POLITECNICO DI MILANO

# Improving DaaS with Kubernetes in Fog Computing Environments

**Relatore:**
Monica Vitali
**Correlatore:**
Pierluigi Plebani

**Student:**
Sun Chao

Department of Electronics, Informatics and Bioengineering

M.Sc. programme in Computer Science and Engineering

# Abstract

Enormous amounts of data are generated from different types of devices, including IOT devices, embedded sensors and personal computers. How to effectively manage these data without breaking related constraints and privacy policies has become a crucial and valuable issue for data-intensive applications, especially in the Fog Computing environment where both resources at the edge of the network and resources in the cloud are involved.

Fog Computing extends the capability of Cloud Computing by exploiting the potential of the edge of the network. With Fog Computing, we can take advantage of both resources on the edge (less latency) and on the cloud (scalability,reliability). There are two possibility when it comes to manage huge volume of data between cloud and edge of the network: to move the data or to move the computation tasks. In this thesis, we mainly focus on the computation task movement. To enable computation task movement, a DaaS approach applied to a Fog environment has been adopted.

Our goal is to propose a user-friendly DaaS platform for the data-intensive application developers where data processing occurs not only on cloud but also at the edge of the network, taking into account the heterogeneous IoT devices, network delay and privacy issues. The proposed platform, which is designed under the principles of Service Oriented Computing, is able to provide the monitoring ability and the movement decisions.

**keyword:** Data-intensive application, Kubernetes, Fog Computing, Computation task movement, DaaS, Data management

# Riassunto

Enormi quantità di dati vengono generate da diversi tipi di dispositivi, inclusi dispositivi IOT, sensori incorporati e personal computer. Come gestire in modo efficace questi dati senza infrangere i vincoli e le politiche sulla privacy è diventato un problema cruciale e prezioso per le applicazioni ad alta intensità di dati, specialmente nell'ambiente di Fog Computing in cui sono coinvolte sia le risorse ai margini della rete sia le risorse nel cloud.

Fog Computing estende la capacità del Cloud Computing sfruttando il potenziale del bordo della rete. Con Fog Computing, possiamo sfruttare entrambe le risorse sul bordo (meno latenza) e sul cloud (scalabilità, affidabilità). Ci sono due possibilità quando si tratta di gestire un volume enorme di dati tra il cloud e il bordo della rete: spostare i dati o spostare le attività di calcolo. In questa tesi, ci concentriamo principalmente sul movimento del compito di calcolo. Per consentire il movimento delle attività di calcolo, è stato adottato un approccio DaaS applicato a un ambiente di nebbia.

Il nostro obiettivo è proporre una piattaforma DaaS user-friendly per gli sviluppatori di applicazioni ad alta intensità di dati in cui l'elaborazione dei dati avviene non solo su cloud ma anche ai margini della rete, tenendo conto di dispositivi IoT eterogenei, ritardo di rete e problemi di privacy. La piattaforma proposta, progettata secondo i principi di Service Oriented Computing, è in grado di fornire le capacità di monitoraggio e le decisioni relative ai movimenti.

**keyword:** Applicazione ad alta intensità di dati, Kubernetes, Fog Computing, Computing task di calcolo, DaaS, Gestione dei dati

# Abbreviations and Acronyms

| | |
|---|---|
| **IOT** | Internet of Things |
| **DIA** | Data Intensive Application |
| **API** | Application Programming Interface |
| **DB** | Database |
| **CLI** | Command-Line Interface |
| **GUI** | Graphical User Interface |
| **IAAS** | Infrastructure as a Service |
| **PASS** | Platform as a Service |
| **SAAS** | Software as a Service |
| **DAAS** | Data as a Service |
| **k8s** | Kubernetes |
| **REST** | Representational State Transfer |
| **SQL** | Structured Query Language |
| **YAML** | Yet Another Makeup Language |
| **JSON** | JavaScript Object Notation |
| **VM** | Virtual Machine |
| **VDC** | Virtual Data Container |
| **VDM** | Virtual Data Management |
| **VMM** | Virtual Machine Monitor |

| | |
|---|---|
| **LXC** | Linux Container |
| **SOA** | Service Oriented Architecture |
| **QoS** | Quality of Service |
| **RBAC** | Role-Based Access Control |
| **CSP** | Cloud Service Provider |
| **CNCF** | Cloud Native Computing Foundation |
| **Cgroup** | Control group |
| **TRP** | Third Part Resource |

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

With the development of the Internet Of Things (IoT), more and more smart devices such as personal computers, laptops, sensors and wearable devices are put into use. At the same time, countless data have also been generated. These data are distributed and consumed in different fields, including smart cities, health care, factory production and even our daily use. These data will become valuable only if they are correctly and effectively analyzed.

In this era of data explosion and information overload, both application developers and consumers are often at a loss when facing too much content. How to put the right data in the right place without breaking legal rules and policies, while having good scalability and security has become a key issue for designing Data Intensive Applications.

In the face of multiple data centers in a cloud environment, data-intensive applications inevitably require data transmission across data centers from where they are collected to the places they are processing. Data-intensive applications, especially process-oriented ones, have encountered some new challenges in using the cloud computing environment. On the one hand, the scale of data is huge and the network bandwidth between data centers is limited. On the other hand, some data can only be stored in designated data centers and cannot be moved. Therefore, data transmission between data centers becomes a challenge.

In Cloud infrastructures, based on the resources on the cloud, applications can be quickly built, shortening the product development cycle while providing greater flexibility and scalability. However, there are still some limitations when it comes to computing IoT-generated data, especially in the cloud environment. Due to the limitations of Cloud Computing, in some situations using the cloud infrastructure will become unfeasible and less efficient [5].

The following two issues becomes the major challenges when handling IoT-generated data in the cloud environment [5].

- *privacy/security*: In some sensitive areas, data cannot be moved from the site of collection due to security and privacy concerns. This means that data cannot be uploaded to the cloud platform for processing and can only be stored and processed on local devices that have permissions. For example, when it comes to business secrets and government information, the use and transmission of relevant data on the cloud platform is legally restricted, which makes it unfeasible to process this part of data through cloud computing.
- *latency*: For some time-critical applications, due to the latency of data transmission from local site to the cloud through network, the processing time on the cloud will not be able to meet their operating time requirements. In some scenarios, uploading large amounts of data from the IOT device to the cloud can cause network congestion, which will increase network latency.

Another challenge comes from device heterogeneity[18]. There are different types of IoT devices which generate data with different format and quality. On the other side, each cloud-based infrastructure deployed with its own characteristics could be managed differently. How to transfer available data and computation resources from the cloud to the edge and vice versa at the right time, right place and with good quality raises to be a critical issue when designing data-intensive applications. To enable data and computation movement with one-size-fit-all approach among all the devices could be really difficult.

To tackle these issues, we introduce the concept of Fog Computing[9]. Fog computing, also referred to as Edge Computing, is an emerging paradigm aiming to extend Cloud Computing capabilities to fully exploit the potential of the edge of the network where traditional devices, as well as new generations of smart devices, wearables, and mobile devices (the so-called Internet of Things)[18]. With Fog Computing, we can take advantage of resources regardless where they are living to exploit the respective advantages. Generally speaking, resources live on the edge (where they are produced) can acquire less latency and response time and resources live on the cloud (where they are processing) can obtain better scalability and reliability.

There are two possibility when it comes to computing huge volume of data generated by IoT devices: to move the data or to move the computation tasks. In Fog Computing paradigm, data could be stored on the edge side if the data cannot leave the boundary of the organization that owns them due to the privacy and security reason. Only after data are transformed to preserve the privacy, they can be moved to the cloud. On the other side, Data computation tasks remain on the edge with the available computation power provided by the edge devices in order to reduce network latency. When the network bandwidth limitation is not an issue, data computation tasks could be transmitted to the cloud. In the thesis, we focus on the movement of computation tasks in a Fog Computing environment.

An approach following Service Oriented Computing principle is adopted to enable data and computation movement in a Fog environment. In order to move the computation tasks of data intensive applications, a DaaS execution environment is proposed, which hides the underlying complexity of an infrastructure made of heterogeneous devices and follows all the constraints of network delay and privacy issues mentioned before. It convenient data-intensive application developers focus on the design of the logic layer, ignoring the complex infrastructure configuration and data preprocessing.

In the proposed architecture, data are provided as a service for different data intensive applications with specific data requirements and data quality. Applications designed and developed on our DaaS platform is able to adapt the features of both Edge Computing (less latency and loose privacy principle) and Cloud Computing (high reliability and scalability).

## 1.2 Research goals

Our goal is to propose a user-friendly DaaS platform for the DIA developers where data processing occurs not only on cloud but also at the edge of the network, taking into account the heterogeneous IoT devices and privacy issues mentioned before. DIA developers are completely transparent about the data processing inside the platform. They do not have to be distracted by how to get different types of data in different locations. This makes them only need to pay attention to the design of the business layer, which will greatly improve the development efficiency of data-intensive applications.

With Fog Computing, data processing tasks of data intensive applications are able to occur either on the cloud side or on the edge side. It provides the flexibility to move tasks with respect to the resources available on both the cloud and the edge on the network. The proposed DaaS platform should be able to provide computation movement strategies to decide when, where and how to deploy functions.

To achieve this goal, some methods and modules will be introduced and implemented. We divide the main goal into several sub-goals from the following perspectives.

1. Based on the state-of-art, a theoretical DaaS architecture in a Fog Computing environment, which is able to provide expected data according to the functionality and non-functionality requirements provided by the DIA developers, is proposed.

2. In order to decide whether to enact a data and computation movement in a Fog Computing environment, it is necessary to track the statues of the deployed application and state of the execution environment. A customized monitoring system should be able to track both the regular monitoring metrics and the custom metrics.

The DaaS execution environment is built on the top of kubernetes cluster. Inner modules such as monitoring system, task movement selector, task movement enactor are all deployed inside kubernetes. By exploiting the capability of kubernetes, the goal is to provide a better support for the DaaS platform to enable computation task movement in case of mixed architecture execution environment. Based on the Docker technology, it provides a series of complete functions such as deployment and operation, resource scheduling, service discovery, and dynamic scaling for containerized applications, improving the convenience of large-scale container cluster management[1].

[1]https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

## 1.3 Thesis structure

The rest of the paper is organized as follows.

Chapter 2 clarify the background concepts which are related with the thesis, including the concepts of Fog computing and Data Intensive application, the comparison of two virtualization techniques and why we choose Google kubernetes as a container management system. By exploiting state-of-art, it summarizes and compares task movement methods between cloud platform and edge of the network.

Chapter 3 presents the overview of the DaaS framework. It introduced the entire architecture at a theoretical level and provided some strategies for the task migration under the kubernetes and a theoretical implementation for the monitoring module.

Chapter 4 is the specific implementation part of DaaS platform based on kubernetes. It illustrates the method for the personalization of DaaS and the management of multiple accesses to data sources. Also, by mapping the methodology to the implementation, a DaaS platform built with a Kubernetes cluster is adopted and some key points for deploying the customized monitoring system is approved.

Chapter 5 will evaluate and test the built platform based on existing data sources. The functionality test for the deployed customized monitoring system and containerized data processing application is enacted.

Chapter 6 summarizes what has been done and the future develop of the proposed approach.

# Chapter 2

# Background Concepts

## 2.1 Data Intensive Applications

With the popularity of WEB applications and the advancement of information collection technology, the ability of humans to produce and collect data is rapidly developing, and the amount of data that people need to face is also growing.

To cope with this challenge, the information integration technology has enabled the current information system to integrate various types of data from different regions, at different times, and in different application fields. Usually these data include one or more of the types of data such as natural observations, industrial production, product information, commercial sales and so on.

The typical characteristics of these data are massive, heterogeneous, semi-structured or unstructured [15]. Through the Internet, providing various types of Internet services or information services based on massive data is a trend in the development of the information society. This trend has brought new technical and research issues to the industry and academia. One of the important features of these new services is that they are all based on massive data processing. Under this background, Data-Intensive Computing naturally attracts extensive attention as a supporting technology for new services.

Data intensive applications are a class of parallel computing applications which collect, analyze, and process a large amount of data. An application can be called data-intensive if data is its primary challenge: the quantity of data, the complexity of data, or the speed at which it is changing [14].

First of all, the object it deals with is data, which is calculated around the data. The amount of data it needs to process is very large and changes rapidly. They are often distributed and heterogeneous. Therefore, traditional database management systems cannot meet their needs.

Second, "calculation" includes the entire process from data acquisition to management to analysis and understanding. Therefore, it is different from data retrieval and database query, and also different from traditional scientific computing and high performance computing. It is a combination of traditional data management, high-performance computing, and data analysis and mining.

Third, its purpose is to promote the development of technological frontiers. The tasks to be promoted are applications that rely on traditional single data sources and quasi-static databases.

Due to these new characteristics of data management tasks in data-intensive computing, traditional data management techniques are no longer applicable in data-intensive computing applications. There are three concerns that are important in most software systems, including the data intensive applications[14]:

- *Reliability*: even if the system has a situation such as hardware damage or human error, it can still operate as we expected.
- *Scalability*: as the system grows (in terms of data volume, traffic, or complexity), there should be a reasonable way to handle this increase.
- *Maintainability*: over time, many different people will work to improve the data system (both to maintain current behavior and adapt the system to the new environment), and they should all be able to work effectively.

Data-intensive computing has a close relationship with the concept of Cloud Computing However, Cloud Computing places more emphasis on the way systems provide services, while data-intensive computing emphasizes the management and processing of data behind the system.

Last but not least, because they are usually deployed on large-scale distributed systems, data-intensive computing applications must provide good fault tolerance to reduce system maintenance, the cost of query/processing redo, and improve system high availability. In order to obtain good fault tolerance, the application can even accept errors in data processing results to some extent. This is a fundamental difference from the traditional transaction processing model[4][14].

## 2.2 Fog Computing

Comparing to cloud computing, fog computing is a decentralized computing infrastructure which aims to extend computing ability by combining both the Cloud and the Edge of the network. Unlike cloud computing which primarily stores everything in the cloud data centers, fog computing places data, data processing ability and applications on one or more collaborative end-user clients or near-user edge devices so that computing happens near data sources.

Since edge devices or sensors have limited power in processing data, usually computing tasks and storage resources are moved to cloud for advanced analytics. In this way, it often takes longer time due to the far distance between the cloud servers and the edge devices. What is more, it may cause privacy, security and legal implications issues when connecting all the end-users to the cloud over the internet, especially when dealing with sensitive data subject to regulation in different countries.

With the rapid development of IoT, countless devices and sensors are producing really a huge amount of data. Connecting all the devices by sending the raw data through the internet to the cloud for analysis would also require a mass of bandwidth which increase economical spending. Also, time-critical applications may be effected by the network delay. The network is becoming the bottleneck of the Cloud computing paradigm. So that is why today's cloud models are not designed for the volume, variety, and velocity of data that the IoT generates [6].

In Fog Computing, in order to reduce network traffic issues, data processing is moved to the edge devices (such as mobile, data sensors, smarter router, etc.) which are closer to the data generators and consumers. But that is not to say we abandon the use of the cloud servers, since Fog Computing is the complement of Cloud Computing. Data processing in the edge networking should have a relative lower response time for short-term analytics while data processing can have a deeper and long-term analytics in the cloud side. In this thesis, I will discuss when and how to move different processing tasks between the cloud side and the edge side in both directions in order to get a better performance for Data Intensive Applications.

Edge nodes and Cloud nodes are two kind of resources in the fog network, in the following, their overview and main characteristics are given:

- *Edge nodes*

  - The resources which physically belong to the certain organization, including computers, laptops, single-board computers, networking hardware and different types of sensors and devices within its boundaries, residing between data sources and Cloud-based datacenter.
  - Receive feeds from IoT devices using any protocol, in real time.
  - Run IoT-enabled applications for real-time control and analytics
  - Typically provide transient storage, often few hours.
  - Send periodic data summaries to the Cloud.

- *Cloud nodes*

  - The resources which are rented from various service providers, mainly used for Cloud data storage (database servers), virtual machine or data analysis and processing tasks (as described in Cloud overview sub-section).
  - Receive and aggregate data summaries from many Edge node
  - Perform analysis on the IoT data and data from other sources to obtain business insight.
  - Can send new application rules to Edge nodes based on these insights.

Developers either port or write IoT applications to support execution on the network Edge. The Edge nodes ingest the data from IoT devices, and then direct different types of data to the optimal place for analysis, taking into account network congestion, execution time requirements, privacy and security issues.

Main advantages of Fog Computing [6]:

- *Greater business agility*: With the right tools, developers can quickly develop fog applications and deploy them where needed. Machine manufacturers can offer MaaS to their customers. Fog applications program the machine to operate in the way each customer needs.
- *Better security*: Protect your fog nodes using the same policy, controls, and procedures you use in other parts of your IT environment. Use the same physical security and cybersecurity solutions.
- *Deeper insights with privacy control*: Analyze sensitive data locally instead of sending it to the cloud for analysis. Your IT team can monitor and control the devices that collect, analyze, and store data.
- *Lower operating cost*: Conserve network bandwidth by processing selected data locally instead of sending it to the cloud for analysis.

The concept of Fog Computing is relevant to this thesis, as we want to enable the flexibility of data-intensive applications by providing the ability to move tasks between Cloud and Edge, according to the constraints imposed by legal regulations on the first place, taking into account the security, privacy, network-related issues and the device heterogeneity.

## 2.3 Virtualization Techniques

As the amount of data generated by devices in the Internet of Things grows rapidly, the need for computing, transmission, and storage of IT resources also increases. Therefore, how to effectively use these resources becomes even more important. This leads to issues such as managing, organizing, and allocating automatic resources and effectively utilizing heterogeneous resources when designing data-intensive applications. One of the possible solutions is to use a virtual computing environment that is built on open Internet infrastructure and provides integrated and distributed services for the end-users.

Virtualization is a kind of resource management technology. It abstracts and converts various physical resources of a computer, such as servers, networks, memory, and storage, and breaks through the uncutting barriers between the physical structures, so that users can apply these resources better than the original configuration. The virtual part of these resources is not limited by infrastructure, geographical or physical configuration of the the existing resources. Generally virtualized resources refer to the computing power and data storage ability[1].

The resources in virtualization are separated from the underlying physical hardware to provide services for the computing environment. It is considered to be the process of creating virtual versions, including hardware platforms, operating systems, storage devices, and other computing resources. In Linux, virtualization can create multiple Linux operating systems on a single host. This is called Linux virtualization. Unlike cloud computing, whether you are in a cloud environment or not, you can first virtualize your servers and then migrate to cloud computing to increase agility and enhance self-service. In this section, two methods are proposed and compared for virtualization technology: hypervisor-based virtualization and container-based virtualization.

### 2.3.1 Hypervisor-based Virtualization

Hypervisor, also known as Virtual Machine Monitor (VMM), is an intermediate software layer running between the physical servers and the operating systems. It allows multiple operating systems and applications to share a set of basic physical hardware. Therefore, it can also be viewed as a meta operating system in a virtual environment. It can coordinate access to all physical devices and virtual machines on the server. The hypervisor is the core of all virtualization technologies. The ability to support non-disruptive migration of multiple workloads is a basic function of the hypervisor. When the server starts and executes the hypervisor, it allocates the appropriate amount of memory, CPU, network, and disk to each virtual machine and loads the guest operating systems of all virtual machines[2][3].

In 1974, Popek and Goldberg defined the basic requirements of Classical Virtualization. They believe that a real VMM must meet at least three aspects of standards[19].

- *Equivalent execution*: in addition to the availability of resources and the

[1]https://en.wikipedia.org/wiki/Virtualization

[2]https://en.wikipedia.org/wiki/Hypervisor

[3]https://www.pluralsight.com/blog/it-ops/what-is-hypervisor

time difference, the execution of the program in the virtual environment and the real environment is exactly the same.

- *Performance*: most instructions in the instruction set must be able to run directly on the CPU.
- *Safety*: VMM must be able to fully control system resources.

Although this set of conditions is based on simplistic assumptions, they still provide a convenient way to determine whether a computer architecture can effectively support virtualization, and also provide guidelines for designing virtualizable computer architectures.

Hypervisors are commonly classified into two types[17], as presented in Figure 2.3.1:

- *Type 1*: The first type of hypervisor is called native or bare metal, which is directly deployed on the host's hardware in order to control the hardware and also monitor the guest operating systems. The classical implementation of this kind of virtual machine architecture are Oracal VM, Microsoft Hyper-V, VMWare ESX and Xen.
- *Type2*: The second type of hypervisor is called Hosted hypervisor which runs inside a traditional operating system as a distinct software layer. Comparing with type 1, the guest operating system becomes a third software level above the hardware. The common examples of a hosted hypervisor are Oracle VM VirtualBox, VMWare Server and Workstation, Microsoft Virtual PC, and KVM.

**Features of Hypervisor**

According to [10], four features of hypervisor are concluded from some analysis.

1. **Transparency**: transparency means that software can be executed directly in a virtual machine environment without modification. Software features like hardware resource sharing, isolation environment, real-time migration and portability remain the same.

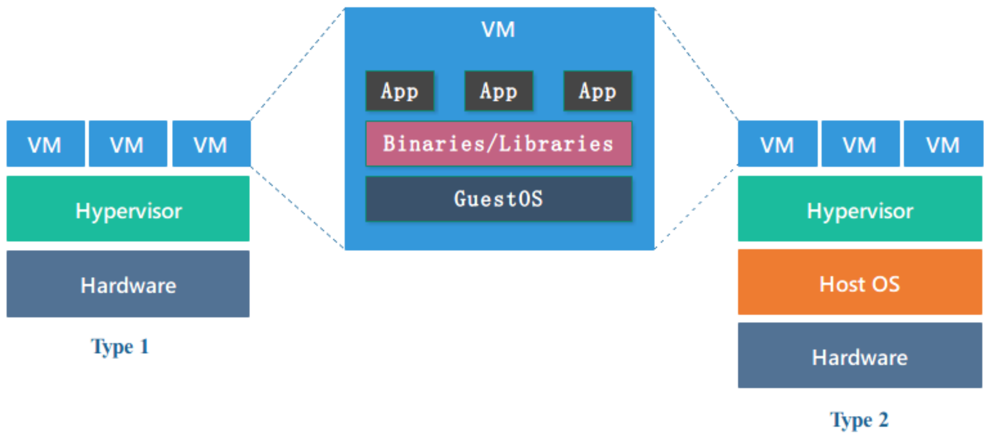


Figure 2.1: Hypervisor-based virtualization architecture

2. **Isolation**: in a physical machine, multiple instances of virtual machine can be hosted to share the hardware resources. In the mean while, all the instances are isolated with each other. In other words, the operations inside one specific virtual machine will not have any influence on others. They have separated runtime environment and different version of softwares.

3. **Encapsulation**: operating systems and applications are encapsulated into Virtual Machines. Real hardware is packaged into standardized virtual hardware, the entire Virtual Machine is documented as an ordinary file of the host machine. In this way, the installation and backing up of Virtual Machines are as easy as copying files from one place to another which increase the flexibility and security of software.

4. **Manageability**: for virtual machine operations, such as boot, shutdown, sleep and even add, modify, or remove virtual hardware, there all have programming interface. Via programming interface, virtual machine can be completely controlled by program.

### 2.3.2 Container-based Virtualization

Containers could be described as a lightweight virtualization approach that creates virtual environment at a software level inside the host machine, also known as operating system-level virtualization [22]. It removes the overhead of using hypervisors by creating virtual machines in the form of containers (act as guest systems) thereby sharing the resources of the underlying host operating system. It provides different levels of abstraction in which a kernel is shared between containers and more than one process can run inside each container. In this way, the whole system becomes more resource-efficient as there is no additional layer of hypervisor, and a full operating system which can occupy a lot of storage space for each Virtual Machine [12][16].

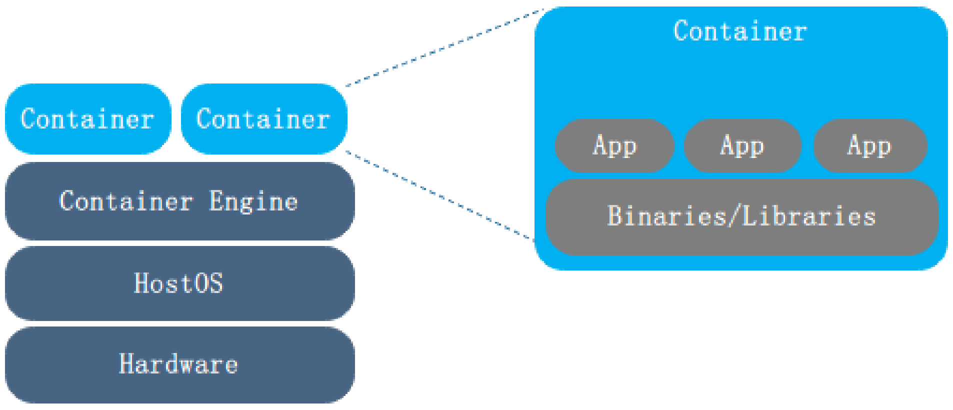


Figure 2.2: Container-based virtualization architecture

As we can see from Figure , containers are an abstraction at the application layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly[4].

The main difference between a container and a virtual machine is the location of the virtualization layer and how operating system resources are used [13][8]. Containers and virtual machines have a similar mission: segregating applications and their dependencies to build a set of self-contained units that can run anywhere. In addition, containers and virtual machines are also freed from the need for physical hardware, allowing us to use computing resources more efficiently, thereby improving energy efficiency and cost-effectiveness.

[4]https://www.docker.com/what-container#/virtual_machines

The virtual machine packages the virtual hardware, the kernel (that is, the operating system), and the user space in a new virtual machine. The virtual machine needs to first virtualize a physical environment, then build a complete operating system, build a layer of runtime, and then use the application to run.

For the container environment, there is no need to install a quest operating system, and the container layer (such as LXC or libcontainer) is directly installed on the host operating system (usually Linux variants). After the container layer is installed, the container instance can be allocated from the available computing resources of the system, and the enterprise application can be deployed in the container. However, each containerized application shares the same operating system (a single host operating system). The container can be viewed as a virtual machine that is installed with a set of specific applications. It directly utilizes the host's kernel. The abstraction layer is less than the virtual machine, it is more lightweight, and the startup speed is very fast.

Containers are more resource efficient than virtual machines because they do not require a separate operating system for each application. This means that a single operating system can carry more containers than virtual machines. Cloud providers are keen on container technology because more containers can be deployed on the same hardware device. In addition, containers are easy to migrate, but they can only be migrated to other servers with compatible operating system kernels, which will place restrictions on migration options.

Because containers do not pack the kernel or virtual hardware as if they were virtual machines, each container has its own isolated user space, allowing multiple sets of containers to run on the same host system. We can see that all operating system-level architectures can be shared across containers. The only ones that need to be built independently are binary files and libraries. Because of this, the container has extremely excellent lightweight properties.

### 2.3.3 The pros and cons of Container-based virtulization

The advantages of Container-based virtualization is listed as follows[1]:

1. Agile environment: the biggest advantage of container technology is that creating a container instance is much faster than creating a virtual machine. Container lightweight scripts can reduce overhead in terms of performance

and size.

2. Improve productivity: containers increase developer productivity by removing cross-service dependencies and conflicts. Each container can be seen as a different microservice, so it can be upgraded independently without worrying about synchronization.

3. Version Control: each container's image is version controlled so that different versions of the container can be tracked, differences between versions can be monitored.

4. the operating environment is portable: The container encapsulates all the relevant details necessary to run the application such as application dependencies and operating systems. This makes the migration of images from one environment to another more flexible. For example, the same image can be run in either Windows or Linux or a testing environment.

5. standardization: Most containers are based on open standards and can run on all major Linux distributions, Microsoft platforms, and more.

6. security: The processes between containers are isolated from each other, and the infrastructure is the same. In this way, the upgrade or change of one of the containers will not affect other containers.

The disadvantage of Container-based virtulization is listed as follows[5][6]:

1. Increased complexity: as the number of containers and applications increases, so does the complexity. Managing such a large number of containers in a production environment is a challenging task, and tools such as Kubernetes and Docker Swarm can be used to manage a certain number of containers.

2. Native Linux support: most container technologies, such as Docker, are based on the Linux container (LXC). Running the container in the Microsoft environment is a bit clumsy compared to running the container in native Linux, and daily usage can also be complicated.

[5]https://www.containerhomeplans.org/2017/11/advantages-of-building-with-shipping-containers/

[6]http://www.aadhan.org/blog/2016/7/2/pro-and-cons-container-architecture

3. Immature: container technology is a relatively new technology in the market and takes time to adapt to the market. The resources available to developers are limited. If a developer is caught in a problem, it may take some time to solve the problem.

## 2.4 Docker Containers

Docker[7] is an open source application container engine that allows developers to package their applications and dependencies into portable containers and then publish them to any popular Linux machine. The containers are completely sandboxed and there is no connection interface between them. Docker-based sandbox environments allow for lightweight isolation where multiple containers do not affect each other. Docker can automatically package and deploy any application, easily create a lightweight private PaaS cloud, and can also be used to build development and test environments and deploy scalable web applications.

The idea of the Docker is to provide the operating environment like it were sea freight. If we compare the operating system with the freighter, then each software is like a container. The contents of the container can be defined by the user or can be made by a professional. In this way, delivering a piece of software is to deliver a standardized set of components. The user only needs to select the suitable container combination and privatize it[8].

Docker is designed and developed using Go language. It is a container engine designed on LXC top layer that is open sourced by dotCloud[9]. Linux Container[10] (LXC) is a kernel virtualization technology that provides lightweight virtualization to isolate processes and resources. At the operating system level, the Linux container enables a single controllable host node to support multiple independent server containers at the same time. It is implemented based on two important kernel features: a control group (Cgroup[11]) and a namespaces[12] to provide isola-

[7]https://www.docker.com/what-docker

[8]https://www.linux.com/news/docker-shipping-container-linux-code

[9]http://dotcloud.co.za/

[10]https://linuxcontainers.org/

[11]`https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01`

[12]https://lwn.net/Articles/531114/

tion within the container and between the host system and the container. In the following, more details about the control groups and namespaces are explained.

### 2.4.1 Control groups

The control group (referred to as Cgroups) is a mechanism provided by the Linux kernel. This mechanism can integrate or separate a series of system tasks and their subtasks into different groups according to the specific behavior according to the resource level, thus providing a unified framework[13] of system resource management. Cgroups can limit, record and isolate the physical resources (including: CPU, memory, IO) used by the process group, providing basic guarantees for container virtualization. Therefore, it is the foundation for building a series of virtualization management tools such as Docker.

The main purpose of implementing Cgroups is to provide a unified interface for resource management at different user levels, from resource control of a single process to operating system-level virtualization. Cgroups gives the following four functions[14].

- *Resource Limitation*: Cgroups can limit the total amount of resources used by process groups. If you set a memory limit for the application, an OOM (out of memory) will be issued when the quota is exceeded.
- *Prioritization*: by assigning the number of CPU time slices and disk IO bandwidth, it is actually equivalent to controlling the priority of the running process.
- *Accounting*: Cgroups can count system resource usage, such as CPU usage, memory usage, and so on. This feature is very suitable for accounting.
- *Control*: Cgroups can perform operations such as suspend and resume on process groups.

The implementation of cgroups is to hook the system process. When a task involves the process of running a task, it triggers the subsystem connected to the hook to perform the detection. Finally, use the appropriate technology based on resource type to limit resources and assign priorities.

[13]https://www.kernel.org/doc/Documentation/cgroup-v2.txt
[14]http://en.wikipedia.org/wiki/Cgroups

### 2.4.2 Namespaces

One of the main goals of the Linux kernel implementation namespace is to implement lightweight virtualization (container) services. Processes in the same namespace can sense each other's changes and have no knowledge of external processes. In this way, the process in the container can create illusions as if it were in a separate system environment to achieve independence and isolation. Currently, the Linux kernel provides six namespaces, described as follows:

- Mount namespaces (mnt): similar to chroot, puts a process into a specific directory. The Mnt namespace allows different namespace processes to see different file structures so that the file directories seen by processes in each namespace are isolated.
- UTS namespaces (uts): the UNIX Time Sharing System (UTS) namespace allows each container to have its own host name and domain name. It can be seen on the network as a separate node, not as a process on the host.
- PID namespaces (pid): the processes of different users are isolated by the pid namespace, and different namespaces can have the same PID.
- IPC namespaces (ipc): process interactions in containers still use Linux's common interprocess communication (IPC) methods, including common semaphores, message queues, and shared memory. However, unlike virtual machines, the interprocess interaction of the container is actually the interaction between processes in the same pid namespace on the host. Therefore, you need to add namespace information when applying for IPC resources.
- User namespaces (user): each container can have a different user and group ID, which means that users inside the container can execute the program inside the container instead of executing the user on the host.
- Network namespaces (net): using the pid namespace, the pids in each namespace can be isolated from one another, but the network ports are still shared host ports. Network isolation is achieved through the network namespace.

### 2.4.3 Features of Linux Containers

According to the description of [12], the four characteristics of the container can be summarized as follows:

1. *Portability*: linux container can run in any environment without changing the operating system's functionality. Applications running in the same container can be deployed together in various environments by being bundled together.
2. *Fast application delivery*: unlike virtual machines, containers are very fast to build as a lightweight virtualization engine and take a few seconds to build a new virtualization container. Because containers can be used in a variety of different environments, the need for infrastructure is no longer relevant to the application environment. This means less time for configuring the environment and debugging environment-specific issues.
3. *Scalability*: on any supported cloud platform, Linux containers can be expanded and scale down quickly. It allows applications deployed on containers to also be scaled as needed
4. *Higher density workloads*: with Linux containers, you can run a large number of applications on a single host system. Because the container does not use a complete operating system, resources are effectively used between different applications compared to the hypervisor.

### 2.4.4 Docker architecture and components

Docker uses server and client mode. The Docker client creates, runs, or deploys a Docker container by interacting with the Docker daemon. Users can install Docker clients and Docker daemons on the same system or on different systems. The Docker client communicates with the Docker daemon through ports or RESTful APIs. The Docker architecture overview is shown in Figure 2.4.4. In the architecture diagram, docker consists of the following components:

- *Docker daemon*: Docker daemon runs on the host, and end users cannot directly interact with the daemon. They can only communicate with the Docker Daemon through the Docker client.

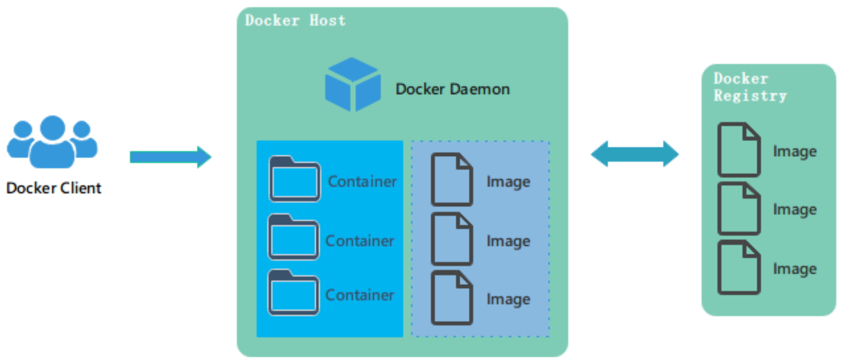


Figure 2.3: Docker Architecture

- *Docker client:* Docker client is a command line tool for users to communicate with the Docker daemon. It accepts commands from the user and then interacts with the daemon. The Docker client communicates with the Docker daemon and returns the results to the user. In addition, it can access remote Docker daemons through sockets or RESTful APIs.

- *Docker images:* Docker images is a read-only template for creating Docker containers. Images can contain Linux operating systems, Apache or Web applications. Users can download Docker images that have been created, or create Docker images for other users. Each image consists of multiple layers. Docker binds these layers in the image via Union File Systems. Each image is based on a preliminary image, and then a new layer is added to these main images by operating instructions. Operational instructions can be running commands, adding files or directories, or creating an available operating environment. These instructions are stored in the Dockerfile.

- *Docker registry:* docker registry is used to save Docker images. It is also divided into public and private use. The public Docker registry is Docker Hub. Users can also create private docker registries and provide docker images for other users.

- *Container:* similar to the directory that holds the VMware virtual machine configuration file, it provides everything the application runs. Docker containers can be run, started, stopped or deleted. Each container is an independent security application platform.

## 2.5 Container Management System

Docker container cluster refers to the deployment of a large number of Docker application containers on the cloud platform. You can deploy the same or different application instances on different Docker application containers. However, messaging middleware, various developments, back-end databases, and administrative toolsets have also been deployed. The large number of dock application containers described above constitutes a set of application containers. On the basic cloud platform, centralized and unified management of these application containers is necessary. The platform realizes efficient and reasonable dynamic scheduling of resources and aims to provide users with highly available and convenient network services. This requires the development of an efficient container cluster management system.

There are some widely used container management system based on Docker, such as docker swarm and Kubernetes. In what follows, their main concepts, features and architectures will be presented in order to decide which one is more suitable for the development of our framework.

### 2.5.1 Docker Swarm

Swarm[15] is a native clustering tool developed by Docker. Swarm uses the standard Docker API, which means that Docker client commands can be used to start the container. Swarm automatically selects the appropriate host to run the container.

Each host in the Docker Swarm platform runs the Swarm agent and the host also runs the Swarm Manager (it can also run agents in the test cluster). The manager is responsible for container orchestration and scheduling on the host. Swarm can be run in high availability mode (any etcd, Consul or ZooKeeper can be used to fail over to the standby manager). When new hosts join the cluster,

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

service discovery in Swarm has several different ways to find newly added hosts. Typically, tokens are used to store a list of host addresses on Docker Hub.

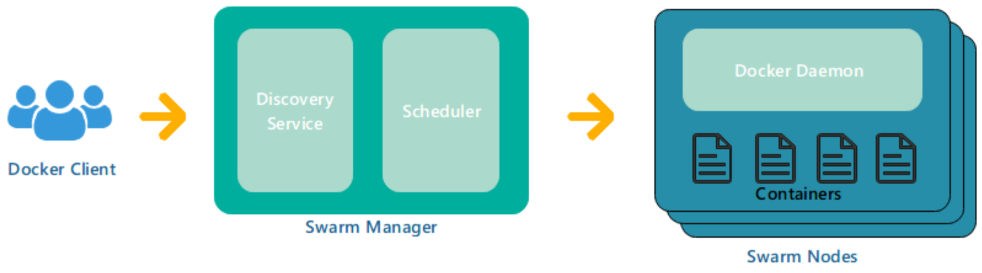


Figure 2.4: Docker Swarm Architecture

The main features of Docker Swarm include:

- Use filtering to start a set of containers for a specific Docker image on the selected node;
- Scale the number of containers based on load conditions;
- Containers health check;
- Perform rolling update of software across containers.

Docker Swarm clusters also provide administrators and developers with the ability to add or subtract iterations as computing needs change. Third-party tools such as Consul, Zookeeper, or etch are needed to ensure high availability and failover to the secondary Swarm Manager.

Swarm has five filters for scheduling containers:

- *Constraint*: which is also known as node labels. Constraints are key-value pairs related to particular nodes, and a user can select a subset of nodes when building a container and specify one or multiple key-value pairs.
- *Affinity*: this filter tells one container to run next to another, so as to make sure that containers run on the same node, which is based on an identifier image or label.

- *Port*: ports represent unique resource with this filter. When a container tries to use an already occupied port, it will move to the next node in the cluster.
- Dependency: when containers depend on each other, this filter schedules them on the same node.
- *Health*: under the circumstances that a node is not working properly, this filter will stop scheduling containers on it.

Docker Swarm also follows the principle of exchange, plug and play. As an initial development, it provides a standard Docker API that supports a pluggable backend that can be easily exchanged with your favorite backend. The discovery service of the manager node is used to provide these backends. This service maintains a list of IP addresses in the cluster assigned to each backend service

### 2.5.2 Google Kubernetes

Kubernetes[16] is an open source cluster manager, which is used for containerized applications across a bunch of physical or virtual hosts, and it also provides automatic deployment, scaling, and maintenance of applications. With kubernetes, applications can be deployed effectively; you can also scale those applications during runtime, roll out new features, and optimize hardware with application specific resources. Thus, Kubernetes gives the benefit to quickly respond to user's requirements and maintain user's the desired state.

Moreover, as considered as a fault-tolerant system, Kubernetes has the ability to maintain the proper execution of the tasks. Fault-tolerance property guarantees that the system proceeding to execute its intended operation instead of failing completely. A fault-tolerant computing system is defined as "system which has the built-in capability to preserve the continued correct execution of its programs and I/O functions in the presence of a certain set of operational faults". This definition indicates the behavior of Kubernetes in practical scenarios.

Kubernetes executes user specific containers by instructing a cluster to run a set of containers. These are executed in the form of pods on particular hosts that

[16]https://kubernetes.io/

are automatically selected by the system. The system then takes into consideration the individual and collective resource requirements, deadlines, workload-specific requirements, hardware/software constraints, and other constraints for the purpose of completing the desired work effectively. Furthermore, Kubernetes follows the idea of persistent storage to keep large volumes of data, which are generated in the discrete applications deployments as containers. There are different methods to store long-lived data that is required by the applications.

**Kubernetes Architecture**

There are several services and components that need to be installed on top of a cloud- based or physical host, so as to configure Kubernetes for managing container. As can be seen from Figure 4.3, the components are differentiated based on the master components such as controller manager, APIs, scheduler and kubelet on master node, and node agents such as kubelet and proxy on worker nodes. This provides an opportunity to fully utilize the capabilities of cluster computing by running containers on multiple worker nodes that are distributed and managed by the master node. Moreover, you can configure a node cluster by installing all the Kubernetes components on a single host that can be master nodes and worker nodes.

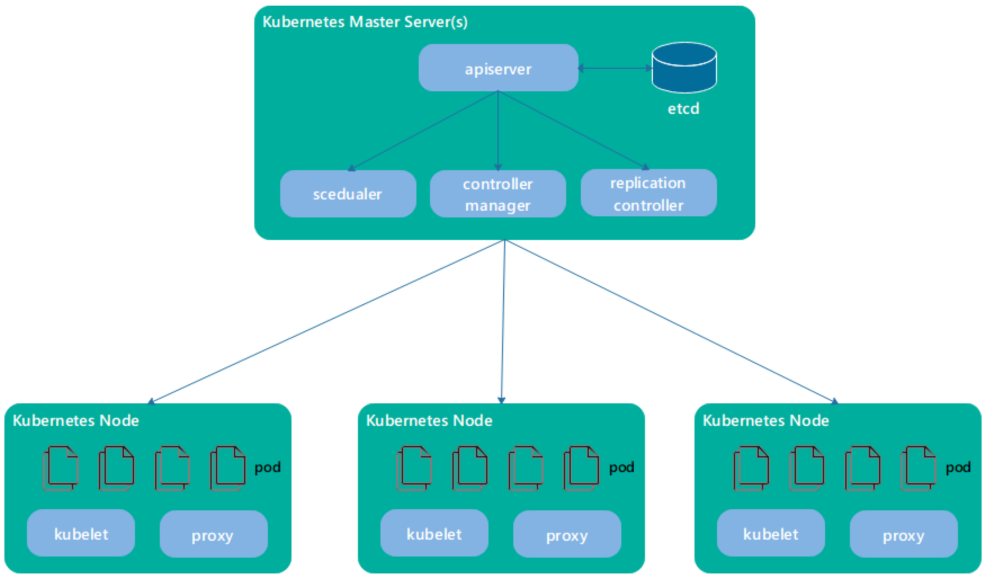


Figure 2.5: Kubernetes Architecture

The master node server is the control unit of the Kubernetes cluster and serves as the primary contact for system administrators and users. It deploys at work nodes, manages and monitors containerized applications. This is achieved through several dedicated components, as described below:

- *API server*: Kubernetes API server is a key component of the entire cluster. It allows users to configure organizational units and workloads to communicate with other nodes in the cluster. The server also validates and configures data by implementing a RESTful API that includes API objects such as pods, replication controllers, services, and so on. Use a separate client named kuberctl and other tools on the primary side to connect users to the cluster. This allows the user to control the entire cluster.
- *Controller manager*: the Controller Manager service is a daemon that processes replication tasks defined by the API server. This service is built using a control loop whose purpose is to monitor changes in the operating state that are stored in parallel in operation. Whenever a change occurs, Control Manager reads the new information and moves the current state to the desired state. Some examples of controllers are replication controllers, namespace controllers, endpoint controllers, and service account controllers.
- *Scheduler*: the scheduler actually assigns the workload to specific work nodes in the cluster. This is mainly used to analyze the working environment, read the operational requirements, and then place the workload on acceptable nodes. It is also responsible for tracking resource utilization on different nodes and tracking the total hardware resources allocated to the process or can be used.
- *etcd*: Etcd is a distributed, reliable, and consistent key-value data store for shared configuration and service discovery. Etcd is considered to be one of the core components of the Kubernetes cluster. Etch provides simple, secure, fast, and reliable storage space to store configuration data that the cluster can use. It follows the principles of the clotting consistency algorithm as a data storage operation.

The following list provides some other common terms related to Kubernetes:

- Pods: Pod is the basic operating unit of Kubernetes. It forms a Pod with one or more related containers. Pod containers usually run the same application. Pod contains containers that run on the same node (host) as a unified management unit that shares the same volume and network namespace/IP and port space.

- Deployments: these building blocks can be used to create and manage a group of pods. Deployment can be used with the service layer to scale horizontally or ensure availability.
- Services: the service is also the basic operating unit of Kubernetes. They are abstractions of real application services. Each service has many corresponding containers. Through proxy ports and service selectors, service requests are passed to the backend service container. The external behavior is that a single access interface does not require external knowledge about the way the backend operates. This is great for extending or maintaining the backend
- Labels: Label is a key/value key pair used to distinguish Pod, service, and replication controllers. Pod, Service, and Replication Controller can have multiple labels, but the keyword for each label can only correspond to one value. Labels are the basis for service and replication controller operations. In order to forward a request to access a service to multiple containers that provide services at the backend, identifying the container's label identifies the correct container. Similarly, the replication controller also uses labels to manage a set of containers created through the pod template so that the replication controller can more easily and conveniently manage multiple containers regardless of the number of containers.

**Comparation between Kubernetes and Docker Swarm**[17]

| | Kubernetes | Docker Swarm |
|---|---|---|
| Application Definition | Applications can be deployed using a combination of pods, deployments and services (or "microservices"). A pod is a group of containers that ar co-located and is the deployed atomic unit. Deployments can have replicas across multiple nodes. The service is the "external face" of the container workload and is integrated with the DNS to cycle through the requests. | Applications can be deployed as a service (or "microserver") in a Swarm cluster. Multi-container applications can be specified using YAML files. Docker Compose can deploy applications. Tasks (instances of services running on nodes) can be distributed across data centers using labels. For example, you can use multiple layout preferences to further distribute tasks to racks in the data center. |
| Application Scalability Constructs | Each application tier is defined as a pod and can be scaled when managed by the deployment specified in YAML. Scaling can be manual or automatic. Pod can be used to run vertically integrated application stacks, such as LAMP (Apache, MySQL, PHP) or ELK/Elastic (Elasticsearch, Logstash, Kibana), co-locate and co-administered applications such as content management systems and application for backups, checkpoints, compression, rotation, snapshots. | Services can be scaled using the Docker Compose YAML template. Services can be global or replicated. The global service runs on all nodes and the replication service runs a copy of the service (task) across the nodes. For example, a MySQL service with 3 replicas can run up to 3 nodes. Tasks can be scaled up or down, and can be deployed in parallel or in sequence. |

[17]https://platform9.com/blog/kubernetes-docker-swarm-compared/

| | | |
|---|---|---|
| High Availability | Deployment allows nodes to be distributed among nodes to provide HA so that they can tolerate application failures. The load balancing service detects unhealthy pods and removes them. High availability of Kubernetes is supported. Multiple master and worker nodes can be load balanced for requests from kubectl and clients. Etcd can be clustered and API servers can be replicated. | Services can be replicated between Swarm nodes. The Swarm manager is responsible for the entire cluster and managing the resources of the worker nodes. Managers use ingress load balancing to expose services to the outside world. Swarm managers use the Raft Consensus algorithm to ensure that they have consistent status information. It is recommended to use an odd number of managers, and most managers must be able to use Swarm clusters (2 of 3, 3 of 5, etc.). |
| Load Balancing | The pods are exposed through a service that can act as a load balancer in the cluster. In general, an ingress used for load balancing. | Swarm mode has a DNS component that can be used to assign incoming requests to service names. Services can be run on a user-specified port, or they can be assigned automatically. |
| Auto-scaling for the Application | Auto-scaling with a simple pod target is defined as declarative definition using deployment. The CPU utilization target for each container is available. Other goals are on the roadmap. | Not directly available. For each service, you can specify the number of tasks to run. When you zoom in or out manually, Swarm managers adjust automatically by adding or removing tasks. |

| | | |
|---|---|---|
| Rolling Application Upgrades and Rollback | The deployment controller supports "rolling-update" and "recreate" strategies. Rolling updates can specify the maximum number of pods that cannot be used or run in this process. | At rollout time, you can apply rolling updates to services. Swarm Manager allows you to control the delay between the deployments of services to different node sets, updating only one task at a time. |
| Health Checks | There are two kinds of health check: liveness (is application responsive) and readiness (is application responsive, but busy preparing and not yet able to provide services) Out-out-the-box K8S provides a basic logging mechanism to pull aggregate logs for a group of containers that make up the pod. | Docker Swarm health checks are limited to services. If the container that supports the service is not running (running state), a new container is started. Users can use the HEALTHCHECK instruction to embed health check functionality into the Docker images. |

| | | |
|---|---|---|
| Storage | Two storage APIs: Two storage APIs: The first provides an abstraction of the personal storage backend (eg NFS, AWS EBS, ceph, flocker). The second abstraction that provides storage resource requests (for example, 8Gb) can be fullfilled with different storage backends.<br>Modifying storage resources used by the Docker daemon on a cluster node requires temporarily removing the node from the cluster.<br>Kubernetes provides several types of persistent volumes that support blocks or files. Examples include iSCSI, NFS, FC, Amazon Web Services, Google Cloud Platform, and Microsoft Azure.<br>The emptyDir volume is non-persistent and can be used to read and write files using containers. | Docker Engine and Swarm support mounting volumes into containers. Shared file systems (including NFS, iSCSI, and Fibre Channel) can be configured nodes. Plugin options include Azure, Google Cloud Platform, NetApp, Dell EMC, etc. |

| | | |
|---|---|---|
| Networking | The networking model is a flat network, enabling all pods to communicate with one another. Network policies specify how pods communicate with each other. The flat network is typically implemented as an overlay. The model requires two CIDRs: one from which pods get an IP address, the other for services. | Node joining a Docker Swarm cluster creates an overlay network for services that span all of the hosts in the Swarm and a host only Docker bridge network for containers. By default, nodes in the Swarm cluster encrypt overlay control and management traffic between themselves. Users can choose to encrypt container data traffic when creating an overlay network by themselves. |
| Service Discovery | Services can be found using environment variables or DNS.<br>DNS Server is available as an addon. For each Kubernetes Service, the DNS Server creates a set of DNS records. If DNS is enabled in the entire cluster, pods will be able to use Service names that automatically resolve. | Swarm Manager node assigns each service a unique DNS name and load balances running containers. Requests to services are load balanced to the individual containers via the DNS server embedded in the Swarm.<br>Docker Swarm comes with multiple discovery backends:<br>Docker Hub as a hosted discovery service is intended to be used for dev/test. Not recommended for production. A static file or list of nodes can be used as a discovery backend. The file must be stored on a host that is accessible from the Swarm Manager. You can also provide a node list as an option when you start Swarm. |

| | | |
|---|---|---|
| Performance and scalability | With the release of 1.6, Kubernetes scales to 5,000-node clusters. Kubernetes scalability is benchmarked against the following Service Level Objectives (SLOs): API responsiveness: 99 of all API calls return in less than 1s. Pod startup time: 99 of pods and their containers (with pre-pulled images) start within 5s. | According to the Docker's blog post on scaling Swarm clusters, Docker Swarm has been scaled and performance tested up to 30,000 containers and 1,000 nodes with 1 Swarm manager. |

Table 2.1: compares the features of Kubernetes with Docker Swarm

## 2.6 Task Migration Methods

In this section, in order to observe better execution performance in a cloud environment, some intelligent decision making methods for task migration are proposed. There have been many solutions about decision making for task migration in cloud computing, especially for mobile cloud computing. They are all for augmenting devices and making applications better executed, but they are different from each other in decision objectives and decision algorithm. A significant difference is that a migration decision can be static or dynamic. Now in the following is some approaches with the two different decision making ways.

**Steps of task migrations:**

1. Looking for available resources and services which can be offloaded by devices
2. Monitoring environment context information, such as surrounding networks and relavent devices
3. Conducting application partions at a suitable granularity and using appropriate approaches for task migration decision making

4. Remote execution control for applications needs to be conducted, including getting users' input and return results to the user-end

The migration decisions can be separated into two dimensions. *static approach:* migration decisions are made before an application's execution, usually once *dynamic approach:* migration decision are made in a dynamic way taking into account the runtime environment and application executions

**wishbone:** A system that uses operator dataflow graphs to generate the best partitioning. Wishbone uses analytics to determine how each operator in a dataflow diagram actually operates on sample data, without the need for cumbersome user annotations. Its partitioning algorithm simulates the problem as an integer linear program, minimizes the linear combination of network bandwidth and CPU load, and uses a program structure (ILP solver) to effectively solve the problem in practice. Partitioning methods only apply to applications that can be represented as data flow diagrams [2].

**parametric analysis**: This work presents a parametric analysis method to achieve optimal program partitioning and scheduling for computational offloading. The application is divided into tasks according to the task control flow diagrams. Then based on the optimal program partition corresponding to the current value of the runtime parameters, the program is converted into a distributed program that schedules its computational tasks to the mobile device or server at runtime. When making partitioning and migration decisions, there is a trade-off between computing workload and communication costs. It obtains program calculation workload and communication costs through cost analysis and represents them as a function of runtime parameters. Then the parameterized partitioning algorithm will find the optimal program partition corresponding to different runtime parameter ranges [21].

**Darwin:** It is an integrated automation framework. It supports source-target workload migration. The goal is to increase speed and reduce the costs and inherent risks of performing the migration. During workload migration, it first discovers the source environment, performs analysis to select migration target candidates, and then creates a workload migration map. Darwin eventually configures the target platform environment and application components for migration. Darwin, known as a framework, can architecturally support different variants, but there are still deficiencies, such as the lack of integration with open virtualization formats that migrate XML specification requests [21].

| | wishbone | parametric analysis | Darwin |
|---|---|---|---|
| objective | minimize a combination of network load and CPU consumption | minimize total cost of computing, communication and scheduling | enhance speed reduce inherent risk of migration |
| algorithm | operation research for data flow graph partition n | parametric analysis partition | none |
| granularity | operation | task | application |
| dynamic/static | static | dynamic | dynamic |
| decision factor | computing time and transmission time, CPU utilization | computation and communication time, data volumn | CPU utilization workload, resource information |
| central/distributed | central | distributed | central |

Table 2.2: task migration methods comparison

# Chapter 3

# DaaS Framework in Fog Computing

The goal of this thesis is to improve data management among data-intensive applications in Fog computing by proposing a framework which adopts the principles of Service Oriented Computing and hides the underlying complexity of an infrastructure made of heterogeneous devices. Through this DaaS framework, data-intensive applications are able to obtain the desired data with specific data requirements and data quality predefined by DIA developers. In this chapter, the overview layout of the DaaS framework is presented.

## 3.1 Data intensive applications in Fog Computing

Massive amounts of data are being generated at ever-increasing rates in various fields: social network, business, healthcare, the government domains, etc. Moreover, these data usually comes not only with a big volume but also in different formats. Some new computing schemas or methods should be taken into consideration for applications which are used to utilize these data, so as to derive business and scientific value from the data. Cloud computing with its promise of seemingly infinite computing resources is seen as the solution to this problem [11]. Based on the unlimited resources in cloud infrastructure, data-intensive applications are able to handle significant amounts of data and do some further processing on them. Furthermore, since storing and managing data in the cloud is typically inexpensive compared to the local physical machines, the cloud-based solution has a positive impact to reduce the cost.

However, relying on a cloud infrastructure for implementing data-intensive applications is not always feasible, for instance, when data are produced outside the cloud and need to be processed in the cloud. The data transmission from the edge of the network to the cloud is limited by the network bandwidth while increasing the latency. For some time-critical applications, the increasing latency may not fit the application requirements. Furthermore, due to privacy and security issues, sensitive data generated from the edge (IoT sensors, smart devices, laptops) can only remain near where they are produced, thus cannot be moved to the cloud for further processing.

On the other side, data are produced by a variety of devices with different characteristics (data format, data quality), thus an issue is raised for the cloud infrastructure to manage these data with different characteristics. It might be very challenging to apply the data processing solutions developed for cloud-based infrastructure directly to the edge.

In such a scenario, Fog Computing [9], often also referred to as Edge Computing, is an emerging paradigm aiming to extend Cloud Computing capabilities to fully exploit the potential of the edge of the network where traditional devices as well as new generations of smart devices, wearables, and mobile devices – the so-called Internet of Things (IoT) – are considered. Cloud and edge are connected to each other for transferring data as two distinct environments with its own specific characteristics. Data are usually generated by the deployed devices located on the edge and then move to the cloud for customized processing. The adoption of Fog Computing facilitates mutual cooperation between the cloud and edge devices, especially in IoT environments. In this thesis, we consider Fog Computing as the sum of Cloud and Edge Computing where these two paradigms seamlessly interoperate to provide a platform where both computation and data can be exchanged in both downstream and upstream direction [18]. A presentation of a Fog Computing environment is shown in Figure 3.1.

Through these definitions, Cloud Computing is responsible for providing compute, storage, and networking services in the center of the network whereas Edge

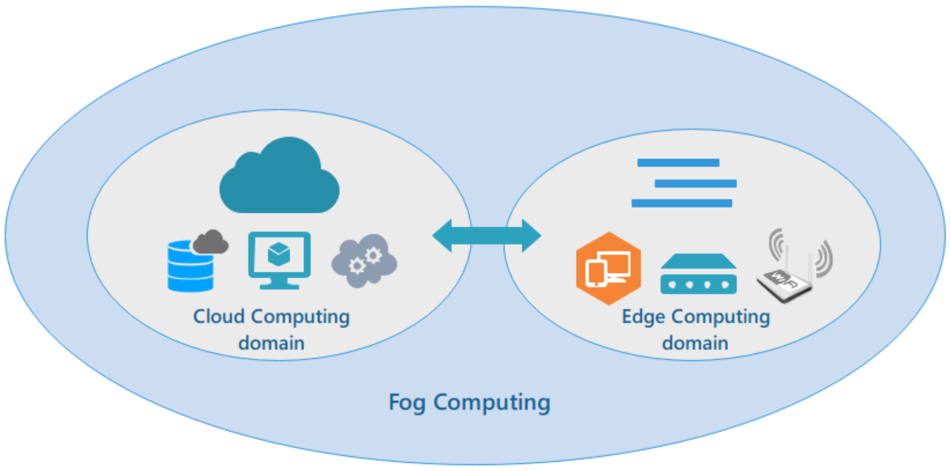


Figure 3.1: Fog Computing environment

Computing is responsible for collecting and preprocessing data before further utilization in the cloud at the edge of the network. As presented in Figure 3.1, Cloud Computing resources are constituted by several physical and virtual machines which are able to store and process data. On the other side, Edge Computing resources are composed by sensors, smart devices, laptops, and mobiles which are capable to generate vast amounts of data in different locations and offer these data to the owner of the resource. Fog Computing environment is generated by combining the capability of Cloud Computing and Edge Computing. In common case, the direction of data and computation movement is from the Edge Computing domain (where data are generated) to the Cloud Computing domain (where applications run).

Furthermore, data and computation movement between the cloud domain and the edge domain should follow constraints like limited bandwidth and strict latency requirements. When the computing power is not sufficient at the edge, data are moved to the cloud for a scalable processing solution. Similarly, if data cannot leave the edge for security and privacy issues, the computation is moved to the edge.

Based on the Fog Computing paradigm, this thesis proposes an architecture designed for developers of data-intensive applications to provide an appropriate mechanism which are able to decide where to move data and computation resources based on the data requirements set by DIA developers and privacy constraints.

## 3.2 Data-as-a-Service architecture

### 3.2.1 Overview of DaaS

As an extension of virtualization, Cloud Computing allows developers to focus on business aspects without getting distracted from hardware procurement, maintenance and capacity planning. Cloud Computing is considered as an emerging concept which provides resources and services through the Internet and these services are distributed in a cloud [25]. The influence of Cloud Computing is increasing and several different models and deployments have emerged to meet the specific needs of different users. Each type of cloud service provides different levels of control, flexibility, and management capabilities.

Since Cloud Computing is growing rapidly, it is further divided into three service models for utilizing the overall computing power [20].

- *Infrastructure-as-a-Service (IaaS)*: IaaS mainly includes IT infrastructure services such as computer servers, communication devices, storage devices, which can provide users with computing power, storage capacity, or network capabilities on demand. Users acquire these resources by purchasing IaaS services.
- *Platform-as-a-Service (PaaS)*:the second layer is PaaS, which is a service for software developers. The cloud computing platform provides hardware, OS, programming languages, development libraries, deployment tools in order to provide a proper software running environment that helps software developers to develop software services faster.
- *Software-as-a-Service (SaaS)*: the SaaS layer provides services for general users and provides a complete and available software system that allows general users to use application services deployed on the cloud through browsers or application clients, without having to pay attention to technical details.

Compared to the traditional categories of Cloud Computing, DaaS is a service module which provides specific data to different users with distributed locations.

*Data-as-a-Service (DaaS)* is a cloud strategy used to facilitate the accessibility of business-critical data in a well-timed, protected and affordable manner. Any data-related service, such as aggregation, data quality management, data cleaning, can take place in a centralized location, and then provide data to different systems and users regardless the geographical separation between data consumers and providers [24].

DaaS provides a new way to access business-critical data within an existing data center. The DaaS approach delivers the following benefits [23]:

- *Agility*: through the integration of different data access methods, customers can access data quickly without considering where the data is located. If the customer needs a slightly different data structure or calls data at a specific location, DaaS can meet the requirements very quickly with minimal changes.
- *Affordability*: the service provider builds the possible architecture to provide suitable data and the presentation layer can be outsourced to other organizations. In this way, it is more flexible when meeting any changes from the requirements.
- *Data quality*: one of the main benefits is to improve data quality. Data access is mainly controlled by the data service itself. Thus, there is only one updating entry. This adds a strong layer of security and improved data quality.

For the Data Intensive Application developer, there is the ability for them to harvest or capture new data from disparate sources, but issues like quality, consistency and access rights arise at the same time. DaaS removes these issues by transferring these hassless to a specialized DaaS provider that gives the agility to integrate quality data into the applications via a cloud service. Based on these characteristics, we adopt DaaS paradigm as a Service-Oriented Architecture in our framework.

In the thesis, we build a DaaS framework based on kubernetes for the DIA to provide integrated and secure data. Kubernetes[1] is an open source cluster

[1]https://kubernetes.io/

management platform which is able to deploy, scale, and maintain containerized applications across a number of physical or virtual hosts. While we are trying to find a flexible solution for improving data management capability in a mix architecture, including cloud data centers and heterogeneous devices at the edge of the network, kubernetes is chosen as the underlying technology for the framework that is going to be developed.

When developing DIA, the major problems are usually amount of data, complexity of data, the speed at which it is changing and the fact that data generated at one location often needs to be sent and processed somewhere else. There are many factors that may influence the design of a data system, including the skills and experience of the people involved, legacy system dependencies, the time scale for delivery, organization's tolerance of different kinds of risk and regulatory constraints. By introducing DaaS infrastructure, data processing is transparent for the end-users. The DIA developer requests data without considering where and how to extract it, everything will be handled by the designed system automatically.

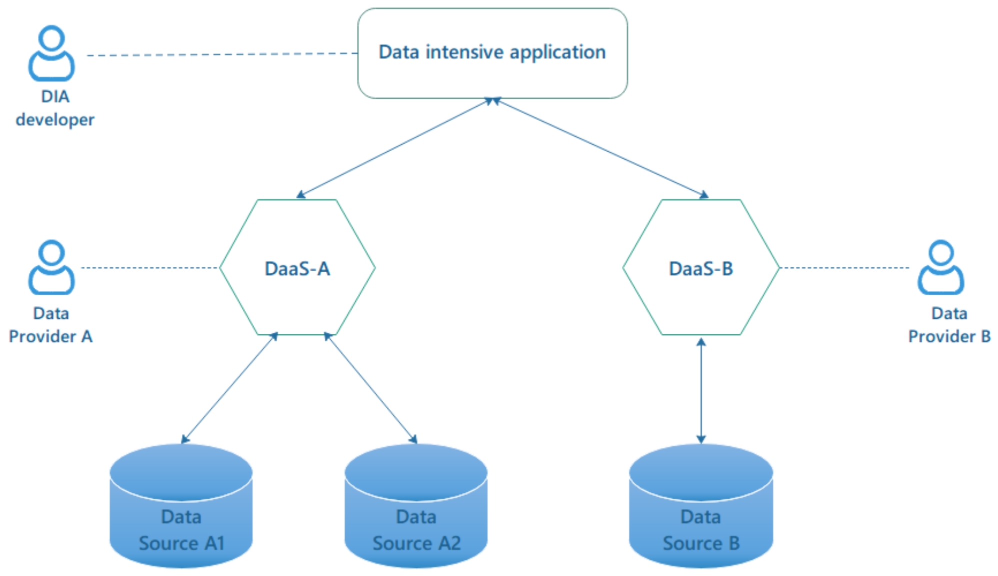


Figure 3.2: Data-as-a-Service architecture

Figure 3.2 provides a high-level overview of the Data-as-a-Service architecture from the perspective of its user. We can identify two types of users involved in the architecture: the data provider and the DIA developer. The responsibility of the data provider is to collect data from heterogeneous devices and make them available to application developers. DIA developers, also seen as data consumers, are able to use the collected data to develop their applications. As to the data sources, they can be deployed on the edge to store data coming from sensors or on the cloud to collect business transactions data.

DaaS is an abstract layer that provides specific data to different DIA developers and data consumers regardless where and how to extract them to fit the basic data requirements in a secure, privacy and good performance manner. The complex data processing happens inside the DaaS infrastructure, which allocates one dedicated and personalized access for each Data Intensive Application.

### 3.2.2 Data provisioning and data consuming

Data provisioning is supported by various of data providers, which are responsible for collecting, storing, and supporting access to data from the data sources they can control. For instance, in IoT scenarios, data are generated at the edge of the network (e.g., sensors, mobile devices) and the data provider has to setup an environment allowing the data consumers to properly access them. Since resources on the cloud are unlimited, moving the data to the cloud for efficient storage and processing is a good solution. However, while cloud resources bring reliability and scalability, network bandwidth has become a key factor restricting the transmission of data between the cloud and the edge domain. Network delay caused by limited bandwidth can negatively affect the processing time of data-intensive applications and the offered service quality, thereby losing the advantages by adopting resources on the cloud.

Data Intensive Applications can be applied in many different areas. For example, in the context of observing and researching the growth status of crops, the data provider plays the role of the farm data manager. Collect data on the color, height, and quantity of crops through sensors and cameras deployed in crops, and then move the data acquired from these smart devices to the cloud so that they can easily and widely accessible. In this scenario, the data provider is able to collect and provide data over which he has complete control.

For data collection in different infrastructures, cloud platforms offer compatible data interfaces to achieve interoperability. However, since agreement about protocols, data formats, and interfaces of smart devices, sensors, and smartphones has not been achieved yet, it is really challenging for data providers to handle these data generated by heterogeneous devices at the edge domain.

We assume that the data provider successfully obtains data from data sources in different regions (either cloud or edge). The next step is to upload the data to the cloud for further processing. Combining the ever-increasing computational and storage capabilities provided by the resources on the edge, part of the data processing can be kept on the edge. Thus, once the data is generated, it is not necessary to immediately move to the storage space on the cloud, they can stay near where they are generated and sent back to the cloud after some data preprocessing directly on the edge.

As a last step, data providers should be able to offer available data to various data consumers based on their customized requirements. Under this premise, principles of Service Oriented Computing are adopted to define the DaaS interfaces that data providers have to propose to allow the final users to properly consume the data with their own needs. The design of such interfaces should take into consideration two aspects:

1. *functional*: the functional requirement is describing how the data can be accessed by data users. There are usually some exposed DaaS interfaces (e.g., REST API) reserved.

2. *non-functional*: the non-functional requirement can be expressed by data quality dimensions (e.g., timeliness, accuracy) and service quality dimensions (e.g., response time, latency, data consistency) according to the expectations of the developers of Data Intensive Applications.

From the perspective of data consumption, data are accessible by invoking the available DaaS. In our approach, data consumers (Data Intensive Applications) need to obtain different kinds of data from heterogeneous data sources which are provided by several data providers. In this scenario, data consumers have to deal with several DaaS, each of them providing specific data with different quality of service. Data intensive applications are built on top of these DaaS with the goal of analyzing and processing provided data to create added value for the customer. To make it more concrete, in combination with the above-mentioned example of observing crop growth status, some new data dimensions (e.g., temperature, humidity, climate) given by other data providers have been adopted to analyze the correlation between crop growth status and external environment. The related analysis is handled by data-intensive applications that extract data through different DaaS .

### 3.2.3 Personalization of the DaaS

The data provider is responsible for exposing the collected data according to the rules of DaaS. During this process, it has to decide where to store data, in which format, how to satisfy the privacy constraints, and many other aspects. This situation becomes even more complex when dealing with a heterogeneous system, where different devices are involved in the data management. For instance, the data collected by different versions of smart devices, even from the same sensor, may not be uniform, which will lead to differences in data quality. This implies that the developers have to manage this heterogeneity as well as to properly distribute the data among edge and cloud, to make applications as much efficient as possible.

From the data consumer standpoint, the development of Data Intensive Application requires to select the appropriate set of DaaS to obtain the desired data based on the functional and non-functional requirements of the application. It ensures that the data obtained meets the agreed quality of data and service by interacting with the selected DaaS. All the aspects involved in the life-cycle of data consuming should be respected in order to help the DIA developers focus only on the business logic.

For this reason, we aim to offer tools for a smart data management trying to hide the complexity related to data retrieval, processing and storage. For this purpose, an abstract layer between data providers and data consumers is adopted. As we can see from Figure 3.3, Data Intensive Applications are not directly connected to the data sources containing the necessary data, but the access to these data sources is mediated by a specific component called Virtual Data Container (VDC).

The VDC is a concrete element that is used to interoperate with both the distributed data sources and the Data Intensive Application with its own requirements. Based on the specific data and service requirements, it provides

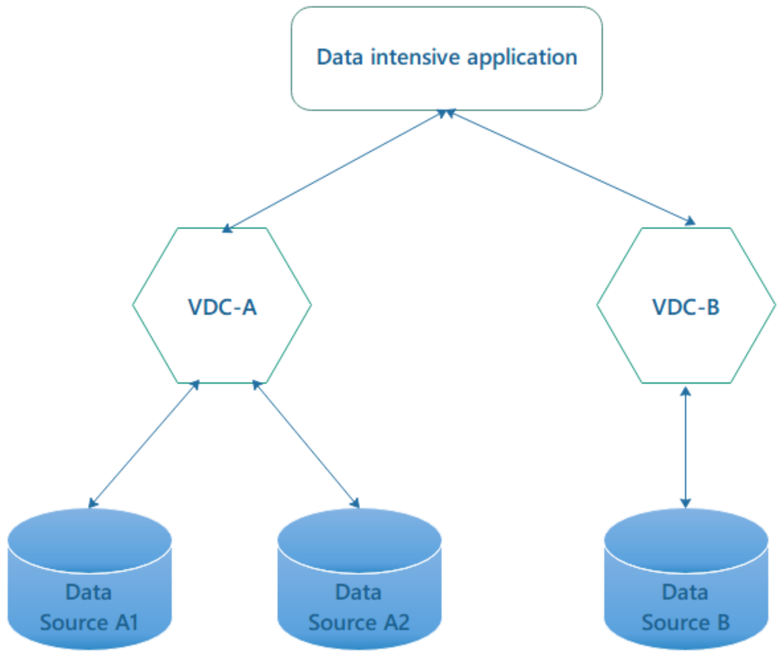


Figure 3.3: The role of VDC in DaaS architecture

personalization of the DaaS for each Data Intensive Application in order to provide the proper data and satisfy requirements by hiding the complex underlying infrastructure composed of different platforms, storage systems, and network capabilities. More characteristics and functionalities of VDC are described as follows.

- Since data sources can be distributed deployed either on the edge and or the cloud, VDC provides an uniform access to them. In this way, the workload of data providers is reduced.
- VDC supports some further processing (e.g., encryption, compression, integration) of the acquired data by embedding a set of data processing techniques.
- With the help of the node-RED programming modle , VDC allow composing these processing techniques in pipelines and executing the resulting application.
- Since data requirements and quality defined by the developers are varied on

different applications, it requires VDC to be easily deployed on resources which can live either on the edge or in the cloud.

Data Intensive Application life-cycle and data sources life-cycle are separated in the designed architecture, in order to manage the indirect access between Data Intensive Applications and data sources. Data Intensive Application life-cycle presents data retrieving activities occurring between a Data Intensive Application and a VDC, while data sources life-cycle defines the relationship between the data sources and a VDC. Focusing on data-intensive application life-cycle, a DIA developer selects the best fitting DaaS to retrieve required data with respect to the predefined functional and non-functional aspects of the data-intensive application. It implies that DIA developer only focus on the business logic of the application with the help by Virtual Data Container (VDC) which is responsible to provide the best fitting data to satisfy both the functional and non-functional requirements specified by the application designer.

In order to illustrate how a VDC instance is generated, a new role is involved in our architecture. A data administrator has a complete knowledge of one or more data sets and is responsible for making data available to applications that might be managed by other developers through a DaaS. As we can see from Figure 3.4, data administrator creates a VDC blueprint over the managed data sources which could be either located in the cloud or on the edge.

The VDC blueprint includes the information about the data sources and the methods exposed through the API. Also, a VDC blueprint describes functional (i.e., which data is needed) and non-functional (i.e., data accuracy, data privacy) aspects of the data source. All the designed VDC blueprints are stored inside a public repository which can be accessed by DIA developers. On the other side, DIA is not aware of the physical location of the data sources. Once VDC blueprints are ready in the repository, the DIA developer selects the most suitable VDC from the repository based on the functional and non-functional requirements of the under-design application. What should be notice here is that the developer could select different VDCs referring to different data. Till now, the VDC instance has been generated by the selected VDC blueprint.

Referring to the VDC deployment, it is necessary to know which are the possible resources on which the VDC can be executed. Combining with the concept of Fog Computing, both the provider and the consumer can provide

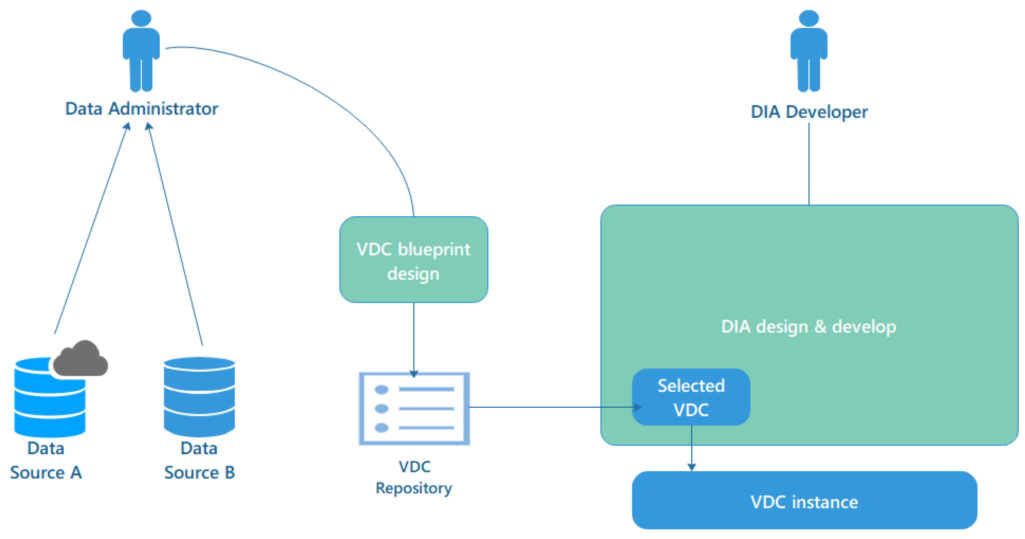


Figure 3.4: Generation flow of VDC instance

resources to realize DaaS. In common sense, we can assume that the consumer resources are always on the edge, while the consumer resources are always in the cloud. Thus, if we deploy VDC on the edge, the data and computation movement latency could be reduced since the processing procedure occurs locally. On the other side, a VDC living in the cloud has more capacity since it lives close to the data source to which it is connected. The location of the VDC deployment could be effected by several factors.

- *resource capability*: if there is no enough available resources at the edge for processing data, the VDC can only be deployed and executed in the cloud.
- *network characteristics*: the network transmission rate could have an effect on the application response time which could be a factor for selecting the location of the VDC.
- *privacy/security*: not all the data can be moved to the consumer side due to privacy/security issues, thus even the processing cannot be placed at the edge.

For each VDC, it dedicated to one user, provides the desired data by following the personalized data requirements. VDC personalize data extraction according to user request.

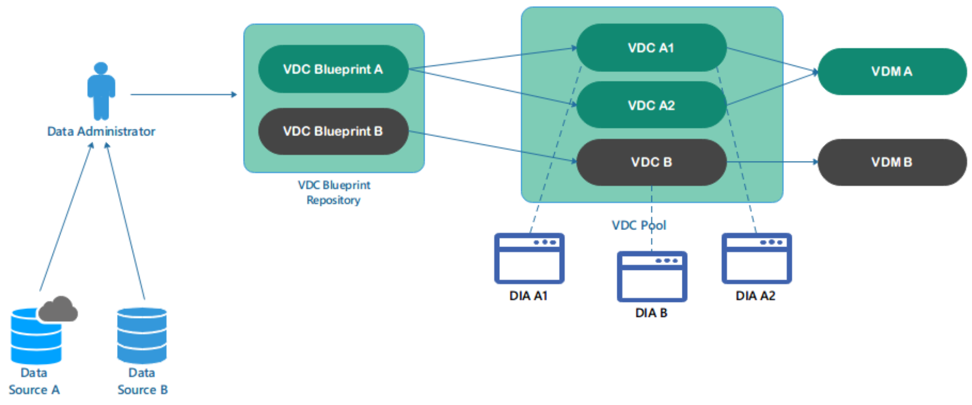


Figure 3.5: The role of VDM in DaaS architecture

### 3.2.4 Management of multiple access to data sources

Developers design and develop data-intensive applications according to the data collected from distributed data sources which could rely on the resources either in the cloud or on the edge of the network. After the data administrator defines and stores a VDC blueprint into the blueprint repository, the application developers can select one for their own applications. As a consequence, there could be several DIAs which operate with different VDCs. Moreover, the same VDC blueprint can be adopted in different applications, each of these applications include instances generated from the VDC blueprint. As shown in Figure 3.5, different applications (DIA A1 and DIA A2) pick their specific VDC (VDC A1 and VDC A2) from the same blueprint (blueprint A). Both DIA A1 and DIA A2 request data from the same data resource (data resource A), but with personalized data requirements and QoS constraints. In order to distinguish and manage these two different VDC instances, we introduce the concept of Virtual Data Manager (VDM).

Virtual Data Manager (VDM) is a module used to control the behavior of the different instances of the same VDC blueprint and ensure there is no conflict when enacting data and computation movements. Data-intensive applications deployed through the platform can access the required data with the help of the abstraction layer provided by the VDC. By detecting the state of the execution environment during the run time, if the current state does not satisfy the functional and non-functional requirements predefined by the developer, VDM is responsible for the relocation of the corresponding VDC. More details about the data and computation movement will be illustrated in the next section.

## 3.3 Data and computation movement in Fog Computing

In a IoT scenario, data are mainly generated from the edge of the network and then they are usually moved to the cloud for storage or deeper processing. Theoretically, the cloud has unlimited amount of resources and computing capability, it ensures high reliability and scalability. Due to the network latency, the advantage of the fast-processing in the cloud might be wasted and the service quality might be negatively effected.

Movement strategies provide solutions for moving data or computation in a Fog environment, taking into consideration all the factors which affect the application execution, data usability and trying to keep the QoS and the data quality at the levels required by the application designer.

Data and computation movement strategies are used to decide where, when, and how to save data - on the cloud or on the edge of the network - and where, when, and how to move tasks composing the application to create a synergy between traditional and cloud approaches able to find a good balance between reliability, security, sustainability, and cost. The driver for data and computation movement is the evaluation of the Data Utility [5]. Data utility is the relevance of data for the us-age context, where the context is defined in terms of the designer's goals and system characteristics. In this way, data utility considers both functional (i.e., which data is needed) and non-functional (e.g., the data accuracy, performance) aspects.

When an application is deployed through the platform, the application designer expresses application requirements about the QoS and quality of data used both to lead the data source selection and to select a proper computation and data location. In the application requirements both hard and soft constraints are expressed. When the evaluation of the data utility does not respect the application designer requests, the VDM will enact the most suitable data and computation movement strategies to balance the posed requirements such as reducing the latency or their size, ensuring a given accuracy, while maintaining — if requested — privacy and security. Data and computation movements are executed in order to satisfy all the hard-constraints and, as much as possible, soft-constraints and requirements expressed by the user with the final objective of executing the requested functionality having in mind also the maximization of the user experience. As computation movement requires a dynamic deployment of the data processing tasks, data movement could require only a transformation of the data format (e.g., compression or encryption) or could also affect the quality of the data (e.g., data aggregation).

Data and computation movement are managed over the entire life-cycle of the application, from its deployment to its dismissal. During this time, the application and its data sources are monitored and evaluated in order to satisfy the hard and soft requirements expressed by the application developer. As the decision of when, how, and where data and computation movement must occur depends on the current situation in which the data-intensive application operates, the execution environment includes a distributed monitoring system.

The management of data and computation movement is a life-cycle composed of the following steps:

- *Monitor*: a DIA is monitored through a set of indicators providing information about both the application behavior and the data source state.
- *Analyze*: The result of the previous phase is used to compare the current situation with the required data utility values. If the data utility provided to the application does not satisfy the application requirements, an exception is raised.
- *Execute*: once the strategies have been selected, they can be enacted in order to fix violations.

We define a movement strategy as a modification in the placement of a computing task or a set of data. The abstract movement strategy is characterized by one movement action and an object category (e.g., data, computation). The abstract movement strategy is also characterized by the effects on the environment that the enactment of such a strategy will cause. This information can be retrieved by a knowledge base built from the observation of previous enactments using machine learning techniques like reinforcement learning techniques.

In order to enact a movement strategy, it is necessary to specify also the actual object of the movement and its initial and final location. We define this as movement strategy instance. A movement strategy instance may be subject to some constraints given by the object of movement. If we consider data movement, a constraint can be related to privacy and security policies on the moved data. These policies are independent from the context and need to be applied anytime a movement action is applied to an object affected by them.

In order to enact a movement action, several alternative techniques may be used. A movement technique is a building block of a movement strategy. Strategies will combine these building blocks according to the specific needs of an application. Similarly, computation movement techniques will be proposed to define how to distribute the tasks to be executed among the available nodes taking into account the requirements of the task, the resource made available by a node and general requirements at application level. As an example, in case of a movement strategy requiring data aggregation, a set of different movement techniques may be alternatively selected each one implementing a different aggregation algorithm.

The selection of the most suitable movement action for a given context should be driven by the expected utility improvement, which is computed on the basis of the detected violation and the known effects of the strategy over the environment.

Based on this definition of movement strategy, the primary goal of is to find a coherent mechanism for deciding which is the best data/computation movement action to be enacted based on the application characteristics, the nature of the data, privacy and security issues, and the application's non-functional requirements. Creating rules and selecting parameters for an automatic selection of data movement actions in different contexts represents now a major challenge. In Figure 3.6 we show a preliminary model specifying the influences that need to be mapped between the several elements of the environment. More specifically, through the analysis of the data collected by the monitoring system some events are raised by violations in the data utility, which compose the context block of the figure. The goal block is a representation of the user requirements composing the data utility evaluation, which can be accessed from the events in the context. When the violations occur in the data utility from the context, some available

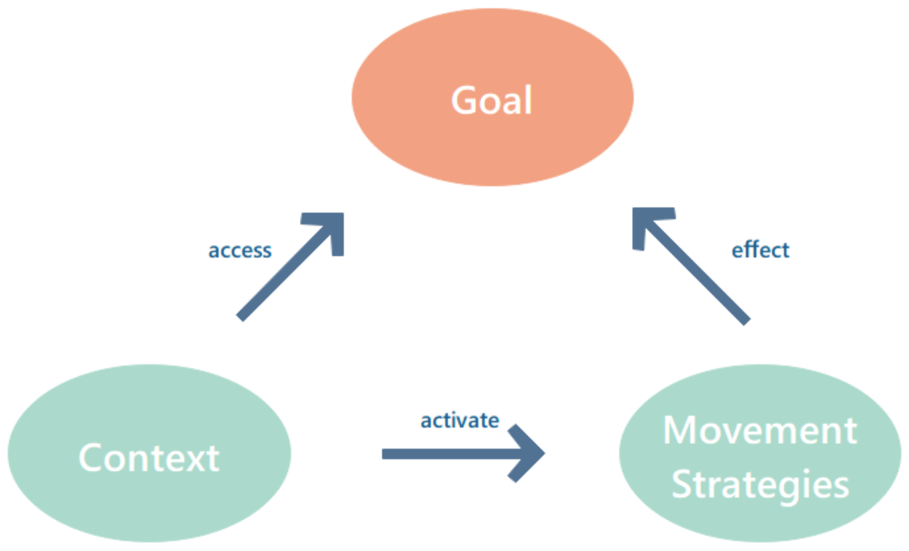


Figure 3.6: Model of movement strategy

movement strategies will be triggered, then they could exert some effects on the goals.

## 3.4 Monitoring system

As described in the above context, the DaaS architecture we are designing should be feasible in a Fog Computing environment. Moreover, the data management life-cycle is supposed to exist alongside with the life-cycle of the data-intensive application developing phase, from where data are generated, to where they are stored and processed. The resources involved in the data management can be located both at the edge of the network (e.g., sensors) and on the cloud.

Since containers have provided us with more flexibility for running cloud-native applications on physical and virtual infrastructure, we decide to adopt container technique in order to package up the services provided by our platform and make them portable across different computation environments. On the other side, since containers rely on resources of the host OS and only keep the different pieces that are unique within the containers, they are much light weight comparing with virtual machines. It implies that it is more efficient for them to use the underlying infrastructure. By the adoption of containers, it is easy to deploy and manage the VDC instances with different data requirements. In order to manage the behaviors of all the containers, a container orchestration tool should be taken into consideration. In this thesis, the DaaS architecture is built on the top of a kubernetes cluster.

Kubernetes is an open source cloud-native platform for deploying, scaling, and managing containerized applications. These applications can run across multiple operating environments, including public clouds, virtualized private clouds, edge-based devices. Kuberntes eliminates infrastructure barriers by providing core capabilities for containers through a combination of features, including Pods, Services, and ReplicaSet.

Monitoring the environment of a distributed system is significant to detect which performance, resilience, and security are important to our architecture. Also, it is an effective way to anticipate problems and discover bottlenecks in a production environment. In our architecture, the applied monitoring system is used to track running states of the application and the computing resources utilization, in order to provide critical information for the following decision making system. To be specifically, if the decision making system detects any violation of the current execution environment with the required data utility values, data and computation movement will be enacted with a proper movement strategy. Referring to the kubernetes cluster, a pod scaling should be attached to fix the violation.

The monitoring indicators collected among different applications vary according to the different functionality of the applications. Kubernetes applications usually consist of multiple, separate services, each running in its own container. Some basic monitoring indicators, like resource usage, could be collected by observing the resource usage of the host containers. However, in some special cases, we may need some specific business indicators collected directly inside the applications. In our architecture, we need a customized monitoring system which can gather a set of indicators providing information about both the application behavior and the execution environment state.

# Chapter 4

# Design and Implementation

As a result of the research, a cloud-based framework is proposed, taking advantage of the reliability and scalability of Kubernetes in order to providing a stable customized monitoring system. The purpose of the designed customized monitoring system is to support the further data and computation movement with the monitoring data at run-time. Also, the developing platform should follow the principles of Service Oriented Architecture.

In this chapter, the framework will be described in details, including framework structure, components, requirements, and some conceptual descriptions. On the other side, a suitable monitoring system, which can not only collect the regularly monitoring metrics of the entire architecture (e.g., resource utilization metrics, cluster health metrics) but also customized monitoring metrics related with specific running applications, is deployed within this framework.

## 4.1 Mapping between the methodology and Kubernetes

The underlying mechanism about data management in a Fog Environment, where both resources at the edge and resources on the cloud are involved, has been described on the theoretical aspect in Chapter3. This section provides a high-level overview of the DaaS architecture by combining with Kubernetes architecture and containerization technology.

Data as a Service (DaaS) is used to take care of all the activities needed to collect, process, store and publish data regardless of the location in which they are stored and from which they are requested. All the inner structure of the framework is transparent to the end-users. The framework provides interfaces for the requests of the customers with different requirements. Then, the framework decide from where and how to retrieve the requested data. Here we adopt Kubernetes which is an open source system for automating deployment, scaling, and management of containerized applications.

In the Kubernetes cluster, each node is considered as a distributed resource located either at the edge of the network or in the cloud. In order to simulate a real Fog environment, part of the generated nodes should rely on some real hardware. Since Kubernetes runs great on bare metal, we can build a Raspberry Pi cluster running Kubernetes as well. However, in this thesis, all the involved nodes are deployed on a cloud data-center using virtual machines of different dimensions to simulate distributed locations in a Fog environment.

The mapping between the proposed methodology and kubernetes is shown in Figure 4.1. The mapping diagram consists of two main components: VDC (lives inside slaver node) and VDM (lives inside master node). In what follows, more details will be illustrated.

**Virtual Data Container (VDC)**

VDC is the concrete component which lays between data-intensive application developers and distributed data sources, to provide required data in the desired format and with a proper quality. Each VDC is dedicated to one DIA developer for specific usage. In the same way, each VDC comes with a single node in a Kubernetes cluster. Each slaver node attaching to the same master node can have different response time due to the different performance provided by the virtual machines and current network workload.

When a data request from DIA developer is received, a new node should be pulled out and attached to the master node. The new generated node works as the destination to deploy a VDC based on the received information, including data format, location of data source, Quality of Service, etc. Each VDC acts as a single pod which deployed inside the new generated node. Once our platform receives any data request by DIA developers, the VDC will try to retrieve the corresponding data regardless where the data source locates. If the queried data

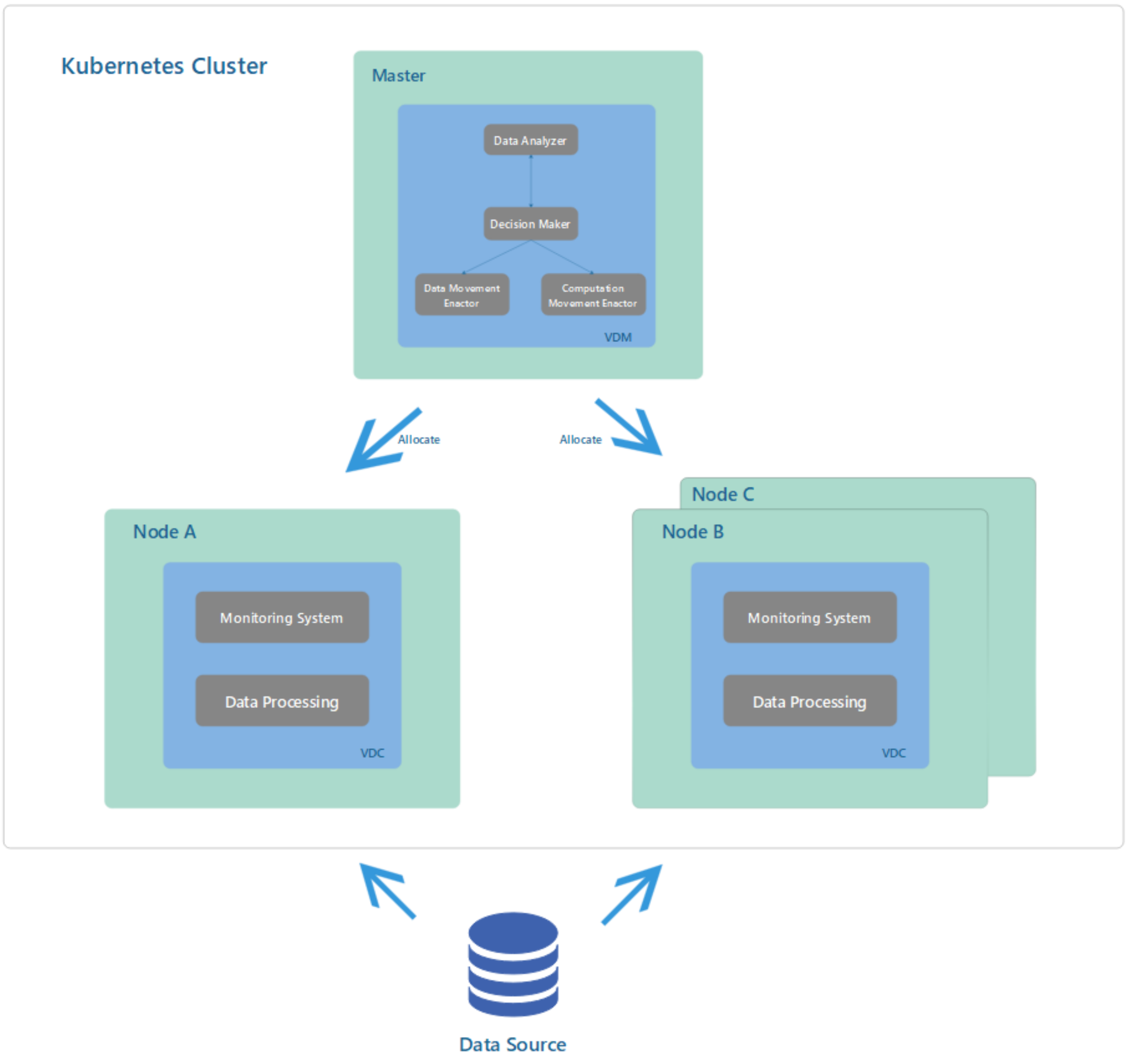


Figure 4.1: The mapping between the methodology and Kubernetes

does not fit the predefined user requirements, a further data processing activity With the received data steam will be activated. At the same time, a dedicated monitoring system will synchronize monitor and record the state of the running environment within the life-cycle of VDC. By tracking the monitoring data, if the currently running environment does not satisfy the predefine requirement, a further movement action should be enacted. When the processed data in a VDC fit all the data constraints, they can be transmitted to the data-intensive application in a good manner.

Since data sources may be distributed in different locations, VDCs should be able to collect data regardless where they are deployed (i.e., on the edge or in the cloud). This kind of issue is solved by the adoption of Kubernetes, which provided scalability support to retrieve data from heterogeneous data sources.

On the other side, a VDC should be easily deployed on resources which can live either on the edge or in the cloud. It implies that the VDC attached node can either rely on the resources provided by any Cloud Service Provider (CSP), or resources at the edge of the network (e.g., Raspberry Pi). Since Kubernetes provides an effective extension to ARM devices, the designed Kubernetes cluster can be feasible to include some Single Board Computers as the edge side in a Fog environment.

There are two main components inside each VDC node.

1. *Data Processing*: once the data are collected, a set of data processing techniques may be required to transform data (e.g., encryption, compression), according to the functionality and non-functionality requirements from the DIA developers. Thus, the resulting data processing application should be executed inside this VDC node.
2. *Monitoring system*: monitoring system is embedded inside each VDC node to track the application running state and the corresponding environment states. Some monitoring metrics will be used as one of the factors to trigger data or computation movement.

**Virtual Data Manager (VDM)** The creation and deployment of a VDC instance is based on the selection of a VDC Blueprint. As the same VDC Blueprint can be adopted by different applications, each of these applications includes instances generated from this VDC Blueprint, thus, they are connected to the same data source. VDM is used to manage this concurrent access to the same data sources. It implies that VDM should have the complete control on the behavior of different VDC instances operating on the same data sources. Through this, we decide to deploy VDM inside the master node which have full privileges to access and control all the slave nodes attaching to this master node.

Usually kube-scheduler will automatically do the assignment with a reasonable strategy, based on the machine resource usage or some related constraints. But it is also possible to constraint a pod facing it to run on a specific node in

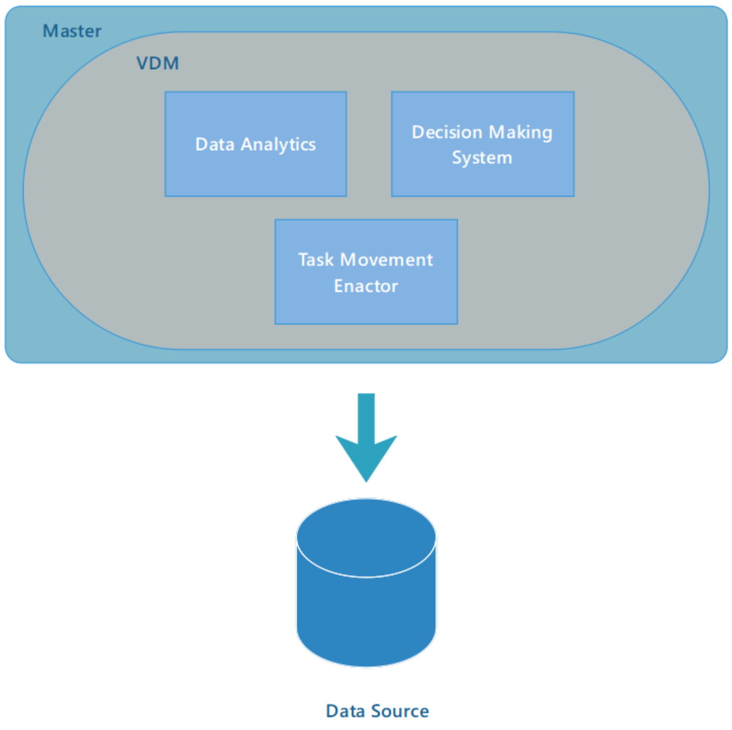


Figure 4.2: VDM components

order to gain more control on a node where a pod lands. After specifying application requirements to the VDC resolution engine, DIA developer can select the most suitable VDC blueprint which is queried from the repository.Once the VDC Blueprint is selected, a VDM is created and a specific VDC node based on the Blueprint will be allocated.

As presented in Figure 4.2, there are several functionality components inside VDM.

1. *Data analytics*: through the analysis of the monitoring data collected from each connected VDC, it provides information to the decision making system.
2. *Decision making system*: based on the information provided by the Data analytics module, it makes the decision about whether to enact task movement.
3. *Task movement enactor*: if the decision making system decide to move tasks, the task movement enactor will start moving them through some suitable mechanisms.

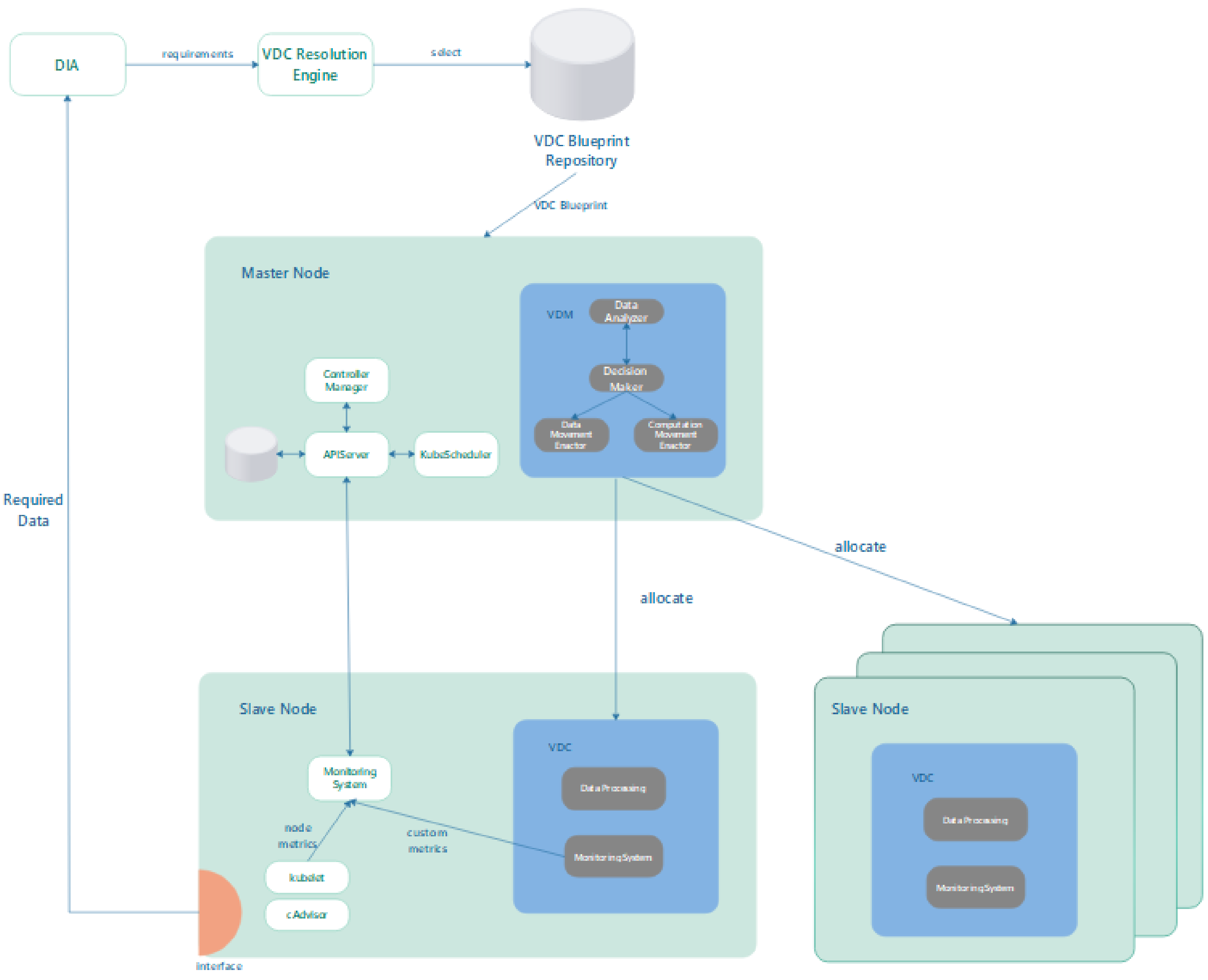


Figure 4.3: Kubernetes architecture overview

Figure 4.3 presents the whole Kubernetes architecture that will be applied in our platform. First, the DIA developers send a data request and the requirement definition for the request data to the VDC resolution engine. Based on the pre-defined data requirements, a VDC Blueprint should be selected. Once the VDC Blueprint is ready, a VDM is created as a running pod inside the master node. At the mean time, a VDC node will be allocated to a feasible location (e.g., virtual machine in the cloud or smart device at the edge). For a single VDC Blueprint, there is only one unique VDM corresponding to it whereas several VDC instances could be generated. Data could be retrieved from distributed data sources. Inside each VDC, a data processing module is deployed to handle the retrieved data in order to fit all the requirements. Referring to the customized monitoring system in the cluster, it can collect the environment monitoring data by using node metrics provided by kubelet, which is embeded natively in each node. Also, it can gather the VDC running states by using custom metrics defined by data processing application. Then, the data analytics module take charge of analyzing the monitoring data collected from each connected VDC. A decision will made to decide whether and where to transmit the current VDC to another location. The final goal is to obtain the most suitable data following all the constraints and requirements. In this way, the Kubernetes cluster provides required data as a service to data-intensive applications through exposed interfaces of the VDC node.

## 4.2 Monitoring System in kubernetes

At any scale, monitoring Kubernetes cluster as well as the health of the deployed applications and the containers running them is essential to ensure good performance. In our designed platform, we need a distributed monitoring system which resides in each node to collect performance indicators about the containerized application states and the health state of each node.

### 4.2.1 Feasible monitoring system in kubernetes

Kubernetes by design do not provide self-monitoring mechanism. In this case, a set of monitoring tools with their own characteristics and unique qualities can be deployed within kubernetes as addons. Currently, there are several tools which can be used to collect and graph Kubernetes metrics. In our platform, we want to provide data-intensive application developers with detailed resource usage information of their running applications at different levels: containers, pods, services, and the entire cluster. Moreover, the selected monitoring system should support custom metrics for each applications running inside the cluster. By exploring state-of-art, what follows present the most frequently-used monitoring system under kubernetes.

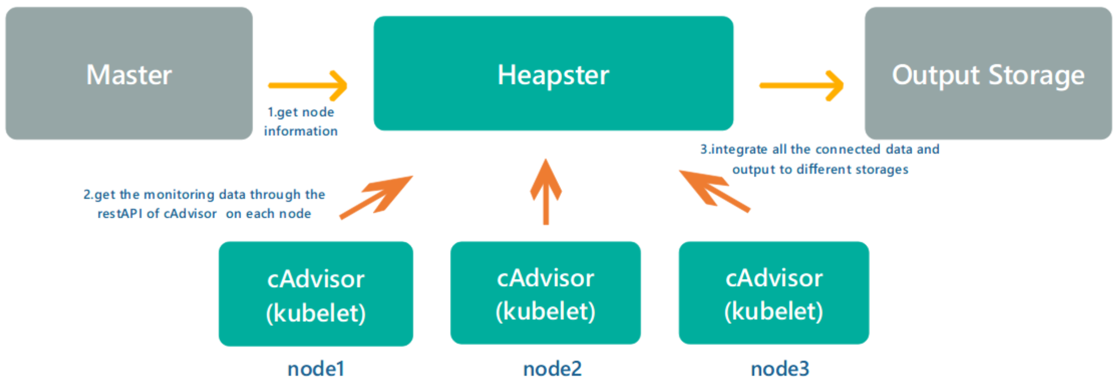

Master
1.get node information
Heapster
Output Storage
3.integrate all the connected data and output to different storages
2.get the monitoring data through the restAPI of cAdvisor on each node
cAdvisor (kubelet)
cAdvisor (kubelet)
cAdvisor (kubelet)
node1
node2
node3

Figure 4.4: Monitoring strategy of heapster

*cAdvisor*[1] is a resource usage and performance analysis agent built for the container. In Kubernetes, cAdvisor is integrated into the Kubelet. The purpose of the cAdvisor is to automatically discover all the containers in the machine and collect CPU, memory, filesystem, and network usage statistics.

*Heapster*[2] is a cluster-wide aggregator of monitoring and event data. It supports kubernetes natively, even they are set up in different approaches. Similar to other containerized application, Heapster runs as a pod in the cluster. As shown in Figure 4.4, Heapster obtains all the node information through the cluster master node and then collects the monitoring data from kubelets which is the primary node agent in every node. Finally, Heapster integrates all the monitoring data and push them to a configurable backend for storage (e.g., InfluxDB) and visualization (e.g., Grafana). It allows the extraction of up to 15 minutes of historical data for any Container, Pod, Node or Namespace in the cluster, as well as the cluster itself.

*Prometheus*[3] is an open source Cloud Native Computing Foundation (CNCF) project to monitor applications and microservices running in containers at scale. Similar to Heapster, it is also native to containerized environments.

Comparing to other monitoring systems, Prometheus has some distinguishing features[7].

[1] https://github.com/google/cadvisor
[2] https://github.com/kubernetes/heapster
[3] https://github.com/prometheus/prometheus

- it provides a multi-dimensional data model, whose timeseries is defined by metric name and set of key/value dimensions.
- a flexible query language (PromQL) is used to leverage this dimensionality.
- no dependency on distributed storage.
- timeseries collection happens via a pull model over HTTP.
- targets are discovered via service discovery or static configuration.
- multiple modes of graphing and dashboarding support.
- support for hierarchical and horizontal federation.

As we mentioned in previous context, cAdvisor is responsible for collecting data about running containers from each node and then Heapster is able to query them through the kubelet of the node. It implies that Kubernetes replies on Heapster to report metrics instead of the cgroup file directly. Control groups (cgroup) is a feature natively provided by Linux, which can limit and isolate the resource utilization for a collection of processes. However, Heapster is not practical for collecting custom metrics from the applications running in the containers. This is one of the reasons why we decide to abandon Heapster and use Prometheus as a monitoring solution.

Prometheus is able to monitor all the components within a Kubernetes cluster, including the control plane, the worker nodes, and the applications running on the cluster. Considering all the described distinguishing features, we decide to select Prometheus as the main monitoring system. Another important factor is that the custom metrics are usually emitted in Prometheus format, while Heapster does not implement Prometheus as a data sink (typically use InfluxDB as a backend storage). Even though, we still deploy Heapster as an auxiliary monitoring system in our platform, in order to directly view the cluster resource utilization from the Kubernetes dashboard.

### 4.2.2 Monitoring metrics

Metrics represent the performance measurements of resource behavior and usage that can be observed and collected. In kubernetes, there are large quantities of monitoring metricses with different purposes and functionalities. In the following list, these metrics are divided into two main parts [3].

1. *Cluster monitoring metrics*: The metrics used for cluster monitoring are mainly focusing on the detection of the health status and resources usage in all the components of the kubernetes cluster. This category could be further separated into three levels.

    - *Resource utilization metrics*: resource utilization, such as network requests, filesystem utilization, CPU and memory utilization, are important to check the health status of the cluster. The target to apply resource utilization metrics could be either the node or some specific pods. By monitoring system resource utilization, it gets easiler to check whether the cluster and deployed applications remain healthy.
    - *Number of nodes metrics*: this metrics is used to check the number of available nodes in the cluster. By using this metrics, we can check if all the configured nodes are correctly connected with the cluster.
    - *Running pods metrics*: this metrics present the number of available pods in each node of the cluster. The desired number of pods is predefined by some configuration files when creating pods. If the observed number of the running pods do not match, error detection actions should be applied.
    - *Kubernetes metrics*: with Kubernetes metrics, we can monitor how a specific pod and its deployment are being handled by the orchestrator. By comparing the number of instances a pod has and the expected number of instance a pod should have, it is easy to detect whether the cluster is out of resources.

2. *Custom metrics*: In order to correctly and effectively monitor the built infrastructure, the resource metrics obtained by monitoring Kubernetes cluster and containers are correlated with the performance of different applications running on top of them. Each image has its own characteristics, so the types of metrics vary from one to another. Even for the same application image, if it is running in the container located in distributed nodes, the performance is not the same due to the unbalanced computing resources between different running environment. The the adoption of custom metrics

Table 4.1: common monitoring metrics in Kubernetes

| Metric Name | Description |
|---|---|
| cpu/node_capacity | CPU capacity of a node |
| cpu/node_utilization | CPU utilization as a share of node allocatable. |
| cpu/usage | Cumulative CPU usage on all cores. |
| cpu/usage_rate | CPU usage on all cores in millicores. |
| filesystem/usage | Total number of bytes consumed on a filesystem. |
| filesystem/available | The number of available bytes remaining in a the filesystem. |
| memory/limit | Memory hard limit in bytes. |
| memory/node_capacity | Memory capacity of a node. |
| memory/node_allocatable | Memory allocatable of a node. |
| memory/usage | Total memory usage. |
| network/rx | Cumulative number of bytes received over the network. |
| network/tx | Cumulative number of bytes sent over the network |
| uptime | Number of milliseconds since the container was started. |

gives us more possibility to explore the data about how the containerization application are performing. For instance, we can set the response time as a custom metrics, since the response time usually will not be the same for each application. In our platform, this kind of metrics is essential for tracking the application performance.

In Kubernetes, there are lots of resource metrics can be collected by cAdvisor. In Table 5.1, some common ones are listed.

Referring to the system metrics monitoring, they are usually exposed through REST APIs which involve several domains, including CPU monitoring, memory monitoring, network throughput monitoring. CPU monitoring includes total usage monitoring, single kernel usage monitoring, and fault monitoring. Memory monitoring includes total usage monitoring and fault monitoring and Network throughput monitoring includes throughput monitoring and fault monitoring. On the other side, custom metrics are specific for different applications and purposes.

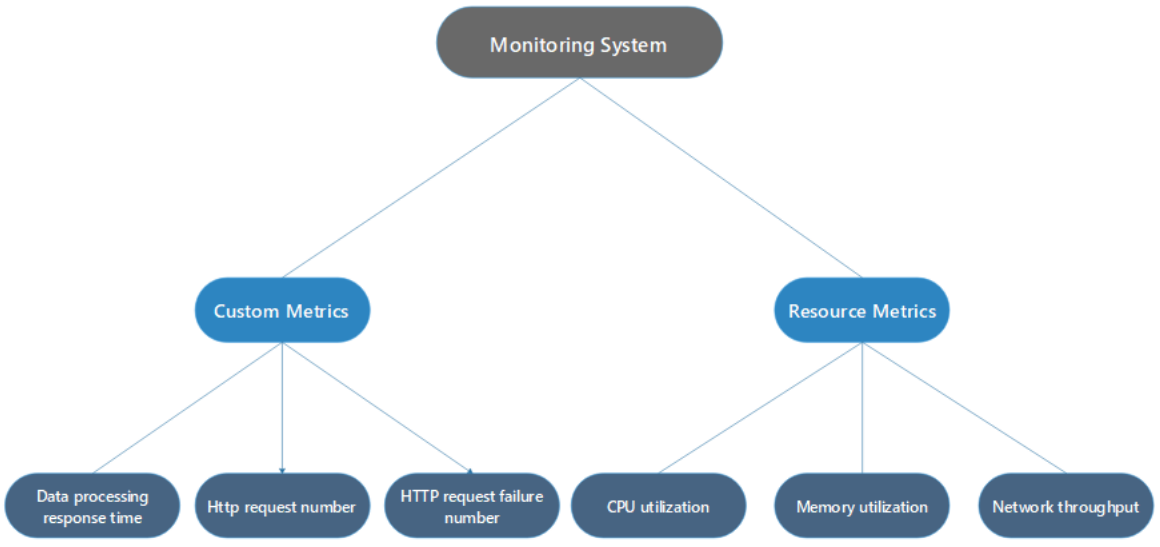


Figure 4.5: Metrics involved in the monitoring system

In our platform, we choose three resource metrics (CPU utilization, memory utilization, Network throughput) as the monitoring indicators. For all of them are on the pod level, which means they are used to track the health state of that pod.

On the other side, response time for the data processing module is set as the one of custom metrics of the containerized application in VDC (Figure 4.5). This value is crucial for measuring if the current VDC satisfies the requirements from the DIA developers. What is more, another two custom metrics are set, HTTP request number and HTTP request failure number. HTTP request number is used to record how many times VDC sends a HTTP request to the external and HTTP request failure is used to record the failure number when meeting a network error.

## 4.3 Deployment of the customized monitoring system

This section is intended to give an introduction to all the components required to integrate Kubernetes cluster with a proper monitoring system. Prometheus is a standalone project devoting to obtain a proper monitoring and altering functionality. Combining the features both from Kubernetes and Prometheus, an easy-to-deploy solution is proposed.

All the configuration files used by Kubernetes to deploy the monitoring system are available on GitHub, that can be accessed through the following link:

`https://github.com/CharlesHouse/MasterThesis/tree/master/manifests`

### 4.3.1 Configure Prometheus for Kubernetes

Before deploying Prometheus server into our Kubernetes cluster, it is necessary to introduce some components and services that could make the setup easier.

**Prometheus Operator**

Operators were introduced by CoreOS as a class of software which are able to reduce the configuration workload by putting knowledge collected by humans into software. An Operator builds upon the basic Kubernetes resource and controller concepts but includes application domain knowledge to take care of common tasks. As one of the most famous Operators, Prometheus Operator gives a native support to integrate Prometheus server with Kubernetes cluster.

The mission of the Prometheus Operator is to make Prometheus setup on top of Kubernetes as easy as possible, while preserving as well as Kubernetes native configuration options.

The Prometheus Operator introduces additional resources in Kubernetes to declare the desired state of a Prometheus and Alertmanager cluster as well as the Prometheus configuration. The resources it introduces are: Prometheus, Alertmanager, ServiceMonitor. These newly defined ThirdPartResources (TPRs) play a key role in simplifying Prometheus setup.

The Operator ensures at all times that for each Prometheus resource in the cluster a set of Prometheus servers with the desired configuration are running. This entails aspects like the data retention time, persistent volume claims, number of replicas, the Prometheus version, and Alertmanager instances to send alerts to. Each Prometheus instance is paired with a respective configuration that specifies which monitoring targets to scrape for metrics and with which parameters.

The user can either manually specify this configuration or let the Operator

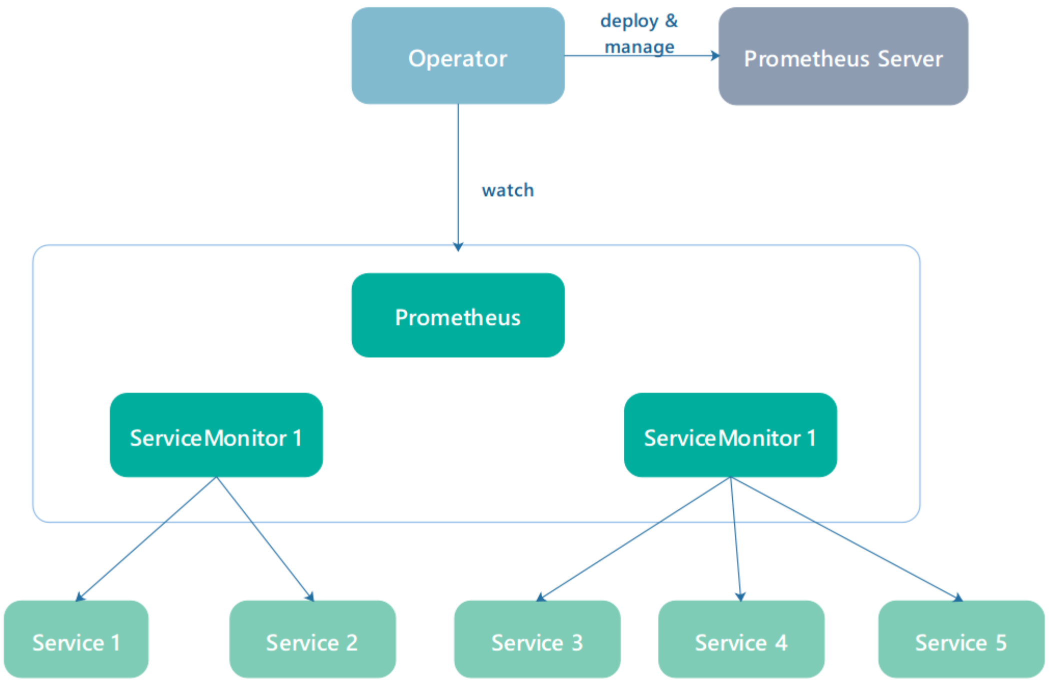


Figure 4.6: Prometheus Operator Architecture

generate it based on the TPR, the ServiceMonitor. The ServiceMonitor resource specifies how metrics can be retrieved from a set of services exposing them in a common way. A Prometheus resource object can dynamically include ServiceMonitor objects by their labels. The Operator configures the Prometheus instance to monitor all services covered by included ServiceMonitors and keeps this configuration synchronized with any changes happening in the cluster.

The Operator encapsulates a large part of the Prometheus domain knowledge and only surfaces aspects meaningful to the monitoring system's end user. It's a powerful approach that enables engineers across all teams of an organization to be autonomous and flexible in the way they run their monitoring.

**Service Monitor**

Prometheus-operator uses a Custom Resource Definition (CRD), named ServiceMonitor, to abstract the configuration to target. The ServiceMonitor resource specifies how metrics can be retrieved from a set of services exposing them in a common way. A Prometheus resource object can dynamically include ServiceMonitor objects by its labels. The Operator configures the Prometheus instance to monitor all services covered by included ServiceMonitors and keeps this configuration synchronized with any changes happening in the cluster.

The Operator encapsulates a large part of the Prometheus domain knowledge and only surfaces aspects meaningful to the monitoring system's end user. It's a powerful approach that enables engineers across all teams of an organization to be autonomous and flexible in the way they run their monitoring.

**Node Exporter**

The Prometheus node exporter is used to monitor a node's resource, including CPU, memory, and disk utilization. By default, kubernetes tracks monitoring metrics through the cAdvisor service on all the nodes. However, for the metrics collected from other sources have to be translated to a compatible format. Prometheus uses exporters to read metrics from other sources and translate them into the Prometheus format.

Kubernetes provides DeamonSets, which ensure pods are added to nodes as nodes are added to the cluster. We can use this to easily deploy Prometheus node exporter to each cluster node. We deploy one node exporter for each cluster node by using a DaemonSet. and expose the running node exporter as a service at port 9100. The node exporter configuration file is written in yaml format and the command to create the corresponding DaemonSet is:

```
$ kubectl create -f node-exporter.yaml
```

Once the Prometheus Server has been deployed, it will automatically scrape all node exporters for metrics.

**kube-state-metrics**

Heapster acts as an intermediary that reformats and integrates metrics already generated by kubernete, whereas kube-state-metrics is focusing on generating new metrics from kubernete's object state. Since the metrics provided by Heapster are so limited, we decide to deploy kube-state-metrics into our kubernetes infrastrure.

kube-state-metrics[4] is a simple service that listens to the Kubernetes API server and generates metrics about the state of the objects. It is not focused on the health of the individual Kubernetes components, but rather on the health of the various objects inside, such as deployments, nodes and pods.

[4]https://github.com/kubernetes/kube-state-metrics

The metrics are exported through the Prometheus golang client on the HTTP endpoint */metrics* on the listening port (default 80). They are served either as plaintext or protobuf depending on the *Accept* header. They are designed to be consumed either by Prometheus itself or by a scraper that is compatible with scraping a Prometheus client endpoint.

The kube-state-metrics service is availiable once it is built as a pod inside a kubernetes which has a service account token has read-only access to the Kubernetes cluster. One replication for this pod is enough.

Next up is the Prometheus object itself. We will deploy the application, and then the roles/role-bindings.

```
find manifests/prometheus -type f ! -name prometheus-k8s-roles.yaml
! -name prometheus-k8s-role-bindings.yaml -exec kubectl --namespace
"$NAMESPACE" apply -f {} \;
kubectl apply -f manifests/prometheus/prometheus-k8s-roles.yaml
kubectl apply -f manifests/prometheus/prometheus-k8s-rolebindings.yaml
```

Figure 4.7 shows all the involved components of monitoring system we are going to deploy in our own Kubernetes cluster.

Since we need to track the running states over all the available nodes, including master node and slave nodes, we decide to deploy one node export running inside a pod in each node. The node exporter is responsible to monitor cAdvisor service in order to pull resource metrics of the cluster. Firstly, we have to expose the cAdvisor service and allow webhook token authentication. The following script is executed in the CLI of the master node to write the valid cAdvisor port parameter into the system configuration file and enable webhook authorization for the following deployed node exporter.

```
sed -e "/cadvisor-port=0/d" -i "$KUBEADM_SYSTEMD_CONF"
if ! grep -q "authentication-token-webhook=true" "$KUBEADM_SYSTEMD_CONF"; then
  sed -e "s/--authorization-mode=Webhook/--authentication-token-webhook=true
  --authorization-mode=Webhook/" -i "$KUBEADM_SYSTEMD_CONF"
fi
systemctl daemon-reload
```

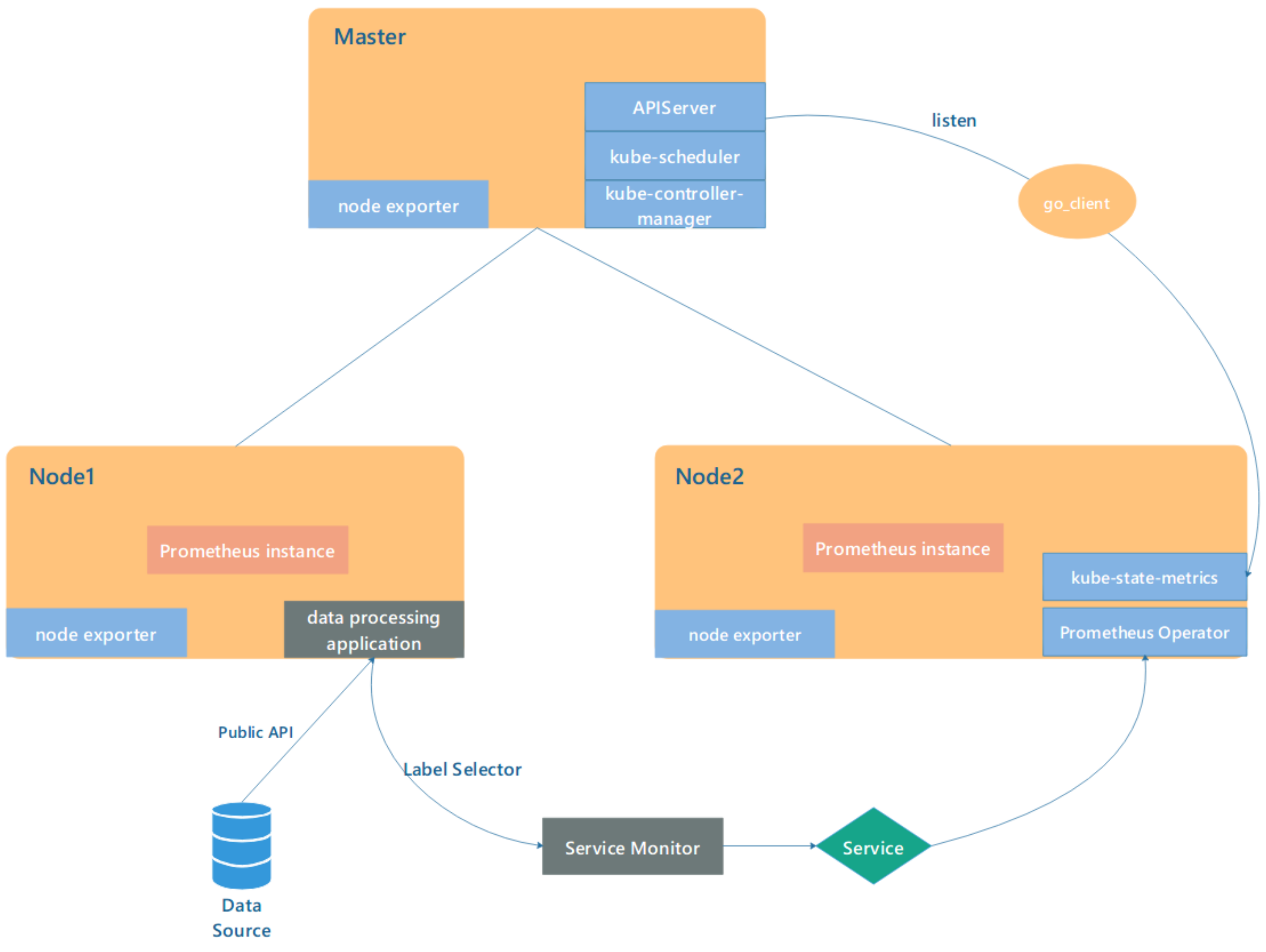


Figure 4.7: Components for the monitoring system

```
systemctl restart kubelet
```

Then, we use DeamonSet to deploy node exporter into each node. DeamonSet is a native controller supported by Kubernetes which ensures all nodes run a copy of a pod. In this way, when a new VDC node is generated and connected to our Kubernetes cluster, a node exporter pod can be synchronized created and attached to this VDC under the control of DeamonSet.

The Prometheus node exporter allows monitoring node resource metrics which are already generated by Kubernetes, including CPU, memory and disk utilization. However, node exporter cannot collect metrics from Kubernetes object state (e.g., deployment, StatefulSet, DeamonSet, job, etc). Thus, we decide to deploy an addition addon which works as a separate project enables access to these metrics. kube-state-metrics is deployed as a pod which listens to the Kubernetes API server to track the health of objects inside the cluster.

Prometheus instance runs as a pod to pull monitoring metrics from the pointing targets. For each Prometheus instance, the Prometheus Operator ensures that there are a set of Prometheus servers running with the desired configuration. The connection between a Prometheus instance and the targets is achieved by Service Monitor, which specified how metrics can be retrieved from a set of services. Service Monitor are written with a declarative language and stored into a yaml file. For instance, the Service Monitor used to connect Prometheus instance with Kubernetes APIServer is described as follows:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: kube-apiserver
  labels:
    k8s-app: apiserver
spec:
  jobLabel: component
  selector:
    matchLabels:
      component: apiserver
      provider: kubernetes
  namespaceSelector:
    matchNames:
    - default
  endpoints:
  - port: https
    interval: 30s
    scheme: https
    tlsConfig:
      caFile: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      serverName: kubernetes
    bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
```

In order to make Prometheus instance available to pull all cluster metrics, several Service Monitors should be deployed and attached to every components of Prometheus and Kubernetes, including kube-control-manager, kube-scheduler, kubelet, node-exporter, prometheus Operator and Prometheus Server. All the configuration files have been uploaded into the GitHub repository.

### 4.3.2 Configure customized Prometheus

In addition to the monitoring the regular cluster metrics of Kubernetes components, we still need to take care of the custom metrics of some specific VDCs at run-time. Since Prometheus supports time-series collection via a pull model over HTTP, we have to provide a mechanism to make a Prometheus instance access the HTTP endpoints exposed by a VDC.

Once a VDC node successfully collects the raw data from some data sources, the data should do a further processing to satisfy the functionality and non-functionality requirements predefined by DIA developers. For the data processing module of the VDC, we developed a sample application which is able to expose some predefined custom metrics through a HTTP endpoint. When the data reach the desired data requirements, they can be transmitted to the end-users. Referring to the customized monitoring system, the VDC should expose some custom metrics to our deployed Prometheus server, in order to the running state of the VDC. In this section, we focus on how Prometheus server pull these custom metrics exposed by a VDC.

Figure 4.8 presents the generation flow of application instrumentation with Prometheus. The data processing application runs inside a dedicated pod and provides a corresponding service to communicate with other components. Since the containerized application exposes a metric interface, we should trigger a Prometheus instance to pull the defined metrics. In order to make this running application communicate seemly with a Prometheus instance, a custom Service Monitor is adopted. The application service should set a specific lable, so that the Service Monitor is able to connect it by

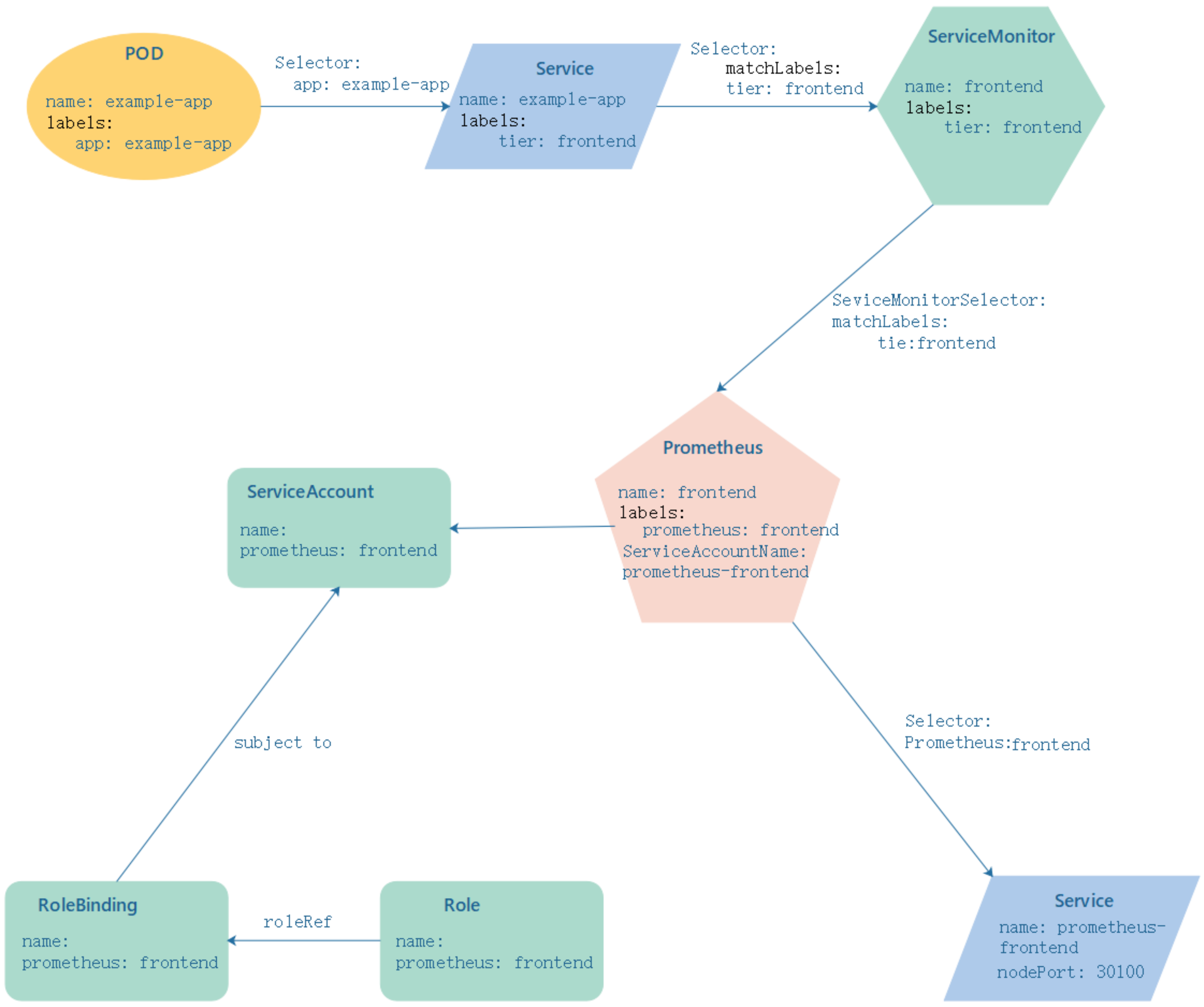


Figure 4.8: Generation flow of application instrumentation with Prometheus

matching this label. For instance, we set *tier: frontend* as the service label. When the Prometheus successfully collected the application custom metrics, it exposes endpoints for them by creating a Prometheus service in Kubernetes.

On the other side, Kubernetes provides a mechanism for controlling access to the Kubernetes API – Role-Based Access Control (RBAC). Kubernetes defines two resources for roles (Role and ClusterRole) and two resources for connecting subjects to those roles (RoleBinding and ClusterRoleBinding). Thus, in order to bypass this authentication, we defined a Role, which is able to access Kubernetes resources (e.g., nodes, services, endpoints, pods), to communicate with the Prometheus instance. By creating the dependences between role and rolebinding, we are able to overpass the authentication and access kubernetes API.

The configuration files used by Kubernetes to deploy the customized Prometheus instance for the predefined VDC can be accessed through the following link:

`https://github.com/CharlesHouse/MasterThesis/tree/master/instrumentApp`

## 4.4 Containerized application development

Each VDC should feed back the desired data with the requirements predefined by DIA developers. In order to provide the desired data to the end-users, a data processing action should be enacted over the received raw data. Thus, we decided to develop a containerized web-service application which is able to provide a heavy data processing and expose a application metrics endpoint through HTTP. Since Prometheus server only supports custom metrics with Prometheus format, the designed metrics should follow this rule.

The source code for the containerized data processing application and the resulting Dockerfile can be accessed through the following link:

`https://github.com/CharlesHouse/MasterThesis/tree/master/instrumentApp`

We decide to choose Public API Server[5] as our data source. The Public API Server collects public API services from all over the world through Internet. The base URL for this is *https://api.publicapis.org/*. The service supports Cross-Origin Resource Sharing and all the responses are sent over HTTPS. We decide to query different API data by using its categories.

Table 4.2provides a collective list of public JSON APIs for use in web development. As we can see, for separated API categories, there are a list of API data which can be retrieved from the Public API Server. In the column *Link*, the specific API base URL is presented.

[5]https://github.com/davemachado/public-api

Table 4.2: common monitoring metrics in Kubernetes

| API | Description | Auth | HTTPS | CORS | Link |
|---|---|---|---|---|---|
| Cats | Pictures of cats from Tumblr | No | Yes | Unknown | Go |
| Dogs | Based on the Stanford Dogs Dataset | No | Yes | Unknown | Go |
| IUCN | IUCN Red List of Threatened Species | apiKey | No | Unknown | Go |
| RandomCat | Random pictures of cats | No | Yes | Yes | Go |
| RandomDog | Random pictures of dogs | No | Yes | Yes | Go |
| AniList | Anime discovery | OAuth | Yes | Unknown | Go |
| Jikan | Unofficial MyAnimeList API | No | Yes | Yes | Go |
| Kitsu | Anime discovery platform | OAuth | Yes | Unknown | Go |
| Studio Ghibli | Resources from Studio Ghibli films | No | Yes | Unknown | Go |

The data pipeline of the data processing application is presented in Figure 4.9. When the data processing application sends a request to the Public API Server for retrieving API data by category, the server should feed back a JSON file consisting with a list of API data. With these data, a heavy data processing algorithm is used to transform this data string. In this thesis, we adopt SHA-512 hash function to encrypt this data string. After this processing, the example application provides a REST API to present the encrypted data string for the end-users. Also, it exposes a HTTP endpoint (e.g., http://localhost:8080/metrics) to present the custom metrics in Prometheus format.

Based on this, we decided to apply three custom metrics.

- *requests_latency_seconds_sum*: this metrics is used to record the HTTP request latency, starting from the sending request to the end of data processing. This metrics is essential for monitoring execution statues of the VDC.
- *requests_total*: this metrics is to count the numbers of the sending requests.
- *requests_failures_total*: when the data server send back a request failure, this metrics is used to count the failure number.

The Prometheus client libraries offer four core metric types. These are currently

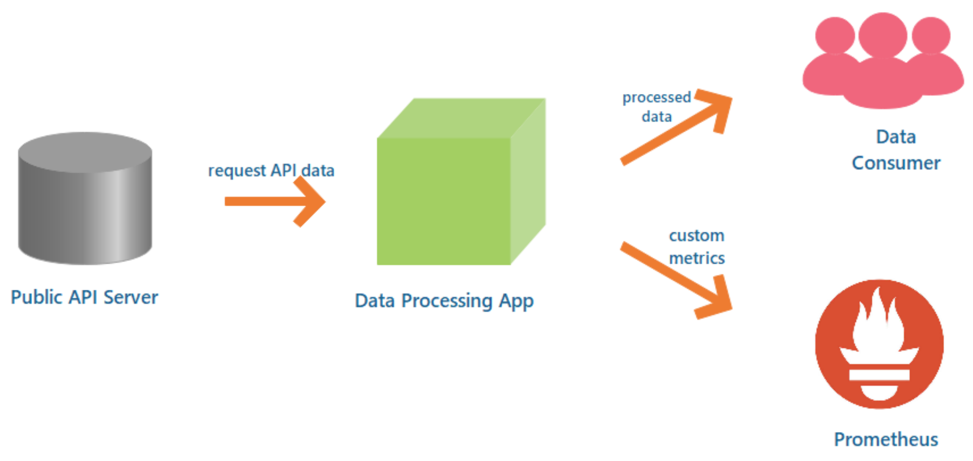


Figure 4.9: Data pipeline of the data processing application

only differentiated in the client libraries (to enable APIs tailored to the usage of the specific types) and in the wire protocol.

- Counter: counters can be used to record metric types that can either increase or decrease, such as recording the total number of application requests, CPU usage time, and so on.
- Gauge: for this kind of increase or decrease indicator, it can be used to reflect the current state of the application, such as the host's current free memory size and available memory size when monitoring the host.
- Histogram: it is mainly used to record the size (such as http request bytes) or the number of events in the specified distribution range.
- Summary: similar to a histogram, a summary samples observations (usually things like request durations and response sizes). While it also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

In order to integrate our application with Prometheus instance, a metrics class is defined according to the Prometheus metrics format. What follows shows the custom metrics class defined in the application

```
public static final Summary requestLatency = Summary.build()
```

```
    .name("requests_latency_seconds")
    .help("Request latency in seconds.").register();
  public static final Counter requestFailures = Counter.build()
    .name("bad-request-number")
    .help("Request failures.").register();
  public static final Counter requestsTotal = Counter.build()
    .name("http-request-number")
    .help("Total number of HTTP requests made.").register();
```

As we decide to use Docker as the container technique, all the applications should be packaged as a Docker image. Thus, we can use this image to directly build and run the developed application in Kubernetes. Since DockerHub is the office image repository, We decide to use it to host our application image.

Once the REST api is ready, the next task to build the web application and build a docker image. By using maven plugin "docker-maven-plugin", a docker image can be created from the project source code. The command used for building docker image is presented in the following command.

```
mvn clean package docker:build -DpushImage
```

Another way to create Docker image is to write Dockerfile directly. The script to build a Dockerfile is presented as follows.

```
FROM tomcat:6
ENV APP_ROOT /myapp
RUN apt-get update && apt-get install -y default-jdk
COPY . $APP_ROOT/
RUN rm -r /usr/local/tomcat/webapps/ROOT
COPY  ROOT.war /usr/local/tomcat/webapps/ROOT.war
WORKDIR $APP_ROOT
```

Docker images used for the application implementation are available inside Docker Hub repository that can be accessed through the following link:

`https://hub.docker.com/u/bestcharles`

# Chapter 5

# Evaluation and Validation

## 5.1 Kubernetes Cluster

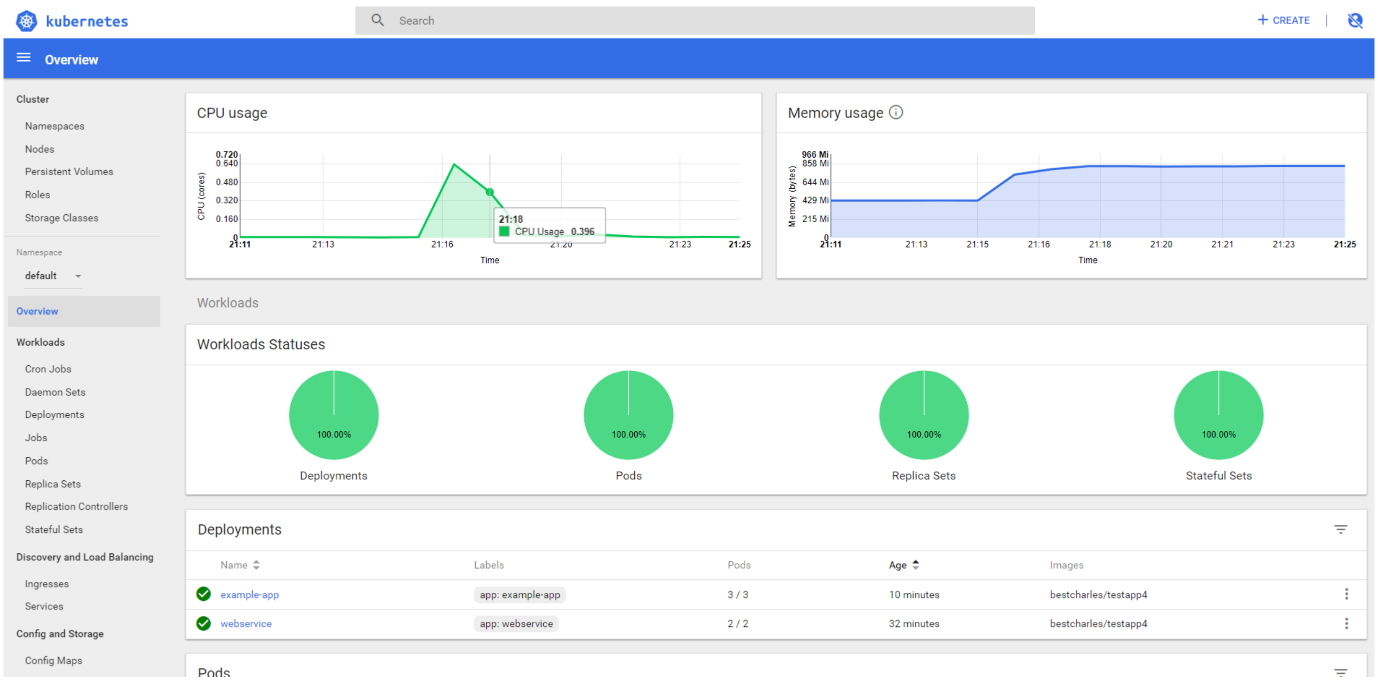
we decide to allocate all the generated VDC instances on cloud resources. However, in order to distinguish them, different amount of resources such as CPU cores, main memory, disk storage, and distributed network will be applied. In this way, we can still simulate to locate some of VDC nodes which are deployed on the cloud, to be at the edge side using proper resource configuration.

Kubernetes can run on various platforms, including Personal Computers, VMs on a cloud provider, and a rack of bare metal servers. The developed kubernetes cluster is hosted in the CloudSigma facilities. We allocated three virtual machines on which Ubuntu 17.04 Server is pre-installed, more details about the computation resources could be viewed from Table 5.1.

By default, the installed k8s cluster will not schedule pods on the master node for

Table 5.1: cloud resources deployed on CloudSigma

| NAME | CPU(GHZ) | RAM(GB) | CONNECTED NETWORK |
|---|---|---|---|
| master | 2 | 4 | 178.22.70.192 |
| node1 | 1 | 2 | 178.22.68.104 |
| node2 | 1 | 1 | 178.22.71.29 |



| Name | Labels | Pods | Age | Images |
|---|---|---|---|---|
| example-app | app: example-app | 3 / 3 | 10 minutes | bestcharles/testapp4 |
| webservice | app: webservice | 2 / 2 | 32 minutes | bestcharles/testapp4 |

Figure 5.1: Kubernetes dashboard

security reasons. As we need to allocate VDM in the master node which is used to manage the corresponding VDCs in several worker nodes, we should also set free to schedule pods on the master:

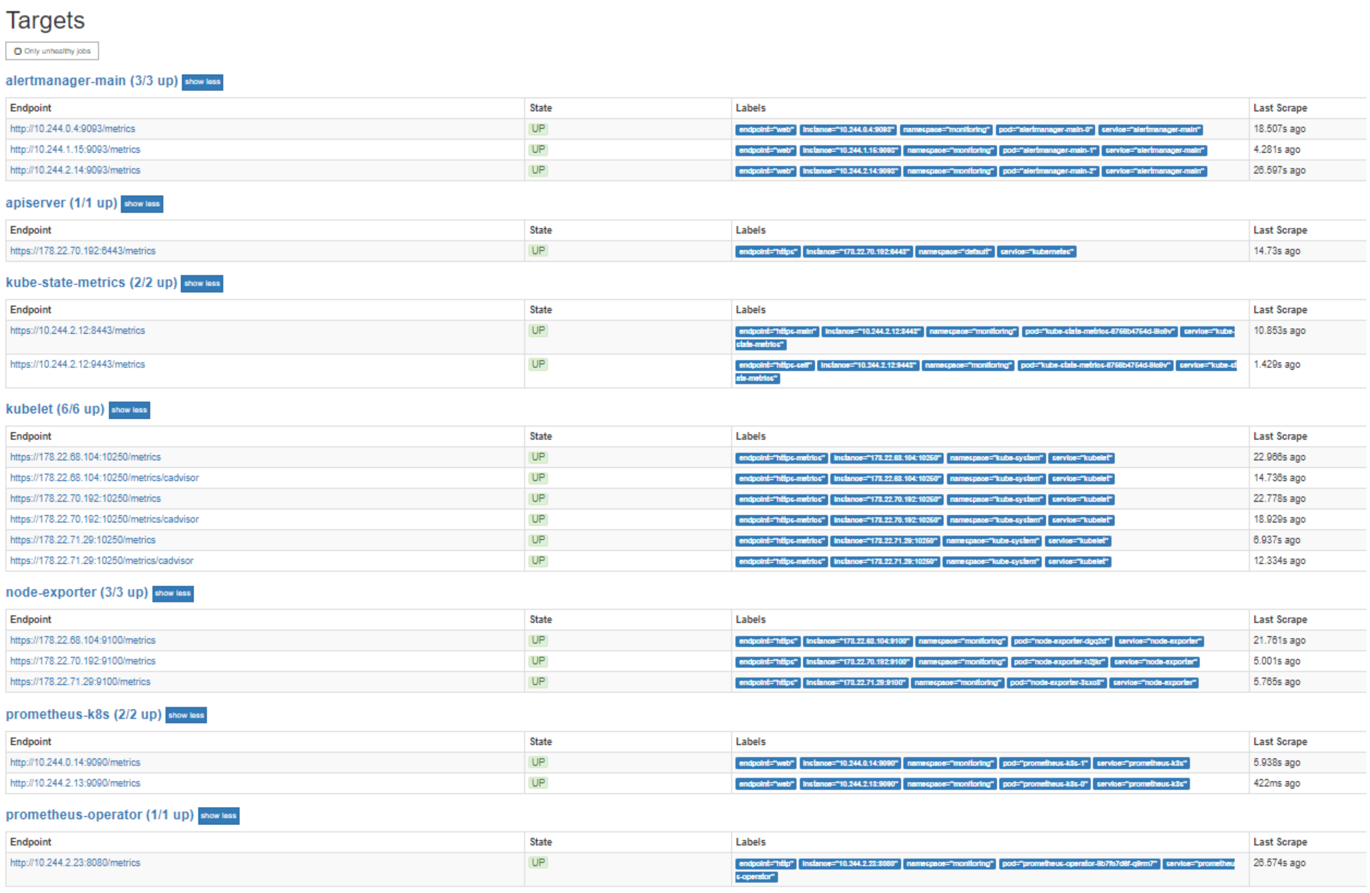
```
kubectl taint nodes --all node-role.kubernetes.io/master
```

This will remove the *node-role.kubernetes.io/master* taint from any nodes that have it, including the master node, meaning that the scheduler will then be able to schedule pods everywhere.

Kubernetes Dashboard is a web-based UI for Kubernetes clusters. By deploying Heapster, we can view some general monitoring information to check the health statues for the Kubernetes resources, including pods, nodes, deployments, Replica Sets and jobs. The screenshot for the dashboard is presented in Figure 5.1.

## 5.2 Monitoring System

Figure 5.2 presents all the monitoring targets by integrating Prometheus and Kubernetes. These connections are triggered by creating and attaching the specific Service-


Targets

Only unhealthy jobs

alertmanager-main (3/3 up) show less

| Endpoint | State | Labels | Last Scrape |
|---|---|---|---|
| http://10.244.0.4:9093/metrics | UP | endpoint="web" instance="10.244.0.4:9093" namespace="monitoring" pod="alertmanager-main-0" service="alertmanager-main" | 18.507s ago |
| http://10.244.1.15:9093/metrics | UP | endpoint="web" instance="10.244.1.15:9093" namespace="monitoring" pod="alertmanager-main-1" service="alertmanager-main" | 4.281s ago |
| http://10.244.2.14:9093/metrics | UP | endpoint="web" instance="10.244.2.14:9093" namespace="monitoring" pod="alertmanager-main-2" service="alertmanager-main" | 26.597s ago |

apiserver (1/1 up) show less

| Endpoint | State | Labels | Last Scrape |
|---|---|---|---|
| https://178.22.70.192:6443/metrics | UP | endpoint="https" instance="178.22.70.192:6443" namespace="default" service="kubernetes" | 14.73s ago |

kube-state-metrics (2/2 up) show less

| Endpoint | State | Labels | Last Scrape |
|---|---|---|---|
| https://10.244.2.12:8443/metrics | UP | endpoint="https-main" instance="10.244.2.12:8443" namespace="monitoring" pod="kube-state-metrics-6756b4764d-8lc8v" service="kube-state-metrics" | 10.853s ago |
| https://10.244.2.12:9443/metrics | UP | endpoint="https-self" instance="10.244.2.12:9443" namespace="monitoring" pod="kube-state-metrics-6756b4764d-8lc8v" service="kube-state-metrics" | 1.429s ago |

kubelet (6/6 up) show less

| Endpoint | State | Labels | Last Scrape |
|---|---|---|---|
| https://178.22.68.104:10250/metrics | UP | endpoint="https-metrics" instance="178.22.68.104:10250" namespace="kube-system" service="kubelet" | 22.966s ago |
| https://178.22.68.104:10250/metrics/cadvisor | UP | endpoint="https-metrics" instance="178.22.68.104:10250" namespace="kube-system" service="kubelet" | 14.736s ago |
| https://178.22.70.192:10250/metrics | UP | endpoint="https-metrics" instance="178.22.70.192:10250" namespace="kube-system" service="kubelet" | 22.778s ago |
| https://178.22.70.192:10250/metrics/cadvisor | UP | endpoint="https-metrics" instance="178.22.70.192:10250" namespace="kube-system" service="kubelet" | 18.929s ago |
| https://178.22.71.29:10250/metrics | UP | endpoint="https-metrics" instance="178.22.71.29:10250" namespace="kube-system" service="kubelet" | 6.937s ago |
| https://178.22.71.29:10250/metrics/cadvisor | UP | endpoint="https-metrics" instance="178.22.71.29:10250" namespace="kube-system" service="kubelet" | 12.334s ago |

node-exporter (3/3 up) show less

| Endpoint | State | Labels | Last Scrape |
|---|---|---|---|
| https://178.22.68.104:9100/metrics | UP | endpoint="https" instance="178.22.68.104:9100" namespace="monitoring" pod="node-exporter-dgq2d" service="node-exporter" | 21.761s ago |
| https://178.22.70.192:9100/metrics | UP | endpoint="https" instance="178.22.70.192:9100" namespace="monitoring" pod="node-exporter-h2jkr" service="node-exporter" | 5.001s ago |
| https://178.22.71.29:9100/metrics | UP | endpoint="https" instance="178.22.71.29:9100" namespace="monitoring" pod="node-exporter-3sxx8" service="node-exporter" | 5.765s ago |

prometheus-k8s (2/2 up) show less

| Endpoint | State | Labels | Last Scrape |
|---|---|---|---|
| http://10.244.0.14:9090/metrics | UP | endpoint="web" instance="10.244.0.14:9090" namespace="monitoring" pod="prometheus-k8s-1" service="prometheus-k8s" | 5.938s ago |
| http://10.244.2.13:9090/metrics | UP | endpoint="web" instance="10.244.2.13:9090" namespace="monitoring" pod="prometheus-k8s-0" service="prometheus-k8s" | 422ms ago |

prometheus-operator (1/1 up) show less

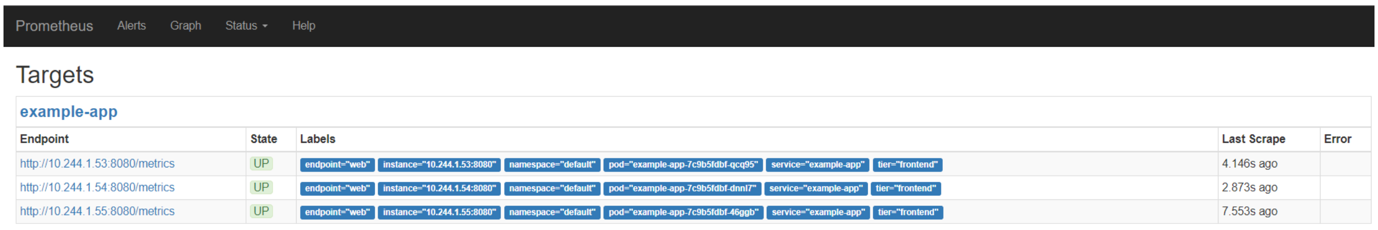
| Endpoint | State | Labels | Last Scrape |
|---|---|---|---|
| http://10.244.2.23:8080/metrics | UP | endpoint="http" instance="10.244.2.23:8080" namespace="monitoring" pod="prometheus-operator-8b7fc7d8f-q8rm7" service="prometheus-operator" | 26.574s ago |

Figure 5.2: Monitoring targets of Kubernetes cluster

Monitor to the prometheus-k8s instance through an appropriate LabelSelector. As we can see from the picture, a working monitoring system within kubernetes integrating with Prometheus has been activated and several targets have been linked to the Prometheus UI dashboard. From these targets, the Prometheus instance for Kubernetes cluster is able to track all the cluster metrics. Following the nodeport forward address, we are able to reach Prometheus UI at node port 30900.

Another Prometheus instance is created to track the custom metrics which are exposed by our developed data processing application. The new generated Prometheus instance dashboard could be reach at port 30100. The application target is presented in Figure 5.3

By sending a request to the Public API Server, the containerized data processing application will automatically encrypt the data string in the received JSON file. The processed data could be viewed by forwarding the container port to the external port.

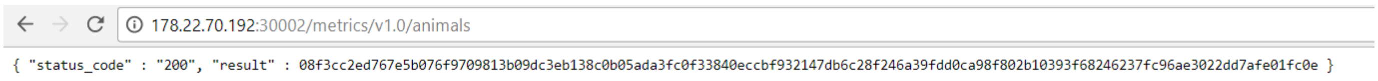


Targets

example-app

| Endpoint | State | Labels | Last Scrape | Error |
|---|---|---|---|---|
| http://10.244.1.53:8080/metrics | UP | endpoint="web" instance="10.244.1.53:8080" namespace="default" pod="example-app-7c9b5fdbf-qcq95" service="example-app" tier="frontend" | 4.146s ago | |
| http://10.244.1.54:8080/metrics | UP | endpoint="web" instance="10.244.1.54:8080" namespace="default" pod="example-app-7c9b5fdbf-dnnl7" service="example-app" tier="frontend" | 2.873s ago | |
| http://10.244.1.55:8080/metrics | UP | endpoint="web" instance="10.244.1.55:8080" namespace="default" pod="example-app-7c9b5fdbf-46ggb" service="example-app" tier="frontend" | 7.553s ago | |

Figure 5.3: Monitoring targets of containerized application



```
178.22.70.192:30002/metrics/v1.0/animals
```

```
{ "status_code" : "200", "result" : 08f3cc2ed767e5b076f9709813b09dc3eb138c0b05ada3fc0f33840eccbf932147db6c28f246a39fdd0ca98f802b10393f68246237fc96ae3022dd7afe01fc0e }
```

Figure 5.4: Processed data

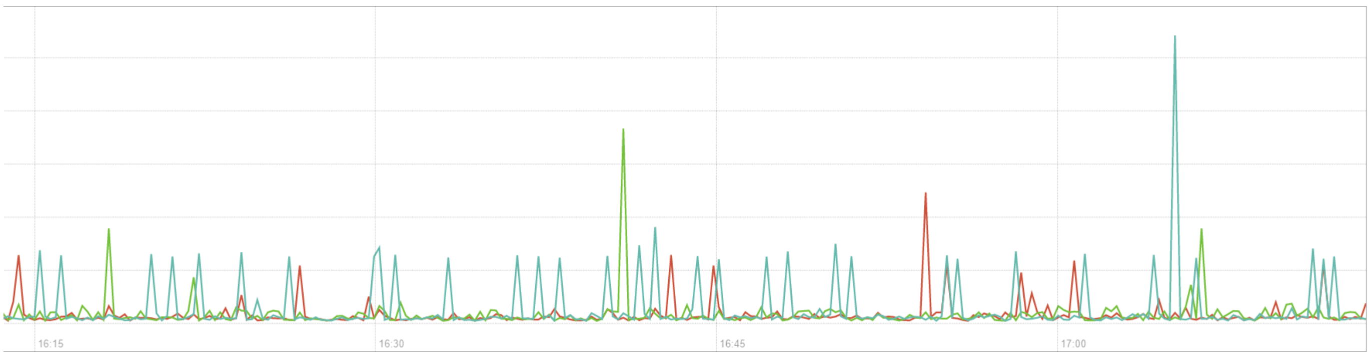


Figure 5.5: Response time for data processing

Figure 5.4 is what we received:

We have deployed three pods inside one node to run our containerized application. Figure 5.5 presents the response time for each pods in one hour.

# Chapter 6

# Conclusion and Future

During the research of this thesis, a improving DaaS framework based on kubernetes for data-intensive applications has been implemented. The platform is deployed in three distributed virtual machines in the Cloud. The adoption of this framework facilitates DIA developer to easily achieve desired data with functionality and non-functionality requirements of the applications.

By adopting the abstraction layer provided by VDC, applications deployed through the platform can access the specific data requested under personalized functional and non-functional constraints in distributed locations. Computation movement is supported by the adoption of VDC that can be easily moved around between the cloud side and the edge side. The criteria of enacting computation movement is decided by an enactor module which is driven by some customized monitoring data about the application behavior and the data source state, providing by the deployed monitoring system in kubernetes.

To validate this architecture, we have deployed a Kubernetes cluster consisting with three nodes, one for master node and others for slave node. Based on this cluster, a customized monitoring system is deployed for retrieving the corresponding monitoring data, including the environment data and the VDC runtime data. In order to make the VDC have the capability to do a further processing on the received data, a containerized webservice application is developed to do a heavy computation on the data. We define three custom metrics for this VDC (response time, total HTTP request number and HTTP failure request number). All the designed custom metrics can be received by our customized monitoring system. The deployed customized monitoring system is responsible for the monitoring data collection in different levels and the monitoring data is essential for the following movement decisions made by VDM.

Even we have already implemented a personalized DaaS platform to provide supporting for different data-intensive applications in a fog environment, there are still some future work underprocessing.

1. In our developed platform, all the nodes we used are supported by Cloud Service Provider which provides several Virtual Machines rely on the cloud data-center. However, in order to manage data and computation movements among a Fog Computing environment, the VDC nodes should be able to easily deployed on resources which can live either on the edge or in the cloud. Since Kubernetes provides an effective extension to ARM devices, the designed Kubernetes cluster can be feasible to include some Single Board Computers (e.g., Raspberry Pi) as the edge side in a Fog environment. In the future, a Kubernetes cluster should be set up in a mix architecture to validate the data movement action in Fog Computing.

2. Since the monitoring data is ready, we still need to decide whether to enact data and computation movement strategies. The decision is made by VDM which is the controller for the corresponding VDCs. In Kubernetes, pod movement is driven by kube-controller which located inside the master node. If we want to force some pods transmitting to a specific node, the VDM should acquire the movement privilege.

3. Through we have already designed three custom metrics for our monitoring system, it could be limited when facing a huge number of DIA. By collecting the characteristics from a reasonable number of Data Intensive Applications, more metrics should be designed and tested.

# Bibliography

[1] Jatin Aneja. *Container Technologies Overview.* https://dzone.com/articles/container-technologies-overview. Accessed: 2018-03-16.

[2] Hind Bangui et al. "Multi-Criteria Decision Analysis Methods in the Mobile Cloud Offloading Paradigm". In: 2014.

[3] Daniel Berman. *Kubernetes Monitoring: Best Practices, Methods, and Existing Solutions.* https://logz.io/blog/kubernetes-monitoring/.

[4] R. E. Bryant. "Data-Intensive Scalable Computing for Scientific Applications". In: *Computing in Science Engineering* 13.6 (Nov. 2011), pp. 25–33. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.73.

[5] Cinzia Cappiello et al. "Utility-Driven Data Management for Data-Intensive Applications in Fog Environments". In: *Advances in Conceptual Modeling.* Ed. by Sergio de Cesare and Ulrich Frank. Cham: Springer International Publishing, 2017, pp. 216–226. ISBN: 978-3-319-70625-2.

[6] CISCO. "Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are". In: (2015).

[7] *github site: Prometheus.* https://github.com/prometheus/prometheus. Accessed: 2018-03-25.

[8] gksinghiet. *A comparison of Container vs. Virtual Machine.* https://www.experts-exchange.com/articles/18440/A-comparison-of-Container-vs-Virtual-Machine.html. Accessed: 2018-03-16.

[9] OpenFog Consortium Architecture Working Group. *OpenFog Architecture Overview (February 2016).* https://www.openfogconsortium.org/ra.

[10] J. Huai, Q. Li, and C. Hu. "CIVIC: a Hypervisor based Virtual Computing Environment". In: *2007 International Conference on Parallel Processing Workshops (ICPPW 2007)*. Sept. 2007, pp. 51–51. DOI: 10.1109/ICPPW.2007.28.

[11] R. Srinivasan Jayalakshmi D. S. and K. G. Srinivasa. *Data Intensive Cloud Computing: Issues and Challenges*. 2015. DOI: 10.4018/978-1-4666-8676-2.

[12] A. M. Joy. "Performance comparison between Linux containers and virtual machines". In: *2015 International Conference on Advances in Computer Engineering and Applications*. Mar. 2015, pp. 342–346. DOI: 10.1109/ICACEA.2015.7164727.

[13] Preethi Kasireddy. *A Beginner-Friendly Introduction to Containers, VMs and Docker*. https://medium.freecodecamp.org/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b. Accessed: 2018-03-16.

[14] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2016. ISBN: 978-1-4493-7332-0. URL: http://shop.oreilly.com/product/0636920032175.do.

[15] Yun Liu, Qi Wang, and Hai-Qiang Chen. "Research on IT Architecture of Heterogeneous Big Data". In: 18.2 (June 2015), pp. 135–142. ISSN: 1560-6686. DOI: 10.6180/jase.2015.18.2.05.

[16] R. Morabito, J. Kjällman, and M. Komu. "Hypervisors vs. Lightweight Virtualization: A Performance Comparison". In: *2015 IEEE International Conference on Cloud Engineering*. Mar. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[17] *Oracle VM User's Guide Release 3.0 for x86*. https://docs.oracle.com/cd/E20065_01/doc.30/e18549/intro.htm. Accessed: 2018-03-16.

[18] Pierluigi Plebani et al. “Moving Data in the Fog: Information Logistics with the DITAS Cloud Platform”. In: 2017, pp. 1–7.

[19] Gerald J. Popek and Robert P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”. In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: http://doi.acm.org/10.1145/361011.361073.

[20] M. N. O. Sadiku, S. M. Musa, and O. D. Momoh. “Cloud Computing: Opportunities and Challenges”. In: *IEEE Potentials* 33.1 (Jan. 2014), pp. 34–36. ISSN: 0278-6648. DOI: 10.1109/MPOT.2013.2279684.

[21] Muhammad Shiraz. “A Lightweight Distributed Framework for Computational Offloading in Mobile Cloud Computing”. In: 2014.

[22] Stephen Soltesz et al. “Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors”. In: *SIGOPS Oper. Syst. Rev.* 41.3 (Mar. 2007), pp. 275–287. ISSN: 0163-5980. DOI: 10.1145/1272998.1273025. URL: http://doi.acm.org/10.1145/1272998.1273025.

[23] Olivier Terzo et al. “Data As a Service (DaaS) for Sharing and Processing of Large Data Collections in the Cloud”. In: *Proceedings of the 2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*. CISIS ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 475–480. ISBN: 978-0-7695-4992-7. DOI: 10.1109/CISIS.2013.87. URL: http://dx.doi.org/10.1109/CISIS.2013.87.

[24] *What is Data as a Service (DaaS) - Definition from Techopedia.* https://www.techopedia.com/definition/28560/data-as-a-service-daas.

[25] Qi Zhang, Lu Cheng, and Raouf Boutaba. “Cloud computing: state-of-the-art and research challenges”. In: *Journal of Internet Services and Applications* 1.1 (May 2010), pp. 7–18. ISSN: 1869-0238. DOI: 10.1007/s13174-010-0007-6. URL: https://doi.org/10.1007/s13174-010-0007-6.