7-28-2021 11:00 AM

A Black-box Approach for Containerized Microservice Monitoring in Fog Computing

Shi Chang, *The University of Western Ontario*

Supervisor: Lutffiya, Hanan, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

© Shi Chang 2021

Abstract

The goal of the Internet of Things (IoT) is to convert the physical world into a smart space in which physical objects, called things, are equipped with computing and communication capabilities. Those things can connect with anything, anyone at any time, any space via any network or service. The predominant Internet of Things (IoT) system model today is cloud centric. This model introduces latencies into the application execution, as data travels first upstream for processing and secondly the results, i.e., control commands, travel downstream to the devices. In contrast with the cloud-model, the cloud-fog-based model pushes computing capability to the edge of the network, which is closer to the data sources. This enables lower latency and a faster response time. The end-device can directly receive the service from the fog node instead of sending all the data to the central cloud server. In addition, with the application of microservice containerization technology, fog nodes can quickly set up various environments for heterogeneous services.

Compared with cloud computing, fog computing needs to consider users' mobility and geographic location. The application scenarios that fog computing are more dynamic and flexible. Therefore, fog computing requires real-time data monitoring and service management. In this thesis, we will explore how to deploy fog computing resources, what data is needed in the deployment process, and how to implement data monitoring.

Keywords

Fog/Edge computing, Docker, Container, Resource virtualization, Fog node container monitoring

Summary for Lay Audience

As more devices are connected to the Internet, there is a need to support real-time analysis and mobility. Cloud computing usually provides computing power support for these interconnected devices. In order to adapt to the new requirements in IoT devices latency and mobility, fog computing is an extension of the cloud to deploy computing resources to the edge of the network.

In the environment of fog computing, containerized microservice is a common service deployment approach. A container is considered to be a more lightweight implementation of computing resource virtualization compared to virtual machines. Container technology uses fewer computing resources than virtual machines, and can be deployed, expanded and migrated faster, which is more suitable for the dynamic computing environment of fog computing. The microservice architecture divides a software application into several microservices representing independent functions that communicate with each other though am API to act as a complete service. This flexible deployment method can deploy different microservices on several different fog service servers, making more efficient use of computing resources.

The cost of this distributed software architecture is the cost of deployment and maintenance. System administrators often have to face complex service dependencies. System administrators need to perform real-time analysis, deployment, expansion, and migration for diverse microservices in heterogeneous servers. Therefore, the container orchestration algorithm of fog computing provides a solution to this problem. The container orchestration algorithm will manage containerized microservices in real time through different algorithms and deployment strategies based on the data of monitoring containerized microservices.

Therefore, we analyzed the existing fog computing monitoring tools and the container orchestration algorithm for fog computing and developed a fog computing monitoring framework for the purpose of providing data for the fog computing container orchestration algorithm.

The framework we proposed can not only provide container-granular virtual hardware resource information, but also black-box monitoring of service layer information related to microservices. We tested the feasibility of the framework on Raspberry Pis and CPU overhead of this framework through experiments and showed what type of data and dashboards this framework can provide. The results show that this framework can be deployed on single-chip microcomputers with relatively insufficient computing performance.

Acknowledgments

Doing research is a lonely journey full of fun and challenges. On this journey, I cannot do without the help from many people.

First of all, I would like to thank my mentor Prof. Hanan Lutfiyya, thank you for your kindness and guidance, for leading me to scientific research, and for giving me full encouragement to explore and explore my scientific research interests. Without your help, this thesis will never be completed.

I also need to express my gratitude to all the faculty members, mentors and administration staff in the Department of Computer Science, University of Western Ontario. The four years of undergraduate studies here have laid a solid foundation for my computer knowledge, and the two years of graduate level study have given me sufficient conditions to explore unknown fields.

I want to express my gratitude to my family. Thank you for your guidance, rising me and support in pursuing the lifestyle that I yearn for, and unconditionally supporting all my decisions.

I would also like to express my gratitude to all the medical staff who have fought for COVID-19 and thank you for your great contribution in this unusual year.

Table of Contents

List of Tables

List of Figures

Chapter 1

1 Introduction

The Internet of Things (IoT) has seen rapid development in the last 10 years [1]. A IoT device consists of sensors and/or actuators, connectivity and compute power. These devices are used to monitor and control physical objects. In 2015, there were 15.41 billion connected devices, and this number is steadily rising every year. In 2025, the estimated number is expected to reach 75.44 billion [1].

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 [21]. In this design model, there is a clear division of tasks between the cloud and IoT devices which are placed at the edge of network. That is, the edge is responsible for data collection and the cloud is responsible for processing of data and management of IoT devices. Cloud systems can aggregate data from different devices for IoT services. For example, for smart cities, a single IoT device is not enough, because it may be necessary to collect information from multiple IoT devices for temperature, humidity and air quality of the entire city. The cloud can aggregate data from temperature sensors, humidity sensors, anemometers, sensitivity meters, and air quality detectors to analyze weather forecast information [39].

However, with the increase in the number IoT devices and data, and the physical distance between the network edge and cloud data centers, the volume of data to be transferred is expected to increase dramatically which result is in a large network load. Even the promise of 5G is insufficient for some applications since 5G applies to the cellular network. In addition to network congestion, cloud computing capabilities may also reach their limits during peak hours. Due to the overload of communication and computing resources, high latency will occur. For some services, high latency is unacceptable e.g., autonomous driving [2].

To address challenges posed by network latency, the concept of fog computing proposed by Cisco [40] has received widespread attention. Fog computing refers to the deployment of servers with low computing performance distributed near the edge of the network that is closer to data sources. These resources are used to provide data analysis, processing of aggregation data and other services. In this way, not all data needs to be sent to the cloud-based system, and much of the data processing can be done in fog nodes. The communication load on the network core is greatly reduced.

In application scenarios of cloud and fog computing and the Internet of Things [45], another notable technology is microservice containerization. The use of microservices has been proposed to be used in software development as opposed to monolithic software. The traditional monolithic software architecture means that all the functions of the software are encapsulated in one program. The microservice architecture can separate different functions in the software into different programs. This design pattern provides more flexibility [48]. These programs that provide part of the software's functions are called *microservices*. Microservices may be encapsulated in a container. An IoT application may consist of multiple microservices. Some of these microservices may be shared by different IoT applications. Microservices can be encapsulated in a container and may be placed on different computing nodes. For example, an IoT application can be used to locate a lost child. The application may require the following types of services: (i) A service that captures surveillance camera images; (ii) A service that provides a face detector classifier for detecting faces on an image; (iii) A service to determine if the face is that of the missing child. If smartphone cameras are to be used, a service that can communicate with smartphone cameras is needed. Services may be replicated in order to better distribute the load of camera data and analyzing it. Services may be shared e.g., a parking lot IoT application may use the service that captures surveillance camera images.

Microservices may need to be replicated due to high demand and this requires a suitable node to place the service on. There is also a need to be able to determine if the IoT application is able to meet the run-time requirements. This requires the monitoring of the interactions among microservices. Furthermore, determining a node to place a microservice should consider the potential for overloading in order to avoid it.

.

1.1 Problem Statement

In the use of fog computing to extend cloud computing, the orchestration of computing resources plays a vital role. Most of the resource orchestration work of fog computing assumes that the monitoring of resources and service interaction is available [20].

The data required by resource orchestration frameworks is diverse and can be categorized into two categories. One is the consumption of hardware resources represented by CPU, RAM, and network bandwidth, which we refer to as the monitoring of usage of computing resources. The other is service layer information, such as the number of requests received by microservices, the rate of error requests processed by microservices, and the dependency between microservices [34]. Currently, fog computing monitoring frameworks mostly provide the first type of data without considering the second type of data related to services. This thesis addresses this limitation.

1.2 Thesis Contribution

We propose a monitoring framework that provides for black box monitoring of containerized microservices in a fog computing environment. This monitoring framework integrates computing resource usage and run-time information of services interaction using a black-box approach.

The current container monitoring methods in fog computing are all indirect monitoring methods. That is, the performance of containerized services is evaluated through the consumption of virtual computing resources. However, in some service orchestration algorithms such as [13] and [43], there are references to container orchestration based on the information of the service layer. Therefore, the framework we propose attempts to integrate service-level information and computing resource information into the same framework.

1. Black-box monitoring of microservice in fog computing.
   In the microservice tracking tools represented by ZipKin[1], Dapper [33], Dynatrace[2], it is necessary to modify the code of the target microservice and insert the monitoring code. However, in the heterogeneous environment of fog computing, making modifications and updates to each microservice will generate a lot of labour work. Therefore, the framework we propose uses black box monitoring, that is, system administrators are not required to understand or even modify the microservices in the container.
2. Acceptable consumption of computing resources.
   Through experiments, we deploy our monitoring framework on a single-chip computer with relatively low computing performance. The results of the experiment show that even when a large number of concurrent requests are made, the computing resources used by this monitoring framework can still be borne by the single-chip microcomputer.

1.3 Thesis Structure

The thesis is organized as follows: Chapter 2 provides the background of container monitoring in fog computing. Chapter 3 presents a literature review on the issues of fog computing orchestration, fog computing monitoring and microservice monitoring. Chapter 4 describes a new monitoring framework. Chapter 5 describes the implementation. Chapter 6 presents the evaluation of the monitoring framework. Finally, we summarize and analyze our work in Chapter 7 and propose future research directions.

[1] https://zipkin.io/

[2] https://www.dynatrace.com/

Chapter 2

2 Background

2.1 Cloud Computing

The development of the Internet has brought more users and over time more complex needs. Applications have higher requirements for computing resources such as more powerful computing power, more storage capacity and faster transmission speed. Enterprises need to continuously manage these computing resources. These tasks include installation, deployment, upgrade, testing, management system, etc. Often this demand for computing resources is dynamic and will continue to change as the number of users changes. It is often difficult for enterprises to accurately assess the appropriate deployment of computing resources.

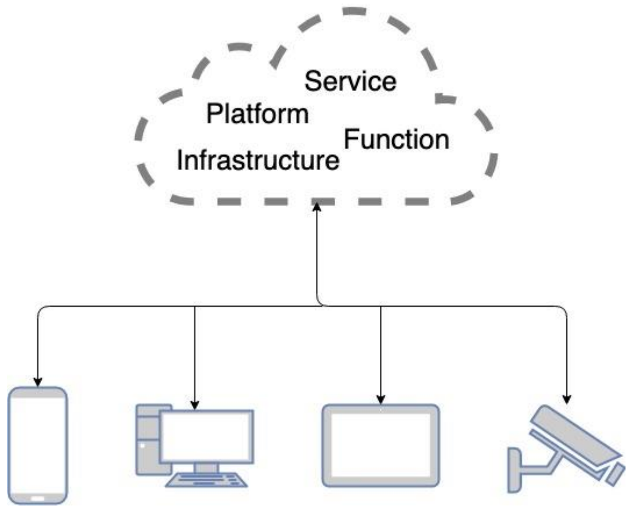


Figure 1: Cloud Computing

The emergence of cloud computing has greatly reduced the cost of computing resources for developers and enables clients to rent computing resources from cloud computing vendors that have data centers with powerful computing resources. Developers can deploy their applications and services in the data centers and pay according to resource usage. With cloud computing, developers can better control costs by using computing resources as needed. Cloud computing can provide dynamic and flexible computing and resources, scalability, stable backup, and simple deployment.

Cloud computing provides different models of service as presented in figure 2. The most common modes are Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS can provide the hardware equipment required by the application, including server, storage and network. Developers can rent these resources to deploy their own 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. Clients only need to focus on application development and maintenance. AWS Elastic Beanstalk and Google App Engine are the typical providers of PaaS.

SaaS is a way in which service providers directly provide services to end users. Users can access data stored in the cloud through any device at any location. Representative examples of such services are Google Drive Storage, Google apps.

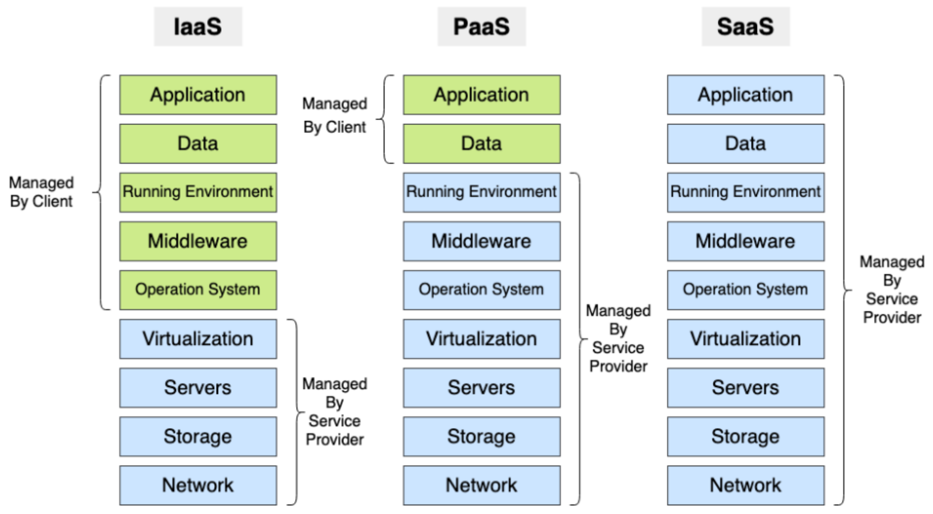


Figure 2: Cloud Service Type and Layers

2.2 Fog Computing

Internet of Things application scenarios typically include sensors that generate data which is often done periodically and often results in large amounts of data. The reliance on cloud computing means long-distance data transmission resulting in high latency for processing and the potential for high amounts of data being transmitted that can result in network congestion. This led to Cisco proposing the concept of fog computing [17], which refers to deployment of computing resources at the network edge which is closer to the data sources. Fog computing is an extension of cloud computing, making up for the shortcomings of cloud computing by placing computing power at the network edge. These computing nodes near the edge of the network are called *fog nodes*. The name fog computing comes from the fact that fog is closer to the ground than clouds.

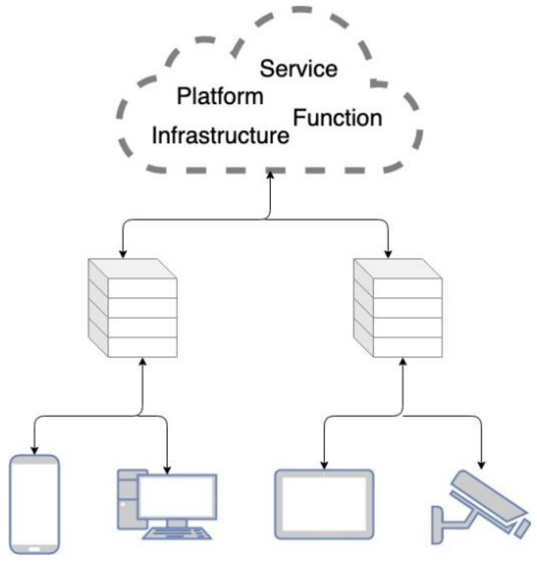


Figure 3: Fog Computing

The main feature of fog computing is that computing resources are distributed at the edge of network. The computing resources / capacity of a single fog node server is less than that of the cloud data centers [50].

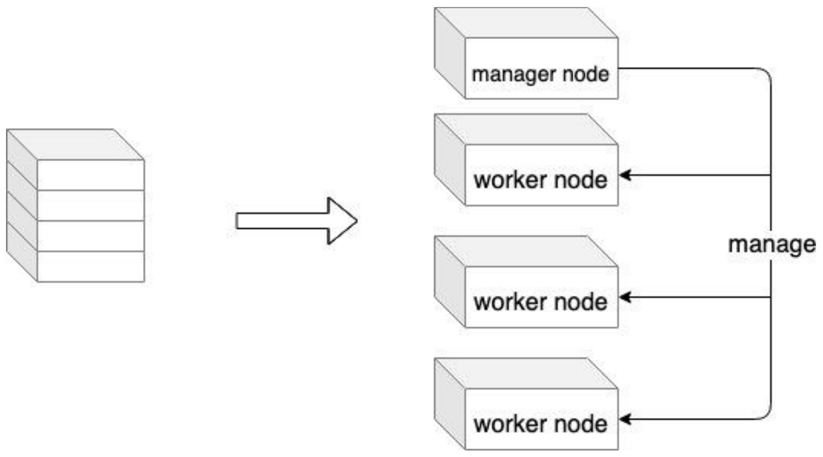


Figure 4: Fog Node

Compared with cloud computing, where all computing tasks are centralized and delivered in the cloud, fog computing is used to support latency sensitive application at the edge of the network which allows for client requests to be processed more quickly. This feature makes it possible to have latency-sensitive application scenarios such as autonomous driving [23], healthcare [24], smart factory [25]. If the computing performance of fog computing cannot support client requests, fog computing can be used as a gateway to aggregate and filter the original data, and thus reduce the communication load from the client to the cloud.

The implementation of fog computing is diverse [26], but the main feature is to combine multiple computing devices with low computing performance into a cluster which is often referred to as a *fog node*. The roles of computing devices in the fog node are either manager or worker. In general, there is only one manager for a fog node. The manager is responsible for the deployment and distribution of tasks, while the worker nodes focus on executing the assigned computing tasks. These tasks may be implemented using microservices.

Among the different fog node implementation methods, Raspberry Pi is considered to be a suitable and promising implementation method. Bellavista et al. [27] showed the feasibility of the Raspberry Pi as a fog node computing device and claimed that the fog node that uses Raspberry Pis has good scalability, flexibility, appropriate cost and easy deployment features. Other research [28] shows that the use of a Raspberry Pi for fog computing to implement a Raspberry Pi fog node for processing real-time data can be used in fog nodes.

Fog computing is essentially an extension of cloud computing, that is, deploying services closer to end users. However, this kind of extension is more than simply increasing the deployment volume and density of servers. Compared with cloud computing, fog computing is closer to users, and relatively more dynamic in response to usage scenarios, such as providing automatic services for autonomous driving [23], providing instant data processing and equipment management for smart factories [25], and providing computing for smart cities Infrastructure [15], etc. These different application scenarios means that fog computing has different characteristics, such as the geographic awareness of equipment, real-time migration of services according to user needs, the diversity of server hardware, and the limited computing performance of hardware equipment. These problems have not been well studied within cloud computing because of the relatively concentrated and unified computing resources.

2.3 Computing Resource Virtualization

In the development of cloud computing, cloud computing users may have various demand for computing resources. If cloud providers customize different hardware for each user, a lot of labor, time and resource costs will be incurred. Therefore, cloud providers address this problem through resource virtualization.

Computing virtualization technology uses virtualization management software (Hypervisor or Virtual Machine Manager) to decouple the hardware resources of the physical server from the upper-layer applications to form a unified computing resource pool, 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 [51], improved utilization, manageability, and reliability [52]. Currently, the two widely used implementations of virtualization are through virtual machine or container technology.

2.3.1 Container and Virtual Machine

Container technology [29] refers to the virtualization of applications, where each application has its own independent user space. The container includes the code, system tools, library and environment configuration required by the application being hosted in the container. The application of container technology allows developers to focus more on developing and deploying applications instead of repeatedly deploying development environments. The components needed to run a program is packaged into an image file.

The image file can be loaded into a container in order to be executed. This reusability and convenience greatly enhance the flexibility and scalability of services. Containers can also download image files from image file storage repositories (such as Docker Hub) for rapid deployment.

Virtual machines differ from containers in that a virtual machine can use an operating system that differs from the server's operating system. The virtual operating system will run like any another program on the server. A virtual machine (VM) can virtualize different operating systems on the host to adapt to different system requirements on the system environment. However, the unique feature of container technology is that all containerized applications can share the same container engine, and thus avoids the usage of system resources by requiring its own operating system. The creation of a container does not need to allocate fixed memory and disk storage like a virtual machine [30]. Therefore, compared to virtual machines, containers are more lightweight, and the utilization of host hardware computing resources is more flexible and dynamic [30]. In terms of data, the minimum amount of RAM resources used by the container can be small as 5MB while the smallest resource usage required by the virtual machine is 250MB [2].

2.3.2 Container Orchestration

Containers placed on the same host share the same operating system. To efficiently run multiple containers on a single host requires that no one container starves the other containers of CPU, memory, or networking I/O. Thus, as the number of containers increases, the complexity of managing resources 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[3] is the official native management tool for Docker container. By using it, users can pack multiple docker servers into a single large virtual docker cluster to quickly

[3] https://docs.docker.com/engine/swarm/

build a container platform. Kubernetes [31] is the container platform designed by Google. Kubernetes has more features such as alert and visualization. In the 2017 Docker Conference, Docker announced that they would provide native support to Kubernetes. Research [4] shows that Kubernetes uses more resources than Docker swarm. However, Kubernetes provide more features. Briefly, Docker Swarm is known for native lightweight deployment, and Kubernetes provides more and powerful functionality.

The tools required for management includes monitoring of containers and container host systems. Management tools are usually integrated with tools designed for container monitoring. For example, Docker Swarm integrates Docker's native monitoring tool *docker stats* and *Kubernetes* integrates *CAdvisor*.

We studied popular monitoring tools used by industry. These tools can be categorized into three categories. The first category represents the basic tools. These tools are known for being lightweight and fast. These tools can efficiently and quickly provide the basic information of the containers and host machines. One tool is the Docker Stats command from Docker. This tool provides CPU and memory usage, network and block I/O, and process identifiers (PIDS) for each container on the physical server. Kubernetes's CAdvisor is able to query this information and provide a web-accessed visualization dashboard. The visualization shows the information for the past 1 minute. The downside of these tools is that they cannot provide advanced features such as an alert system, monitoring multiple docker machines and long-term data storage. CAdvisor can be integrated with most of the management tools e.g., Kubernetes, Amazon ECS.

The second category is hosted tools. These tools provide monitoring services from third-party companies. The administrator pays for an account, and then deploys the configuration files onto the docker machines. The configuration files are used by Docker machines to link to the account on the cloud. The monitoring services (e.g., Scout[4], Datadog[5])typically

[4] https://scoutapm.com/

[5] https://www.datadoghq.com/

use basic monitoring (e.g., Docker Stats) Additional monitoring services may provide a flexible visualization dashboard and an alert system with customized thresholds.

The third category is the self-hosting tools. For some administrators, self-deploying, controlling and customizing the monitoring tool is a better option. These kinds of tools are open source. They usually provide different kinds of monitoring such as virtual machine, system process, user customized metrics, as well as the containers. Even if it requires some extra effort for deployment, the tools can also provide complete functionality by integrating stack of different tools. These tools usually have a well-developed docker image. One tool stack example is Prometheus[6] which consists of four containerized components: MySQL for data storage, Prometheus for data aggregation, Grafana for the visualization and Node Exporter for information query.

2.4 Microservice and Tracing

With the dynamic and flexible computing resource planning brought about by virtualization technology, the concept of *microservices* has also been proposed. In traditional software service development, all parts of application software are packaged in the same program and run on a server. This traditional application software architecture is a *monolithic application*. The concept of microservices is intended to adapt to a more dynamic and flexible computing resources environment [49]. The microservice architecture splits a monolithic application into different parts, and each part is called a microservice. The services provided by the monolithic application is through the collaboration of microservices. In this section, we introduce the characteristics of microservices, the advantages and disadvantages compared to monolithic applications, and the challenges that microservices currently face.

[6] https://prometheus.io/

2.4.1 Microservice

Microservices is a software architecture, which means that the software is composed of multiple independent services where each service is responsible for a single function.
The idea is not to develop a huge monolithic application, but to decompose the application into small, interconnected microservices. One microservice completes a specific function, such as passenger management and order management. Each microservice has its own business logic. Some microservices also provide API interfaces for other microservices and application clients.

Compared with monolithic applications, the microservice architecture has the advantages of low coupling and better maintainability [16]. In a monolithic application, a small change may affect the deployment of the entire application. The modification of a single module may require coordination of other modules. This type of maintenance requires programmers to have a sufficient understanding of the entire application architecture. In the microservice architecture, changes made by the programmer to a single microservice will not affect other microservices.

2.4.2 Microservice Tracing

The disadvantage of the current microservice architecture compared to monolithic applications is troubleshooting. When a single application fails, the system maintainer can troubleshoot the problem by reading the application log on a single server [22]. The microservice architecture is different. Each microservice may have its own log storage format and method, and each microservice may be deployed on a different server. This feature increases the cost of troubleshooting system failure points. In order to solve this problem, the industry proposed the concept of microservice tracing.

Compared with traditional monolithic service systems, in the microservice architecture, a user's request may need to access multiple microservices deployed on different servers. In a monolithic system, the system architecture is relatively fixed and stable; and all modules are deployed on the same server. If errors and abnormalities are found with real-time monitoring, the system administrator can quickly locate the abnormal server and deal with it quickly [22].

With microservices, different containerized components may have multiple replicas as working instances, and these replicas are deployed on different machines. This low-coupling system architecture layer has flexibility and scalability advantages in large cluster deployments. However, this distributed deployment brings challenges to monitoring and tracking. Each request may be passed between multiple stateless microservices through API interaction, and these microservices may be distributed on different servers. Therefore, in the industry, there are also many service layer monitoring tools developed for tracking the performance of microservices such as ZipKin [32], Dapper [33], Dynatrace [34], etc.

The approach of these tools is to assign a trace identifier to each request that is being tracked. A complete microservice trace chain record is generated by combining records with the same trace identifier together. The limitation of this method is that the system administrator needs to have a certain degree of understanding of the application design and this approach requires the modification of the code of the service. This method is referred to as white box monitoring, that is, the service function and monitoring function of the system are mixed together. From a development perspective, this increases the difficulty and complexity of development. Developers not only need to pay attention to the business algorithm, but also need to understand monitoring, communication and DevOps logic [49].

On the other hand, when the maintainers and developers of the system are from different parties (for example, when the service is hosted in an AWS cluster), this typically increases the difficulty of operation and maintenance.

We will discuss the pros and cons of white box monitoring in more detail in section 3.3, as well as the comparison with black box monitoring.

2.5 Containerized Microservice

Since containers and microservices are both dynamic and flexible, the combination has become popular. Cockcroft et al [53] believes that containerized microservices can multiply the dynamic and flexible characteristics, making microservices more elastic and flexible. In this section, we will briefly discuss the usage scenarios of containerized microservices and a monitoring method that is being developed for containerized microservices.

2.5.1 Usage and Application

In this section, we will introduce two representative application scenarios of containerized microservices: providing computing support for the Internet of Thing (IoT) applications and the realization of network function virtualization. Both of these application scenarios involve pushing computing functions to the edge of the network and deploying microservices in a fog computing environment closer to the data source.

2.5.1.1 Container and IoT

IoT applications are characterized by their integration with sensor data. Sensor data may be shared by multiple applications as well as some of the analysis of data. For example, data collected from wearable sensors that monitor patient vitals can be continuously sent to data aggregators and, in the event of detection of abnormal behavior, hospital personnel can be immediately notified in order to take appropriate measures. In this example, there could be multiple conditions that could be detected that use a particular sensor but not necessarily use all the same sensors. Furthermore, more conditions can be detected. It is possible to create a single stand-alone application for this but would be difficult to maintain with the implementation of new features that may require different analysis. This is addressed by using microservices that allows applications to be composed from multiple microservices and microservices can be shared by multiple applications. Microservices can be replicated or have additional resources allocated to it as needed e.g., a newly deployed application may use an existing microservice; mobile movement of sensors may require services to be deployed on new fog nodes. This implies high dynamicity. Containers are often used to host a microservice. Under this trend, the deployment of fog nodes at the edge of the network as gateways for IoT devices can effectively deal with the latency problem. However, the resource-constrained nature and diversity of these gateways pose a challenge to the development of widely deployable applications. Cziva et al. [55] focus on this issue and proved through experiments that deploying gateways through containerized services can improve the computing performance of IoT gateways. On the other hand, because of the lower consumption of computing resources and faster deployment speed of containers, the deployment of containers [56] is more flexible and faster than virtual machines and can adapt to dynamic user needs faster to achieve real-time expansion. Scaling and migration.

2.5.1.2 Container and Network Function Virtualization

As an increasing number of network middleware devices are deployed in the network. Problems such as high development cost, fast update, and difficulty in upgrading and deployment based on dedicated hardware have become increasingly prominent. These middleware or proprietary services often require specific hardware to work together. 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 traditionally run-on proprietary hardware.

At present, industry and academia tend to use virtual machine technology to implement NFV platforms [53]. With the rise of container technology, containers are considered to be the technology to implement NFV in the future. For example, Cziva et al. [54] have conducted in-depth research using containerized NFV. Cziva et al [54] believe that with the increase number of users and new mobile devices, telecommunications service providers (TSP) often encounter the problems of low resource utilization, tight coupling with specific hardware and lack of flexible control interfaces and cannot support multiple mobile applications and service. Therefore, the authors proposed a framework for implementing NFV by container instead of Virtual Machine (VM) at the edge of the network. Since containers take less hardware computing resources and are more flexible, TSP could reduce unnecessary core network usage, better troubleshoot faults, and provide users with location-aware and transparent services.

2.5.2 Service Mesh

When containers are chosen as the running environment of microservices, the service mesh[7] is considered to be a way of complete microservice tracking in the future. The service mesh is an infrastructure component for the processing of service communication.

The service grid monitoring solution is to deploy a corresponding traffic proxy called "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 is the data layer (data plate). This layer deploys sidecars for each container. The sidecar could completely proxy request and response related to the container. These tasks include processing data packets, forwarding, routing, load balancing, monitoring, etc. [35] These sidecars can communicate with each other, and these communication records can be used to track microservice requests. The other layer is the control layer (control plate). This layer does not directly parse the data packets, but communicates with the sidecar of the data layer, collects the information of the data layer and assign distribution/routing policy. Also, the control layer could provide APIs to system administrators to facilitate configuration, monitoring, visualization, continuous integration and deployment.

This architecture splits out service communication, allowing developers to focus more on service code logic. The related management and control functions of the communication and network layers are lowered to the infrastructure layer. In this design, service code and communication are fully decoupling.

Service mesh also has limitations. The most frequently mentioned limitation of the service mesh is the increase in system complexity and latency. The integration of sidecar's additional agents into the entire distributed microservice system will make the entire

---

[7] https://linkerd.io/

system even more troublesome and increase the difficulty of operation and maintenance. Since all information is proxied by the sidecar instead of directly communicating with the container, this will cause a slight delay. In some business scenarios, this kind of delay cannot be tolerated.

Moreover, the current research on service mesh is all in the cloud environment without considering the characteristics of fog computing. When the computing performance is limited and the configuration process needs to be simplified, whether the service mesh is still applicable has not been well studied [36].

Chapter 3

3 Related Work

This chapter reviews the current work on containerized microservice monitoring and container management using the evaluation criteria described below which focusses on supporting IoT applications.

1. **Support for fog computing.**
   Fog computing is expected to host applications composed of services that are dynamic and that to have more dynamic application scenarios than cloud computing. In the fog computing environment, we need to consider the diversity of fog server devices, the mobility of the services, the geographical location of clients, and the limited computing resources of the server devices.

2. **Support for the collection of run-time information of microservices and containers**
   The monitoring of containers and microservices are usually carried out separately with different design and tools. Container monitoring is a type of virtual resource monitoring. The focus of virtual resource monitoring is to provide the visibility of virtual machines' resource and performance. Therefore, the indicators that the container monitoring solution focuses on are Health (On/Off), Performance (CPU, RAM, Bandwidth, Storage), capacity, security and power [32]. The focus of microservice monitoring is to ensure the stable operation and optimization of service applications. Therefore, the indicators of concern are at the service layer, such as request tracking, specific service error rates and service interaction and dependencies [34]. These two monitoring methods have different design logics and purposes, so they are often not integrated into the same framework. In the fog computing environment, the combination of these two technologies can be of good use of various service scenarios. Therefore, when we conduct a literature review, we will also pay attention to whether there is a framework that can combine the different needs of the

two parties.

3. **Support for Black box monitoring.**
   Black box monitoring refers to the monitoring that doesn't require modifying the source code of the microservices. This is in contrast to white box monitoring, where the source code of the microservice is modified. With the diversity of services, requiring system administrators to have a clear understanding of each service will increase learning costs.
4. **Support for run-time information of microservices, containers and resource usage**.

We conducted a comprehensive literature review in the fields of fog computing monitoring, container orchestration and microservice tracing. In the field of container orchestration of fog computing, our research purpose is to explore what kind of data is needed in this field to support the container orchestration algorithm. We have selected [44], [43], [13] and [42] to represent the information we obtained in the literature study in this field. [44] is a review survey, where they analyzed that the orchestration of fog computing should be looped by Probe(Monitoring), Analyze, Plan and Execute, and the purpose of monitoring should be to provide necessary data for the orchestration and analysis. Then [43], [42] and [13] represent the different algorithms in three different scenario which may require different monitoring data. [43] represent general fog computing and IoT environment which may need QoS data such as request latency, packet loss rate etc. [13] proposed the collection of task specification files describing the characteristics of microservices to load appropriate tasks on the appropriate servers; [42] proposed fog computing applications in the NFV field and mentioned to use packet processing rates in the algorithm. In [43], Yousefpour et al also mentioned that most of the data required for container orchestration work assumptions have been well collected, but the availability of these data depends on the implementation of monitoring work.

Then, we put the research focus on the implementation of fog computing monitoring. [11] and [12] represent the general work for fog computing monitoring. Their research focus includes the topology of fog computing, the communication between nodes and the support for IoT devices. However, these works do not consider the usage of container. And our primary concern is the monitoring of container granularity at the edge of the network. The articles [4],[5],[7],[10] are the work that we found through search engines to monitor the granularity of containers in the fog computing environment. We will discuss in detail their research motivations, solutions, advantages, and disadvantages in section 3.2.

Finally, we also refer to the containerized microservices in the cloud computing environment. There are [22] and [37] respectively. When analyzing this type of article, our research focuses on whether the solutions proposed in these articles are feasible in a fog computing environment.

After analyzing the current research in these three categories in detail, we believe that none of the papers in these three categories address the problem of monitoring microservices in the fog computing environment. In the field of fog computing monitoring, the current monitoring implementation work focuses on the consumption of container computing resources, such as CPU and RAM. In terms of monitoring at the service level, current microservice tracking solutions are aimed at cloud computing environments, which does not consider the resource constrain in the fog computing environment. We define this as the research gap and discuss each article and this research gap in detail in section 3.4.

3.1 Fog Node Monitoring Requirements

This section describes monitoring solutions designed specifically for fog computing. The motivation and goal of this type of work is usually in the context of MAPE [44]. MAPE stands for monitoring, analysis, plan and executing. The set of processes is planned and managed by monitoring some quantitative metrices of the fog node or virtualized resources which is followed by an analysis of the collected metrics to support management tasks. These management tasks include migrating services, restarting servers, changing the connection path between servers, etc.

3.1.1 Orchestration and Monitoring in Fog Computing

Bonomi et al [44] analyzed the characteristics of fog computing from the perspective of application scenarios and composition structure. Cloud computing resources are managed in a centralized manner, and the composition of server resources is relatively more homogeneous. As an extension of the cloud computing layer, the fog computing layer is composed of heterogeneous server devices. This heterogeneous scope includes high-end servers, edge routers, single board computer, set-top boxes and end devices such as vehicles or mobile phones. In the fog computing environment, the network infrastructure may also be heterogeneous e.g., LTE, WiFi.

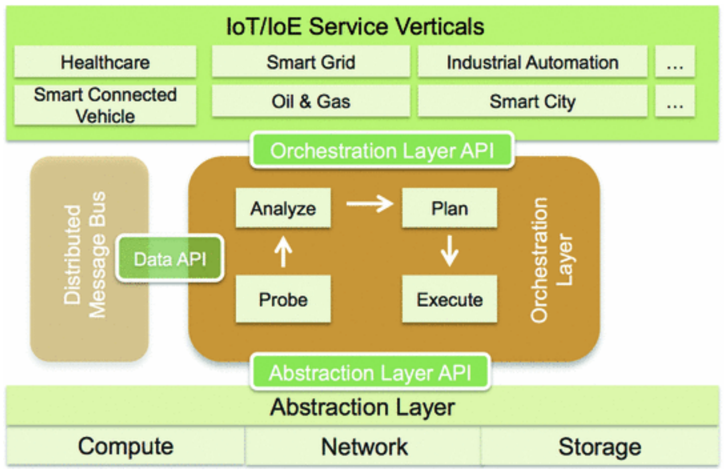


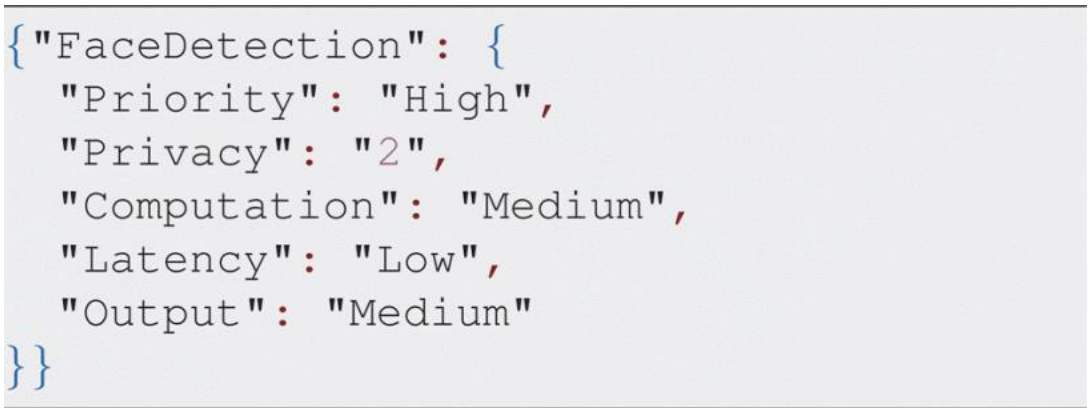
Figure 5: Fog Abstract Layer and Orchestration Loop [44]

In order to standardize the management of fog node devices, Bonomi et al [44] defined a fog abstraction layer as shown in figure 5. This fog abstraction layer hides the heterogeneity of the devices by defining device from the perspective of computing resources. Computing, storage, and networking resources may be virtualized. Monitoring data is to be used for service provisioning and orchestration.

To better understand the monitoring needs for fog computing, we studied the literature on orchestration. This section describes several papers that are representative of recent work. Although container monitoring solution is to provide monitoring data for container management/orchestration algorithm, under different needs, there are different requirements for the monitoring data. In order to show in more detail what kind of data is demanded for container orchestration, we describe some of the orchestration work that includes Foggy [13], FogPlan [43] and a containerized NFV orchestration framework [42].

FogPlan: Yousefpour et al [43] designed FogPlan which is a container orchestration framework that uses a greedy algorithm to optimize service latency. This algorithm focuses on QoS metrics such as service delay, hardware resource capacity and traffic rate. The design of FogPlan includes a monitoring module. There is no discussion on how monitoring is done due to the assumption that the fog node already has the required monitoring capability that is able to parse network traffic packets. FogPlan assumes that the monitoring solution extracts IP addresses and packet header information (such as HTTP header) to map the request to the service by IP address and port number.

Foggy: Yigitoglu et al. [13] proposed ***Foggy***, which is an orchestration framework for containerized microservices in a fog computing environment. User preferences and desired container behaviour are specified using a JSON file. An example is presented in Figure 6. The Computation field is used to indicate the computing power required, the Latency field is used to indicate the time sensitivity of the service and Output field is used to indicate the size of the output.



```
{"FaceDetection": {
  "Priority": "High",
  "Privacy": "2",
  "Computation": "Medium",
  "Latency": "Low",
  "Output": "Medium"
}}
```

Figure 6: Foggy Container Behavior Specification File Example

Foggy monitors the resource usage of containers and fog nodes through CAdvisor. Foggy uses self-matching algorithms to match each microservice with the most suitable fog node in order to support maximizing the quality of service. Placement can be adjusted as resource needs change.

Containerized NFV orchestration framework: Application scenarios considered by Zou et al. [42] is a fast-moving containerized NFV (Network Function Virtualization). They designed a Rate Limit Strategy that needs to monitor the packet throughput and rate of the container granularity.

We observe that for container-based management the physical hardware constraints requires knowledge of CPU, RAM and network bandwidth. However, as indicated by the orchestration papers described in this section there is work on managing containerized services through quality of service (QoS) metrics e.g., service delay. The work improves the utilization of computing resources through optimization approaches that considers quality of service indicators. This requires information that is not hardware usage but rather service based.

3.2 Fog Computing Monitoring Implementation of Framework

This section describes five monitoring frameworks designed for a fog computing environment framework. Four of these focus on container monitoring (e.g., [4], [5], [7], [10]) and one focusses on monitoring server hardware utilization [11].

PyMon: Großmann et al [4] developed *PyMon* which is a container monitoring framework for fog nodes. Großmann et al [4] observed that cloud monitoring solutions transplanted to a fog computing environment were not effective since these solutions do not consider that fog computing nodes are not as powerful as cloud servers. Großmann et al [4] introduced PyMon a lightweight monitoring solution for single-chip computers. In PyMon, the collection of monitored data is through Monit[8]. Monit is an open-source Unix/Linux system monitoring tool. Monit [6] can query the system information and send the monitoring data though HTTP. Monit is installed on each worker node. Monit can periodically collect the host information and create an XML file. However, Monit does

[8] https://mmonit.com/monit/

not natively support monitoring of container information, so the authors modified Monit with additional information that includes container CPU and RAM usage, image name and status. The host information and the additional container information is put in an XML file. The XML file is periodically sent to a manager server though HTTP. On the server (master node), PyMon provides support for data aggregation and filter processing. The pre-processed data is put into a Postgres SQL database on the manager server for long-term storage.

After completing PyMon, Großmann et al conducted an evaluation study of PyMon [5]. The purpose of the evaluation study was 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 the hardware usage overhead. Therefore, they chose two commonly used tools in the field of cloud computing monitoring and deployed them in a fog computing environment. The tools are Prometheus and CAdvisor. Prometheus, as a data aggregation server deployed on the manager node, corresponds to the role of MonitCollector in PyMon. CAdvisor in the Prometheus stack is responsible for collecting hardware information on the worker nodes. This is similar to the use of Monit in PyMon. They evaluated two fog computing monitoring solutions from the perspectives of CPU overhead and RAM usage. The results of the evaluation are shown in Table 1.

Table 1: PyMon vs Prometheus Comparison [5]

	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	/	/

As shown in Table 1, PyMon has better performance in RAM usage, while Prometheus and CAdvisor use less CPU. When the number of monitored containers increases, PyMon will also increase CPU usage correspondingly, and serious delays occur when the number

of containers reaches more than 32, forming a bottleneck in the system. The combination of CAdvisor and Prometheus does not have such a problem.
The conclusion was that the current open-source tools CAdvisor and Prometheus can be part of a monitoring framework for a fog computing environment.

Rule Based Monitoring Framework: Bali et al [7] evaluated PyMon with a focus on monitoring efficiency in a fog architecture. They proposed the use of rules to be executed by nodes to determine when data filtering, aggregation and other operations are to take place. For example, a rule on a worker node can specify not to send monitoring data to a manager unless the CPU usage exceeds a threshold value. This reduces communication load. The rules can also be used by master nodes.
This rule-based framework not only enables workers nodes to evaluate simple conditions, but also provides simple management capabilities for master nodes. The Master Analyzer is deployed on the Master node. The responsibility of the Master Analyzer is to manage, apply and update the rule set. In comparison to the deployment algorithm described in [13] and the machine learning model in [14], this approach of using simple rules could reduce the communication load between the client and the server by applying rules to filter monitoring data.

The use of rules can be used to reduce monitoring overhead, but it also means that large amounts of data cannot be saved for the long term. In some applications, detailed system logs may be required e.g., fog applications with real-time communication such as auto-driving [2] and auto-drone [9].

FMonE: FMonE [10] is designed for container monitoring on fog nodes. The framework is made of three components: worker node (FMonE Agent), pipeline, and backend. The pipeline is the communication link between different components in the system. The backend is the database for the data analysis and long-term storage. FMonE uses the concept of *region*, which refers to a group of worker nodes that are geographically near to each other.

On each worker node, there is a plugin referred to as the FMonE agent, that manages the data stream. FMonE agents consist of three components: inplugin, midplugin and outplugin. The inplugin is designed to collect the information from the containers and the node. The   metrics includes the CPU usage, RAM usage and network I/O usage. If these basic indicators are not enough to help users monitor fog computing clusters, FMonE also supports user to define customized filter/aggregation function on the collected data. The midplugin is responsible for aggregating and filtering the data. Finally, the outplugin takes the pre-processed data and sends it to the corresponding backend database though pipelines. The backend is only responsible for the storage of data and system logs without corresponding data aggregation and analysis capabilities. The work of data aggregation is done by a FMonE Agent, while the work of data analysis is handled by applications hosted on the central cloud center. In the design of FMonE, the backend is just an abstract data storage concept rather than a specific tool. Developers can use different database tools for different application scenarios. For example, in the case of a small amount of monitoring data, MySQL is sufficient if there are not too many monitored nodes. Developers focused on scalability may select Cassandra.

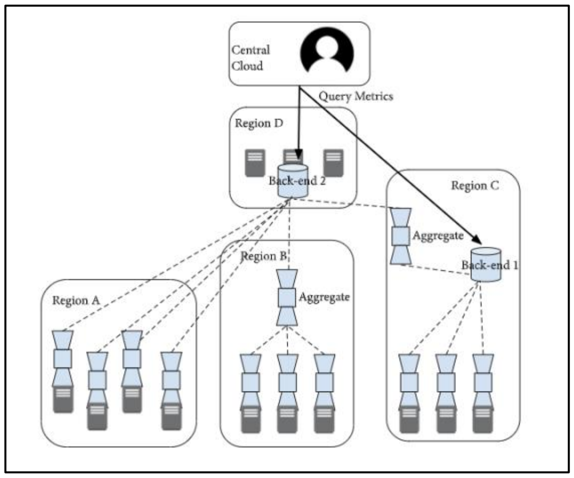


Figure 7 FMonE Architecture

The worker nodes in same region share the same backend, and the backends are connected to the worker nodes though the pipelines. The system administrator can easily modify the pipelines to change the hierarchical structure. Moreover, the pipelines can connect different backends together. As the figure 8 shows, the pipeline can also link up two backends for linkage operation.

Fog computing clusters are not as stable as cloud computing clusters. Any network node in fog computing may leave the cluster at any time, and an uncertain number of devices may join the cluster at any time. Therefore, the monitoring of fog computing needs to meet the two conditions of dynamic and flexibility to adapt to the unstable network environment. In order to meet the flexible and dynamic requirement, all the components are encapsulated into a container for easy setup and deployment.

FMonE has been compared to Prometheus and DataDog. These tools do not fully meet the nine requirements of fog computing monitoring described by the FMonE design team. PyMon is the closest work FMonE's, but FMonE has paid more attention to efficiency and provides support that provided by PyMon.

M3: A. Souza et al [12] proposed an integrated monitoring system referred to as M3. The goal is a monitoring system that provides enough information for container management. The work starts with CLAMBS [41], which is a monitoring framework designed for cloud environment, but it lacks the ability to monitor the edge device such as container information and bandwidth monitoring.
In M3, the monitoring is supported by four components: The SmartAgent handles registration, configuration and communication between the nodes. The SystemAgent collects resource information in the container, including process, system, container and hardware information. The NetworkAgent monitors the inflow and outflow of network traffic. The DeviceAgent is responsible for the communication between nodes and IoT devices, including various sensors. The DeviceAgent is designed to be adaptable and compatible with a large number of different devices and communication protocols.
The above four components are deployed on the worker node, and all the information is aggregated on the manager node. The Manager Agent deployed on the manager node aggregates data from the worker nodes assigned to it.

FogMon: Brogi et al [11] proposed a monitoring framework FogMon which is designed for the fog computing environment. Compared with the previous papers in this section, they do not include the container in the scope of monitoring. The monitoring content of FogMon is limited to the hardware information of the fog server. More specifically, the resources they monitor represent hardware indicators, QoS network indicators and IoT device indicators.

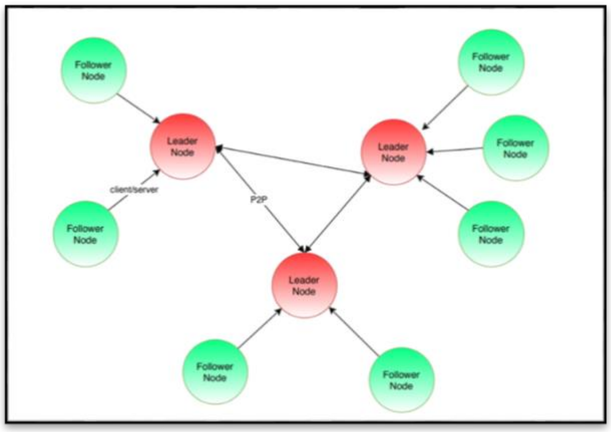


Figure 8: FogMon Topology

FogMon defines nodes as leader node and follower nodes. FogMon is designed to optimize Quality of Service (QoS). When a new follower node joins the cluster, it will also operate for the purpose of optimizing QoS. The follower node will first connect to an arbitrary leader node (assuming that the follower node already knows the leader node's IP address). The follower node will get all other leader node information from the leader node. The follower node will then send a message to all leader nodes to measure the delay, and then select the leader node with the shortest delay time as its leader.

FogMon's unique feature is P2P communication between leading nodes. Unlike the work in Section 3.1, FogMon does not make use of a cloud center. As shown in the figure, the highest level of the entire framework is a group of leader node. They designed a P2P communication algorithm with a complexity of O (log L) to ensure that all leader nodes can share monitoring data. The L in O (log L) represents the number of leader nodes. This approach reduces fog computing's dependence on cloud servers.

3.2.1 Comparison and Discussion

As noted earlier, Prometheus, has lower CPU consumption than PyMon. PyMon developers suggests that with the vigorous development of the open-source community, open-source software (e.g., Prometheus) can be used to monitor containers in fog nodes. Currently, there are no studies comparing the performance of other monitoring frameworks (e.g., FMonE, rule-based framework).

There are three lessons. First, the monitoring system cannot overuse too many system resources. The monitoring system should not affect the performance of applications due to monitoring.

Second, scalability and flexibility. In the fog/edge computing environment, devices consist of heterogeneous resources. Therefore, the monitoring system should minimize the difficulty of deployment and reduce parameter configuration.
Third, the future research direction should not be limited to the performance and architecture of the monitoring system but can explore more indicators that can be used by the orchestration system.

3.3 Monitoring Microservices

This section presents work related to monitoring microservices.

3.3.1 White box direct monitoring

Direct monitoring [22] refers to directly collecting performance metrics such as the number of requests, the amount of concurrency, and the average response waiting time. This is done by inserting code in the services that enables the assignment of a unique trace identifier for a request. This allows for request tracking. Examples of tools that use this approach include Dynatrace [34] and Dapper [33]. These tools require developers to have a certain degree of understanding of the application.

3.3.2 Black box direct monitoring

Black box monitoring does not require code to be inserted in an application for monitoring purposes. Currently, in the application scenario of fog computing, there is no direct monitoring solution that takes into account the characteristics of fog computing. Moreover, in the possible application scenarios of fog computing, we need to consider the diversity and dynamics of services [10]. According to the characteristics of diversity, it is concluded that the monitoring system of fog computing needs to meet the requirements of installation-free. Modifying the source code of each service to collect metrics is a very inefficient behavior. Black box monitoring meets this requirement. The monitoring framework of fog computing needs to assume that we do not know what services are running in the container, and the code cannot be modified. This section focusses on black box monitoring in fog computing usage scenarios.

Cinque et al [22] proposed a black box service log collection solution for microservices. The application layer is complex as reflected in the diversity and complex dependencies of microservices. When system administrators perform maintenance and troubleshooting, diverse application logs and heterogeneous platform architectures pose huge challenges. System administrators need to traverse the system logs and do analysis. Therefore, the researchers propose the use of a unified format log that is convenient for system administrators to view through a black box monitoring method. They developed the “Metro Funnel” tool to sniff HTTP request information on each service port. This solution sniffs the request packets through different service ports, perform filtering and aggregation operations, and then generates a log file for system administrators to view. It is assumed that the system administrator does not know the service running inside of the container. The log information is used to help experienced system administrators to quickly locate system bottlenecks, error information and run-time bugs through the generated logs. This work is not targeted to measuring containerized microservice performance or error detection. However, this black box monitoring method can be used to improve service performance by increasing the number of microservice replicas based on the analysis of the timeout rate of the requests.

Pina et al [37] uses blackbox monitoring to monitor the number of requests per microservices, response times and the dependencies between microservices. A centralized gateway collects and routes all microservices requests and response. Any request from the client, or communication information between the containers will be routed by this centralized gateway. In other words, they collect the requested information in an accessible, programmable and controllable central server. Through the collected information, the framework can clearly provide information, such as the number of requests per microservice, response time, and dependencies between microservices.
In order to implement this centralized proxy monitoring architecture, they modified the Netflix gateway ZUUL to route requests between microservices. The experimental environment is for Docker containers and the HTTP communication protocol. ZUUL collects, aggregate, and filters the communication information of microservices by reading HTTP header.

The metrics collected are similar to those found in [22]. They also collect access timestamps, sender IP, receiver IP, request method, and request path. This is used for the monitoring log. In comparison with the work presented by Cinque et al with [22], the difference is how they use the collected information. Cinque et al [22] generate a unified format log record from the collected data, passively viewed by the system administrator used to help the system administrator locate system abnormalities and bottlenecks. Pina et a [37] stores the collected data in the form of structured data to provide a data source for the visual dashboard of system monitoring.

3.4 Discussion and Problem Statement

Table 2: Summarizes the attributes of the different frameworks discussed in this chapter

Framework	Year	Support container	Support microservice	Support fog	Hardware	Blackbox
PyMon	2017	yes	no	yes	yes	yes
PyMon & Prometheus	2018	yes	no	yes	yes	yes
FMonE	2018	yes	no	yes	yes	yes
Rule-based	2019	yes	no	yes	yes	yes
FonMon	2019	no	no	yes	no	yes
M3	2018	no	no	yes	no	yes
	2019	no	no	yes	no	yes
Foggy	2017	yes	yes	yes	not applicable	no
Cognitive IoT Gateways	2017	not applicable	not applicable	not applicable	not applicable	no
Dapper	2010	no	yes	no	no	no
MetroFunnel	2019	yes	yes	no	no	yes
Service mesh	2016	yes	yes	no	no	yes
non-instructive monitoring	2018	yes	yes	no	yes	yes

3.4.1 Discussion on Fog Computing

In summary, 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 these fog computing container monitoring tools are related to the system, such as flexible data backends, acceptable performance overhead [4], geographic awareness, etc. [10]. This type of work considers the service dynamics in the fog computing environment with current research that focusses on indirect monitoring at the hardware level for CPU, RAM and Bandwidth. This kind of monitoring methods is primarily used in response to frequent migration and expansion of containerized microservices in the fog computing environment. If the system needs to monitor the service-related content of microservices, the heterogeneity and diversity of services in the fog computing environment will bring a huge and tedious workload for administrators.

3.4.2 Discussion on Microservice Monitoring

Compared with fog computing, the cloud computing environment has fixed capacity of servers in a stable data center environment [6]. The cloud servers are usually more powerful and can provide more computing resources. There has not been any work to determine if the black box monitoring method in cloud computing environment could be adapted in the fog computing environment.

Among the four methods mentioned in Section 3.2, Dapper requires the system administrator to modify part of the code in the container, so it is not a completely black box containerized microservice solution. The growth of service mesh to computing resources will increase with the growth of the number of containers, which is not conducive to the scalability of services. Some representative service meshes, such as AnyPoint Service Mesh [58], have minimum hardware requirements of at least 8GB of RAM. Netflix ZUUL in [37] also has requirements for hardware. This requirement is still too high for devices such as Raspberry Pi and Arduino, which represent computing power suitable for fog service devices.

Although MetroFunnel [22], which is very similar to the work described in Pina et al [37], does not describe hardware requirements, this framework is not for real-time monitoring. MetroFunnel is a tool that helps system administrators to output unified logs when they observe abnormalities in the system. MetroFunnel itself does not have the ability to monitor, store, and analyze monitoring data in real time.

In fog computing, it may have different system architecture than cloud computing, difference computing resource limitations and different application user scenarios.

3.4.3 Research Gap

Monitoring frameworks have been developed specifically to support fog computing. However, while there is support for monitoring the resource usage of containers and available resource capacity of fog nodes there isn't specific support for microservice-specific metrics e.g., service delay.

Existing monitoring frameworks designed specifically for the cloud require resources that may not be available in a fog computing environment. Furthermore, a fog node may host more than one instance of containers that encapsulate the same microservice. Each of these have different IP addresses. Some management applications need information for each instance in order to determine if there is sufficient capacity to support the demand.

According to the monitoring of fog computing containers and the black box monitoring of containerized microservices, we can conclude that in the monitoring of fog computing, the current focus of work is to consider the scalability of services that adapt to dynamic needs in the fog computing environment. Therefore, the monitoring of the fog computing environment needs to meet the requirements of real-time monitoring and low computing performance overhead. There is no service-level monitoring for microservices for the time being in fog computing monitoring field.

For containerized microservices, the current black box monitoring methods will incur considerable overhead. Hence, these black box monitoring solutions do not meet the low-overhead requirements of fog computing.

Therefore, we put forward a research question, can we design and implement a black box direct monitoring method for the characteristics of fog computing in the environment of fog computing? The black box monitoring method means that there is no need for system administrators to modify and understand microservices. It can not only greatly reduce the workload of system administrators, but also completely separate the work of microservice development and monitoring.

Combining the current research from the two fields of fog computing monitoring and indirect monitoring, we hope that our framework can combine the needs of the two fields to meet the following requirements:

1. Use reasonable computing resources.
2. Use black box monitoring so that there isn’t a need for system administrators to understand and modify the microservice code.
3. The granularity of the monitoring information is accurate to the container level.

In Chapter 4, we will introduce a solution for black box monitoring of containerized microservices for fog computing environments.

Chapter 4

4 Architecture of Monitoring Framework

This chapter describes the architecture of our proposed monitoring framework. The monitoring measures the run-time behavior of applications that consist of one or more microservices deployed in containers that may be placed in multiple fog nodes. The framework assumes a black box approach to monitoring. For each container, we monitor hardware resource usage and for each microservice we monitor the number of requests received, the average response time, and response code for each request. In a dynamic run-time environment, the states of containers change frequently. These measurements can be used by management applications to diagnose performance bottlenecks and determine when a microservice may need to be replicated and where it could successfully be replicated.

Section 4.1 describes the format of the target collected data and discussed why we need these data. In section 4.2, we will discuss where these data are generated and how to design the system architecture to collect these target data. In section 4.3, we will discuss how to process the collected raw data and generate structured data in the target format. Section 4.4 will discuss and compare our design framework with other similar work. Section 4.5 will discuss some of the limitations of our framework.

4.1 Measurements

4.1.1 Network Connection Information

Requests often span multiple services. Each service handles a request by performing one or more operations, e.g., database queries, publishes messages. Request tracing is a method used to profile and monitor applications built using microservices architecture. Request tracing helps pinpoint where failures occur and what causes poor performance.
Network connection information can be retrieved by analyzing the headers of network traffic packets from the application and transport layers. The traffic packets are parsed in

order extract the network connection information. The sniffer attempt to parse each packet in HTTP protocol. The relevant network connection information is presented in Table 1.

We have summarized this indicator information in the following table:

Table 3: Microservice Trace Metrics Table

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

4.1.2 Hardware Usage Measurements

This section focusses on measuring the resource usage of containers e.g., CPU, RAM. These measurements are used to determine if a container's resource usage is within a reasonable range and does not exceed the upper limit of the server. This can be used in making decisions on replication and migration of containers. Memory and CPU usage are considered to be two most critical pieces of information. For fog computing, researchers [19] have emphasized that sufficient network bandwidth is critical for minimizing service delay and therefore there is a need to collect the network traffic throughput of each container.

The collection methods of network traffic information and hardware are different. Network traffic information is passive in that network tracking information is generated every time a client sends a request. The monitoring of hardware is proactive. The monitoring agent periodically collects the hardware consumption of the container. The scape cycle is defined as the interval time between two captures of container resource usage information is referred to as the scape cycle. The scrape cycle typically ranges from 5 seconds to 1 minute [5]. Each scrape provides the following information:

Table 4: Hardware Metrics Table

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

4.1.3 Derived Information

The information shown in Table 3 and Table 4 is stored in a database. This information can be used to derive service-level information through different query methods to adapt to different needs. The derived information includes but is not limited to the following:

1. Calculate the error request rate by querying and filtering the request/response status code from the header information of each container.
2. Quantify the dependency and communication load between the containers by querying the number of requests between the container and the container.
3. Calculate the packet rate that each container needs to process in real time.
4. Evaluate the container working performance by packet average processing time.

4.2 System Architecture

The fog node architecture consists of multiple computing nodes. One of these computing nodes serves as a local manager and the rest are used to host microservices.

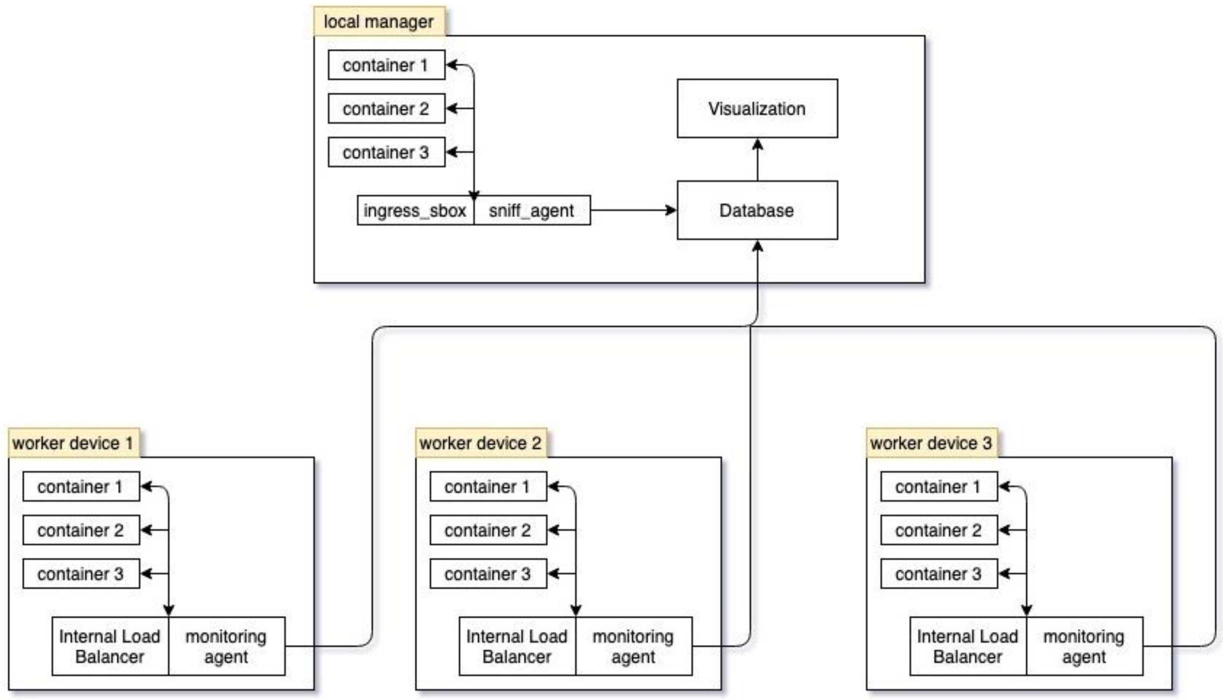


Figure 9: System Architecture

Figure 10 shows the architecture of the monitoring system. Each worker device hosts a monitoring agent. Monitoring agents are used to collect, filter and aggregate data measurements described in Section 4.1. The data processed by the monitoring agent will be sent to the manager server on the fog node. On the manager server, the   saved data can be used for management applications e.g., visualization, making resource management decisions (e.g., container migration or replication) and identifying performance bottlenecks.

Each of the worker devices has an internal load balancer. The purpose of the internal load balancer is to distribute client requests to different containers. The internal load balancer is provided by a container management tool. The rest of this section describes how the monitoring agent collects   the network traffic information of all containers from the internal load balancer in section 4.2.2.

4.2.1 Monitoring Agent

Each worker device hosts a monitoring agent that is used to collect, filter and aggregate the data measurements described in 4.1. This monitoring agent will use the existing packet sniffing tools to monitor the data packets to and from the internal load balancer. The data processed by the monitoring agent will be sent to the manager server on the fog node. The processed data is stored on the database deployed on the manager node. On the manager server, the saved data can be used for visualization, making resource management decisions (e.g., container migration or replication) and identifying performance bottlenecks. Management applications can query the database for the monitoring information of the working nodes. Section 4.2 describes how the monitoring agent processes raw data.

4.2.2 Internal Load Balancer

With container management tools, e.g., Docker Swarm, Kubernetes, one microservice usually has multiple containerized replicated instances. In order to assign requests to these replicas from clients in a balanced fashion, the container management tool provides a module to process all requests from clients, and then forwards client requests to different replicated containers according to the distribution policy. In Docker Swarm, this module is referred to as the ingress sandbox and in Kubernetes this is referred to as the internal load balancer. Each server has its own module. The module contains the information of all replicas for the cluster, including the replicas in the other servers. The module responsible for these requests is referred to as the internal load balancer. To be more specific, whichever a server within the cluster receives a request, the internal load balancer of the server which receives the request, processes the request and forwards it to a replica. The replica could be in the same server and or be in another server within the cluster.

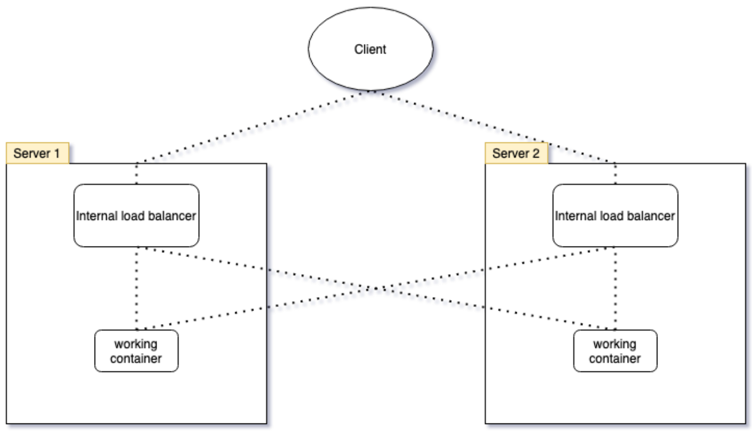


Figure 10: Internal Balancer in container orchestration tools

Each container has its own independent network namespace which consists of an IP address and network interfaces. The packets received and sent by each container may pass through different network interfaces. If the monitoring agent monitors the network name space of each container and each network interface, there is a need to create a thread for each container to monitor network information in different network name spaces. This processing method increases the thread and computational cost (resulting in additional CPU usage) due to the increase in the number of containers. This design is disadvantageous to the scalability of the system [5]. With the existence of an internal load balancer, there is a network namespace that all containers share. Therefore, instead of using multiple threads to monitor all containers, the monitoring agent monitors the internal load balancer to collect the network request information of all containers.

4.3 Microservice Tracing Monitoring

To support microservice tracing, the following information is needed: the sender of the request, the receiver, the sending time, the container that responded to the request, and the response time. Microservice tracing monitoring is used to determine how long it takes for the container to respond to the request after receiving the request. This information comes from multiple sniffed packets monitored by the packet sniffing tool in the monitoring agent. The information needed for a trace often comes from multiple sniffed network packets. Therefore, our monitoring framework needs to be able to aggregate and analyze multiple interrelated packets and integrate this information together. For example, the monitoring framework uses the timestamp of an HTTP request packet and the corresponding http response packet timestamp to calculate the duration between two packets. The time difference between these two packets timestamps is also referred to as packet processing time. This kind of metric is often used to measure the working status of the server, especially in the service of network function virtualization. Section 4.3 describes the approach used to extract the target trace information from the raw data of several packets.

4.3.1 Request Flow in the Internal Load Balancer

In this work, a trace consists of the following data as presented in Table 3: request sender, the container instance used to handle the request, the timestamp of the request, the processing time of the request, the response code and the requested URL path.

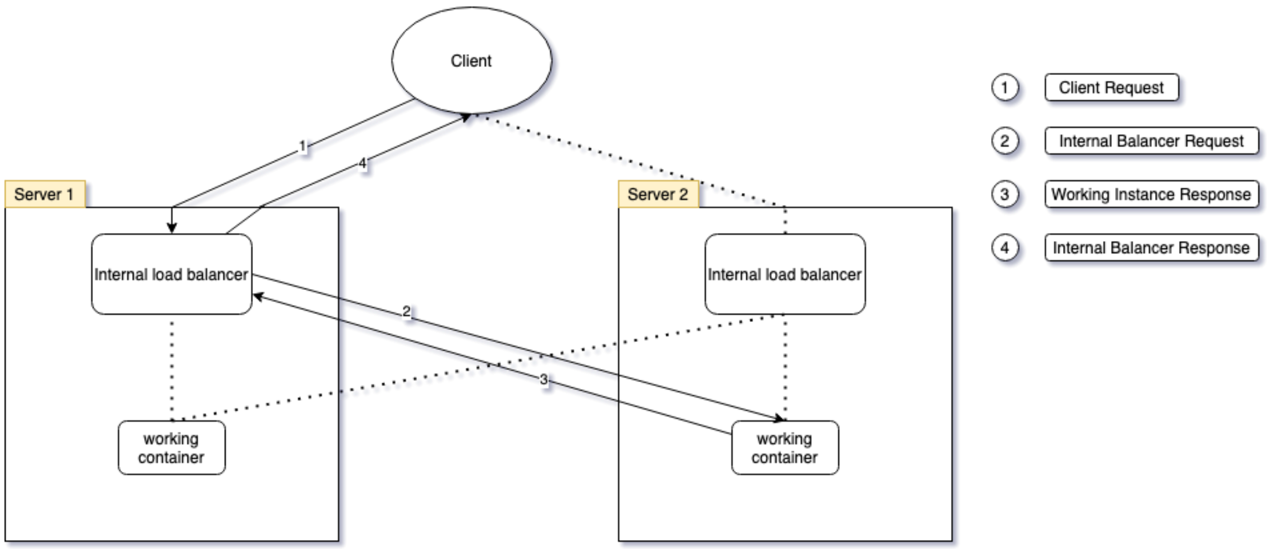


Figure 11:Request Flow

When the client sends a request to the server, the internal load balancer can generate four pieces of information as shown in figure 12. We can extract the data needed for the trace information as follows:

1. The network connection information of the request sent by the client to the internal load balancer, from which the time that the fog node received the request, method, requested URL path, and sender's IP address of the request is extracted.

2. The container that the request is forwarded to by the internal load balancer can be extracted from the internal load balancer.
3. The response of the container instance that hosts the internal load balancer. The response information provides the HTTP header information, which can be used to provide the response status code and the IP address of the selected container.
4. The response time can be calculated by the internal load balancer by calculating the duration between receiving the client's request and receiving the response from the container that was selected by the internal load balancer to handle the request.

The Monitoring Agent uses a packet sniffing method to obtain these four pieces of information.

4.3.2 Packet Pairing

After collecting the information described in the previous subsection, the pairing module of the monitoring agent needs to find the connection between the corresponding pieces of information and generate one complete network tracing information as shown in table 5. With four pieces of information as figure 12 shows (client request, internal balancer request, selected container response and response to the client), we define the communication between the fog node and the client as the *external trace,* which consists of the request sent by the client and the response from the internal load balancer. The request consists of the requested URL and the request method (e.g., GET, POST). The external trace consists of the following: the client's request to the internal load balancer, the IP addresses of the client and the internal load balancer which received the client's request, the timestamp recorded upon 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 the *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: the internal load balancer's request to the selected container, the IP addresses of the internal load balancer and the selected container, the timestamp of the internal balancer sending and receiving the request / response, and the selected load balancer's response to the client.

Table 5 summarizes the source and destination IP addresses for the requests and responses. Rows 1 and 4 represent the request and response which makes up the external trace. Rows 2 and 3 represent the pair of request and response which makes up the internal trace. The external and internal traces differ on the IP addresses but not on the TCP packet and the requested URL.

Table 5 Source and Destination IP addresses of the Trace Information

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

Row 2 represents the internal load balancer forwarding the client's request to the selected container. It contains the same information as row except for the IP addresses. In row 2, the source IP address is for the internal load balancer on the server which received the request, and the destination IP address is for the selected container.
 Row 3 represents is the response from the selected container to the internal load balancer and it is same as found in row 4 other than the IP addresses.

We use source IP, source port, destination IP, and destination port from the HTTP header in the application layer as identifiers to pair the corresponding request and response together. When the paired trace is the external trace, the source IP address is the client, and the internal load balancer on the server which received the request. If the paired trace is an internal trace, the source IP address is the internal load balancer on the server which received the request, and the destination IP address is the selected container.

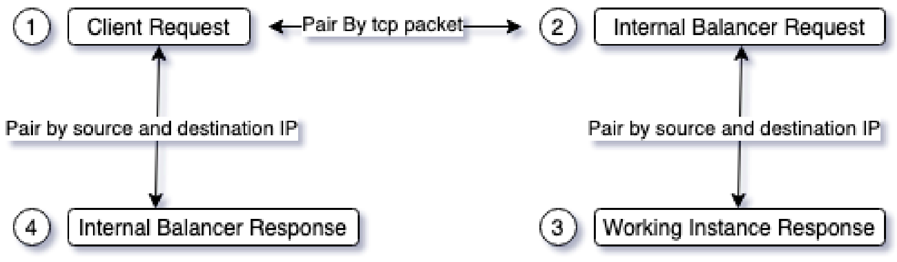

1
Client Request
Pair By tcp packet
2
Internal Balancer Request
Pair by source and destination IP
Pair by source and destination IP
4
Internal Balancer Response
3
Working Instance Response

Figure 12: Trace pairing relation

Internal trace and external trace cannot be connected by the information in the HTTP header by matching the HTTP header information of the application layer. The request and the response are matched by the IP address of the source and destination. However, the sender and destination may have multiple requests and responses. The HTTP header cannot identify which request corresponds to which response. We found that when the internal load balancer forwards the client's request, the internal load balancer only modifies the IP address on the HTTP Header but not the TCP packet information of transmission layer. Therefore, the corresponding external trace and internal trace have the same TCP packet information/We can combine the internal trace and outside trace by comparing the information of the transport layer (such as ack, seq) to form a complete trace chain.

The matching relationship between this information can be intuitively seen in Figure 13.

4.3.3 Trace Monitoring Working Process

In the previous two sections, we described how we theoretically capture all the information we need. However, this information consists of network traffic information, and we cannot yet combine the traced information with the container. Therefore, in this section we will discuss the workflow of how the monitoring agent collects and processes this information.

The monitoring agent on each server needs to obtain the network addresses used by all containers in the cluster. This information is placed in a table with a mapping relationship between the container identifier and the container IP address. In this way, when the monitoring agent obtains all the information needed for a trace, the monitoring agent can use the mapping table to determine which container the trace corresponds to. If the IP address of the request sender is known in the mapping table, it means that this information is a communication between the containers in the fog node. The monitoring agent will mark this information as internal container communication within the cluster and label the request by the identifier of the container for the following process as figure 14 shows. The request will temporarily be saved into the pending dictionary with container identifier. At the end, it will be saved into the database with the container identifier.

Second, we need to consider the real-time nature of the requested information. When we try to pair requests and responses, we need to consider that the same sender may send the same request repeatedly in a short period of time. Therefore, the request and response need to be paired in real time, otherwise we need to obtain information other than the source IP address and destination IP address to complete the pairing. We used a dictionary data structure to temporarily store the request information. The request information includes the request sender’s IP address, the requested URL, requested port and the timestamp. If the request is a known container within the container, the information would also include the request sender container’s identifier. The dictionary key is the IP address of the request sender. This dictionary data structure is referred to as the pending dictionary. When the monitoring agent collects the response information, it will find the matching request in the pending dictionary, take it out of the dictionary and store it into the database for long-term data storage.

Inside trace and outside trace do not need real-time processing. Therefore, the existence of the TCP packet identifier makes it unnecessary for us to complete the matching of inside trace and outside trace in a short time. In order to reduce the computing resource consumption of real-time monitoring, we choose not to match inside trace and outside trace in real time. The system administrator can use the TCP packet identifier to match the inside trace and the outside trace.

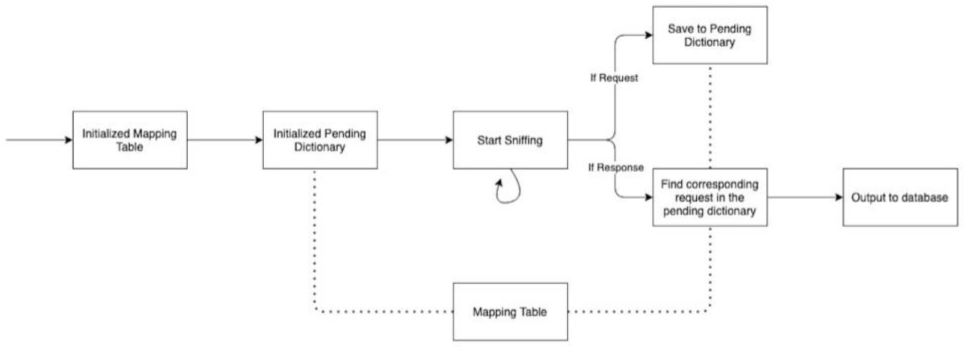

Initialized Mapping Table
Initialized Pending Dictionary
Start Sniffing
If Request
Save to Pending Dictionary
If Response
Find corresponding request in the pending dictionary
Output to database
Mapping Table

Figure 13: Monitoring Agent Workflow

4.3.4 Hardware Data Processing

Acquiring hardware resource usage data is relatively simple and straightforward. We can access the system cgroup, or directly access docker's stats API, or use third-party tools such as CAdvisor[9].

To reduce the amount of monitoring data sent to the manager node, monitoring agents on work nodes are able to filter. Data is filtered based on the parameter setting that controls the Scrape interval of hardware data. The scrape time is the time difference between two measurements of the hardware resource usage.

The scrape interval ranges from 5 seconds to 30 seconds [5]. When the scrape interval is too short, the monitoring agent will take up too much computing resources due to excessive calculations, thereby affecting the quality of other services [5]. When the scrape interval is too long, the monitored data may miss some data generated during the interval. Therefore, the specific interval period should be determined by factors such as the monitoring purpose and the computing power of the device.

[9] https://github.com/google/cadvisor

4.4 Design Discussion and Novelty

In this section, we compare related work [22], [37] to our system design. Our design focuses on monitoring network flow information using a black box approach [22],[37] and service mesh [35]. These three works can represent two typical ideas in black box microservice monitoring.

Service-mesh is a container-oriented collection method, that is, at the system level, communication requests belonging to a container are concentrated into a sidecar, and the sidecar is solely responsible for all requests corresponding to the container.

Clinque et al. [22] propose a service-oriented collection method (MetroFunnel), which does not need to be integrated with the system, and it will also change the request routing method of each container. The request are routed to a central gateway instead of access the container directly. Their approach is to monitor and sniff each packet passing through the target port according to the corresponding relationship between the service and the port.

Pina et al. [37] uses a network proxy collection method. This method is not deployed in the place where the data is generated but rather it is set up an independent, centralized proxy gateway. The advantage of this is that the monitoring task is completely separated from the worker nodes in the cluster. In this way, the worker node is not affected by monitoring tasks. The expansion of microservices will not significantly increase the overhead of monitoring tasks. On the other hand, all monitoring, analysis, and load balancing will be undertaken by the server where the gateway is located. This processing method has extremely high requirements on the computing performance of the gateway server, and in the fog computing environment, we cannot guarantee the hardware quality of the server. On the other hand, this processing method concentrates all request information into one point, which can easily lead to the problem of single-point failure. This potential hazard reduces the fault tolerance of the entire system.

As described above, Clinque et al. [22] and service mesh use similar approach where information collection components are deployed where the request is generated. This method can minimize the impact on the service and the code (one collector per container), but it will also impact the scalability of the system. Each additional container will bring some additional system cost. The method described by Pina et al. [37] using a proxy can be thought of as affecting the forwarding of service requests and concentrating information to the same point (one collector per cluster). This is analogous to the use of a data centralization approach in a local area. The advantage of this is that the service has good scalability and flexibility, but at the same time there are certain hardware requirements for the machine where the gateway is deployed.

We propose a compromise. Between the decentralized approaches and the centralized approached, there is another computing resource unit which is the server. We believe that each server has an internal load balancer that has a one-to-one correspondence with the device. The internal load balancer centralizes the communication information of all containers on the device. In theory, the expansion of the container will not cause additional unnecessary overhead to the monitoring system, and there is no need to concentrate all performance overhead on one machine, reducing the requirements for hardware.

4.5 System target and limitations

Our system provides service-level data and indicators for system administrators or system orchestration algorithms. For example, error response codes on the service level. 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 error occurs in the service, it cannot be detected by indirect monitoring. Such error scenarios include too long response time, too high timeout rate, and too many error responses.

This system does not replace white box monitoring. Although we have designed service-level tracking for the system, we cannot achieve request link monitoring similar to white box monitoring. We also have no way of knowing the nested relationship between records. For instance, if a request from a client triggers different requests for multiple microservices, we can only collect the information of each request separately, but we cannot know how they cooperate to complete the service to the client.

Chapter 5

5 Implementation

In this chapter, we will introduce how we choose a container management tool and implement the development of a container monitoring system for the internal load balancer of the selected container management tool. We choose Docker Swarm as the container management tool. In section 5.1, we discuss the reasons why we chose docker swarm and the working mechanism of internal load balancer in docker swarm. Then, we choose to use Golang to implement the monitoring agent. In section 5.2, We will discuss the reasons for choosing Golang, the library used and the details of implementing the monitoring agent. In section 5.3, we discussed the deployment of database and visualization software on the manager node.

5.1 Docker Swarm and Ingress Sandbox

The two most popular container orchestration tools in the industry are Docker Swarm and Kubernetes. In [4], researchers compared the functions and overhead performance of Docker Swarm and Kubernetes. The results show that docker swarm consumes fewer computing resources, but it also has less support for container orchestration functions than Kubernetes. Considering that our research environment is fog computing, and the server is often a single broad microcomputer with relatively weak computing performance, we choose Docker Swarm as the container orchestration work.

With the Docker Swarm, the internal load balancing is implemented by Docker Mesh routing mode. In our design framework, all servers on a fog node belong to a docker swarm cluster. This distribution policy means that each service joining the docker swarm cluster has a docker swarm load balancer, which is responsible for monitoring all service ports. When the internal node balancer captures a user request, it forwards the request to a container called ingress sandbox through the modified DNS forwarding rules. This container is a default container generated by Docker Swarm. The ingress sandbox container will distribute requests to containers located on different devices in the entire cluster to complete the service according to the set routing distribution rules. The monitoring tool we developed should obtain the network traffic information of all containers on the server by monitoring the ingress sandbox container. For this reason, when we run our monitoring agent, we should switch the network namespace of the system to be consistent with the ingress sandbox so that the monitoring agent be able to access all the network traffic of the ingress sandbox container.

5.2 Monitoring Agent Implementation

In the development of the monitoring agent, we chose to use Golang as the implementation language. The reason for choosing Golang is that Golang has excellent features in system development and distributed environments. Golang can also be binary compiled, which is convenient for us to quickly deploy monitoring tools on different servers.

The monitoring agent uses libpcap library to sniff the network packets forwarded via the ingress sandbox. Each network packet will be analyzed through the GoPacket Library. If the network packet is using the HTTP protocol, then there will be four situations as described in section 4.3.2. The monitoring agent has a dictionary indexed by the IP and port of the sender. If the network traffic packet contains request information, the monitoring agent will temporarily store the request in the dictionary; if the network traffic packet contains response information, the monitoring agent will match the corresponding request from the dictionary and generate a complete trace information.

5.3 Backend Implementation

We deployed MySQL on the local manager server as the back-end database for monitoring data storage. We also deployed Grafana on the local manager server as a tool for monitoring data visualization. System administrators can customize the data dashboard they want to monitor according to their needs. We will introduce the visualization results of monitoring data in detail in the chapter 6.

Chapter 6

6 Experiment

6.1 Experimental Environment

6.1.1 Servers Deployment

The Raspberry Pi [27] is considered a viable fog node component device. A Raspberry Pi is a single chip microcomputer that provides relatively lower computing power at an inexpensive cost. This cost-effective feature also makes Raspberry Pi a strong competitiveness in the large-scale deployment of the Internet of Things in the future.[15] The experiment environment used four Raspberry Pis for a fog node. Table 5 presents the specification of the Raspberry Pis. As seen in table 5 there is diversity in the hardware.

Table 6: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 was designated as the manager. The manager node hosts the request sniffer, the database MySQL, and the visualization tool Grafana. The other Raspberry PIs are designated as workers and host a request sniffer. Although these Raspberry Pis are different with respect to RAM these Raspberry Pis have the same 64-bit Quad-Core Processor.

The Raspbian[10] operating system was installed for each Raspberry Pi. Raspbian is based on the Debian system and is optimized for Raspberry Pi hardware.
The Raspberry Pis communicate with each other through Wi-Fi. We configure a fixed static IP for each Raspberry Pi in order to facilitate the identification and analysis of the communication information between the Raspberry Pis.

6.1.2 Microservice Deployment

In order to verify the feasibility of our monitoring system, we developed three related containerized microservices. The three microservices are the analyzer, data_provider and Client API.

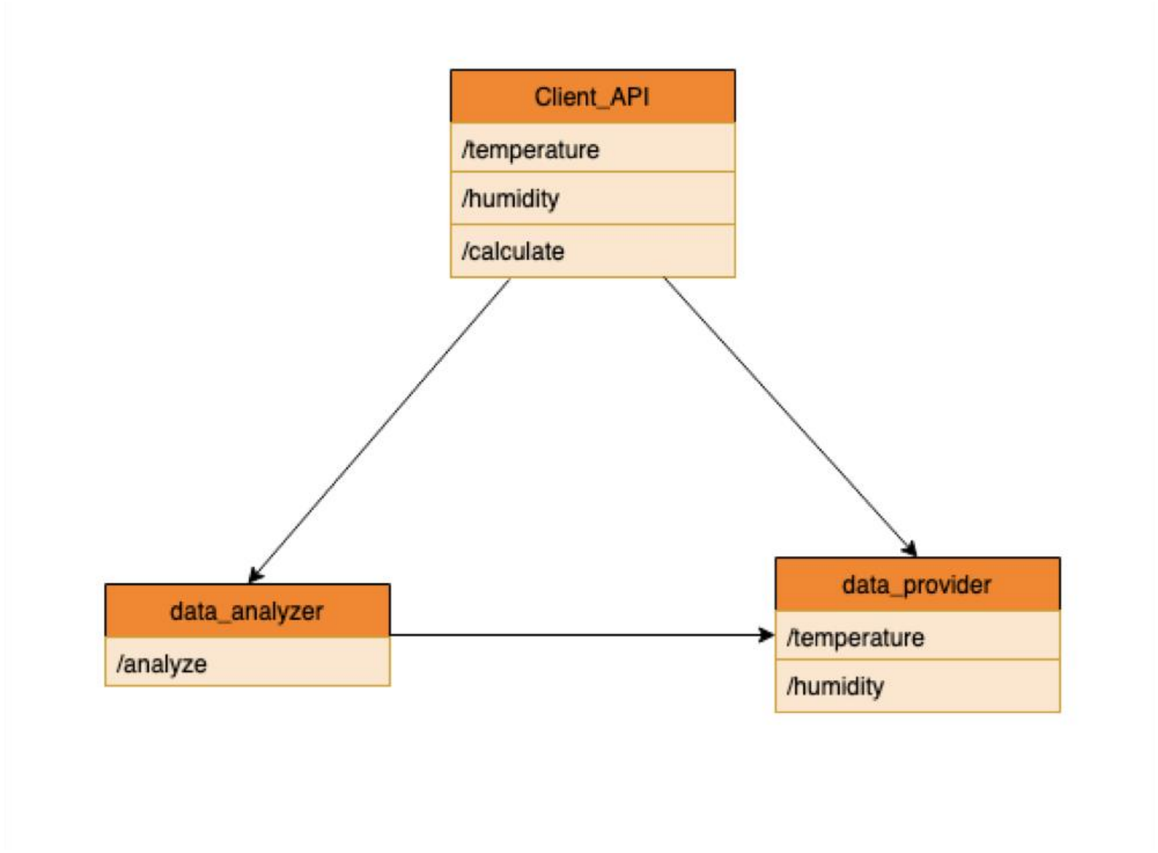


Figure 14: Microservice Dependency

[10] https://www.raspbian.org/

The data_provider simulates the collection of data from the sensor. This data may need to be temporarily saved for aggregation purposes. To support this, we deployed several random length arrays in the data_provider image. The data_provider exhibits characteristics of high RAM usage. The Analyzer is used to process the collected sensor data in real time. This is simulated with the use of loops that perform random calculations. The Analyzer container was designed to have high CPU usage. The Client_API is mainly responsible for centrally processing client requests. These services use 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 APIs to each other. The relationships are shown in figure 15.

Table 7: Microservice API Path

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

The microservices are placed into container images and uploaded to Docker hub. The microservices were then deployed on the four Raspberry Pi devices in the Raspberry Pi cluster through the Docker Swarm on the manager server. For the Client_API and data analyzer microservices there are three replicas. For the data_provider microservice there are five replicas since this microservice is intended to receive sensor data. Table 3 summarizes the deployment configuration.

The containers use different network interfaces for communication, which are bridge network and ingress network. The ingress network is used for distributing the request to the selected container. Whenever the internal load balancer receives a request, it can identify a container by the ingress IP address. The bridge IP network is used for the containers to communicate with the Docker Swarm. When a container is a sender of the request, the internal load balancer can identify the container by the bridge IP address to the selected container. The internal load balancers are configured to be able to query this necessary information from the docker swarm directly so it can identify each request to each container. However, our system does not have this information to match containers to the IP address. Hence, our monitoring system will initialize a mapping table to match containers to the IP address as the Table 7 shown.

Table 8: Container Deployment

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

6.1.3 Purpose of the Experiment and Assumption

We have two main purposes with our experiments. The first is to verify the feasibility of our framework. We want to evaluate what kind of information our monitoring framework can collect by simulating the real microservice application environment. The second purpose is that we want to evaluate the CPU overhead of the monitoring framework.

We use JMeter to simulate the users' actions, which sending requests to the servers, and we assume the users know the servers' IP addresses. In a real environment, users will not directly access the IP address of the server. Application providers usually allow users to access a domain name, and then use request proxy tools such as Nginx[11] to forward the user's request to the selected server. Our research work does not cover this process of work. Our framework only pays attention to the information monitoring after the server

[11] https://www.nginx.com/

receives the request. Therefore, our experiment uses JMeter to send requests directly to the server.

6.2 Data Visualization

We use Grafana for visualization. Grafana connects to the MySQL Database. The data is used for monitoring dashboards in real-time.

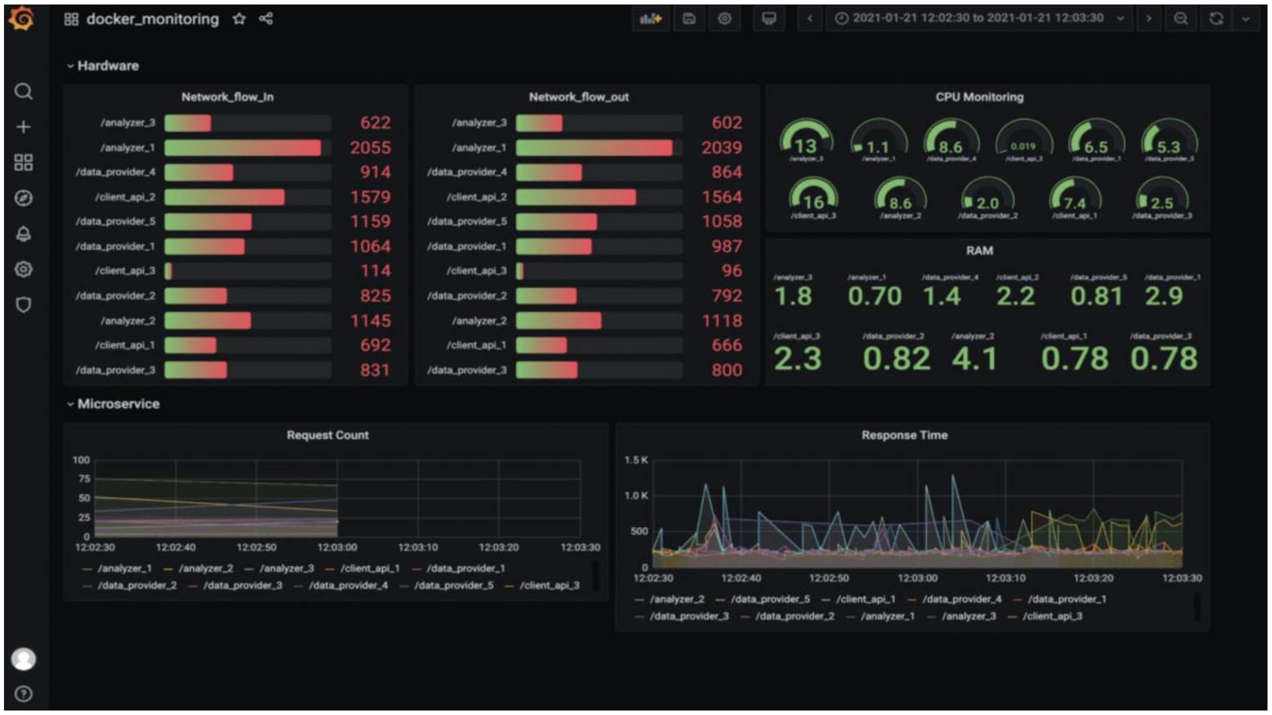


Figure 15:Real-Time Monitoring Dashboard

6.2.1 Monitoring the containers

Figure 16 shows a Grafana dashboard that container CPU usage information. The system administrator can easily obtain real-time CPU information, configure alerts, and refer to this information for real-time container management.

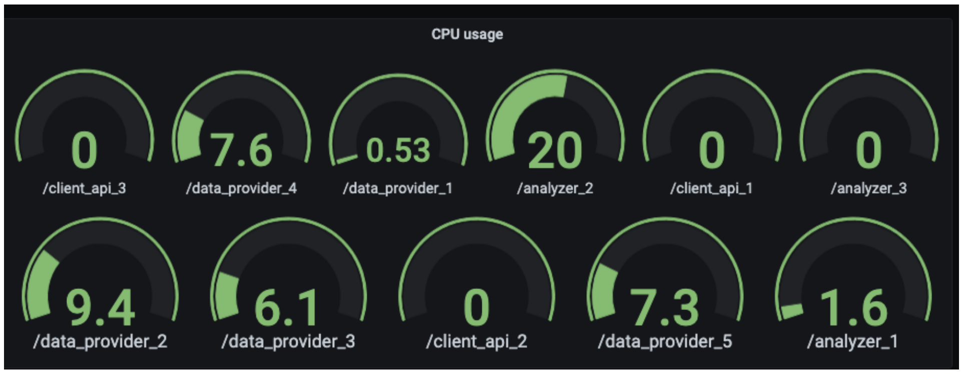


Figure 16: Real-Time CPU Monitoring

Figure 17 shows a line graph of memory usage. The reason why we choose to use the line graph is that we want to show the trend of each container's consumption of memory resources.

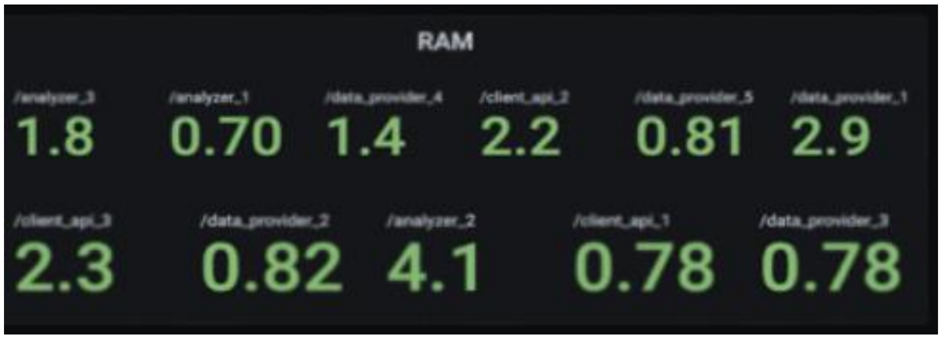


Figure 17:Real-Time RAM monitoring

Combining figures 17 and 18, we can clearly see that the host container of the analyzer and data_provider services consume more computing resources than the container host the client_api service. This is to be expected since the client_api service only provides forwarding and aggregation of requests and thus does not consume more computing resources than the other two services.

In addition to displaying real-time hardware information, we can also configure real-time container requests and average container processing time on the dashboard. The information shown in Figure 19 is the request accepted by each container at the container granularity. We can clearly see that the data_provider service, as the data source of the entire microservice system, receives the most requests. Figure 20 shows the average response time for each container. The response time is measured by calculating the difference of the request and response. 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, we find that data_provider 4 and analyzer 3 are containers that accept more requests, and their response time is longer than other similar containers.

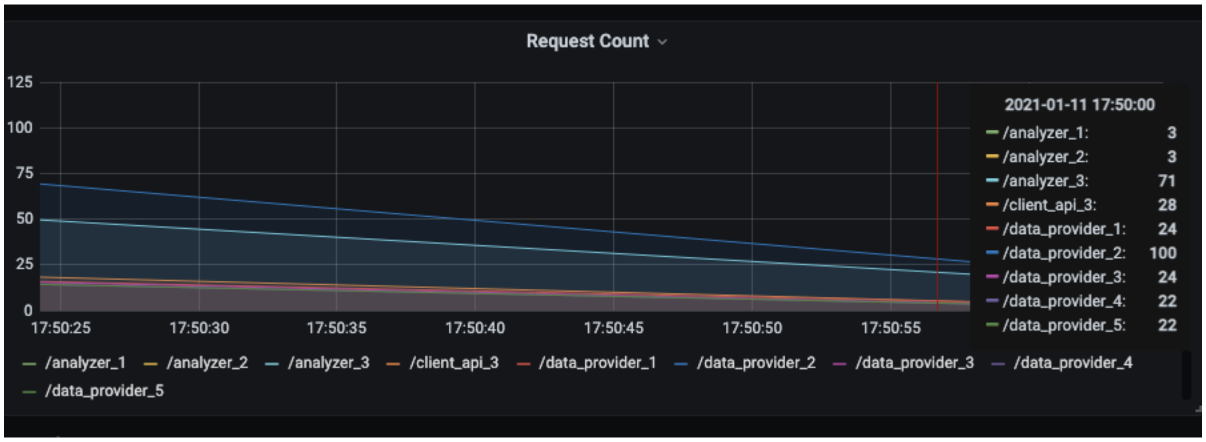


Figure 18 Request Count of each Container

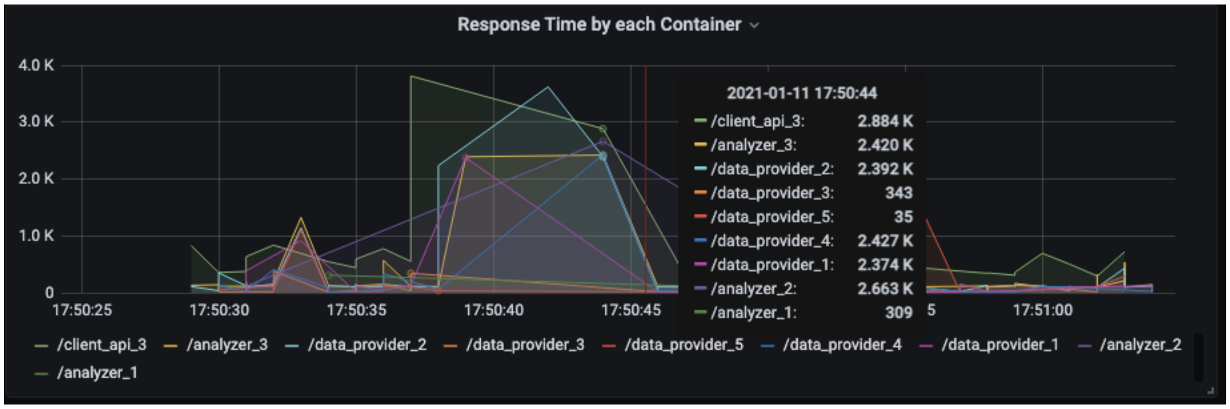


Figure 19: Response Time of each Container

6.2.2 Monitoring the Communication

In this section we present a management application that makes use of asynchronous visualization of monitoring information for a specific period of time. This is different from using Grafana which is real-time.

The information shown in Figure 21 is the request count information between two containers in this figure. To be more specific, the request count represents how many requests are communicated by the containers. 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 known by the monitoring system.

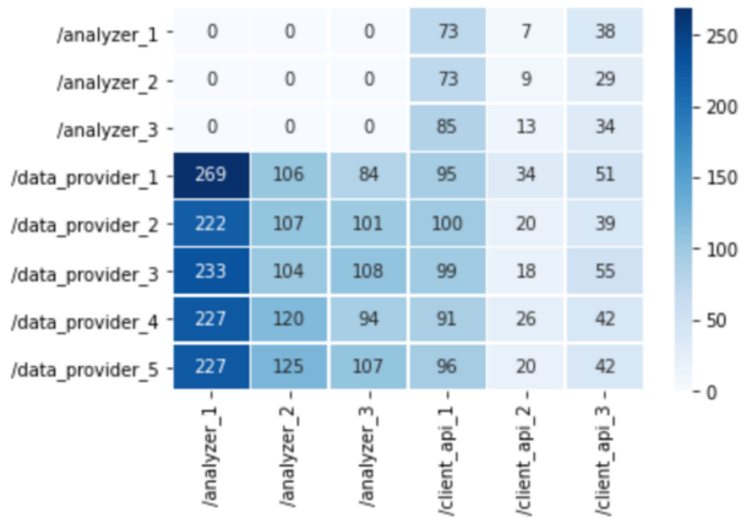


Figure 20:Communication Dependency Count Table

The Figure 22 represent the communication latency between the containers. The latency is calculated by the timestamp difference of packet sending time and the response receiving time.

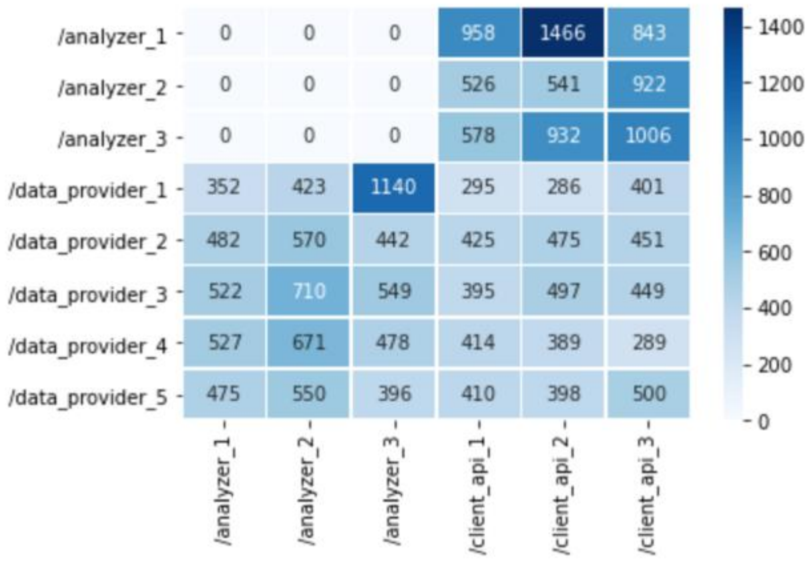


Figure 21: Communication Response-Time Table

6.2.3 Monitoring the Microservices

The charts and information we have shown above all use the container as the analysis granularity to evaluate the working state of the container. Our framework also supports analysis with microservices as the granularity. As shown in Figure 22, we aggregated the average response time of three microservices. As can be seen from the figure, client_api y takes longer to respond to requests than the other two services. The data_provider has the fastest response time. This data result is consistent with the characteristics of the service we deployed. The main functionality of Client_API is to receive requests from clients, then send requests to other services, and finally aggregate information to clients.

Therefore, Client_API related requests often need to send requests to other services, wait for a response, and then reply to the user. Client_API is highly dependent on other services. Network delay and congestion can easily affect the response speed of Client_API. Data_provider has no similar problems since it designed to accept requests from other two microservices and provide data immediately. In this process, there is no need to send additional requests, nor do a lot of complicated calculations. Analyzer microservice will firstly query small amount of data from the data_provider and then does small amount of calculation before response back to the client.

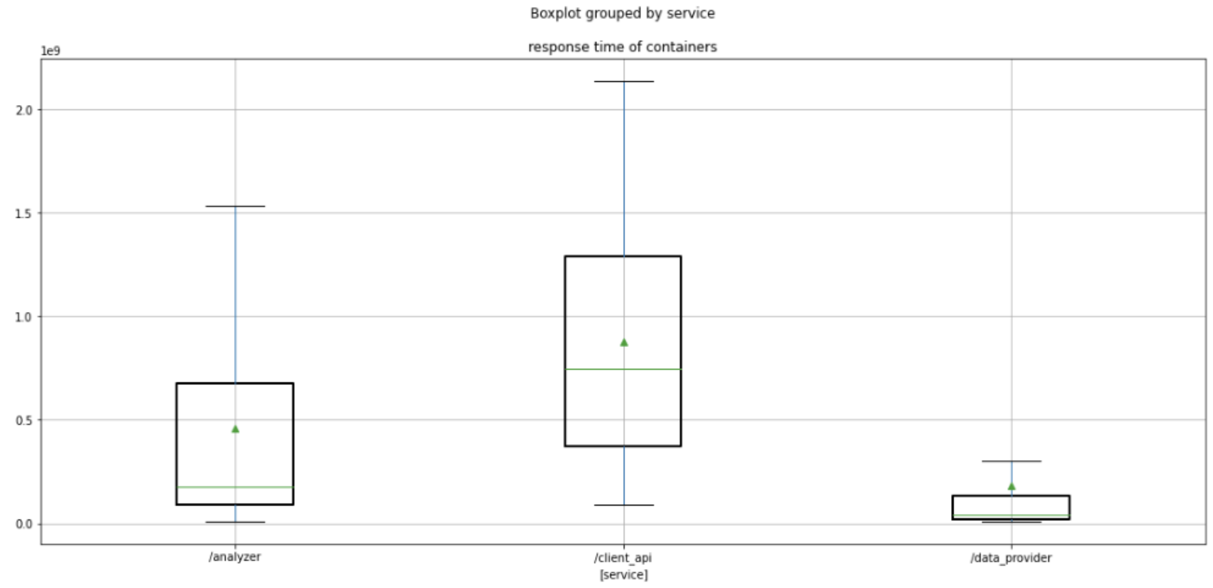


Figure 22: Response Time of Services

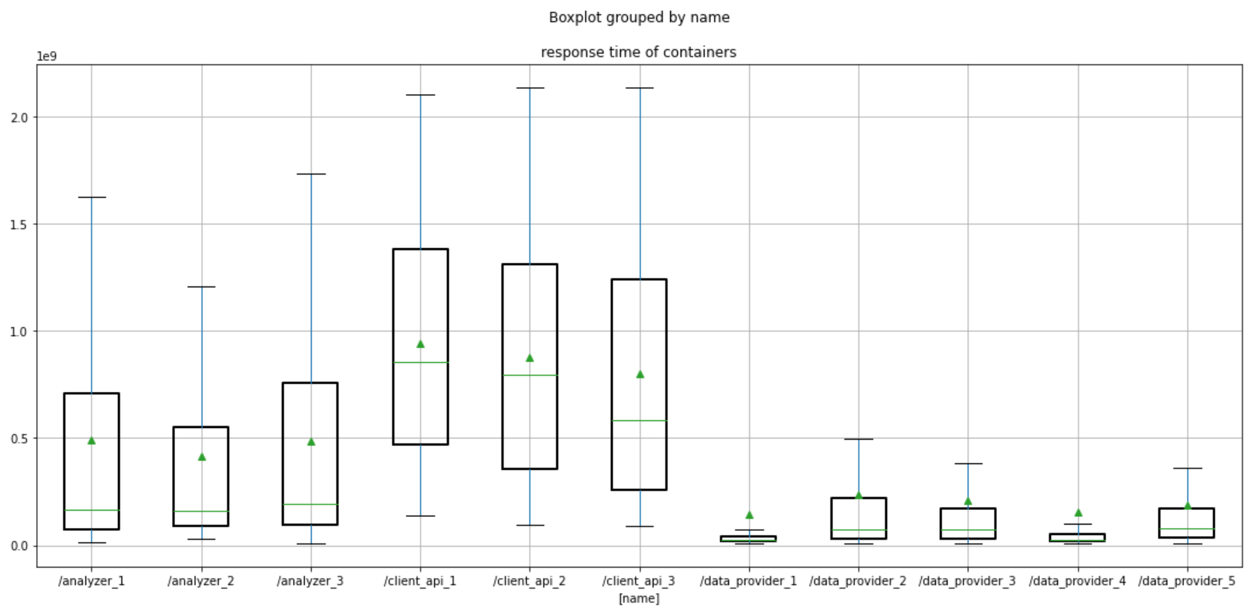


Figure 23: Response Time of Containers

The above microservice characteristics can be directly reflected on the monitoring system. Although the system administrator's internal knowledge of microservices is completely black box, the system administrator can still determine the service's network traffic, hardware usage, received and sent request data statistics, response time, dependency, etc. The characteristics of microservices.

6.3 Performance Overhead

Table 9: Framework CPU usage

	With agent	Without agent	
concurrency	CPU usage	CPU usage	difference
0	0.29	0.69	0.4
1	4.89	20.98	16.09
10	21.557	28.47	6.913
100	62.67	70.68	8.01
1000	84.72	94.01	9.29

In order to analyze the use of computing resources of the framework, we collected the CPU usage of the system when the monitoring agent was turned on, and the CPU usage of the system when the monitoring agent was turned off. When the test is in progress, no other applications on the tested Raspberry Pi server occupy system resources except for the monitoring agent and docker microservices. Concurrency represents how many clients continue to send requests to the server at the same time. We calculate the overall CPU usage of the system of the monitoring agent 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 turned on.

In each test, we continue to collect the CPU usage of the system for five minutes. We then calculated the average value of CPU usage in five minutes and fill it in Table 8. When the number of concurrent clients is 10, the monitoring agent consumes about 6.9% of the system CPU time. As the number of concurrent clients increases, the CPU usage of the monitoring agent also increases.

It is also worth considering that our sampling method is immature. Due to the uncertainty of throughput and the complexity of service, a large number of tracking tasks themselves will bring a large number of computing requirements. Therefore, in order to limit the encroachment of computing resources by monitoring tools, in white box microservice monitoring tools such as Dapper, their solution is to limit the performance loss of the server by setting the sampling rate. When the amount of data is too large, the monitoring system will strictly control the CPU usage, and only sample and monitor microservices within the range allowed by the CPU usage limit. Monitoring records are only a small portion of all information. System administrators can infer overall system performance by analyzing sample data. Our framework collects all request information, and also completes real-time analysis, matching, aggregation, storage, and visualization. This is undoubtedly a huge consumption of CPU. We will optimize this problem in future work.

Chapter 7

7 Future Work

In this work, we discussed how to collect service information of microservices in a black box in the case of fog computing and implemented our ideas through the framework. The purpose of our preliminary work is to obtain sufficient information through complete information collection to determine the operating status, service characteristics and dependencies of each container. We propose a solution for black box monitoring of containers by monitoring the load balancer of the container management tool. Through experiments, we successfully demonstrated that we can provide operational data for visualization that can help system administrators evaluate the running status of containers using a black box approach. The system administrator does not need to understand and modify the target microservice to collect the service characteristics of the containerized microservice. And our method is suitable for the edge of the network, that is, it can run smoothly on the microcontroller with relatively weak computing performance.

In future work, we have two research directions, namely the use of monitoring data and the continued development of the system.

- **Application of monitoring data**

we hope to use the collected information for some dynamic container deployment, such as dynamic horizontal expansion of containers and real-time migration of containers. What caught our attention is that in the work of container deployment and migration, several articles represented by [13] mentioned the concept of task function. We believe that our framework can provide the characteristics of containerized services in a black box manner. For example, in [13], they manually mark tasks using tags such as priority, calculation density, and waiting time requirements, and then use algorithms to match the appropriate hardware to run. Our service collection method can not only use hardware information to determine the service's propensity for hardware resources (powerful CPU, powerful RAM or powerful bandwidth), but also can obtain similar features of high request density and high concurrency through service requests. These black box analysis functions can be combined into algorithms for automatic deployment and management. Therefore, we want to study the relevant quantitative indicators required by the container migration algorithm, rather than the visualization functions provided by the framework at this stage. Then we will try to generate these metrics in real time using our framework. If feasible and with sufficient features and data sets, we can consider training machine learning models for container management.

Our current work can also only collect every request information for each container. However, in microservices, there is often a correlation between requests and requests. For example, in an access event, request A is triggered by request B. Our framework currently cannot associate these requests together. To solve this problem, we consider researching and using a nesting algorithm [8] proposed by M. Aguilera et al. in 2003. This algorithm can judge multiple requests by analyzing request information (such as source, target, timestamp, etc.) obtained through black box monitoring. Whether the request belongs to the same event.

- **Extend the exploration of system information.**

In this preliminary framework, we are using the ingress sandbox of the docker swarm to perform all the controls and aggregate all services into one network. This approach actually abandons the network isolation caused by the overlay network itself and avoids port conflicts. We will further learn how to integrate the overlay network into this framework to reduce development problems caused by port conflicts. We also hope to try to modify part of the container management tool code so that our framework can provide a real-time quantitative index for the load balancing algorithm to help optimize the load balancing algorithm.

- **Optimize the Framework For Real Production Environment**

Our current work is only considering the monitoring prototype in the fog computing environment, and we have not been able to put this framework in a real environment to work. In order to adapt to the complex network edge environment, there are several issues that need to be considered. For example, for cross-node communications at the edge of the network, we need to encrypt communications for information security issues. Secondly, we currently use structured databases like MySQL to store data. Considering the problem of database capacity, we will consider using a time series database such as Influx DB[12] for storage, and only retain the data for a certain period (such as a week).

- **Support other Protocols**

Finally, we will continue to develop this framework to support more service layer network protocols; in fog computing, the most noteworthy application prospect is the development of IoT applications. Therefore, we will extensively learn the common communication protocols of the Internet of Things environment represented by the MQTT protocol and enable our framework to support these protocols. In this case, our future usage scenarios will move closer to the Internet of Things.

12 https://www.influxdata.com/

Bibliography

[1] https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[2] Gillam, L., Katsaros, K., Dianati, M., & Mouzakitis, A. (2018, April). Exploring edges for connected and autonomous driving. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (pp. 148-153). IEEE.

[3] Chen, N., Chen, Y., Song, S., Huang, C. T., & Ye, X. (2016, October). Smart urban surveillance using fog computing. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)* (pp. 95-96). IEEE.

[4] 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.

[5] Großmann, M., & Schenk, C. (2018, December). A comparison of monitoring approaches for virtualized services at the network edge. In *2018 International Conference on Internet of Things, Embedded Systems and Communications (IINTEC)* (pp. 85-90). IEEE.

[6] 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).

[7] Bali, A., & Gherbi, A. (2019, December). Rule based lightweight approach for resources monitoring on IoT Edge devices. In *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds* (pp. 43-48).

[8] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., & Muthitacharoen, A. (2003). Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, *37*(5), 74-89.

[9] Ahanger, T. A., Tariq, U., & Nusir, M. (2018). Mobility of Internet of Things and Fog Computing: Concerns and Future Directions. *International Journal of Communication Networks and Information Security*, *10*(3), 534.

[10] Brandón, Á., Pérez, M. S., Montes, J., & Sanchez, A. (2018). Fmone: A flexible monitoring solution at the edge. *Wireless Communications and Mobile Computing*, *2018*.

[11] Brogi, A., Forti, S., & Gaglianese, M. (2019, October). Measuring the Fog, Gently. In *International Conference on Service-Oriented Computing* (pp. 523-538). Springer, Cham.

[12] Souza, A., Cacho, N., Noor, A., Jayaraman, P. P., Romanovsky, A., & Ranjan, R. (2018, June). Osmotic monitoring of microservices between the edge and cloud. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (pp. 758-765). IEEE.

[13] 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.

[14] Jalali, F., Smith, O. J., 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).

[15] Muralidharan, S., Song, G., & Ko, H. (2019). Monitoring and managing iot applications in smart cities using kubernetes. *CLOUD COMPUTING*, *11*.

[16] Hassan, S., Bahsoon, R., & Kazman, R. (2020). Microservice transition and its granularity problem: A systematic mapping study. *Software: Practice and Experience*, *50*(9), 1651-1681.

[17] 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).

[18] de Brito, M. S., Hoque, S., Magedanz, T., Steinke, R., Willner, A., Nehls, D., ... & Schreiner, F. (2017, May). A service orchestration architecture for fog-enabled infrastructures. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)* (pp. 127-132). IEEE.

[19] Santos, J., Wauters, T., Volckaert, B., & De Turck, F. (2019, June). Towards network-aware resource provisioning in Kubernetes for fog computing applications. In *2019 IEEE Conference on Network Softwarization (NetSoft)* (pp. 351-359). IEEE.

[20] Battula, S. K., Garg, S., Montgomery, J., & Kang, B. (2019). An efficient resource monitoring service for fog computing environments. *IEEE Transactions on Services Computing*, *13*(4), 709-722.

[21] 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.

[22] Cinque, M., Della Corte, R., & Pecchia, A. (2019). Microservices monitoring with event logs and black box execution tracing. *IEEE Transactions on Services Computing*.

[23] Huang, C., Lu, R., & Choo, K. K. R. (2017). Vehicular fog computing: architecture, use case, and security and forensic challenges. *IEEE Communications Magazine*, *55*(11), 105-111.

[24] Kraemer, F. A., Braten, A. E., Tamkittikhun, N., & Palma, D. (2017). Fog computing in healthcare–a review and discussion. *IEEE Access*, *5*, 9206-9222.

[25] De Brito, M. S., 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.

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

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

[28] Kristiani, E., Yang, C. T., Wang, Y. T., Huang, C. Y., & Ko, P. C. (2018, October). Container-based virtualization for real-time data streaming processing on the edge computing architecture. In *International Wireless Internet Conference* (pp. 203-211). Springer, Cham.

[29] 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.

[30] 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).

[31] Sayfan, G. (2017). *Mastering kubernetes*. Packt Publishing Ltd.

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

[33] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., ... & Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure.

[34] 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.

[35] Li, W., Lemieux, Y., Gao, J., Zhao, Z., & Han, Y. (2019, April). Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)* (pp. 122-1225). IEEE.

[36] https://istio.io/latest/docs/ops/deployment/performance-and-scalability/

[37] Pina, F., Correia, J., Filipe, R., Araujo, F., & Cardroom, J. (2018, November). Nonintrusive monitoring of microservice-based systems. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)* (pp. 1-8). IEEE.

[38] Kumar, V., Laghari, A. A., Karim, S., Shakir, M., & Brohi, A. A. (2019). Comparison of fog computing & cloud computing. *Int. J. Math. Sci. Comput*, *1*, 31-41.

[39] Kaur, M. J., & 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.

[40] Cisco. 2015. Cisco Fog Computing with IOx. Retrived August 8, 2020 from

http://www.cisco.com/web/solutions/trends/iot/cisco-fog-computing-with-iox.pdf

[41] Alhamazani, K., Ranjan, R., Jayaraman, P. P., 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.

[42] Zou, J., Li, W., Wang, J., Qi, Q., & Sun, H. (2018, September). NFV Orchestration and Rapid Migration Based on Elastic Virtual Network and Container Technology. In *2018 IEEE International Conference on Information Communication and Signal Processing (ICICSP)* (pp. 6-10). IEEE.

[43] Yousefpour, A., Patil, A., Ishigaki, G., Kim, I., Wang, X., Cankaya, H. C., ... & Jue, J. P. (2019). FOGPLAN: A lightweight QoS-aware dynamic fog service provisioning framework. *IEEE Internet of Things Journal*, *6*(3), 5080-5096.

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

[45] Tran, M. Q., Nguyen, D. T., Le, V. A., Nguyen, D. H., & Pham, T. V. (2019). Task placement on fog computing made efficient for iot application provision. *Wireless Communications and Mobile Computing*, *2019*.

[46] https://docs.docker.com/engine/swarm/ingress/

[47] https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer

[48] Sun, L., Li, Y., & Memon, R. A. (2017). An open IoT framework based on microservices architecture. *China Communications*, *14*(2), 154-162.

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

[50] 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).

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

[52] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C., Anderson, A. V., ... & Smith, L. (2005). Intel virtualization technology. *Computer*, *38*(5), 48-56.

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

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

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

[56] Dolui, K., & Kiraly, C. (2018, December). Towards multi-container deployment on IoT gateways. In *2018 ieee global communications conference (globecom)* (pp. 1-7). IEEE.

[57] Dupont, C., Giaffreda, R., & Capra, L. (2017, June). Edge computing in IoT context: Horizontal and vertical Linux container migration. In *2017 Global Internet of Things Summit (GIoTS)* (pp. 1-4). IEEE.

[58] https://docs.vmware.com/en/VMware-Tanzu-Service-Mesh/services/tanzu-service-mesh-environment-requirements-and-supported-platforms/GUID-D0B939BE-474E-4075-9A65-3D72B5B9F237.html