker nodes. This is similar to using Monit in PyMon. The evaluation results are shown in Table 1. Compared with the implementation algorithm described in Yigitoglu et al. (2017) and the machine learning model in Jalali et al, (2017) this approach to using simple rules can reduce the communication burden between clients and servers by applying rules to filter monitoring data. Using rules can be used to reduce monitoring overhead, but it also means that large amounts of data cannot be stored for the long term. In some service orchestration algorithms as described by Yigitoglu et al, (2017) there is a reference to container orchestration based on information from the service layer. Therefore, the framework proposed in this study seeks to integrate service-level information and computing resource information into the same framework. Prometheus has lower CPU consumption than PyMon. The PyMon developers suggest that with active open-source community development, open-source software can be used to monitor containers on mist nodes.

	PyMon			Prometheus		
	Component	CPU Usage	RAM usage	Component	CPU Usage	RAM usage
container info	Monit	28.69%	13 MB	CAdvisor	12.51% CPU usage	50-60MB
Aggreatation	Monitcollector	20% - 90%	75MB	Prometheus	6%	500MB
Visualization	Web UI	/	/	Grafana	/	/
Storage	SQLite	/	/	MySQL	/	/

Table 1: Comparison of PyMon vs Prometheus

White-box live monitoring refers to directly collecting performance metrics such as the number of requests, amount of concurrency, and average response timeout. This is done by inserting code in the service that allows assigning a unique trace identifier to requests. This enables request tracking. Black box direct monitoring does not require any code to be entered into the application for monitoring purposes. The black box service log collection solution for microservices in this study is proposed, and the "Metro Funnel" tool in this study is developed to sniff HTTP request information on each service port. This research is not targeted to measure the performance of microservices in containers or error detection. However, this black box monitoring method can be used to improve service performance by increasing the number of microservices replicas based on request timeout rate analysis. Use black box monitoring to monitor the number of requests per microservices, response times, and dependencies between microservices. The centralized gateway collects and routes all microservices requests and responses. Any requests from clients or information communication between containers will be routed by this centralized gateway. Through the information collected, the framework can provide information, such as the number of requests per microservices, response times, and dependencies between microservices. To implement this centralized proxy monitoring architecture, they modified the ZUUL Netflix gateway to route requests between microservices. The experimental environment is for Docker

containers and the HTTP communication protocol. ZUUL collects, aggregates, and filters microservices communication information by reading HTTP headers.

Computing Resource Virtualization

If a cloud provider customizes different hardware for each user, it will cost a lot of manpower, time, and resources. Therefore, cloud providers solve this problem through resource virtualization. Compute virtualization technology uses virtualization management software (Hypervisor or Virtual Machine Manager) to separate physical server hardware resources from upper-layer applications to form a unified pool of computing resources, which can then be flexibly allocated to logically isolated virtual machines or containers for shared use. The advantage of this virtualization technology lies in dynamic computing resource planning (Arora et al, 2011), increased utilization, manageability, and reliability (Uhlig et al, 2005). Currently, the two widely used virtualization implementations are through virtual machines or container technologies.

Containers and Virtual Machines

Container technology (Jimenez et al, 2015) refers to application virtualization, where each application has its own independent user space. The container includes the code, system tools, libraries, and environment configuration required by the applications hosted in the container. Deployment of container technology allows developers to focus more on developing and deploying applications rather than deploying a development environment over and over again. The components needed to run the program are packed into an image file. Image files can be loaded into the container for execution. This reuse and convenience greatly increase the flexibility and scalability of services. Containers can also download image files from image file storage repositories (such as Docker Hub) for fast deployment. Virtual machines are different from containers in that virtual machines can use a different operating system than server operating systems. The virtual operating system will run like any other program on the server. A virtual machine (VM) can virtualize different operating systems on a host to adapt to the different system requirements of the system environment. However, a unique feature of container technology is that all applications in a container can use the same container engine, avoiding consuming system resources by requiring its operating system. Container creation does not need to allocate fixed memory and disk storage like virtual machines Sharma et al, (2016). Therefore, compared to virtual machines, containers are lighter, and the utilization of the computing resources of the host hardware is more flexible and dynamic Sharma et al, (2016). In terms of data, the minimum amount of RAM resources used by a container can be as small as 5MB while the smallest resource usage required by a virtual machine is 250MB.

Container Orchestration

Containers placed on the same host share the same operating system. To run multiple containers efficiently on a single host, no container starves CPU, memory, or other network I/O containers. Thus, as the number of containers increases, the complexity of resource management also increases. Container orchestration tools are needed to manage containers and applications. Tools widely used in the industry include Docker Swarm and Kubernetes. Docker Swarm is the official native management tool for Docker containers. By using it, users can package multiple docker servers into one large virtual docker cluster to quickly build container platforms. Kubernetes Sayfan (2017) is a container platform designed by Google. Kubernetes has more features such as

alerts and visualizations. At the 2017 Docker Conference, Docker announced that it will provide native support for Kubernetes. Research by Großmann & Klug (2017) shows that Kubernetes uses more resources than Docker swarm. However, Kubernetes provides more features. In short, Docker Swarm is known for its native lightweight deployment, and Kubernetes provides more robust functionality. The tools necessary for management include container monitoring and container host systems. Management tools are usually integrated with tools designed for container monitoring.

Microservices and Tracking

The concept of microservices is intended to adapt to a more dynamic and flexible computing resource environment Nadareishvili et al, (2016). Microservices is a software architecture consisting of multiple independent services where each service is responsible for a single function. The idea is not to develop a large monolithic application but to decompose the application into small interconnected microservices. One microservice completes certain functions, such as passenger management and order management. Each microservice has its business logic. Some microservices also provide API interfaces for other microservices and application clients. Compared to monolithic applications, microservice architecture has the advantages of low coupling and better maintenance. In a monolithic application, small changes can affect the deployment of the entire application. Modification of one module may require coordination of other modules. This type of maintenance requires the programmer to have a sufficient understanding of the entire application architecture. In a microservices architecture, changes made by programmers to one microservice will not affect other microservices.

The disadvantage of today's microservices architecture compared to monolithic applications is troubleshooting. When one application fails, system maintainers can solve the problem by reading the application logs on a single server. Each microservice has its log storage format and method, and each microservice can be deployed on a different server. This feature increases the cost of troubleshooting system failure points. Compared to the traditional monolithic service system, in a microservices architecture, user requests may need to access multiple microservices deployed on different servers. In a monolithic system, the system architecture is relatively fixed and stable. If errors and abnormalities are found by real-time monitoring, system administrators can quickly find abnormal servers and deal with them quickly. With microservices, components in different containers may have multiple replicas as working instances, and these replicas are deployed on different machines. This layer of low-coupling system architecture has the advantages of flexibility and scalability in large cluster deployments. However, these distributed deployments present challenges for monitoring and tracking. Each request can be forwarded between multiple stateless microservices via API interaction, and these microservices can be distributed across different servers. Therefore, in the industry, there are also many service layer monitoring tools developed to track the performance of microservices such as ZipKin (Chandran & Walvekar, 2014), Dapper (Sigelman et al, 2010), Dyna-trace (Mayer & Weinreich, 2017), etc. The approach of this tool is to assign a trace identifier to each request being tracked. A complete microservices trace chain record is created by combining records with the same tracking identifier together. The limitations of this method are that the system administrator needs to have a certain level of understanding of application design and this approach requires modifying the service code. This method is called white box monitoring, that is, the service function and system monitoring function are combined into one. From a development perspective, this increases the difficulty and complexity of development. Developers not only need to pay attention to business algorithms, but also need to understand DevOps monitoring, communication, and logic Nadareishvili et al, (2016). On the other hand, when system maintainers and developers are from different parties, this usually increases the difficulty of operation and maintenance.

Microservices in Containers

Because containers and microservices are dynamic and flexible, this combination is popular. Cockcroft et al (2014) believe that containerized microservices can multiply dynamic and flexible characteristics, making microservices more elastic and flexible. IoT applications are characterized by their integration with sensor data. Sensor data can be shared by multiple applications as well as multiple data analyses. Microservices can be replicated or have additional resources allocated as needed. Containers are often used to host microservices. Under this trend, the deployment of fog nodes at the edge of the network as gateways for IoT devices can solve latency problems effectively. However, the resource-limited nature and diversity of these gateways pose challenges to the development of widely applicable applications. Cziva et al. (2017) focused on this issue and demonstrated through experiments that deploying gateways through containerized services can improve the computational performance of IoT gateways. On the other hand, due to lower computational resource consumption and faster container deployment speed, container deployment is more flexible and faster than virtual machines and can adapt to dynamic user needs more quickly to achieve real-time expansion. Scaling and migration.

Virtualization of Container and Network Functions

As the number of network middleware devices that are deployed on the network increases. Issues such as high development costs, rapid updates, and difficulty in upgrading and deploying based on specific hardware are becoming increasingly prominent. These middleware or proprietary services often require specialized hardware to work with. Network function virtualization (NFV) aims to change the current situation faced by network operators. Network function virtualization (NFV) is a method of virtualizing network services (such as routers, firewalls, and load balancers) that have traditionally run on proprietary hardware. Currently, industry and academia tend to use virtual machine technology to implement NFV platforms (Cockcroft et al (2014)). With the emergence of container technology, containers are considered a technology to implement NFV in the future. Cziva et al. (2015) have conducted in-depth research using containerized NFV. Cziva et al (2015) believe that with the increasing number of users and new mobile devices, telecommunication service providers (TSP) often face the problem of low resource utilization, tight coupling with specific hardware, and lack of a flexible control interface and cannot support many mobile devices. applications and services. Therefore, the authors propose a framework for implementing NFV with containers instead of Virtual Machines (VMs) at the network edge. Because containers take up fewer hardware computing resources and are more flexible, TSP can reduce unnecessary use of the core network, better solve error problems, and provide users with location-aware and transparent services.

Service Network

When a container is selected as the environment running microservices, the service net is considered a way to supplement the tracking of microservices in the future. The service network monitoring solution is to use an associated traffic proxy called a "sidecar" for each container. All communication services related to the container will be processed through the sidecar. The service mesh architecture is relatively simple and consists of a two-tier architecture. One of them is the data layer (data plate). This layer deploys sidecars for each container. The sidecar can fully represent the request and response associated with the container. This task includes data packet

processing, forwarding, routing, load balancing, monitoring, etc. These sidecars can communicate with each other, and these communication records can be used to track microservices requests. The other layer is the control layer (control plate). This layer does not directly parse data packets, but communicates with the sidecar from the data layer, collects information from the data layer, and defines distribution/routing policies. In addition, the control layer can provide system administrators with APIs to facilitate configuration, monitoring, visualization, continuous integration, and deployment.

This architecture divides service communication, allowing developers to focus more on the logic of the service code. The related management and control functions from the communication and network layers are passed down to the infrastructure layer. In this design, service and communication codes are completely separated. The integration of sidecar auxiliary agents across distributed microservices systems will make the entire system less cumbersome and increase the difficulty of operation and maintenance. Since all information is proxied by the sidecar instead of communicating directly with the container, this will cause a slight delay. In some business scenarios, this kind of delay is intolerable. In addition, the current research on service mesh is all in the cloud environment without considering the characteristics of fog computing. When computational performance is limited and configuration processes need to be simplified, whether service mesh is applicable has not been well studied.

In the currently known literature, the current research focus of container monitoring in the fog computing environment is based on the system architecture, which mainly considers the flexibility and scalability of the system. Therefore, the monitoring requirements put forward by this fog computing container monitoring tool are related to the system, such as flexible data backend, acceptable performance overhead (Großmann & Klug, 2017), geographic awareness, etc. This type of work considers service dynamics in a fog computing environment with current research focusing on indirect monitoring at the hardware level for CPU, RAM, and Bandwidth. Compared to fog computing, a cloud computing environment has a fixed server capacity in a stable data center environment (Firdhous et al, (2014)). Cloud servers are usually more powerful and can provide more computing resources. There has been no work to determine whether black box monitoring methods in a cloud computing environment can be adapted to a fog computing environment. Among the four methods mentioned so, this is not a completely black box boxed microservices solution. Service mesh growth to computing resources will increase with the growth of the number of containers, which is not conducive to service scalability. Some representative service nets, such as AnyPoint Service Mesh, have a minimum hardware requirement of at least 8GB of RAM. Netflix ZUUL also has hardware requirements. This requirement is still too high for devices such as the Raspberry Pi and Arduino, which represent computing power suitable for a fog service device. In fog computing, it may have different system architecture from cloud computing, different computing resource constraints, and different application user scenarios.

Experiment Objectives and Assumptions

The two goals of this research experiment are to verify the feasibility of the framework and evaluate the CPU overhead of the monitoring framework. This is to evaluate what kind of information can be collected from the monitoring framework by simulating a real microservices application environment. In addition, JMeter is used to simulate user actions, which send requests to the server, assuming the user knows the server's IP address. In a real environment, the user will not directly access the server's IP address. Application providers typically allow users to access domain names and then use a request proxy tool such as Nginx to forward user requests to selected servers. The framework proposed here is only concerned with monitoring information after the

server has received a request. Therefore, the test used to send requests directly to the server is JMeter.

MONITORING FRAMEWORK ARCHITECTURE

Monitoring the run-time behavior measurement of applications consisting of one or more microservices deployed in a container that can be deployed across multiple mist nodes. The framework assumes a black-box approach to monitoring. For each container, hardware resource usage and performance are monitored to determine when microservices need to be replicated and where services can be successfully replicated.

Network Connection Information

Each service handles requests by performing one or more operations. Request tracking is a method used to profile and monitor applications built using the microservices architecture. Request tracking helps determine where failures are occurring and what is causing poor performance. Network connection information can be retrieved by analyzing network traffic packet headers from the application and transport layers. Traffic packets are parsed to extract network connection information.

Table 2 Table of Microservices Tracking Metrics

Metrics	Data type	Description
Source IP	string	The IP address of sender of the request
Source Port	string	The source port of the request sender
Method	string	The RESTFUL method such as "GET", "PUT"
Path	string	The request URL such as "/data"
Response Code	integer	The HTTP response status code
Container instance	string	The container ID of the container that which responds to the client
Start Time	Unix timestamp	The UNIX timestamp of representing when the fog node received the client request
Duration	Big int	The time difference between the fog node receiving the request and sending the response

Hardware Usage Measurement

Memory and CPU usage are considered the two most important pieces of information. In fog computing, sufficient network bandwidth is essential to minimize service delays, and therefore there is a need to aggregate the network traffic throughput of each container. The methods of gathering hardware and network traffic information differ. Network traffic information is passive because network tracking information is generated each time a client sends a request. Hardware monitoring is proactive. The monitoring agent periodically collects hardware consumption from the container. A scrape cycle is defined as the time interval between two captures of the container resource usage information which is referred to as a scape cycle. Scrape cycles usually range from 5 seconds to 1 minute. Each stroke provides the following information:

Table 3 Table of Hardware Metrics

Metrics	Data Type	Description
Container ID	string	The container ID of the container which corresponding to this record
CPU percentage	float	The CPU time usage since the previous check
Memory Usage	int	The memory usage in bytes
Memory Limit	int	The maximum memory of the server
Memory Percentage	float	The usage of the container
Netflow_in	int	The network traffic inflow bytes
Netflow_out	int	The network traffic outflow bytes
Check_time	big int	The timestamp for the checking time

Derivative Information

The information shown in Table 2 and Table 3 is stored in the database. This information can be used to obtain service level information through different request methods to adapt to different needs. The fog node architecture consists of several compute nodes. One of these compute nodes serves as the local manager and the rest are used to host microservices.

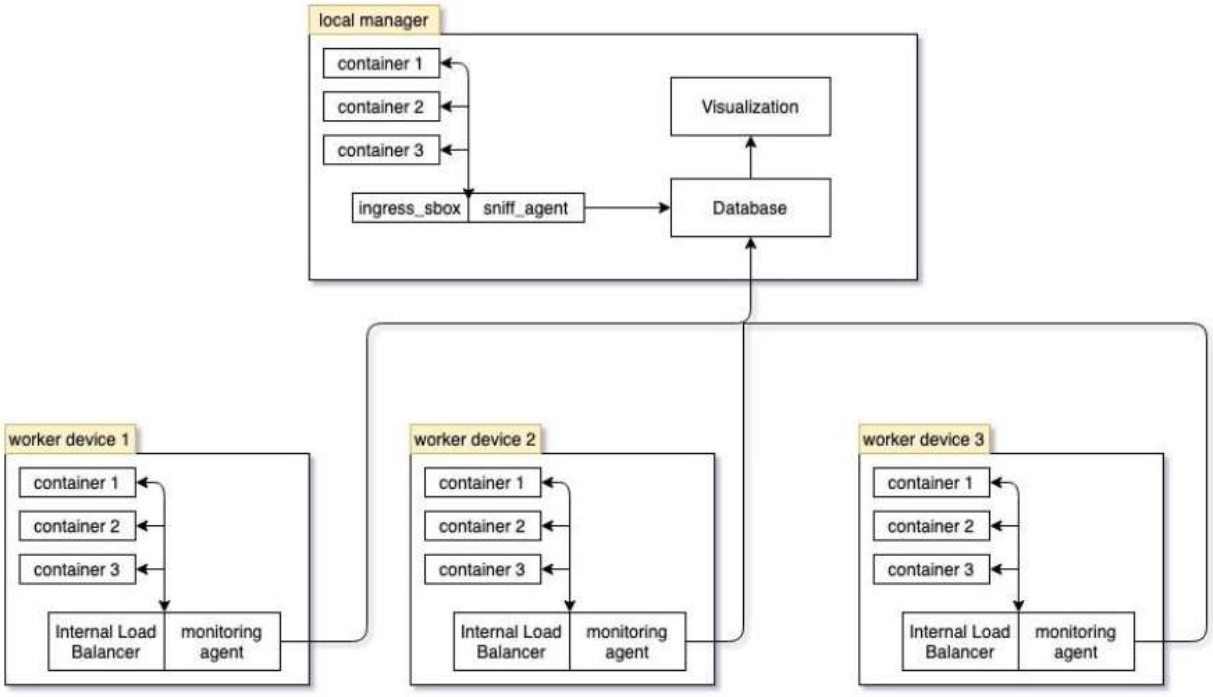

Figure 6: System Architecture

Figure 7 shows the monitoring system architecture. Each worker device hosts a monitoring agent. The monitoring agent is used to collect, filter, and aggregate measurement data that is processed by the monitoring agent to be sent to the manager server in the mist node. On the management server, the data stored can be used for management applications eg, visualization, making resource management decisions, and identifying performance bottlenecks. Each worker device has an internal load balancer. The purpose of an internal load balancer is to distribute client

requests to different containers. Internal load balancing is provided by a container management tool.

Monitoring Agent

Each worker device hosts a monitoring agent that is used to collect, filter, and aggregate measurement data. This monitoring agent will use existing packet sniffing tools to monitor data packets to and from the internal load balancer. The data processed by the monitoring agent will be sent to the manager server in the fog node. The processed data is stored in the database which is deployed on the manager nodes. On the management server, stored data can be used for visualization, making resource management decisions (e.g. container migration or replication), and identifying performance bottlenecks. The management application can query the database for monitoring information of the running nodes.

Internal Load Balancer

With container management tools single microservices usually have multiple replication instances packaged in containers. To assign requests to these replicas from clients in a balanced way, the container management tool provides a module to process all requests from the client, then forward the client requests to different replication containers according to the distribution policy. In Docker Swarm, this module is called an ingress sandbox and in Kubernetes, it is called an internal load balancer. Each server has its module. The module contains information on all replicas for the cluster, including replicas on other servers. The module responsible for these requests is known as an internal load balancer. To be more specific, whichever server in the cluster receives the request, the internal load balancer of the server that receives the request processes the request and forwards it to the replica. Replicas can be on the same server or other servers in the cluster.

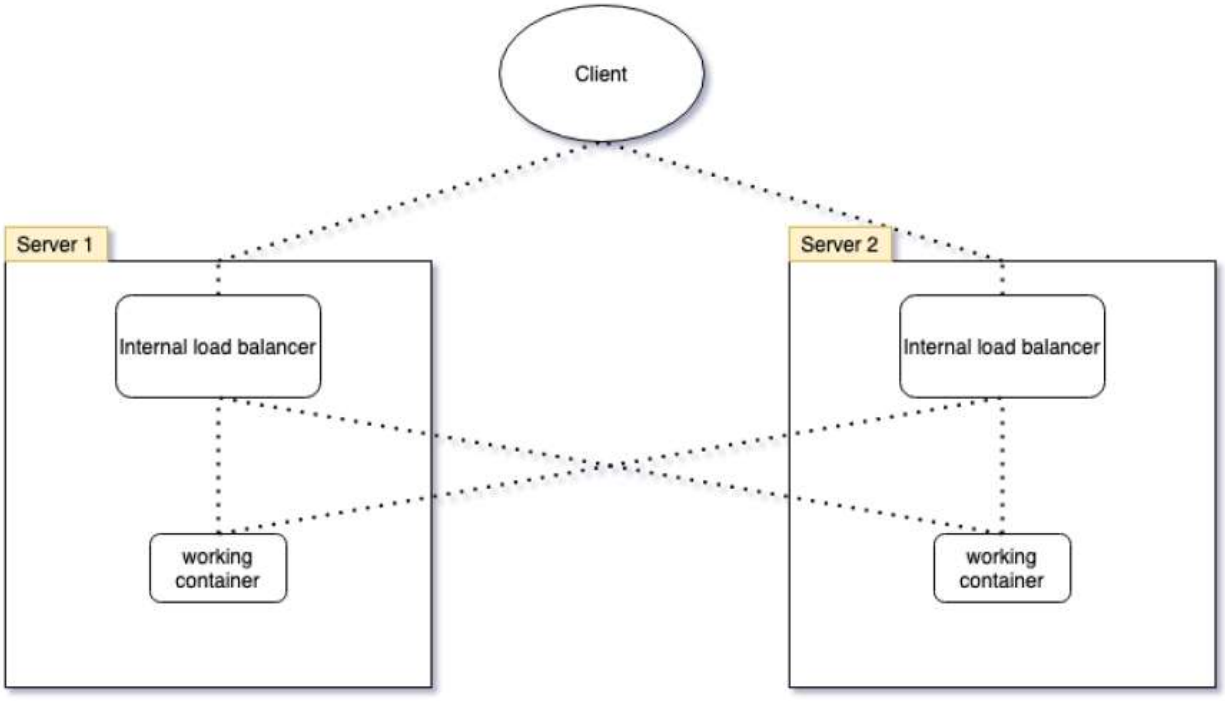

Figure 7 Internal Balancer in the container orchestration tool

Each container has its independent network namespace consisting of IP addresses and network interfaces. The packets received and sent by each container can pass through different network interfaces. If a monitoring agent monitors the network namespace of every container and every network interface, there is a need to create threads for each container to monitor network information in different network namespaces. This processing method increases thread and computation costs (resulting in additional CPU usage) due to an increase in the number of containers. This design is detrimental to system scalability. With an internal load balancer, there is a network namespace that is shared by all containers. Therefore, instead of using multiple threads to monitor all containers, the monitoring agent monitors the internal load balancer to collect network request information from all containers.

Microservices Tracking Monitoring

To support tracking microservices, it requires information on the sender of the request, the recipient, the time it was sent, the container that responded to the request, and the response time. Microservices trace monitoring is used to determine how long it takes the container to respond to a request after receiving the request. This information comes from multiple packets sniffing that is monitored by the packet sniffing tool in the monitoring agent. The information needed for tracking often comes from multiple sniffed network packets. Therefore, the monitoring framework must be able to collect and analyze several interrelated packages and integrate this information.

Request Flow in Internal Load Balancers

In this study, tracking consists of the data presented in Table 2. When a client sends a request to a server, the internal load balancer can generate four pieces of information as shown in figure 8.

Package Pair

After collecting the information described in the previous sub-sections, the monitoring agent installation module needs to find the relationship between the corresponding pieces of information and generate one complete network tracking information as shown in table 4. With four pieces of information as shown in figure 8 communication between fog nodes and clients is defined as external traces, which consist of requests sent by clients and responses from internal load balancers. The request consists of the requested URL and the request method (eg, GET, POST). An external trace consists of the following: a client request to the internal load balancer, the IP address of the client and the internal load balancer that received the client request, a timestamp recorded after receiving the request, and the internal load balancer's response to the client. The communication between the internal load balancer and the selected container in the fog node is referred to as an internal trace, which consists of the client request forwarded by the internal load balancer and the response returned by the container. The internal trace consists of the following: internal load balancer requests to the selected container, the IP address of the internal load balancer and the selected container, the timestamps of the internal balancer that sent and received the request/response, and the selected load balancer. response to clients. Table 4 summarizes the source and destination IP addresses for requests and responses. Lines 1 and 4 represent the requests and responses that make up the external trace. Lines 2 and 3 represent the request and response pairs that make up the internal trace. External and internal traces differ on the IP address but not on the requested TCP packet and URL.

Info #	Description	Source IP	Destination IP
1	Client request	Client	Internal load balancer
2	Internal load balancer request	Internal load balancer	Selected container
3	Selected container response	Selected container	Internal load balancer
4	Internal load balancer response	Internal load balancer	Client

Table 4 Source and Destination IP Addresses of Tracing Information

Line 2 represents the internal load balancer that forwards client requests to the selected container. It contains the same information as the line except for the IP address. In line 2, the source IP address is for the internal load balancer on the server receiving the request, and the destination IP address is for the selected container. Line 3 represents the response from the selected container to the internal load balancer and is the same as found in line 4 other than the IP address. This research uses the source IP, source port, destination IP, and destination port of the HTTP header in the application layer as identifiers to pair the corresponding request and response together. When a paired trace is an external trace, the source IP address is the client, and the load balancer is internal on the server receiving the request. If pair tracing is internal tracing, the source IP address is the internal load balancer on the server receiving the request, and the destination IP address is the selected container.

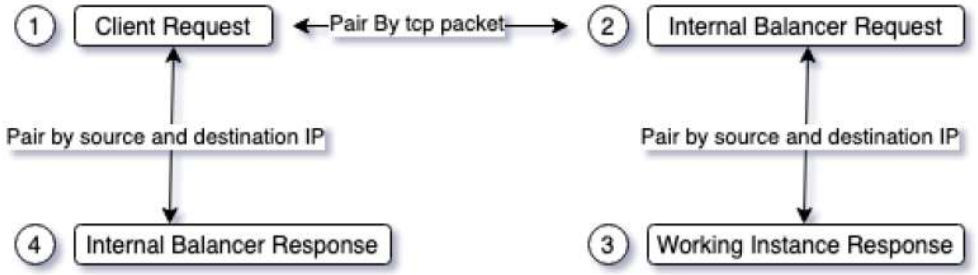

Figure 8 Trace the couple's relationship

Internal traces and external traces cannot be linked to the information in the HTTP header by matching the HTTP header information from the application layer. Requests and responses are matched by source and destination IP addresses. However, the sender and destination may have multiple requests and responses. HTTP headers cannot identify which request corresponds to which response. This research found that when the internal load balancer forwards client requests, the internal load balancer only changes the IP address in the HTTP Header but not the TCP packet information from the transmission layer. Therefore, the corresponding external traces and internal traces have the same TCP packet information/can combine internal traces and outer traces by comparing the information from the transport layer (such as ack, seq) to form a complete trace chain. The matching relationship between this information can be seen intuitively in Figure 9.

Track Monitoring Work Process

The monitoring agent on each server needs to get the network address used by all the containers in the cluster. This information is placed in a table with a mapping relationship between

the container identifier and the container IP address. This way, when the monitoring agent has all the information it needs to search, it can use the mapping table to determine which container the search is associated with. If the IP address sending the request is known in the mapping table, then this information is communication between containers on the mist node. The monitoring agent will mark this information as internal container communication within the cluster and label requests with container identifiers for the following processes as shown in figure 10. This study uses a dictionary data structure to temporarily store requested information. Request information includes the request sending IP address, requested URL, requested port, and timestamp. If the request is a known container within the container, the information will also include the request sending container identifier. The dictionary key is the IP address of the request sender. This dictionary data structure is called a pending dictionary. When the monitoring agent collects response information, it finds matching requests in the pending dictionary, takes them out of the dictionary, and stores them in the database for long-term data storage. The inner trace and outer trace do not require real-time processing. To reduce the consumption of computing resources from real-time monitoring, this check does not match inner and outer traces in real-time. System administrators can use TCP packet identifiers to match in and out traces.

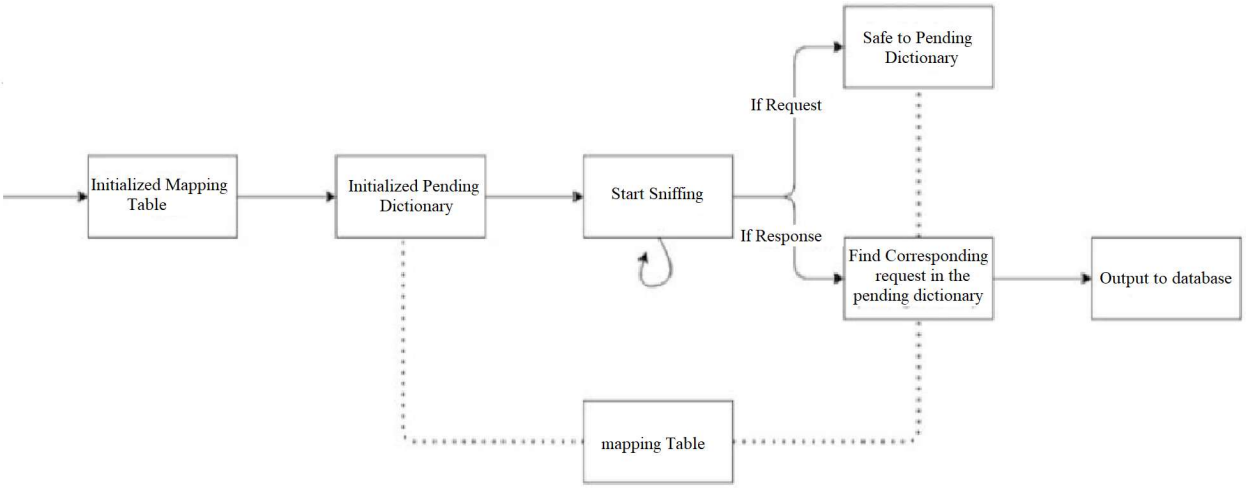

Figure 9 Monitoring Agent Workflow

Hardware Data Processing

To reduce the amount of monitoring data sent to the manager node, monitoring agents on the work node can filter. Data is filtered based on parameter settings that control the Scrape interval of hardware data. Swipe intervals range from 5 seconds to 30 seconds. When the scrape interval is too short, the monitoring agent will consume too many computing resources due to excessive computation, thereby affecting the quality of other services. When the scraping interval is too long, the data being monitored may miss some of the data generated during the interval. Therefore, the specific interval period must be determined by factors such as the purpose of monitoring and the computing power of the device.

Discussion of Design and Novelty

The design in this study focuses on monitoring network information flow using black box and service net approaches and this can represent two typical ideas in black box microservices monitoring. The service-mesh method is not implemented at the place where the data is generated but is prepared by a centralized, independent proxy gateway. The advantage is that the monitoring task is completely separate from the worker nodes in the cluster. In this way, worker nodes are not affected by monitoring tasks. Expanding microservices will not significantly increase monitoring

task overhead. On the other hand, all monitoring, analysis, and load balancing will be done by the server where the gateway is located. This processing method has very high requirements on the computing performance of the gateway server, and in the fog computing environment. This processing method concentrates all request information to a single point, which can easily lead to single-point failure problems. This potential hazard reduces the fault tolerance of the entire system. Besides this method can also minimize the impact on services and code (one collector per container), but also impact system scalability. Each additional container will bring some additional system costs. The described method using proxies can be considered to affect the forwarding of service requests and concentrate information to the same point (one collector per cluster).

System targets and limits

The system provides service-level data and indicators to system administrators or system orchestration algorithms. In the current fog computing container monitoring, all indirect monitoring can only obtain hardware information about the container, such as CPU usage, RAM, etc. When a service level fault occurs in a service, it cannot be detected by indirect monitoring. These error scenarios include response times that are too long, timeout levels that are too high, and too many error responses. This system does not replace white box monitoring. Although service level tracking is already designed for the system, request link monitoring similar to white box monitoring is not achievable.

RESULTS AND EXPERIMENTS

Docker Swarm and Ingress Sandbox

In Großmann & Klug (2017) the functionality and overhead performance of Docker Swarm and Kubernetes are compared. The results show that docker swarm consumes fewer computing resources, but also supports fewer container orchestration functions than Kubernetes. Considering that the environment in this study is fog computing, and servers are often large single microcomputers with relatively weak computing performance, this study chose Docker Swarm as the container orchestration job. With Docker Swarm, internal load balancing is implemented by the Docker Mesh routing mode. Within the design framework of this study, all servers on the mist node belong to the docker cluster. This distribution policy means that every service that joins a docker swarm cluster has a docker swarm load balancer, which is responsible for monitoring all ports of the service. When the internal node balancer catches a user request, it forwards the request to a container called the ingress sandbox via modified DNS forwarding rules. This container is the default container created by Docker Swarm. Inbound sandbox containers will distribute requests to containers located on different devices across the cluster to complete services according to defined routing distribution rules. The monitoring tool developed should obtain network traffic information from all containers on the server by monitoring the incoming sandbox containers. For this reason, when running the monitoring agent, the system network namespace must be changed to be consistent with its inbound sandbox so that the monitoring agent can access all incoming sandbox container network traffic.

Monitoring Agent Implementation

In developing the monitoring agent, Golang was chosen as the implementation language. The reason for choosing Golang is that it has superior features in system development and

distributed environments. Golang is also binary compileable making it easy to quickly deploy monitoring tools on different servers. The monitoring agent uses the libpcap library to sniff network packets that are passed through their inbound sandbox. Each network packet will be analyzed through the GoPacket Library. The monitoring agent has a dictionary indexed by sending IP and port. If a network traffic packet contains request information, the monitoring agent temporarily stores the request in a dictionary; if the network traffic packet contains response information, the monitoring agent will match the appropriate request from the dictionary and generate the complete trace information.

Backend Implementation

As a back-end research database II uses MySQL on a local management server to monitor data storage. This research also implements Grafana on the local management server as a tool for monitoring data visualization. System administrators can customize the dashboard of the data they want to monitor according to their needs.

EXPERIMENT

Experimental Environment - *Server Deployment*

Raspberry Pi (Bellavista et al, 2017) is considered a viable fog node component device. The Raspberry Pi is a single-chip microcomputer that provides relatively lower computing power at an inexpensive cost. These cost-effective features also make the Raspberry Pi a strong competitive advantage in future large-scale Internet of Things deployments. Experimental environment using four Raspberry Pis for the fog node. Table 4 presents the specifications of the Raspberry Pis.

Table 5 Raspberry Pi Specifications

Hostname	RAM	IP	Role	Deployment
4GB01	4GB	192.168.0.24	manager	request sniffer, MySQL, Grafana
2GB01	2GB	192.168.0.23	worker	request sniffer
1GB02	1GB	192.168.0.22	worker	request sniffer
1GB01	1GB	192.168.0.21	worker	request sniffer

The Raspberry Pi with the most RAM is appointed as the manager. The manager node hosts the request sniffer, MySQL database, and the Grafana visualization tool. Another Raspberry PI is set up as a worker and hosts the sniffer requests. Although this Raspberry Pi differs concerning RAM, it shares the same 64-bit Quad-Core Processor. The Raspbian operating system is installed for every Raspberry Pi. Raspbian is based on the Debian system and optimized for Raspberry Pi hardware. Raspberry Pis communicate with each other via Wi-Fi. A fixed static IP is configured for each Raspberry Pi to facilitate the identification and analysis of communication information between Raspberry Pi.

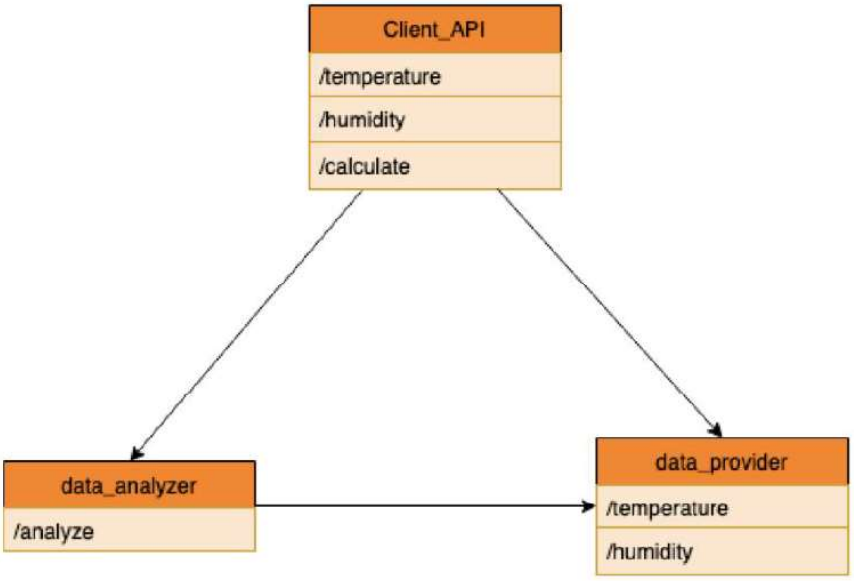

Figure 10 Microservices Dependencies

Application of Microservices

To verify the feasibility of the monitoring system, this research develops three related containerized microservices. The three microservices are analyzer, data_provider, and Client API. data_provider simulates collecting data from sensors. The analyzer is used to process the collected sensor data in real-time. It is simulated by using a loop that performs random computations. The Analyzer Container is designed to have high CPU usage. Client_API is primarily responsible for processing client requests centrally. This service uses the RESTFUL interface, which can obtain different services and data content by sending different HTTP requests. There is also a dependency relationship between these three services that call each other's APIs. The relationship is shown in Figure 11.

Table 6 Microservices API Paths

micro-service	Path	description
Client_API	/temperature	send a request to data provider
	/humidity	send a request to data provider
	/calculate	send a request to analyzer
sensor data provider	/temperature	response random data
	/humidity	response random data
data analyzer	/analyze	send a request to data_provider

Table 8: Application of Containers

Container_id	ingress_ip	bridge_ip	image_name	Host IP	Microservice name	Container Name
1075f4a5e4	10.0.0.13	172.18.0.5	sharlec/client_api: latest	192.168.0.22	/client_api	/client_api_1
16baa556d3	10.0.0.8	172.18.0.5	sharlec/data_provider: v2	192.168.0.24	/data_provider	/data_provider_4
1a5461e110	10.0.0.17	172.18.0.3	sharlec/analyzer: v2	192.168.0.21	/analyzer	/analyzer_3
25329ecfe6	10.0.0.22	172.19.0.3	sharlec/client_api: latest	192.168.0.23	/client_api	/client_api_2
35b27d7f72	10.0.0.7	172.18.0.6	sharlec/client_api: latest	192.168.0.24	/client_api	/client_api_3
3a3e18c8a8	10.0.0.20	172.18.0.4	sharlec/data_provider: v2	192.168.0.22	/data_provider	/data_provider_3
6f4a23acb5	10.0.0.9	172.18.0.3	sharlec/analyzer: v2	192.168.0.24	/analyzer	/analyzer_2
d113ea3720	10.0.0.24	172.18.0.3	sharlec/analyzer: v2	192.168.0.22	/analyzer	/analyzer_1
db19a6fc20	10.0.0.23	172.19.0.4	sharlec/data_provider: v2	192.168.0.23	/data_provider	/data_provider_5
e0470d44ba	10.0.0.16	172.18.0.4	sharlec/data_provider: v2	192.168.0.21	/data_provider	/data_provider_2
fac22af288	10.0.0.12	172.18.0.4	sharlec/data_provider: v2	192.168.0.24	/data_provider	/data_provider_1

Microservices are placed into container images and uploaded to the Docker hub. The microservices are then deployed on the four Raspberry Pi devices in the Raspberry Pi cluster via Docker Swarm on the manager server. For the Client_API microservices and data analyzer, there are three replicas. For microservices data_provider there are five replicas because these microservices are intended to receive sensor data. Containers use different network interfaces for communication, namely the bridge network and the access network. The inbound network is used to distribute requests to the selected containers. Whenever the internal load balancer receives a request, it can identify the container by its incoming IP address. The bridge IP network is used for containers to communicate with Docker Swarm. When the container is the sender of the request, the internal load balancer can identify the container by the connecting IP address to the selected container. The internal load balancer is configured to be able to query this required information from the docker swarm directly so that it can identify each request to each container. However, the system in this study does not have this information to match containers with IP addresses. Therefore, the monitoring system will initialize the mapping table to match containers with IP addresses as Table 6 shows.

Data visualization

In this research, Grafana is used for data visualization. Grafana is connected to MySQL Database. The data is used to monitor the dashboard in real time.

Figure 11: Real-Time Monitoring Dashboard

Monitor containers

Figure 12 shows the Grafana dashboard which contains information on CPU usage. System administrators can easily get real-time CPU information, configure alerts, and refer to this information for real-time container management. Figure 13 shows a line graph of memory usage.

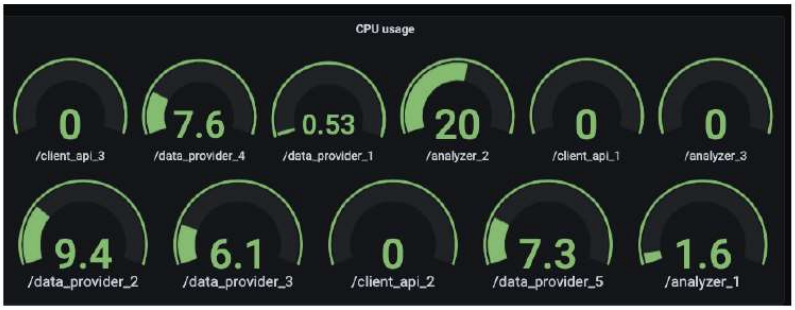

Figure 12 Real-Time CPU Monitoring

Figure 13 Real-Time RAM Monitoring

By combining numbers 13 and 14 it can be seen clearly that the container host of the analyzer and data_provider services consumes more computing resources than the container host of the client_api service. This is to be expected since the client_api service only provides request forwarding and aggregation and thus does not consume more computational resources than the other two services. In addition to displaying real-time hardware information, it is also possible to configure real-time container requests and average container processing time on the dashboard. The information shown in Figure 15 is the requests received by each container in the container breakdown. It can be seen that the data_provider service, as the data source of the entire microservices system, receives the most requests. Figure 16 shows the average response time for each container. Response time is measured by calculating the difference between requests and responses. The Client_API service is responsible for forwarding client requests as an intermediate gateway, so each Client_API container takes a relatively long time to respond to client requests. In the replicas of data_provider and analyzer, and found that data_provider 4 and analyzer 3 are the containers that receive more requests, and their response times are longer than other similar containers.

Figure 14 Number of Requests from each Container

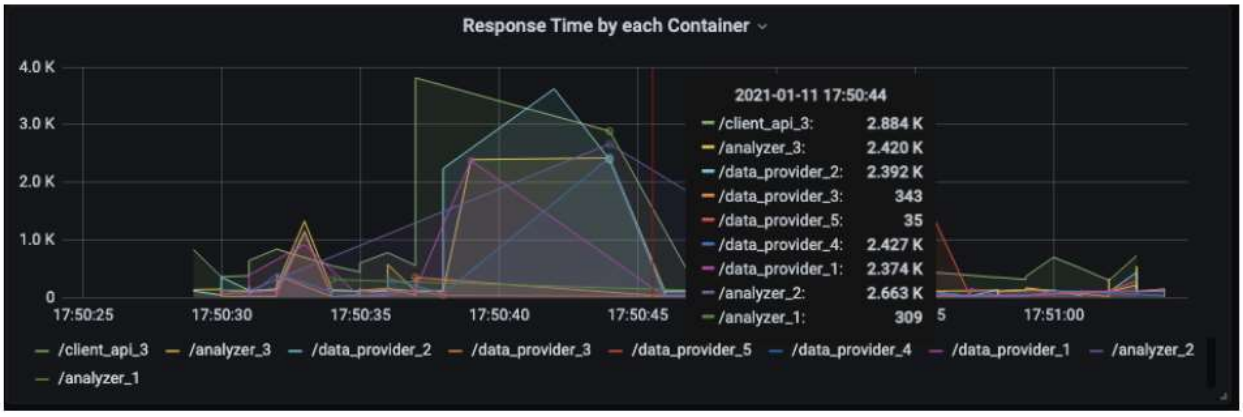

Figure 15 Response Time for each Container

Monitoring Communications

The information shown in Figure 17 is the number of requests between the two containers in this figure. To be more specific, the number of requests represents how many requests the container is communicating. The y-axis represents the receiving container, and the x-axis is used as the sending container. The IP address does not belong to a container that is recognized by the monitoring system.

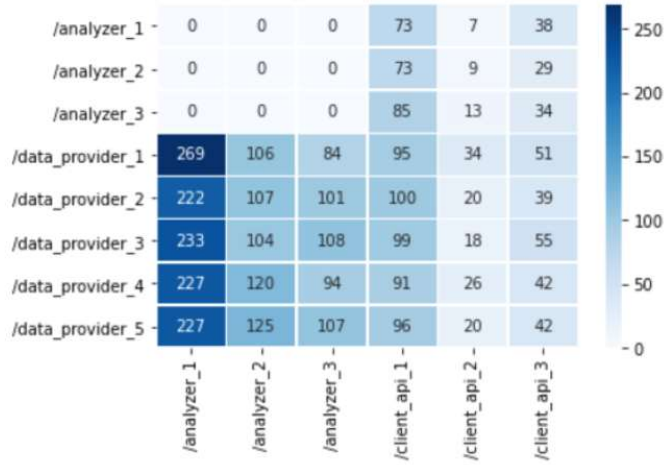

Figure 16 Communication Dependency Calculation Table

Figure 22 represents the communication latency between containers. Latency is calculated by the difference between the timestamp of the time the packet was sent and the time it was receiving the response.

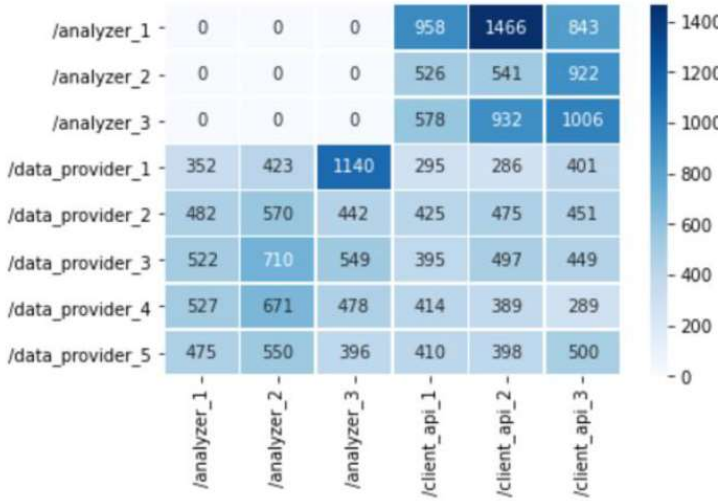

Figure 17 Table of Communication Response Time

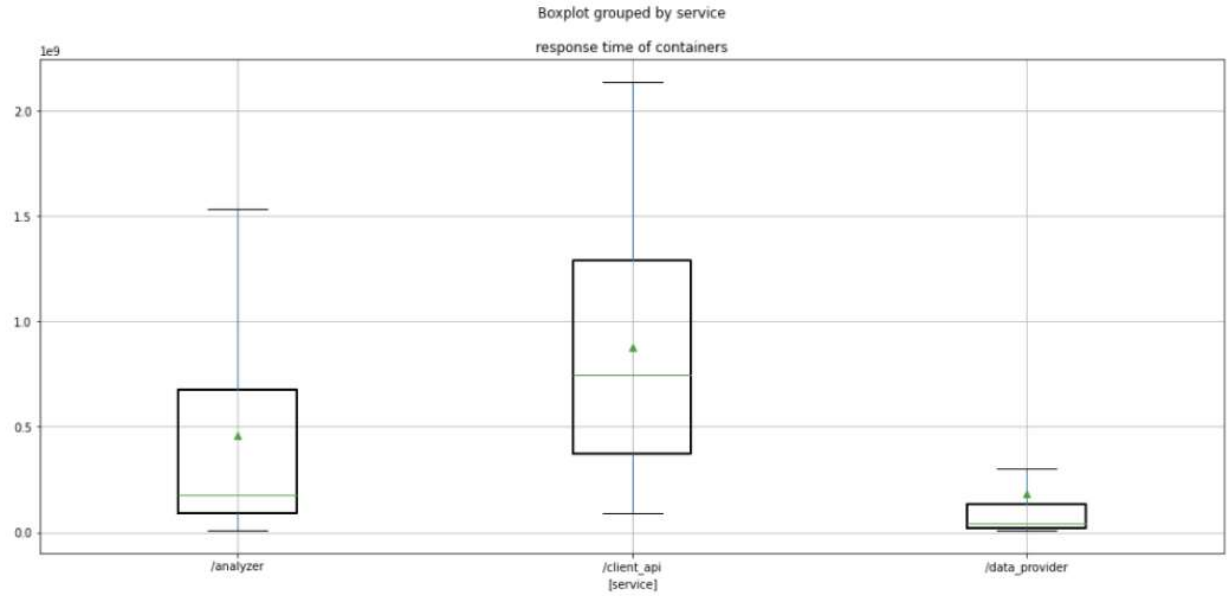

Figure 18 Service Response Time

Monitor Microservices

The chart and information shown above all use the container as a detailed analysis to evaluate the working condition of the container. The proposed framework also supports analysis with microservices as a breakdown. As shown in Figure 18, the average response time of the three microservices combined. As can be seen from the image, client_api takes longer to respond to requests than the other two services. data_provider has the fastest response time. The results of this data are consistent with the characteristics of the services used. The main functionality of Client_API is to receive requests from clients, then send requests to other services, and finally collect information from clients. Therefore requests related to Client_API often need to send a request to another service, wait for a response, and then reply to the user. Client_API depends heavily on other services. Network delays and congestion can easily impact Client_API response speed. Data_provider doesn't have a similar problem because it's designed to accept requests from two other microservices and provide data immediately. In this process, there is no need to send additional requests, or do many complex calculations. The analyzer microservices will first request a small amount of data from the data_provider and then perform a small amount of computation before responding to the client.

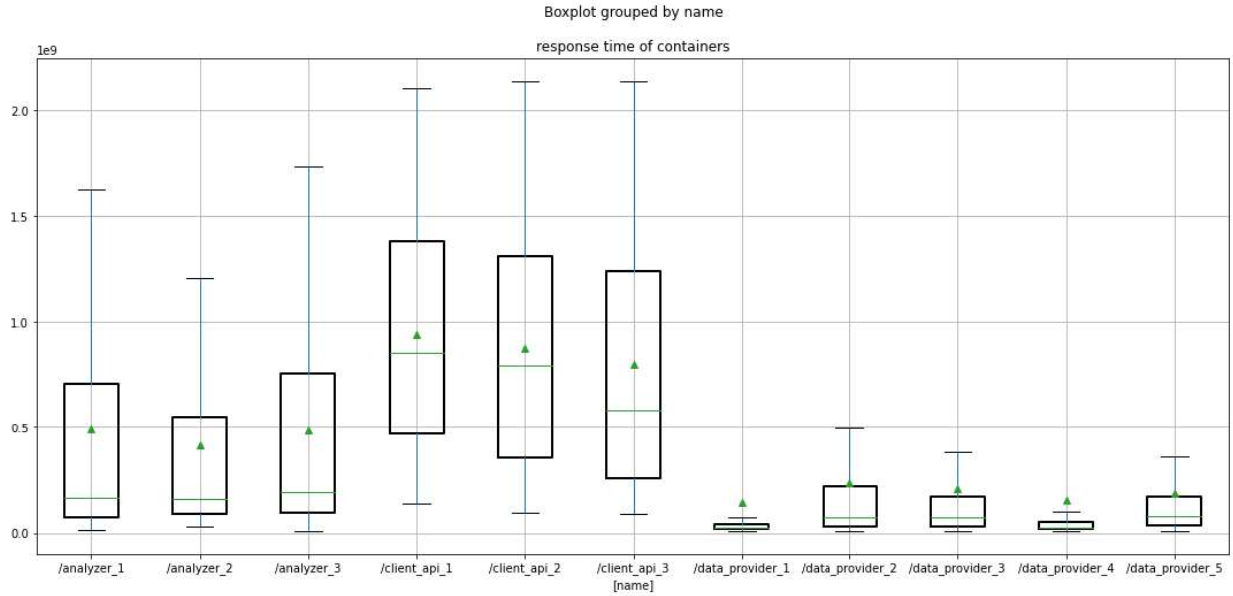

Figure 19 Container Response Time

Performance Overheads

To analyze the computational resource usage of the framework, the CPU usage of the system is collected when the monitoring agent is turned on, and the system's CPU usage is when the monitoring agent is turned off. While testing is in progress, no other applications on the Raspberry Pi server under test are using system resources except for the monitoring agent and docker microservices. Concurrency represents how many clients are continuously sending requests to the server at the same time. The overall CPU usage of the monitoring agent system is calculated by calculating the difference between the CPU usage when the monitoring agent is turned on and the CPU usage when the monitoring agent is not enabled. In each test, the CPU usage of the system is continuously collected for five minutes and then calculates the average value of CPU usage is over five minutes and fills it in. When the number of concurrent clients is 10, the monitoring agent consumes about 6.9% of the system's CPU time. As the number of concurrent clients increases, the CPU usage of the monitoring agent also increases. It is also worth considering that the sampling method is immature. Due to throughput uncertainty and service complexity, a large number of tracking tasks alone will bring a large number of computational requirements. Therefore, to limit the encroachment of computing resources with monitoring tools, in white-box microservices monitoring tools such as Dapper, their solution is to limit server performance loss by adjusting the sampling rate. When the amount of data is too large, the monitoring system will strictly control CPU usage, and only sample and monitor microservices within the range allowed by the CPU usage limit. Monitoring records are only a small part of all the information. System administrators can infer overall system performance by analyzing sample data. The proposed framework collects all request information and also completes the real-time analysis, matching, aggregation, storage, and visualization. This is undoubtedly a huge CPU consumption.

CONCLUSIONS AND RECOMMENDATIONS

This study discusses how to collect microservices information in a black box in the case of fog computing and implements the ideas that arise during research through the framework. The initial aim of this study is to obtain sufficient information through comprehensive information gathering to determine the operating status, service characteristics, and dependability of each container. The proposed solution is to monitor the container black box by monitoring the container management tool load balancer. Through experimentation, this study succeeded in showing that operational data for visualization that can help system administrators evaluate the status of containers that are currently running using the black box approach can be provided properly so that system administrators do not need to understand and modify target microservices to collect service characteristics from containerized microservices. The method used in this study is also suitable for network edges, which can run smoothly on microcontrollers with relatively weak computational performance.

For future research, this study provides suggestions in terms of continuous use of monitoring data and system development, whereby the information collected for multiple dynamic container deployments, such as dynamic horizontal container expansion and real-time container migration can be used.

1. Part of the framework container management tool code proposed here can be modified to provide a quantitative index of real-time algorithms to help optimize load-balancing algorithms.
2. Optimize Framework For Real Production Environment. Considering the database capacity issue, future research should consider using a time series database such as Influx DB12 for storage, and only storing data for a certain period (a week).
3. This framework should be further developed to support more service layer network protocols. The common communication protocols of the IoT environment represented by the MQTT protocol must be studied extensively.

REFERENCE

Aguilera, MK, Mogul, JC, Wiener, JL, Reynolds, P., & Muthitacharoen, A. (2003). Performance debugging for distributed systems of black boxes. ACM SIGOPS Operating Systems Review, 37(5), 74-89.

Alhamazani, K., Ranjan, R., Jayaraman, PP, Mitra, K., Liu, C., Rabhi, F., ... & Wang, L. (2015). Cross-layer multi-cloud real-time application QoS monitoring and benchmarking as-a-service framework. IEEE Transactions on Cloud Computing, 7(1), 48-61.

Arora, D., Feldmann, A., Schaffrath, G., & Schmid, S. (2011). On the benefits of virtualization: Strategies for flexible server allocation.

Bellavista, P., & Zanni, A. (2017, January). Feasibility of fog computing deployment based on docker containerization over raspberry pi. In Proceedings of the 18th international conference on distributed computing and networking (pp. 1-10).

Bonomi, F., Milito, R., Natarajan, P., & Zhu, J. (2014). Fog computing: A platform for the internet of things and analytics. In Big data and internet of things: A roadmap for smart environments (pp. 169-186). Springer, Cham.

Bonomi, F., Milito, R., Zhu, J., & Addepalli, S. (2012, August). Fog computing and its role in the internet of things. In Proceedings of the first edition of the MCC workshop on Mobile cloud computing (pp. 13-16).

Chandran, M., & Walvekar, J. (2014). Monitoring in a Virtualized Environment. GSTF Journal on Computing (JoC), 1(1).

Corkcroft, Adrian. Dockercon State of the Art in Microservices. dec. 4th, 2014, https://www.slideshare.net/adriancockcroft/dockercon-state-of-the-art-in-microservices

Cziva, R., & Pezaros, DP (2017). Container network functions: Bringing NFV to the network edge. IEEE Communications Magazine, 55(6), 24-31.

De Brito, MS, Hoque, S., Steinke, R., & Willner, A. (2016, September). Towards programmable fog nodes in smart factories. In 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W) (pp. 236-241). IEEE.

Doukas, C., & Maglogiannis, I. (2012, July). Bringing IoT and cloud computing towards pervasive healthcare. In 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (pp. 922-926). IEEE.

Firdhous, M., Ghazali, O., & Hassan, S. (2014). Fog computing: Will it be the future of cloud computing? The Third International Conference on Informatics & Applications (ICIA2014).

Großmann, M., & Klug, C. (2017, September). Monitoring container services at the network edge. In 2017 29th International Teletraffic Congress (ITC 29) (Vol. 1, pp. 130-133). IEEE.

Han, B., Gopalakrishnan, V., Ji, L., & Lee, S. (2015). Network function virtualization: Challenges and opportunities for innovations. IEEE Communications Magazine, 53(2), 90-97.

Huang, C., Lu, R., & Choo, KKR (2017). Vehicular fog computing: architecture, use cases, and security and forensic challenges. IEEE Communications Magazine, 55(11), 105-111.

Jalali, F., Smith, OJ, Lynar, T., & Suits, F. (2017). Cognitive IoT gateways: automatic task sharing and switching between cloud and edge/fog computing. In Proceedings of the SIGCOMM Posters and Demos (pp. 121-123).

Jimenez, I., Maltzahn, C., Moody, A., Mohror, K., Lofstead, J., Arpaci-Dusseau, R., & Arpaci-Dusseau, A. (2015, March). The role of container technology in reproducible computer systems research. In 2015 IEEE International Conference on Cloud Engineering (pp. 379-385). IEEE.

Kaur, MJ, & Maheshwari, P. (2016, March). Building smart cities applications using IoT and cloud-based architectures. In 2016 International Conference on Industrial Informatics and Computer Systems (CIICS) (pp. 1-5). IEEE.

Kraemer, FA, Braten, AE, Tamkittikhun, N., & Palma, D. (2017). Fog computing in healthcare–a review and discussion. IEEE Access, 5, 9206-9222.

Mayer, B., & Weinreich, R. (2017, April). A dashboard for microservice monitoring and management. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 66-69). IEEE.

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). Microservice architecture: aligning principles, practices, and culture. "O'Reilly Media, Inc.".

Sayfan, G. (2017). Kubernetes mastering. Packt Publishing Ltd.

Sharma, P., Chaufournier, L., Shenoy, P., & Tay, Y.C. (2016, November). Containers and virtual machines at scale: A comparative study. In Proceedings of the 17th International Middleware Conference (pp. 1-13).

Sigelman, BH, Barroso, LA, Burrows, M., Stephenson, P., Plakal, M., Beaver, D., ... & Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure.

Sun, L., Li, Y., & Memon, RA (2017). An open IoT framework based on microservices architecture. China Communications, 14(2), 154-162.

Uhlig, R., Neiger, G., Rodgers, D., Santoni, AL, Martins, FC, Anderson, AV, ... & Smith, L. (2005). Intel virtualization technology. Computers, 38(5), 48-56.

Yi, S., Hao, Z., Qin, Z., & Li, Q. (2015, November). Fog computing: Platforms and applications. In 2015 Third IEEE workshop on hot topics in web systems and technologies (HotWeb) (pp. 73-78). IEEE.

Yi, S., Li, C., & Li, Q. (2015, June). A survey of fog computing: concepts, applications, and issues. In Proceedings of the 2015 workshop on mobile big data (pp. 37-42).

Yigitoglu, E., Mohamed, M., Liu, L., & Ludwig, H. (2017, June). Foggy: A framework for continuous automated IoT application deployment in fog computing. In 2017 IEEE International Conference on AI & Mobile Services (AIMS) (pp. 38-45). IEEE.