Evolving Geospatial Applications: From Silos and Desktops to Microservices and DevOps

by

Bing Gao
B.Sc., Dalian JiaoTong University, 2007

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Evolving Geospatial Applications: From Silos and Desktops to Microservices and DevOps

by

Bing Gao
B.Sc., Dalian JiaoTong University, 2007

Supervisory Committee

Dr. Yvonne Coady, Supervisor
(Department of Computer Science)

Dr. Neil Ernst, Departmental Member
(Department of Computer Science)

ABSTRACT

The evolution of software applications from single desktops to sophisticated cloud-based systems is challenging. In particular, applications that involve massive data sets, such as geospatial applications and data science applications are challenging for domain experts who are suddenly constructing these sophisticated code bases.

Relatively new software practices, such as Microservice infrastructure and DevOps, give us an opportunity to improve development, maintenance and efficiency for the entire software lifecycle.

Microservices and DevOps have become adopted by software developers in the past few years, as they have relieved many of the burdens associated with software evolution. Microservices is an architectural style that structures an application as a collection of services. DevOps is a set of practices that automates the processes between software development and IT teams, in order to build, test, and release software faster and increase reliability. Combined with lightweight virtualization solutions, such as containers, this technology will not only improve response rates in cloud-based solutions, but also drastically improve the efficiency of software development.

This thesis studies two applications that apply Microservices and DevOps within a domain specific application. The advantages and disadvantages of Microservices architecture and DevOps are evaluated through the design and development on two different platforms—a batch-based cloud system, and a general purpose cloud environment.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **viii**

Acknowledgements **x**

Dedication **xi**

1 Introduction **1**

1.1 Virtual Machines 3

1.2 Docker Containers 3

1.2.1 Kubernetes 5

1.2.2 Mesos 6

1.3 Microservices 8

1.4 DevOps 10

1.5 Continuous Integration/Delivery 11

1.6 Serverless Framework 11

1.7 Motivation 12

1.8 Contributions 12

1.9 Thesis Overview 13

2 Background and Related Work **14**

2.1 Background 14

2.2 Related Work 16

2.3 Summary 18

3 Geospatial Images Processing Application with Big Data 20
3.1 Background 20
3.1.1 Images Processing On a Single Machine 24
3.1.2 Data Set 25
3.1.3 Compute Canada 25
3.1.4 Singularity Container 25
3.1.5 Polymer Algorithm 26
3.2 Architecture of GIPA 27
3.3 Implementation Details 29
3.4 DevOps Plus Container Highlight 31
3.5 Summary 31

4 Design and Implement a GUI Satellite Imagery Application On AWS 32
4.1 Design Characters 33
4.2 TestKitchen Application 37
4.2.1 Overview 37
4.2.2 Front-end Development 39
4.2.3 Back-end Development 40
4.2.4 Architecture of TestKitchen 41
4.3 Implementation Details 42
4.4 Highlight and Future Goal 43
4.5 Summary 44

5 Evaluation, Analysis and Comparisons 46
5.1 Lightweight Versus Full Virtualization 47
5.2 Testbed Architecture 47
5.3 Kubernetes VM on VMware vSphere 48
5.4 Systematic Evaluation: VM Launch Times 49
5.5 Kubernetes with Minikube 50
5.5.1 Setup: Kubernetes with Minikube 51
5.6 Systematic Evaluation: Container Launch Times 53
5.7 Test Plan For Example Applications 56
5.7.1 Spectral GIPA On Compute Canada 57

5.8 GIPA Result Analysis . . . 57
5.8.1 Binning Analysis . . . 59
5.9 TestKitchen On AWS . . . 60
5.9.1 Docker on Mesos . . . 60
5.9.2 Mesos On AWS . . . 60
5.10 Development and Maintenance Experiences . . . 62
5.11 Summary . . . 63

6 Conclusions and Future Work **65**
6.1 Contributions . . . 66
6.2 Limitations . . . 67
6.3 Microservices and DevOps Successful Factors . . . 67
6.4 Summary . . . 68
6.5 Future Work . . . 70

A Appendix **72**
A.1 Playground Dockerfile . . . 72
A.2 Docker-Compose File . . . 75
A.3 Singularity Recipe . . . 77
A.3.1 Polymer Image . . . 77
A.3.2 SNAP GPT Command Line Utility Image . . . 78
A.4 Polymer Docker Dockerfile . . . 79

Bibliography **81**

List of Tables

Table 5.1 Compute Canada Machine Type Specifications 47
Table 5.2 Amazon AWS Machine Type Specifications 47
Table 5.3 Measuring Container Launch Time 56
Table 5.4 Measuring Duplication Rate on Compute Canada 59
Table 5.5 Comparison of Different Platforms 63

List of Figures

Figure 1.1 Cloud Categories 2
Figure 1.2 Container vs VM Architecture 5
Figure 1.3 The Architecture of Kubernetes 6
Figure 1.4 The Architecture of Mesos 7
Figure 1.5 Monolithic vs Microservices 9
Figure 1.6 DevOps 10

Figure 3.1 SNAP Open Raw Image 22
Figure 3.2 SNAP Open Polymer Output 23
Figure 3.3 Raw Image vs Polymer Output 23
Figure 3.4 Single Machine Artifact 24
Figure 3.5 Cluster Artifact Diagrams 28
Figure 3.6 Interaction Diagrams 29

Figure 4.1 Test Kitchen Structure 38
Figure 4.2 TestKitchen Home Page 39
Figure 4.3 Map Reduce Diagram 41
Figure 4.4 TestKitchen DevOps Work Flow 44

Figure 5.1 VMware vSphere Cloud Platform 49
Figure 5.2 A Simple 4-Nodes Kubernetes Cluster 50
Figure 5.3 Dashboard on VMware vSphere Cloud Platform for Kubernetes Cluster 51
Figure 5.4 Setup for Minikube 52
Figure 5.5 Virtual Machine Launch Time in a VMware vSphere Cloud Platform 53
Figure 5.6 Starting Two Nginx Pods 53
Figure 5.7 Kubernetes Dashboard for Two Nginx Pods 54
Figure 5.8 Container Launch Time 55

Figure 5.9 Single Machine vs Cluster 58

Figure 6.1 Google Trends for Serverless and Microservices 68
Figure 6.2 From DevOps to Clouds 69

ACKNOWLEDGEMENTS

I would like to sincerely thank:

my wife, Fei for supporting me in the low moments.

Supervisor Yvonne Coady, for her guidance and support throughout this study and specially for her confidence in me.

Dr. Issa Traore, for his time and help in my thesis examining committee.

DEDICATION

I dedicate this to my mother and father who have always loved me unconditionally and whose good examples have taught me to work hard for what I aspire to achieve.

Chapter 1

Introduction

Cloud computing is now embraced by industrial companies and communities. Everyday billions of users access cloud services provided by cloud infrastructure providers, like Google, Amazon and so on. People are using online services everyday. Most of them use these services not only for their private life, but also their public work. For example, they can share information with friends on social media, and collaborate with colleagues and clients through cloud documents. It is hard to project this scene even a few years ago.

In order to guarantee that every user in different regions can be provided with the same quality of service (QoS). In other words, the service providers need to build robust and resilient services, either by building their own data centers or using public clouds. An increasing focus has been shifted towards public clouds. Many cloud solutions are proposed, for instances, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), Container as a Service (CaaS) and Function as a Service (FaaS). Figure 1.1 displays a catalog of these cloud solutions.

Although cloud computing does not equal to virtualization, most of cloud platforms use virtualization techniques in their platform. Because of virtualization, cloud providers can maintain simplicity and keep their customers away from manually manipulating the physical resources. Virtualization can provide a multi-tenant environment that improves the total utilization of hardware. Overall cloud providers and customers get a win-win result from virtualization. High performance helps tenants to provide better service to their clients, and eventually, cloud providers get more income from their tenants.

From Virtual Machines (VMs) to containers, and IaaS to SaaS, the latest attempt is to apply Serverless frameworks. The trend is to handover more and more responsibilities to cloud providers. Cloud users can focus on their own business. Then it comes the question: how to fully utilize the cloud servers' capacity to secure the best investment return?

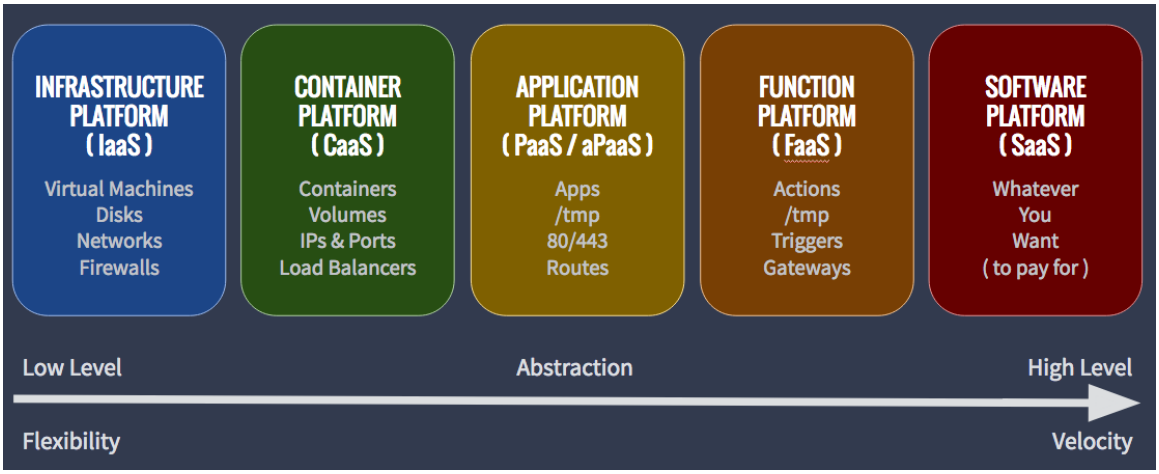

Figure 1.1: Cloud Categories
Source: [6]

If there is a way telling potential users which architecture or technology is better, that would answer the above question. Fortunately, software communities do have a method widely used to evaluate the performance of cloud platforms. Bench-marking is relatively widely applied and accepted by the computer science community. Bench-marking measures many metrics that are collected from products, which are geospatial applications in our case. In this thesis, I will show how to design cloud-native applications and what I have achieved from the design. Later I will demonstrate the performance of these applications on different cloud platforms.

In the rest of chapter 1, I will introduce methodologies, architectures and frameworks that I have used in this work.

1.1 Virtual Machines

When people think about virtualization, the first thing comes into their mind probably is VMs. A Virtual Machine is essentially an emulation of a real computer. It has dedicated hardware, namely CPU, memory and disk. VM has been widely used in the past decades. The hypervisor is a key component to make sure the VM work. There are two types of hypervisor [54], Type 1 and Type 2. The most significant difference between them is that Type 1 is installed on a bare metal server. On the other hand, Type 2 needs to be run on a hosted Operating System (OS). Type 1 has less overhead. It doesn't have to load an underlying OS. Type 2 hypervisor relies on the host machine's pre-existing OS to manage calls to CPU, memory, storage and network communication. Therefore, Type 2 hypervisor have more limitations. Google Cloud is using Type 1 Kernel Virtual Machine (KVM) [48] as their hypervisor. Amazon Web Services (AWS) [23] recently has announced that they are planning a move to KVM from Xen [27]. KVM is a virtualization infrastructure for the Linux kernel that allows Linux to act as a Type 1 hypervisor. It supports a Linux distributions to run as an unmodified guest OS.

On the other hand, VM overhead is heavy because it emulates a full OS. It requires the hypervisor to allocate a number of virtual CPUs (vCPUs), disks and RAMs. All the resources are exclusive. As a consequence, a couple of instances running simultaneously is enough to bog down a server with its overhead. As a result, the software communities are still working on better solutions, which can dynamically allocate resources from the hardware directly.

1.2 Docker Containers

VM was a great innovation. Despite that, some of its disadvantages are bothering the communities. To give an example, VM images are not portable. If users want to switch to a different hypervisor, they need to re-do every step in the new hypervisor. It's also hard to automate maintenance tasks. Its maintenance requires a lot of human intervention and more to list. For that reason, people are looking for lightweight virtualization solutions. Container technology is quickly emerging in recent years. It is not a new idea. Linux world has been using a similar concept more than a decade.

Docker container is the one that makes it famous. Before talking about Docker container, let's discuss the container technology first.

Containerization was invented two decades ago (FreeBSD Jails [46] has been around since 2000). This technology did not seem to attract the public attention until Docker [18] (2013) released their first container product. Container basically is a minimized OS. It uses techniques/tools, like chroot, cgroup to implement resources isolation, security etc. Containers can greatly reduce the overhead compared to VMs. Figure 1.2 shows the VMs and Containers architectures. A container shares resources with other container instances. This key improvement makes it easy to build and deploy containerized applications. Containers allow users to deploy Microservices stacks locally, which can incredibly speed up development and deployment cycles [52].

Docker container immediately attracts users' attention. After its release, it has been popular among other container solutions. It has built an ecosystem to support the upstream and downstream users in recent years, and, it is still rapidly evolving.

Docker container can help to set up modular systems. Its portability feature allows users to build images in one machine and run everywhere where Docker engine is installed. Essentially, many applications leverage lightweight virtualization, in particular, Docker container, which compose themselves as Microservices.

Rather than running a full OS on a hypervisor, container can run many instances in an existing OS. By doing this, container actually acts as a process in the hosted OS. This involves adding the container ID to the process, and attaching new access control checks to every system call. Thus containers can be viewed as another level of access control in addition to user and group permission systems. In practice, Linux uses a more complex implementation. Whereas containers are just a subset of kernel processes, which maintains the core of an OS. Containers are extremely fast to boot. They also consume less resources, for example, smaller per-instance memory footprints which results are higher density of containers in a hosted system. However, a container is less secure. Because it shares its host with other container instances.

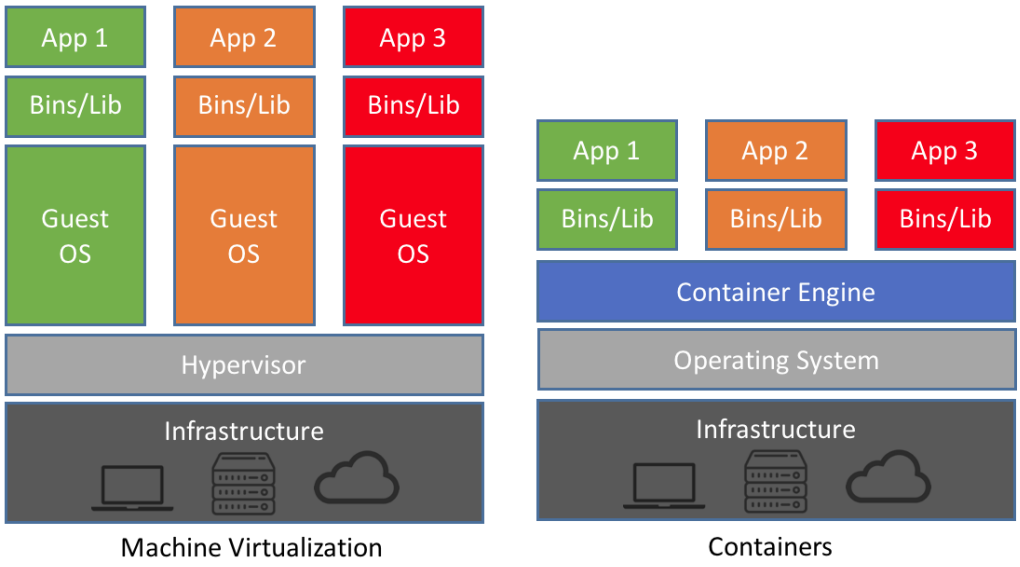

Figure 1.2: Container vs VM Architecture
Source: [4]

1.2.1 Kubernetes

It is easy to manage individual or a small set of containers. But a production system merely consists of only a handful of containers. More often, a company has many applications. Furthermore, one application contains a number of containers. Eventually, when aggregating together, a production system can have thousands of instances running in their clusters. It is unimaginable that any team is able to properly manage this large volume containers manually. People have found that they need a container management system that can mitigate from the heavy operation burden. Kubernetes and Mesos are two examples.

According to Kubernetes website, "Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications." Kubernetes was built by Google based on their experiences running containers in production systems using an internal cluster management system called Borg [63]. The architecture of Kubernetes, which relies on Google's experiences, is shown in Figure 1.3. A Kubernetes cluster consists of at least one master node and multiple worker nodes. The master is responsible for exposing the application endpoint, scheduling the deployments and managing the overall cluster. Worker node runs a container

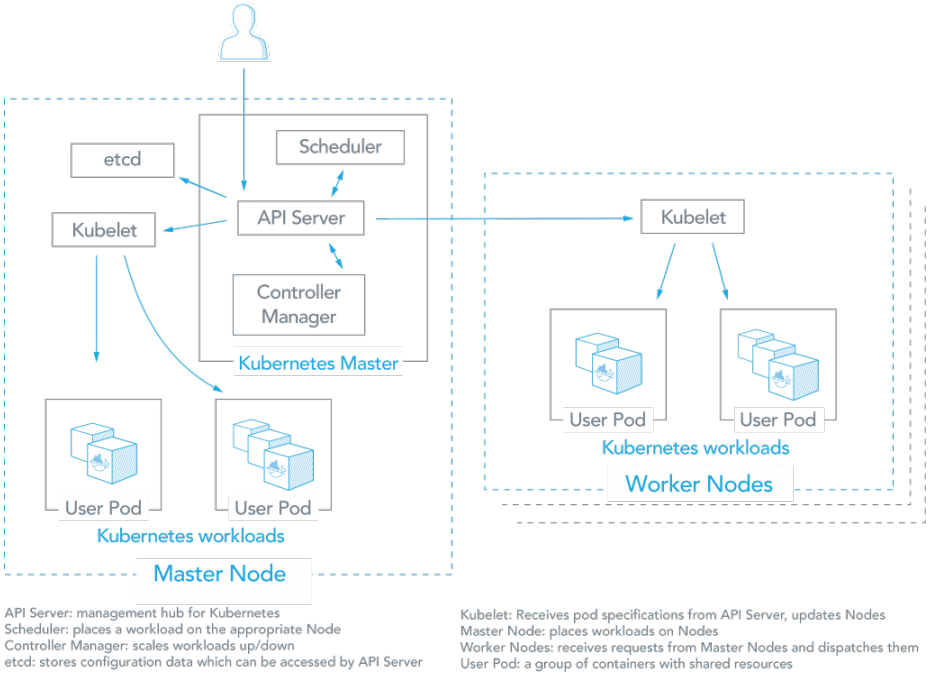

Figure 1.3: The Architecture of Kubernetes
Source: [20]

runtime, such as Docker, along with an agent that communicates with the master.

Major development of Kubernetes began at Google in 2014. Google had a long history of using container technology by then. In this paper [32], Google revealed some background information about their internal container development in last decade. The Kubernetes is inherited from its predecessors Borg and Omega. Kubernetes is not the first open-source project for container management, not even the first cluster manager developed at Google. Ever since its first release, it has succeeded in winning the attention of the software community. Open-source developers have invested lots of effort in Kubernetes at the time of this thesis is written. Over a decade of Google's experiences in cluster management field, Google has unique experiences in handling large volume traffic. Google shares these lessons to the open-source communities.

1.2.2 Mesos

Mesos [40] is a centralized fault-tolerant cluster manager, distributing workloads across a cluster of slave nodes. It provides efficient resource isolation and share across the cluster. It was initially developed at University of California at Berkeley.

Later on they collaborated with Apache Foundation to make this project open source.

Mesos system consists of the parts below:

1. *Masters*: Mesos master is used to manage the slaves. It also collects information about resources, and tasks to be executed from one entity and passed on to the other entity.
2. *Slaves*: These are servers that actually run the tasks.
3. *Frameworks*: also known as a Mesos application. It has two important components: scheduler and executor. The scheduler, which registered with the master to receive resource offers. It can also accept or reject the offers based on its requirement and algorithm. The executor, which launches tasks on slaves.
4. *ZooKeeper*[44]: Mesos uses ZooKeeper for cluster membership and leader election.

Figure 1.4 gives a very clear picture about Mesos's architecture.

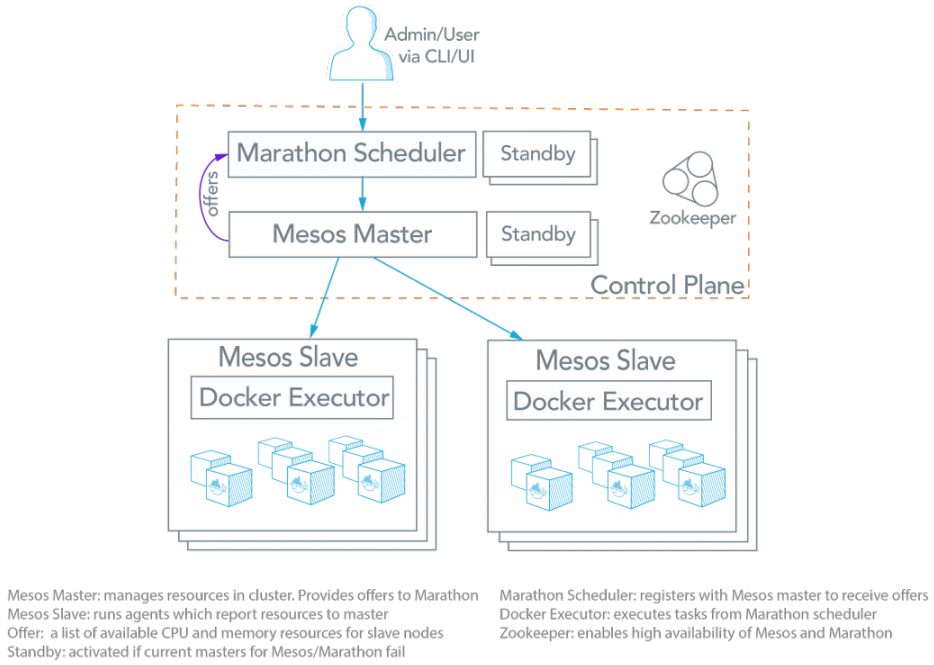

Figure 1.4: The Architecture of Mesos
Source: [20]

Mesos leverages features of modern Linux kernel, like "cgroups", to provide isolation for CPU, memory, I/O, file system, rack locality etc. The key innovation is

resource offer. Mesos introduces a distributed two-level scheduling mechanism called resource offer. Mesos decides how many resources to offer to each framework, while frameworks decide which resources to accept and which computations to run on them.

Marathon

Marathon is a production-grade container orchestration platform for Mesospheres Datacenter Operating System (DC/Operating System) and Apache Mesos. A cluster-wide init and control system for services in cgroups or Docker container. It is a framework for Apache Mesos. Marathon exposes a REST API for managing tasks. Users can also use Marathon to run and manage other frameworks as well. Marathon will receive a resource offer from Mesos. If it accepts the offer, it will provide Mesos information about the task which will be passed on to the Marathon Executor running on a slave.

Comparing to Kubernetes, Mesos is a cluster management system with long history. Its history confers both advantages and disadvantages. Long-running projects like Mesos have more resources, for instance, books, online documents and successful implementations on different companies. However, on the opposite side, the older one was not designed to run containers initially. It has significant overheads if users want to run containers on Mesos. Such as, users need to install Marathon plugin on Mesos first, then, users can manage Docker containers in Mesos. On the other hand, Kubernetes supports Docker containers from the beginning. Besides that, Google added number of features to Kubernetes based on their experiences, which dramatically decrease the management overheads.

1.3 Microservices

Software developers already know traditional monolithic architecture does not work well in the cloud era. From time to time, a successful application will grow. Such an application has a habit of enlarging over time, and eventually, it becomes a huge block, then it will turn into a monolithic hell. Its test and deployment will become extremely slow and error-prone. Developers hesitate to make any change, as it can

break the application unexpectedly. In order to overcome these drawbacks, Microservices is a great alternative for software development teams.

Basically speaking, Microservices is an idea that illustrates software build block that are loosely coupled with each other. In another word, this idea means to split an application into a set of smaller, interconnected services. Services are fine-grained, they are designed to do one thing and do it well. By doing so, users can reduce the complexity of an application [38]. This makes applications easier to understand, develop, test, and more resilient to architecture erosion. It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently. It also allows the architecture of an individual service to mature through continuous refactoring. The outcome helps development team to rapidly, frequently and reliably deliver software. Figure 1.5 displays the different between monolithic and Microservices architecture.

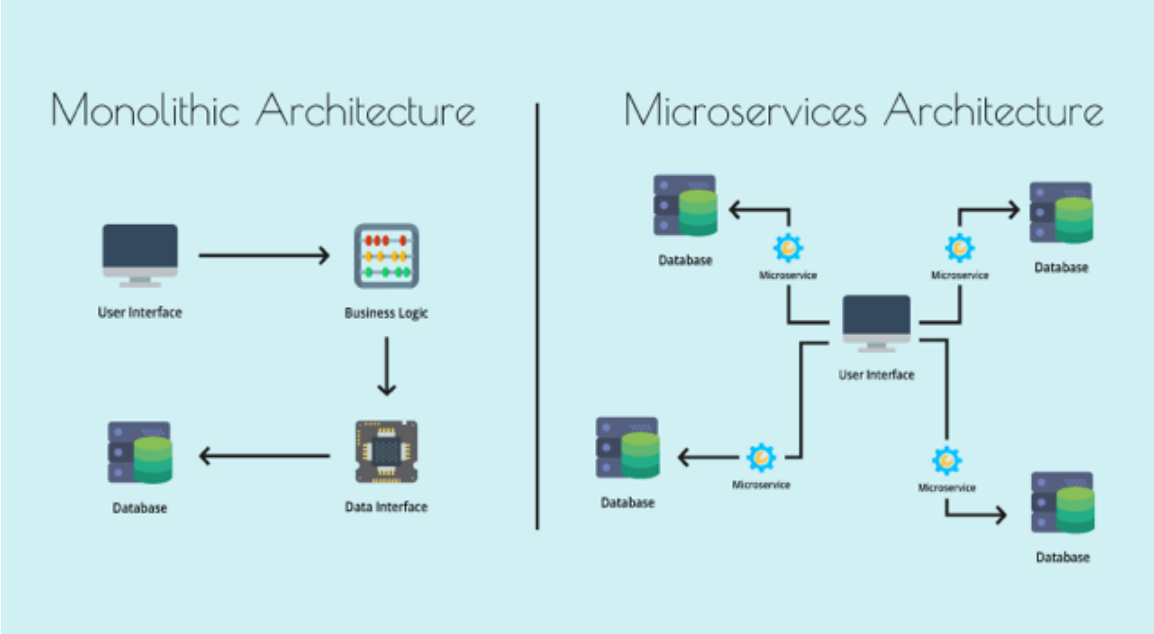

Figure 1.5: Monolithic vs Microservices
Source: [3]

1.4 DevOps

As software communities keep improving traditional monolithic software architectures, they are involving the old software development and maintenance procedure as well. Development and operation teams are separated normally. Each team has different tasks and work scope in their daily work. Nonetheless, this way is no longer working in this rapidly developing world. Companies want to increase the frequency of releases and improve the quality of deployments. At the same time, companies also want to keep their system robust, meanwhile, introducing innovation and increasing its risk-taking capability. Based on the experiences researchers have obtained from Waterfall and Agile model, the concept of DevOps is founded on building a culture of collaboration between teams that historically functioned in a relative split.

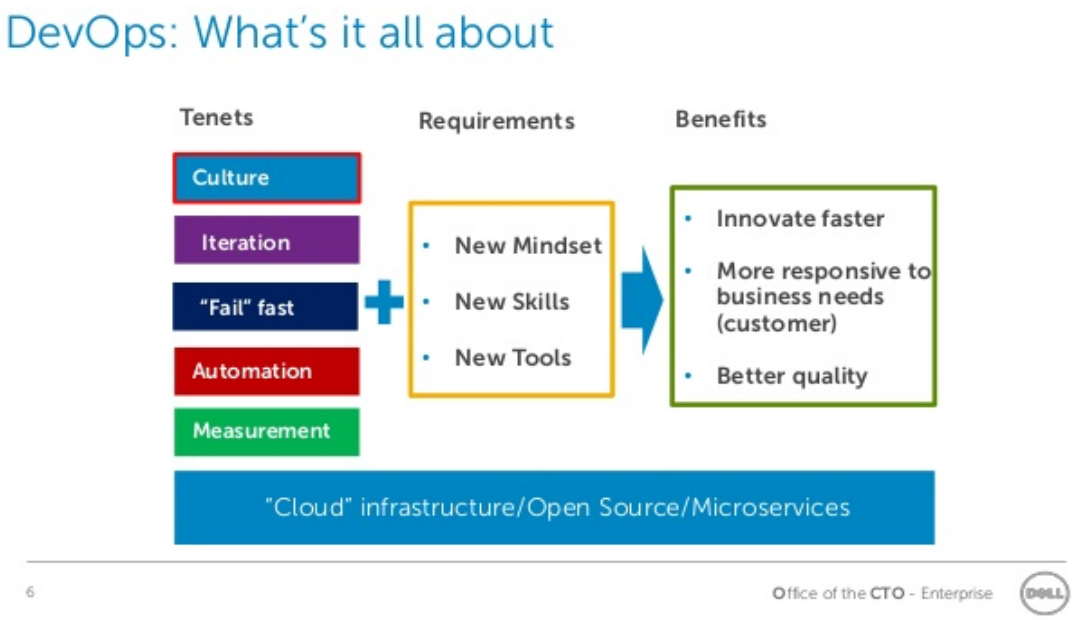

Figure 1.6: DevOps
Source: [1]

DevOps is a set of practices that automates processes between software development and maintenance teams. So that they can build, test, and release software faster and more reliable. The idea of DevOps is more like a collection of practices. Figure 6.2 shows a clear picture of DevOps. Everyone can pick suitable ones for their own project. The core idea behind DevOps is clear: removing the barrier between

developers and maintenance team members, who writes the code also need to take care of it after deployed to a production system [34].

1.5 Continuous Integration/Delivery

One of the key elements to practice DevOps is automation. Without automating the development and deployment procedures, software teams can not enjoy DevOps benefits. Accordingly, practicers need tools/approaches to fulfill the automation. Continuous Integration (CI) is a software development approach where members of a team integrate their work regularly leading to multiple integrations per day. Each integration test is verified by an automated build to detect integration problems as quickly as possible [37].

Software communities have learned experiences by practicing CI in their work. Some people extend this concept further. Continuous Delivery [43] is a software development discipline that enables on demand deployment of software to any environment. With Continuous Delivery, software delivery life cycle will be automated as much as possible. Microservices leverage techniques like Continuous Integration and Continuous Deployment and embrace DevOps.

1.6 Serverless Framework

Except for some extreme cases, no single application receives non-stopping requests from users. For both private data centers or public cloud platforms, application owners need to pay bills once their servers start running. The application owners waste their money when there is no request at all. As a cloud market leader, Amazon noticed this issue through their customers feedback. It proposed a new way to help their customers to reduce their costs. This new way is called the Serverless framework [26]. The core idea is that users pay their bills based on the actual amount of resources consumed by their applications, rather than charging on pre-purchased units. This new billing strategy has gained strong support from the industry. All mainstream cloud platforms, such as Google, Amazon, Microsoft and IBM supply a similar ser-

vice now. It also has another name called Function as a Service (FaaS).

Currently, there is no necessity to change all codes in order to adapt to the Serverless. Hence, the best way is in conjunction with existing applications deployed in Microservices styles. The Serverless architecture can save costs dramatically as users only pay what their applications actually consume. Moreover, users do not need to provision their servers. This can help many small teams if they do not have enough resources.

1.7 Motivation

Customers want to gain performances as good as their own servers when deploying to cloud platforms. As public cloud providers usually provide fixed units of resources (CPU cores, Disk, Bandwidth and RAM) for their customers. The cloud users can get a better performance by renting more powerful nodes, but this is not the only option. Another way is to optimize their applications so that they consume fewer resources.

How to design and develop a software that can reach this goal? How to fully utilize cloud computing resources? These interesting questions have driven us to design and implement cloud-native applications.

1.8 Contributions

In this work, I used Kubernetes to analyze microbenchmarks for the overheads of containers. I tested Kubernetes scalability by using our scripts to increase system load.

Then, I designed and deployed GIPA prototype on Compute Canada, accelerate data processing time from months to hours. GIPA process a large volume of geospatial image data using Singularity container.

At the end, I designed and deployed a generalized, multi-algorithm, tiled system with an industry stakeholder. At the same time, I established the role of Microservices and DevOps during the development and deployment of TestKitchen.

1.9 Thesis Overview

This thesis studies the use of Microservices pattern and DevOps practices to do software applications development and implementation, and the performance of these applications on cloud platforms. The main structure of the thesis is listed below:

Chapter 2 describes background and state-of-the-art work in the software communities.

Chapter 3 demonstrates an Geo-spatial Application running on Compute Canada.

Chapter 4 gives details of our system design, its methodology, the procedures involved, and example applications that I have designed to prove Microservices and DevOps concepts.

Chapter 5 is where the experiments and the methodology used for design and develop geospatial applications is fully described. As well as evaluation of the result data I have obtained from the experiments.

Chapter 6 gives conclusions and future work.

Chapter 2

Background and Related Work

Clouds are discussed everywhere in the software communities. Companies are rapidly moving their infrastructure from private data centers to public clouds. The ultimate goal is to reduce costs and improve productivity at the same time. Cloud computing is an active area in development and research. This chapter briefly describes some background information and related work.

2.1 Background

Public clouds have been overwhelmingly popular in the past decade. With a public cloud solution, start-up companies and researchers can focus on their core businesses or ideas. Whereas, the low-level details of cloud platforms are hidden by the cloud providers, such as Amazon Web Services (AWS) [23] and Google cloud platform [19].

Cloud computing is trying to present everything as a service. The service model brings many financial benefits. More specifically, there are different models that users can choose, such as Infrastructure as a service (IaaS), Platform as a service (PaaS), Software as a service (SaaS), and Serverless computing/Function as a service (FaaS). Since virtualization is invented, it has been about the isolation of resources. Virtualization solutions are trying to create a virtual version of a resource. It could be a CPU, a memory, a network device or even an Operating System (OS) where a hypervisor separates the resource into one or more execution environments.

A hypervisor is a tool to manage hardware and to maintain a cloud that runs as expected. In order to cut the operation costs, companies have to make some tradeoffs in efficiency of hardware. When users install software on bare metal hardware, it can use all the resources. On the other hand, virtualization means users need to allocate some system resources, such as memory and CPU to the hypervisor. This part of overhead would be significant if users manage a large size cloud server. Hence, many people are trying to reduce the overheads to make virtualization solutions more efficient.

This is where container comes into context. That is the reason container technology is emerging in the past few years. It is a new milestone in IT industry. Containers are a software component enables the execution of applications in an isolated environments. Currently, the Big Data and the Artificial Intelligence(AI) technology are overwhelmingly popular in the software fields. These types of software release their features frequently. The traditional way of developing and maintaining software does not work here any more. Microservices and DevOps have become popular. Because these new methods can overcome some defeats that embedded in the old software development flow. By combing with container technology, Microservices and DevOps completely change the way how software teams develop software in cloud era.

In the paper [62], the authors have described a cloud-native application which is implemented by using Microservices and DevOps. This example application demonstrates the principles of cloud-native application. As a cloud native application, it must embrace resilience and elasticity. The traditional monolithic software does not fit in the rapidly changing world. Monolithic software limits the application applying of new technologies and the integration of new practices to the existing software. Microservices is a new idea to develop the cloud-native software. It separates an application into many small, independent services. Each service is a small application that has its own hexagonal architecture consists of business logic along with various adapters. A service communicates through a designed API with each other, whereas an individual service can update itself without impacting the whole application at all if it does not change its API endpoint.

2.2 Related Work

There is plenty of work to focus in the area of cloud computing and its variations. Especially in recent years, researchers and software communities are working on how to fully optimize cloud servers and secure a maximum business return. In order to achieve the goal, there are many methodologies and technologies proposed by the researchers in the communities. Today Microservices, DevOps and Serverless architecture and their paradigms are the hottest topics in cloud computing.

VMs was originally introduced by IBM in their mainframe machine. Though VMs were only widely expended after VMware [56] was reinvented on X86 platform in late 1990. Many papers have studied the performance penalty of virtualization technologies. Xen [27], VirtualBox [22] and KVM [48] brought VMs to the open source world in the 2000s. Researchers have been working on VMs overhead elimination for a long time. The good news is that the overhead of VMs was only initially high. However, it has been steadily reduced over recent years due to hardware and software optimization.

Leveling virtualization in OS has a long history as well. The Unix chroot feature has been used to implement rudimentary "jails" [46] in FreeBSD. For example, in work [42], authors introduced jails as an isolation mechanism to run multiple tenants(aka virtual operating systems) in FreeBSD system. It is the prototype of container technology. The Linux system finally supported native containerization starting in 2007 in the form of using kernel namespace and LXC [30] userspace tool to manage it. In the paper [59], authors discussed an alternative solution to replace hypervisor, the container virtualization could reduce overhead and provide sufficient support for isolation and superior system efficiency. Google is a big advocate of container technology. Google has been using container in their internal system for a decade and published detailed information in paper [32]. Google did not reveal it until 2016. Before that, computer science communities knew the container technology through an open source project called Docker. Docker is the first well-known container product.

In paper [35], IBM compared VMs and Linux container performance based on CPU, Memory, I/O and Network. They found that in most cases, container is better

than VM. Yet, they did not compare different container cluster management tools. After Docker released its first public container product and received positive feedback, researchers started to do performance analysis for Docker container. In paper [51], authors gave an outstanding introduction about Docker container. The paper describes the workflow of Docker container and hands-on material to practice Docker on your own machine.

Docker container gives relief from dependency hell by keeping the dependencies contained inside the containers. In spite of that, how to manage these containers become a new challenge. As projects grow, and companies expand, users may face a scenario that they need to run and operate hundreds of instances of containers at a same point. They can not manually control containers anymore. For that reason, a container cluster management system becomes necessary for these container users. Kubernetes [32] was released on 2014. Google shared its internal container cluster management experience. As Google has been practicing container technology for more than a decade, it has collected many first-hand information. Paper [32] describes an evolving path regarding of Kubernetes.

Mesos [40] is another cluster management tool. It was invented by UC Berkeley. Later on, it is released as an open source project. It is a largely deployed cluster management system in many production systems. Container rarely deploys to a production environment without using cluster management systems. Knowing Docker container's performance under the cluster management system is critical. In this article [50], the author compared current mainstream container technologies' performance on different container management system. The result shows that Docker Compose or Swarm is good for a small deployment or for testing used by developers. For large deployment, Kubernetes should be selected. Although it is much more complicated, it brings more benefits to the whole infrastructure.

There are a lot of existing monolithic applications serving customers at this time. Many companies want to migrate their application to a cloud-native architecture, as cloud platform brings many benefits. In the work of [25], the authors described how they converted and refactored an existing application to Microservices architecture.

Microservices, DevOps and containers are very interrelated to each other that

they form an ecosystem. Researchers in IBM did experiments [47] on an IaaS platform to explore how Docker container and Microservices could fully utilize the cloud environment, and they also practiced DevOps in their work. They found some challenges that needed to be solved, such as how to reduce configuration overhead, how to maintain running services state etc. They propose a new prototype to fix these problems.

Evolution is continuing. Severless framework is one of the next step. This model is far more elastic and scaleable than previous platforms [39]. Nevertheless, it is a new thing, so almost no one has a clue what is its best usage scenario. Therefore, paper [26] describes current trends and open problems. The authors discussed what is Serverless framework. How to move your applications to the Serverless and what problems that current platform can and can not handle.

In paper [45], authors at UC Berkeley gave their view about Serverless. They also discussed the current status of Serverless and its limitations. The authors also described what Serverless should become in the future. Especially the new virtualization technology, such like gVisor [16] and Firecracker [15]. gVisor is a new kind of sandbox that can be used to provide secure isolation for containers that is less resource intensive than running a full VM. Firecracker, a new virtualization technology that makes use of KVM. You can launch lightweight micro-virtual machines (microVMs) in non-virtualized environments in a fraction of a second. Both tools are taking advantage of the security and workload isolation provided by traditional VMs and the efficiently allocate resource that comes along with containers.

2.3 Summary

All in all, public clouds will supersede traditionally private-owned data centers eventually. This trend is irreversible. Therefore, in order to reduce the overhead of virtualization solutions, the communities should find a way that can better utilize cloud's compute power. That's why the next generation application should be in cloud-native style. The container is the latest concept that attracts great attention. Its faster boot-time and better resources utilization are the advantages, even though its security is not perfect.

Currently Virtual Machine still remains dominant in cloud solutions. However, container technology has an increasing demand in recent years. Both of these virtualization solutions are trying to mimic real hardwares. It is not an easy decision to select one between them, because both technologies have its pros and cons. What's more, a container itself is not running alone in a cloud environment. Even a small company would run hundreds of container instances. Manually managing them could be extremely difficult. So a container orchestration system becomes indispensable for container users.

The cloud-native application does not just involve new technologies, but also embraces some new software development practices. Typical examples are Microservices and DevOps. Modern software has become sophisticated due to the highly complicated reality. Microservices can decouple components connection, DevOps can improve collaboration across organization by developing and automating a continuous delivery pipeline.

Recently, Serverless framework is discussed widely in software communities. Users who are using public clouds still worry about the high cost in their IT infrastructure in old billing model. If the billing model changes to pay-as-you-go, that means companies are able to avoid unnecessary expense. Frankly speaking, this is a new model that everyone is investigating for the best usage. There are always too many options in the table, users themselves need to figure out which one is the best shot for their own needs.

Chapter 3

Geospatial Images Processing Application with Big Data

In this chapter, I will introduce a Geospatial Image Processing Application (GIPA). GIPA is an example of applying Microservices and the container technology for software development. GIPA is running on Compute Canada platform. First, I will introduce the background of GIPA and the data set to be processed by GIPA. Further, I will introduce the Compute Canada cloud platform and Singularity technology. Moreover, it follows by the architecture and implementation details of GIPA. At last, I will summarize the achievements from the development of GIPA.

3.1 Background

University of Victoria (UVIC) Spectral Lab is located at UVIC Geography Department. The Spectral Lab focuses on investigating the interaction of light energy with organic and inorganic material in ocean waters, and remote sensing of the ocean. Remote sensing technology is advancing at a much faster speed than our traditional understanding of how to interpret the spectral information. The Spectral Lab focuses on developing research methods that help to more effectively use the remotely sensed imagery, in order to understand and monitor the biogeophysical processes in ocean waters and in wetlands. The Spectral Lab also researches light attenuation in coastal and riverine waters, and figures out possible effects caused by human use of the land and climate change.

The Spectral Lab collaborates with many space agencies, like the NASA/ESA (European Space Agency). As a result, the Spectral Lab can accesses ocean images data from these agencies. Ocean images are the raw data that were taken by cameras installed on satellites. Unfortunately, sometimes these images cannot be used directly due to various reasons, which may include, clouds obscuring measurements of ocean reflectance and atmospheric correction of the contaminating effects of sun light.

The Spectral Lab scientists need to pre-process the raw data in order to analyze these data. Previously, the Spectral Lab wrote the following script to process the raw images.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import time
import polymer
import os
tic=time.clock()

from subprocess import call

from polymer.main import run_atm_corr, Level1, Level2
from polymer.level2 import default_datasets
from glob import glob

print("enter polymer dir")
for filename in glob('/spectral/OLCI/tests/S3A*.SEN3'):
        print("filename is ", filename)
        run_atm_corr(Level1(filename),
         ↪ Level2(outdir='/spectral/outdir1/',  # level2 filename
         ↪ determined from level1 name, if outdir is not provided it
         ↪ will go to the same folder as level1
                        fmt='netcdf4',
                         ↪ datasets=default_datasets+['SPM']),
                         ↪ multiprocessing=-1 # thres_Rcloud = 0.13
```

```
                )
        toc=time.clock()
        print('Processing time',filename,': (seconds)')
        print(toc-tic)

toc=time.clock()
print('Total processing time: (seconds)')
print(toc-tic)
```

In order to better understand the procedure, here is an example. The Spectral Lab uses a tool named SNAP to open and analyze satellite images.

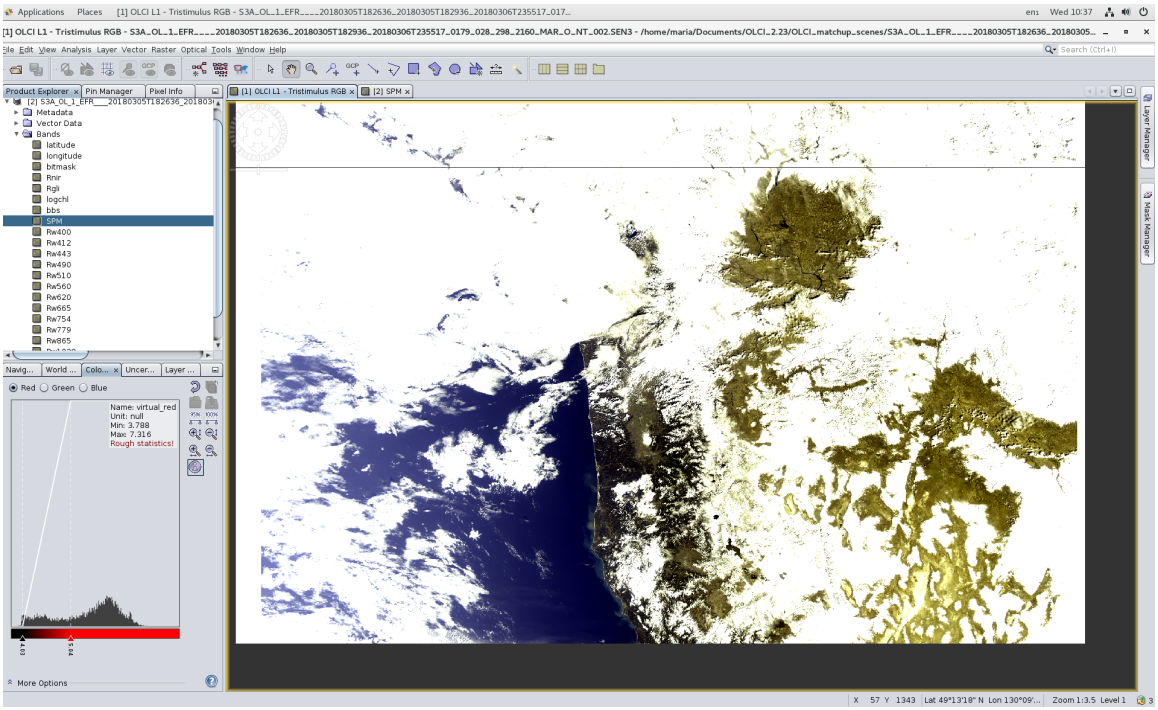

Figure 3.1: SNAP Open Raw Image

Figure 3.1 displays a raw image. In this image, it is hard to distinguish land and water. After using Polymer algorithm to perform the atmospheric correction, Figure 3.2 shows the generated Polymer output file. The new image shows land and water clearly.

Figure 3.3 gives a more impressive view of Polymer algorithm result compared with the Raw image.

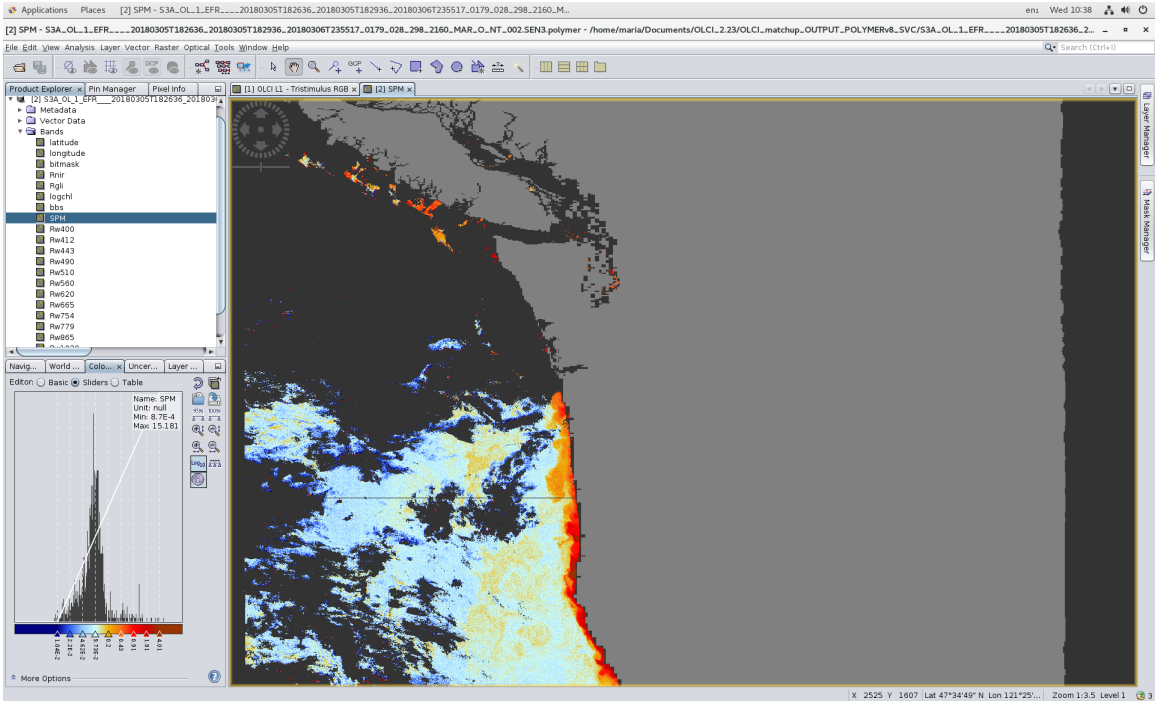

Figure 3.2: SNAP Open Polymer Output

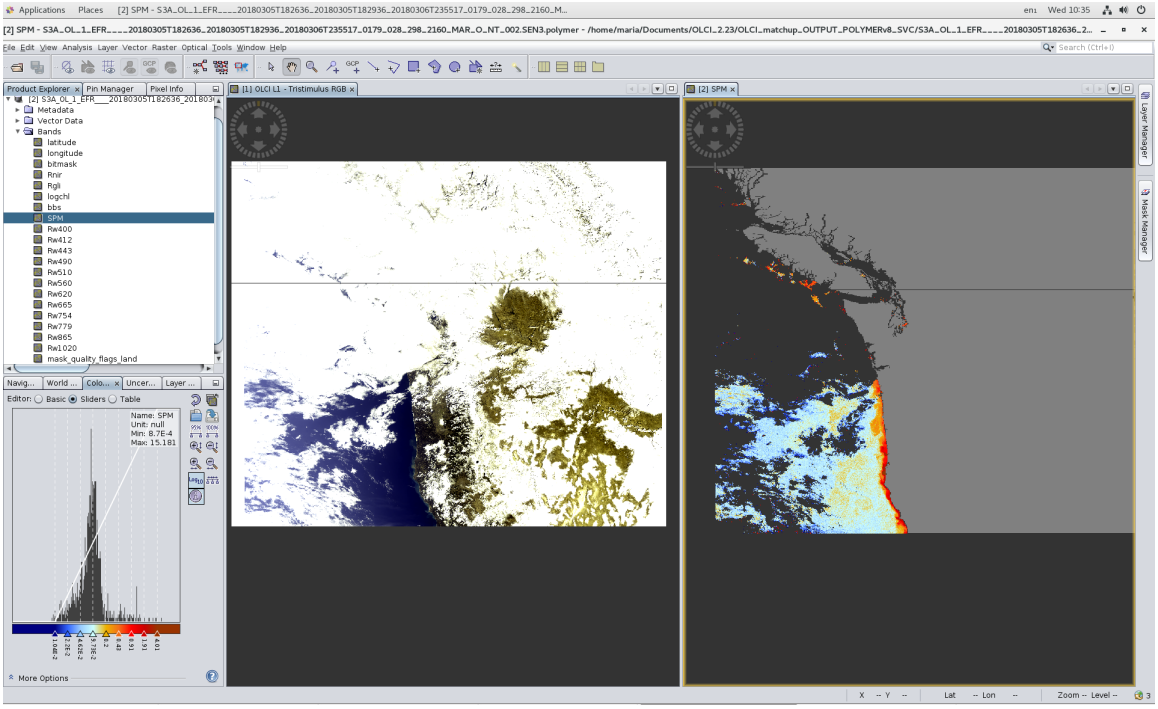

Figure 3.3: Raw Image vs Polymer Output

3.1.1 Images Processing On a Single Machine

GIPA originally runs on a desktop in the Spectral lab. This machine is a powerful modern computer with 32G memory and 12 cores Intel i7 CPU. The scripts that are running in this single machine follow a sequential order. The running time in a single machine is taking too long for the Spectral Lab to accept. It costs 15 to 20 days to process one year's data. As a result, it is approximately two months or more to process the entire data set. For this reason, Spectral Lab wants to find a new way to process the data, so that it can save time.

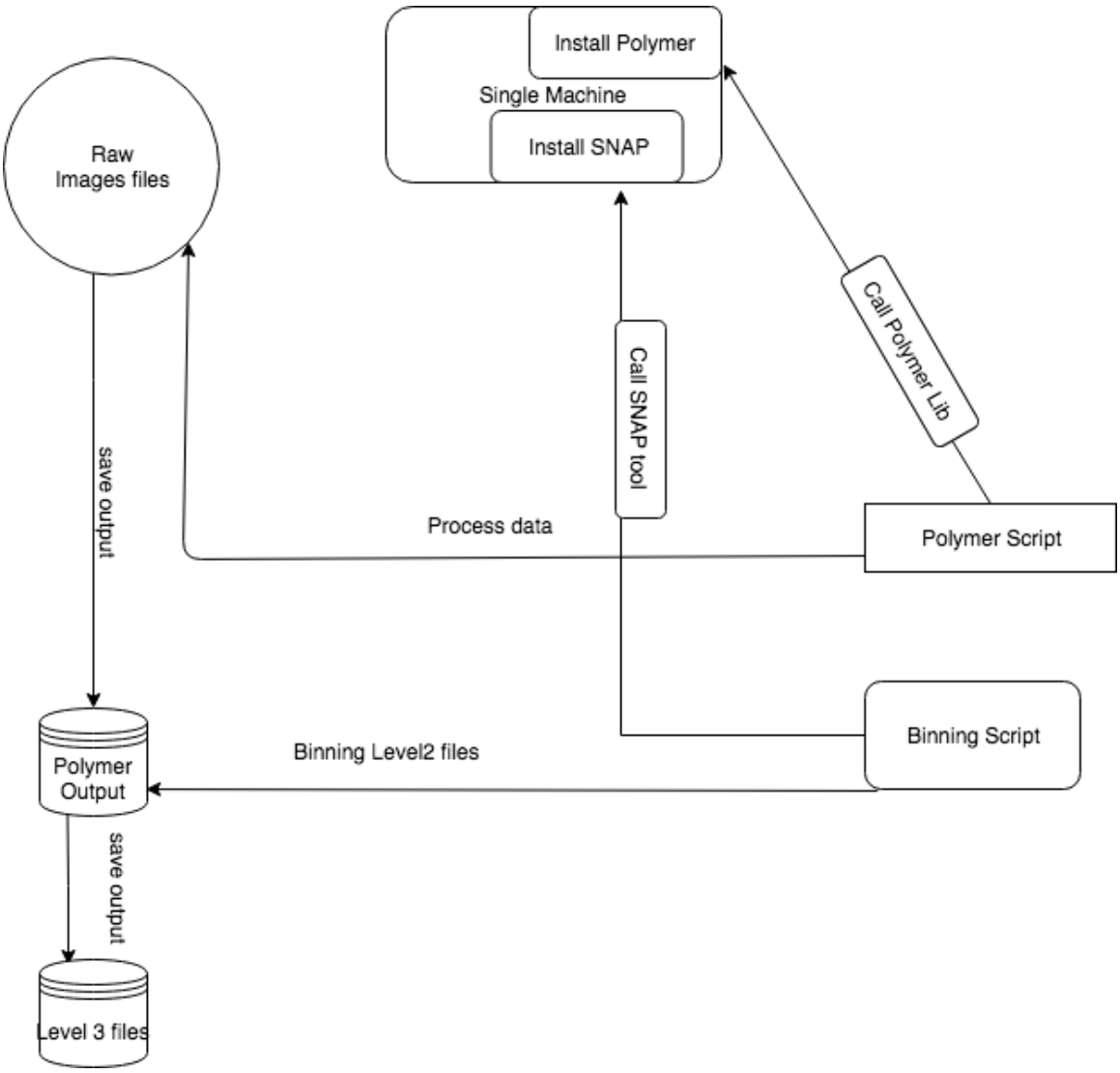

Figure 3.4: Single Machine Artifact

In order to verify the original Spectral Lab scripts, the development team at UVIC Computer Science Department runs the same script on a single node on Compute Canada Platform. Figure 3.4 describes the single node structure of GIPA. As an example, there are 147 raw images. It takes 36 hours to process these images on a single node on Compute Canada. By contrast, processing the same files only needs 3 hours for 14 nodes on Compute Canada. There are 1086 images in total in the year of 2016. Processing these images will take 3 hours for 150 nodes on Compute Canada.

3.1.2 Data Set

In this project, our data is from ESA. Three years (2016-2018) of Sentinel-3 Ocean and Land Colour Instrument (OLCI) imagery are processed from level 1 to level 3. The data set size of ESA Sentinel-3 OLCI image is 1.7 terabytes in total. It contains 3651 image files. The Spectral Lab wants to process these satellite images to remove noise information, and regenerate a better version by applying specific algorithms on them.

3.1.3 Compute Canada

Because of the massive dataset, a powerful compute power is needed to pre-process the satellite images. A cluster system can fulfill our requirement at this point. Compute Canada is a High Performance Computing (HPC) system, which provides essential Advanced Research Computing (ARC) services and infrastructure for Canadian researchers and their collaborators in all academic and industrial sectors; It helps accelerate scientists' innovation. One of the Compute Canada's systems is named Cedar and locates on Simon Fraser University and contains more than two thousand powerful nodes. The experiment of this thesis runs on the Cedar system.

3.1.4 Singularity Container

Singularity [12] container is a free, cross-platform and open-source containerization solution created by Berkeley Lab. It is accepted by research communities to meet their scientific demands in the HPC environment, mainly because it brings containers

and re-producibility to scientific computing and HPC world. Currently, Compute Canada does not support Docker container, however, GIPA uses the Singularity container to build the application. The Singularity container is fully compatible with Docker container. As a consequence, Singularity communities do not need to re-write the tools. Docker container can be easily transferred into a Singularity one. The Singularity container is similar to other container solutions, in particular, Docker container. Except that Singularity was specifically designed to enable containers to be used securely without requiring any special permissions on multi-user compute clusters.

Singularity also provides a secure way to use Linux containers on Linux multi-user clusters. It enables users to have full control of their environment. Moreover, Singularity uses a way to package scientific software and deploy such to different clusters that have the same architecture. Please refer to A for more detail of Singularity container information.

3.1.5 Polymer Algorithm

The purpose of Polymer algorithm is on recovering the radiance that scattered and absorbed by the oceanic waters (also called Ocean Colour) from the signal measured by satellite sensors in the visible spectrum. This algorithm has been applied to multiple sensors from ESA (MERIS/ENVISAT, MSI/Sentinel-2 and OLCI/Sentinel-3), NASA (SeaWiFS, MODIS/Aqua, VIIRS), and the Korean Geostationary Ocean Colour Imager (GOCI). One of the strengths of this algorithm is the possibility to recover the Ocean Colour in presence of sun glint. This leads to a much more improved spatial coverage compared to some previous products [9].

The Polymer atmospheric correction algorithm is developed, in particular, for retrieving the Ocean Colour when the observation is contaminated by the direct reflection of the sun on the wavy air-water interface, also called sun glint. Polymer is a spectral matching algorithm which uses the whole spectral ranges from the blue to the near-infrared, to decouple the atmospheric and surface components of the signal from the water reflectance. This algorithm is described in [60], but continuous development has been done since then. The latest stable version is 4.9.

3.2 Architecture of GIPA

The Spectral Lab has already written a series of scripts to process images on their lab computer. As mentioned, the process is very slow on a single machine. In order to overcome this bottleneck, the Spectral Lab wishes to execute the scripts in a cluster system. As a result, the architecture of the scripts needs to be changed from sequential mode to a distributed way. In a single machine every thing is sequential. Users do not need to worry about the synchronous issue. This is contrary to a cluster environment that runs jobs in a distributed mode. Users request multiple nodes or compute units from a scheduling manager, which means multiple nodes may compete to process the same image file. Users want to prevent this situation from happening. As computing resource is precious, researchers do not want to waste it doing duplicate task. Therefore, the Spectral Lab wants to fully utilize Compute Canada compute resources to process the images data as much as possible.

There are two tasks to perform. The first is recovering the radiance scattered and absorbed by the oceanic waters from the signal measured by satellite sensors in the visible spectrum. The second is binning the images generated from the first task. Binning is a procedure of combining a cluster of pixels into a single pixel. The Polymer algorithm is used to execute the first task. Moreover, SNAP [13] Graph Processing Toolkit (GPT) tools are chosen to perform the second task. To give a clearly view, Figure 3.5 displays how GIPA works on Compute Canada.

The development team has designed a mechanism to prevent the duplication work. First of all, the names of all pending process files are stored in a text format file. This step is done by unzipping raw data from zip format. Every time a running container instance picks a random file name from the text file. Then Polymer script checks if the chosen file already existed in two temporary lists named as processing and completed. If it exists in one of the two lists, it means the file is either in processing or already finished. Then the container goes back and get a new name from the text file, until it finds a name that does not exist in either list. In another words, the container must locate a file that has not been processed. Secondly, the script puts the name of

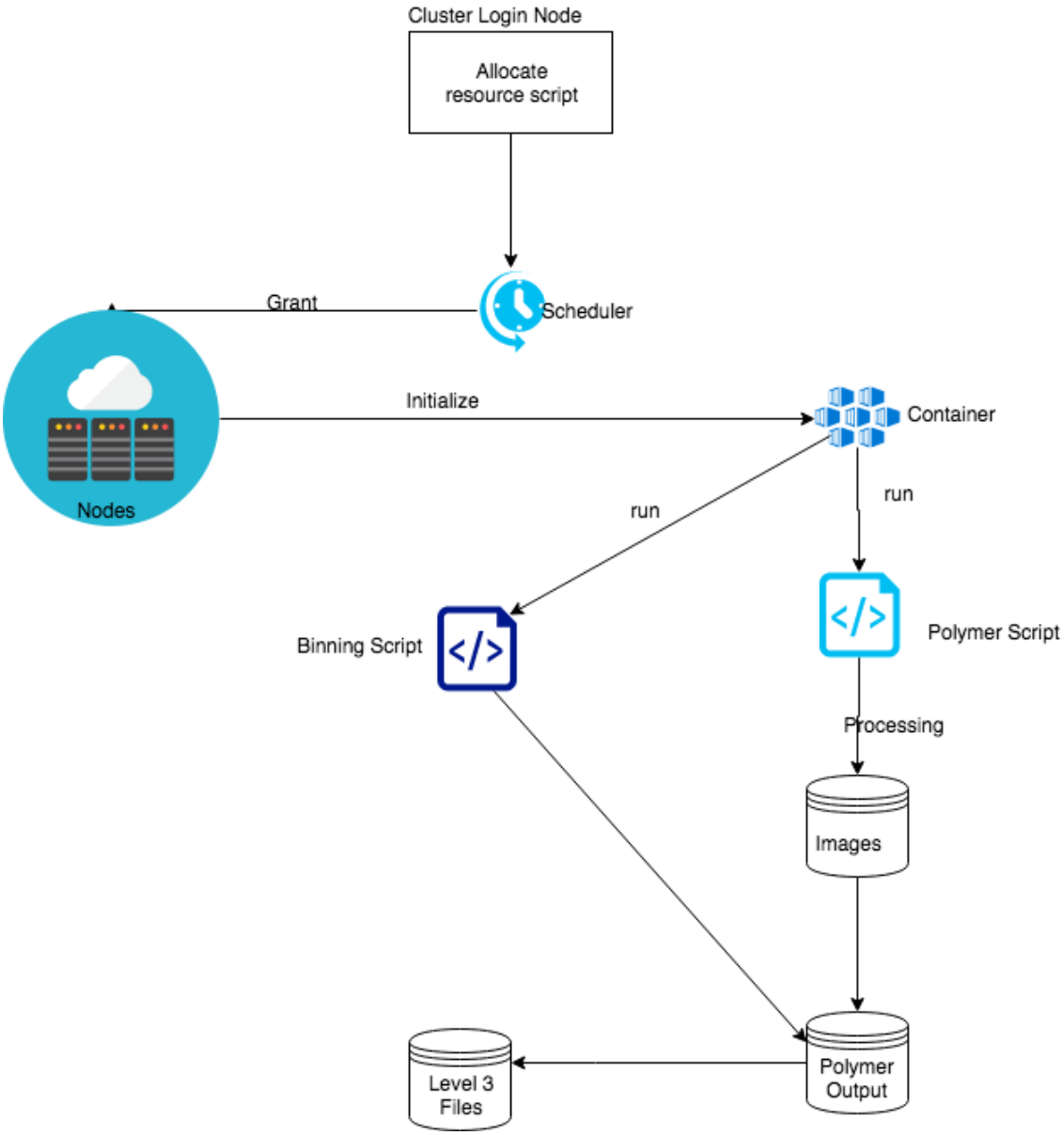

Figure 3.5: Cluster Artifact Diagrams

the picked file in the processing list, and then it calls the Polymer library to perform atmospheric correction operation. At last, when the file is proceeded, the script saves the file name into the completed list.

3.3 Implementation Details

The procedure is designed as a pipeline to fit for the Compute Canada platform. Level 1 files are the original data downloaded from ESA. Then the Spectral Lab gets level 2 files after level 1 files are processed by the Polymer algorithm. After the Spectral Lab has level 2 files ready, the Spectral Lab wants to bin files for further operation. The Spectral Lab first submits jobs to handle level 1 files to get the level 2 files, after that, the Spectral Lab bins level 2 files to get level 3 files. The Spectral Lab has various requirements for binning tasks, such as based on a weekly or monthly, or select a random number of files, at last applies resulting files for further comprehensive analysis.

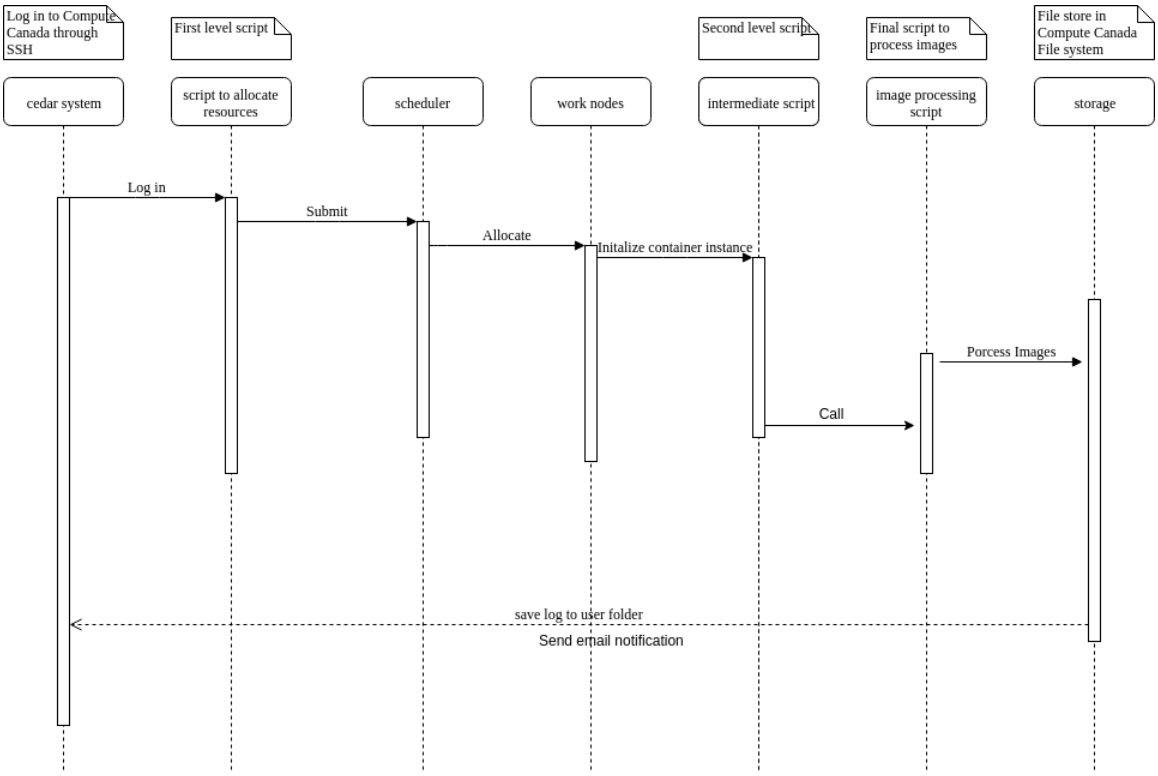

Figure 3.6: Interaction Diagrams

Consequently, the development team designs and implements some scripts to automate the procedure. Figure 3.6 describes the details of the workflow. The first script decompresses all zipped source files. The source files are compressed to save disk space, then they must be decompressed first before executing our tasks. The next

step is submitting a request to Compute Canada scheduler for computing resources. After the scheduler grants the resources, the main script starts a container instance to run a image processing script inside the container.

The image processing script is written on Python 3. As recommended by Polymer website, Python 3 was chosen as the development language. Basically, the script reads files in the file system as well as calls Polymer library to generate NetCDF [55] (Network Common Data Form) formatted data files. These NetCDF formatted data files are so called level 2 images. The following step is to bin these intermediate data files to get level 3 data files. As the hierarchy of level 2 data files are structured to follow the calendar, their paths are following by year/month/day structure. The tool that the Spectral Lab use to bin requires us to put the files in a same folder. Therefore, the first task is to copy level 2 files to a folder.

Currently the data is processed on a monthly base. As a result, the development team creates twelve folders which named from January to December. I creates a shell script for the copy task. After that the Spectral Lab starts to bin data files, the rest part is same as the first task. At last, twelve level 3 files are generated and saved to their designated folder.

Compute Canada is, in fact, a HPC cluster. The Polymer library is not pre-installed in the cluster, so the development team must install it ourselves. It is a time-consuming and complicated process to install software in a cluster with only normal user privilege. Fortunately, there is an alternative solution. Instead of installing the Polymer software on Compute Canada users home directory, users install it on the Singularity container. The development team builds GIPA together with Polymer tools into a Singularity container image and allows scheduler to initialize a container instance to process the data.

From the usability perspective, this is a command line style application. Users need to remember many commands in order to process the data. On one hand, the command line style is more efficient. Users can control their applications in a direct way. On the other hand, a Graphical User Interface(GUI) application is easy to use. GUI style applications can do multiple tasks at the same time. GUI application's learning curve is not steep compared to command line style applications.

3.4 DevOps Plus Container Highlight

In traditional HPC environments, HPC maintenance engineers are responsible for install software. Despite that, HPC system engineers can not install all software for their users. If the HPC can not provide a software that users want to run, users need to install the software themselves. This step causes various issues. Most of the time, HPC users are basic users without root privileges. Even if users can install software eventually, the maintenance may be difficult and error-prone. If users use container technology, users can set up necessary software to a container, then deploy the container to a HPC environment. This is exactly what the development team does to solve the problem. Below is an example that demonstrates how container technology is applied.

When the Spectral Lab runs the Polymer script to process the satellite images, users have noticed a lot of segmentation fault errors. This kind of error will terminate a running container immediately. After careful investigation, the development team has found out the root cause is a bug in the version of NetCDF that the application is using. It would be complicated to fix the bug in a traditional HPC environment. Though, it is easy to fix in a container. The development team builds a new Singularity container image with the desired NetCDF version and put the new image to Compute Canada. All the scripts will be using this new image. Users do not need to ask Compute Canada to get support engineer involved in this situation.

3.5 Summary

In this example, Computer Science development team builds a pipeline of scripts on Singularity container that is deployed at Compute Canada. Container technology gives us a great example on how it can help us run jobs on a HPC environment. Comparing to the traditional way, it is easier to fix a software error in a container than having the software installed under user directory. Users do not need to touch the HPC environment configuration. This can dramatically reduced the development and maintenance overhead in a HPC system.

Chapter 4

Design and Implement a GUI Satellite Imagery Application On AWS

This chapter describes how to design and implement a GUI-based geospatial imagery processing application. Chapter 3 has discussed an application which runs on Compute Canada. The application works as our expectation, though it is not user-friendly. We believe we can revise the batch processing system with a web GUI-based cloud-native application. This is a new experiment that explores cloud computing power. We hope that the new user-friendly application will get better resilience and elasticity. in addition, the cost of the new platform can be reduced.

Designing and building a software is not an easy task. There are a lot of trade-offs to be made. Back to this case, because most existing satellite imagery software are using client-server architecture. Users need to install a client tool on their machine first. This model introduces two problems. One is that users need to hold a powerful computer due to high resource consumption from the imagery software. Even though almost all commercial software permit researchers are using their software freely for non-profit activities, researchers still need to get a powerful computer themselves.

The first problem was the users need for powerful computers. Another problem is troubleshooting, whenever software stop working. Users must call the product's customer service first. Then they wait for an engineer to be dispatched to their case.

The support engineers can troubleshoot either remotely or on site. Either way, users have to suspend their task. On the other hand, a cloud-based application does not face these kinds of problems at all. Users need not to learn how to configure servers or what parameters need to be configured based on their own situation. All these steps are done by development teams on the cloud side. More importantly, we call this application a platform because this application does not only process images, but also works as a Microservices style application, it can add different kind of feature to this platform as plug-ins services. Furthermore, by applying container technology in our design and implementation, combining with DevOps, we have made this tool become a self-managing application.

4.1 Design Characters

In this section, I will discuss the whole picture of our design philosophy and at last provide an implementation as an example. There are many different types of cloud in general, such as PaaS [21], IaaS [8] and SaaS [14]. Each one suits for certain scenarios and has its own advantages. For our research, we used PaaS. Because PaaS is the best solution that fit our user scenarios because we need to manage hardware and low level details of our application. PaaS can provides the flexibility that we want.

Cloud environment means bare metal level infrastructure will be carried out by public cloud providers. As a user, he or she needs to focus on his/her applications. We name an application that is deployed on a cloud environment as a cloud application. However, not all applications are designed for the cloud environment originally. Some applications simply switch their running environment from a private data center to a cloud platform. Those applications can not fully exploit benefits of the cloud. Hence, we would like to design an application that can optimize the cloud platform. This is so-called cloud-native application [49]. It binds with the cloud to deliver better performance. In the past decade, there were many attempts to accomplish this goal. Some of attempts have been accepted as best practices by the software communities. We could not reach our final stage without the knowledge we have learn from these predecessors, such as Agile, Service Oriented Architecture (SOA) etc. They are the supporting pillars of our product.

Microservices [38] came out after developers faced many issues when they were developing software on cloud platforms. Thus, people were proposing various new solutions for software development in the cloud era. In order to shorten development life cycle during features delivering and bug fixing in close alignment with business requirements, DevOps [28] has became a popular practice after Agile movement. It is because the traditional software development flow cannot satisfy the current fast changing market. Due to the rapid evolution, applications cannot wait too long to be released to the public. Otherwise, competitors would deliver their version of a similar or identical feature in their applications earlier. This new situation requires a development team to build and release their products as fast as possible. So monolithic style software is unable to satisfy this situation anymore. If one part of an application has a problem, then, the whole development team need to stop and wait for the problem to be fully resolved. What's more, the test team cannot start to test until the development team finish the project.

Microservices break a monolithic software into a series of small pieces. Each piece is expected to fail if it runs independently. As a result, each piece gains some levels independence but it is not an complete application. We have an executable product if only all the pieces are combined together. Each piece communicates with another through RESTful [36] API or other similar protocol if needed. We call these small pieces microservices. Any change to a specific service should not affect the execution of the whole application. The upstream or downstream services only need to be modified if a service changes its exposed API endpoints. We can upgrade a single service without breaking the whole application.

Yet, splitting the application is only one part of the story. How to develop and deploy every service is also challenging. Regarding software development workflow, the traditional waterfall model is not popular anymore. Agile software development has gained more and more attraction in the recent years. Furthermore, DevOps has won a lot of support. It mixes the developer and operator roles together in the software life cycle. Developers also has responsibility to maintain it after the software has been deployed to production environment. Google introduced this concept in the famous book Site Reliability Engineering [31]. DevOps makes software products robuster and more flexible.

This Microservices architecture has gained a growing popularity by using container technology. By employing Docker container, we can put a service into a container, then build our application making use of multiple containers. Each service is running on one or more containers according to their workload. Microservices plus container fully fits the DevOps philosophy. As containers can make local development environment close to a production one, and one service stays in one container to keep each service isolated.

By default, Docker containers are not easy to coordinate with each other. Thanks to open source community, there are open source tools that focus on this requirement. As an illustration, Kubernetes is a container orchestration system. Kubernetes supports the user to monitor their system's health. If Kubernetes detects that a container is not working properly, it kills the problematic container and starts a new identical one to replace the old one. Therefore, the operation team does not need to get involved in this situation, which previously required manual intervention. This is one of the features that Kubernetes provides for operation teams.

In addition, Kubernetes has one more feature that helps operation teams to manage their application in a smooth way. Basically speaking, you can get rid of the old way of software maintenance. If a server got overloaded in the old days, maintainers had to manually initialize more servers to share the burden. Nowadays, this is unacceptable. Because when new server ramps up, customers may already leave due to a long waiting period. As a result, many teams choose Kubernetes to manage containers. It orchestrates containers and empowers them to self-manage. It provides monitoring, logging, auto-scaling and other functionalities. More importantly, Kubernetes provides more advanced features to regulate applications, such as load balance, namespace discovery and firewall. Previously, many of these features required to be configured by cloud users or even a third party product. Now Kubernetes has most of these features and is free.

Under Kubernetes, cloud providers, such as Amazon, Google and Microsoft, offer a fundamental infrastructure. In our case, we choose AWS. As AWS has the leading role in the cloud market and it provides a comprehensive solution for everything from computing resources to data storage, from databases to pre-installed application software. All we need to do is to select a right bundle products for our application.

Especially when processing satellite images, one single image could be as big as several gigabytes. When accumulating them together, the size of data can easily reach hundreds of gigabytes or even terabytes. Our top priority is to find a cheap storage to store these images. AWS S3 is a perfect option. It is stable and cheap. We do not need to worry about data lost issue either. In spite of that, other clouds also have similar solution, so it is not vendor-lock.

Even using tools like Mesos or Kubernetes, an application can still carry a significant operation overhead. Particularly, when we start up a container in AWS. We are paying Amazon as soon as the container is running, even when the container is inactive. This means customers are wasting money. This has became a serious issue when a customer has many containers that are idle for a longer time. For example, users may be located in different regions, or they are not using the service all the time.

More recently, Serverless architecture has won growing popularity as it reduces even more burden from cloud users. In the VM/container era, even though the users do not own machines, they still need to manage their fleet. Users need to decide how many nodes they need, then preserve these computing resources from a cloud provider. After that, cloud users also need to operate these nodes, like installing necessary software in these nodes. The ops team monitors metrics of these servers and gets ready to fix any issues as well as planning activities. The planning includes getting more compute power and setting them up as soon as possible and upgrading current machines, if the current fleet capacity cannot support their business.

Planning is a critical point for cloud users. It is a tough problem especially for a small team. Because it is hard to predict an application's traffic volume in advance. Finally, even if the planning is calculated precisely, it is still impossible to balance nodes size and cost spending in the cloud environment. This is because an application has peek and non-peek time, except for extraordinary situations. The Serverless architecture has completely changed the situation. Taking AWS Lambda as an example, a service that runs users code in Lambda in response to events and AWS automatically manages computing resources for users. The point here is that users never provision, allocate or configure a server, for the code to run on, and nor can the users. This is entirely AWS's responsibility and decision.

The last goal of our design focuses on the end users. Most of them are not computer experts. They may know little about computer programming. It is difficult to teach them installing and configuring tools on their local computers. Instead of that, this application gives users a user-friendly interface, users only need to connect to the Internet and access their browser to open the website. In this way, the users only focus on how to improve their core ideas and leave other concerns to the clouds.

In conclusion, the design of the application must be easy to use. As an experimental project, the requirements change frequently. It also needs flexibility to upgrade or modify. At the same time, it should be easy to scale as the number of the satellite images may increase. Furthermore, the cost must meet the budget as our funding is limited. At last, it should be easy to operate and maintain.

4.2 TestKitchen Application

In order to examine the design, we have developed a prototype application. We used our experiences that learned from GIPA. We named this prototype as TestKitchen.

4.2.1 Overview

TestKitchen is an AWS hosted application. It enables multiple users to collaborate on algorithm development and to share computational resources in order to run and test those algorithms. The platform of TestKitchen features a python development environment with debugging tools, a React front-end, a map-reduce computational framework for executing algorithms, and a Leaflet-based mapping framework for displaying the results of algorithms.

The Help systems in front-end are provided in-frame to allow inexperienced developers to quickly come up to use TestKitchen. Users are able to share algorithms under development through sending email links to the work-in-progress workplace, or by jointly maintaining a source code repository of the algorithms. Google Earth Engine and Geonotebook offer some of the feature set of the TestKitchen. Nevertheless, from both an architectural perspective and a user-interface perspective, we believe that the TestKitchen offers valuable lessons in improving cloud-based algorithm de-

velopment efforts.

TestKitchen platform is designed to process satellite images. We hope that can help academic users, such as professors, graduate students and researchers to analyze satellite images through their algorithms with ease. The images taken by the satellites are various regarding the size. It could be as big as gigabits. As described in chapter 2, because of the uncertainty of images size, we could not estimate exactly how much computing resource TestKitchen need to support the clients' requirement. If we preserve as much as we can, we will waste a lot of money. On contrary, If we keep the average size of the nodes, some jobs may run out of resource and eventually fail.

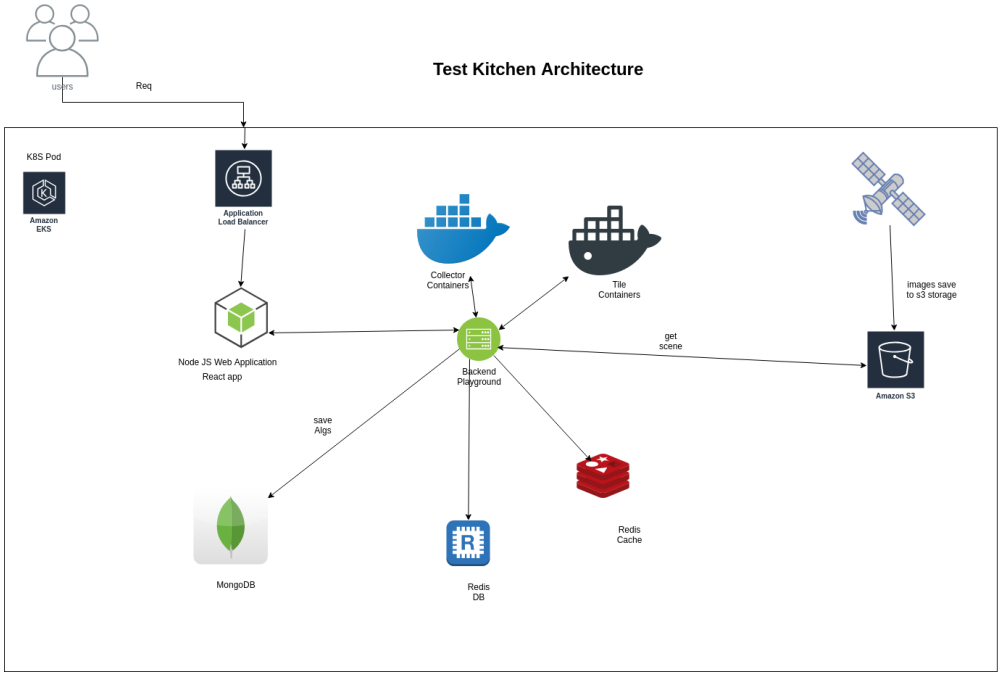

Figure 4.1: Test Kitchen Structure

Figure 4.1 displays TestKitchen structure. We want to achieve dynamic resources allocation. This feature could give the application development team more flexibility.

In TestKitchen platform, there are 5 containers that running 5 services. The first one is a front-end container. The front-end container includes the UI of TestKitchen, what the end users see when they access TestKitchen. TestKitchen's main window is

split into three parts. The left part shows the satellite images. The right upper part displays the algorithm window. The right bottom part is reducer, which is responsible to collect the map partial result and to generate the final outcome. The Docker container definition file is located in appendix part.

4.2.2 Front-end Development

From a user-interface perspective, the innovations in the TestKitchen consist of user assistance features and collaboration features. The TestKitchen is a React [10] application built on top of a windowing framework called Golden Layout, which allows us to have a relatively large number of windows that can be arranged and hidden as the user desires.

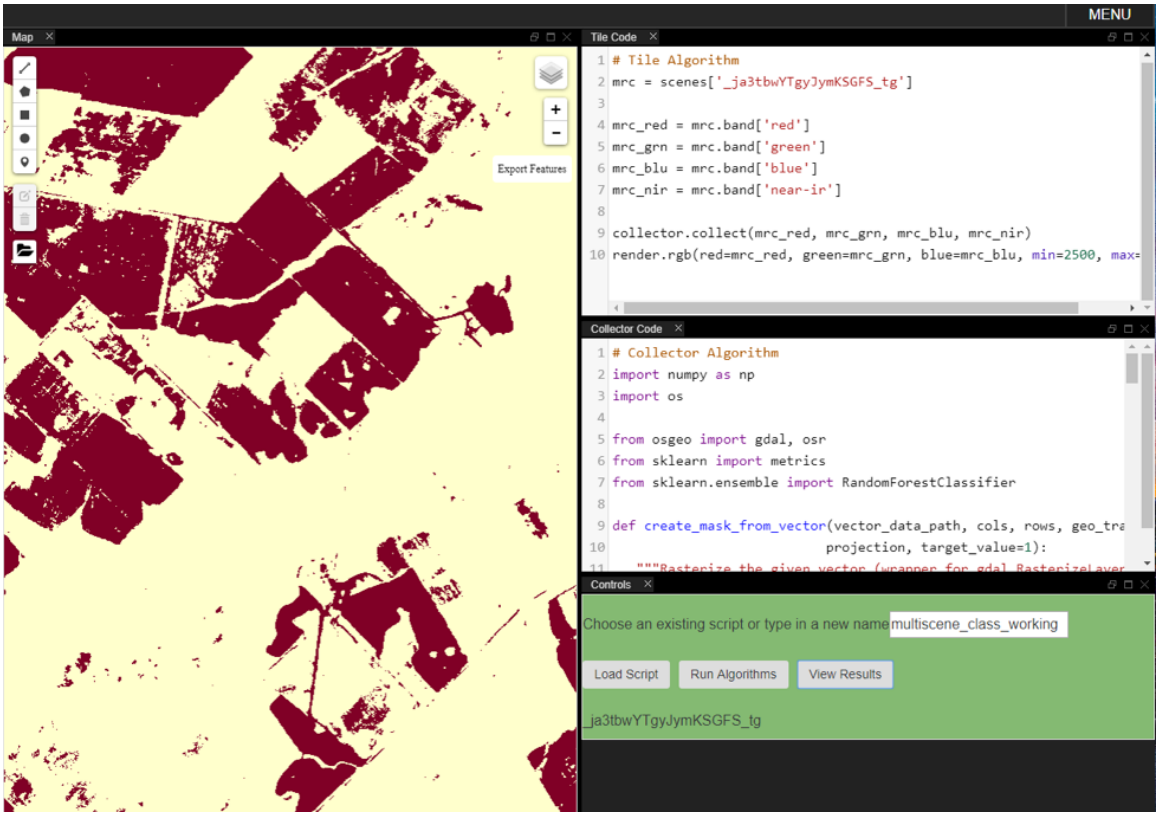

Figure 4.2: TestKitchen Home Page

The whole front-end UI is a Node.js [61] application which runs in a Docker container. These windows include script management windows, code editor windows,

map display windows with layers and drawing tools, satellite scene selection windows and help windows. These can all be arranged as the user desires and then sent as a complete console to other TestKitchen users through a URL link in an email. This allows a user to develop a portion of an algorithm and then ask others on the team for help without having to worry about the environment setup or explaining how to get to the point that they get stuck. Algorithms can also be stored in a source-controlled repository that is viewable to other users, so that they can use it as a starting point for their own algorithm development. The main left part is a map display window. It shows the working area and the results return from algorithms. Figure 4.2 is a screen shot of the home page of TestKitchen.

4.2.3 Back-end Development

The core of the TestKitchen is tile and collector containers. The names come from our programming model. The programming model for TestKitchen consists of two different types of image processing scripts, which are identified as tile scripts and collector scripts. The tile script is running on a tile-by-tile basis on the satellite imagery data, while the collector script is running only once after all tile scripts have been fully processed. This programming model allows the processing of the algorithm to completely use a map-reduce framework where the tile scripts are running in parallel on multiple processors. We use MapReduce [33] model to process the images because it fits our scenario perfectly. The MapReduce model cuts a image to small pieces and feed the pieces to the tile server. Figure 4.3 shows how this mode works. After every tile server gets a partial result, the collector container collects results and generates final output. As we don't know how big an image can be in advance, we need the capability to scale our tile and collector containers as soon as the default compute nodes have exhausted. We also have a database container saving data to a persistent storage. In our case, it is the AWS S3 storage system. The last container is a caching service. We use Redis [11] as TestKitchen cache layer to support frequent user requests. Because retrieving information from S3 can be very slow compared to caching mechanisms. It is much faster if we deploy a caching service.

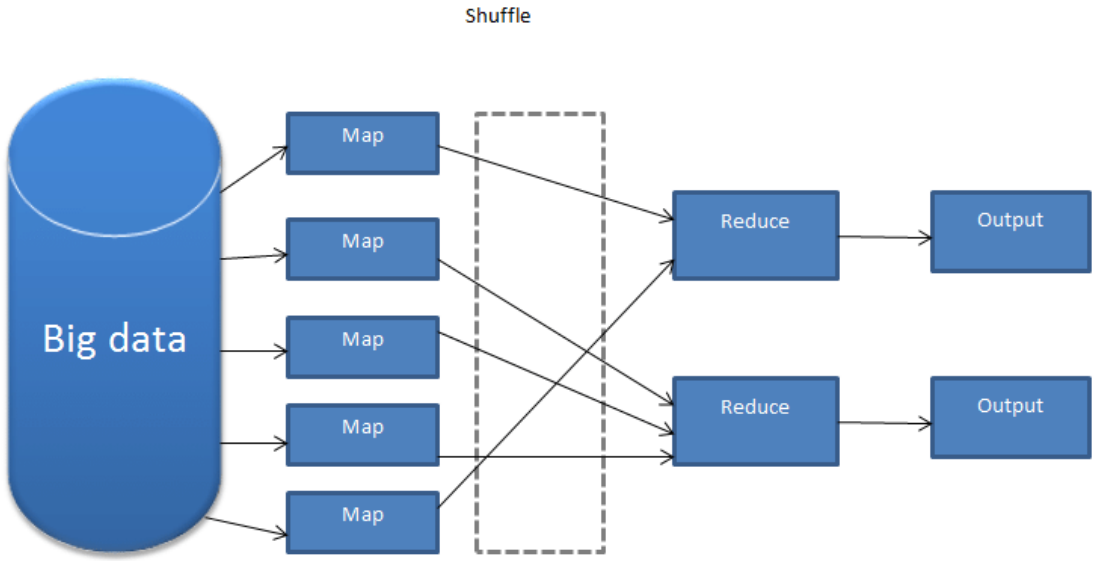

Figure 4.3: Map Reduce Diagram
Source: [24]

4.2.4 Architecture of TestKitchen

The architectural element which enables this model to occur seamlessly is the use of a containerized environment for executing the tile and collector scripts. These Docker containers are deployed on Apache Mesos using Marathon as the container orchestration solution. The ability to quickly spin up a large number of independent tile containers allows results to be processed in a highly parallel fashion, as well as confers a degree of fault tolerance to the system. Even if a particular container crashes for some reason, it does not bring down the entire system.. This built-in fault tolerance feature enables the algorithm developer to be given much more ability to access the core system resources, than the system which has to be more protected from the script authoring mistakes. It also allows the use of the multiple types of execution containers to depend on the type of facilities required by the script authors. For instance, the Geospatial Abstraction Data Library (GDAL) is a base element of the system included in all containers, but only a small number of the scripts need the access to classifier training capabilities from scikit-learn [53], so the provision of specialized container constructions allows an optimized use of system resources.

When a user logs on to TestKitchen, the user can input his/her own algorithms

on the algorithm code window. By clicking the run algorithm button, front-end will dispatch the request to tile containers, tile containers process the images and pass the results to the collector container. The collector container collects all the intermediate results and combines them together to output a final outcome. As mentioned, the images size is unbounded, so we don't know the size in advance. If the size is huge , pre-existing instances of the tile and collector containers may not have enough resource to process it in a reasonable time. For this case, Mesos can initialize more containers to share the load, but this is a manual process.

A better solution is Kubernetes. Users can define specific metrics and Kubernetes will monitor the containers. When the metrics reach over the defined threshold, Kubernetes can automatically add new instance to the application. For instance, if the container's CPU utilization passes 85%, Kubernetes can start a new container to share the load. On the other side, when a metric is lower than a specific threshold, Kubernetes can stop routing request to this instance and eventually terminate the instance after existing requests have been processed. This approach will certainly save the cost for cloud platform users.

The TestKitchen platform is built upon AWS cloud. It uses Mesos to manage Docker containers and implement TestKitchen application by using Microservices pattern and DevOps practice.

4.3 Implementation Details

We believe that this architecture has several benefits. Firstly, users do not need to install any thing on their local computer. Users only need a computer with a browser that can connect to the Internet. Then, users can focus on their businesses. Secondly, TestKitchen is a user-friendly application. Users also do not use command line to submit their jobs like GIPA did on Compute Canada platform. Thirdly, by using Microservices, TestKitchen application is divided into few small independent microservices. Hence, every single one can be upgraded or modified without breaking the whole application. If there are any new features to be added to the application, we only need to work on the specific container. If coding and testing parts finished, we just stop the specific container in the system, and build a new image and initialize

a new container instance. Comparing to the previous software upgrading method, this is a substantial improvement. Previously, operation team must stop an application to do upgrade. So during this period users can not access the system. The new code needs to be compiled firstly, then. operation teams uploaded the package to a cloud server. At last administers restart the system. The users then can use the new version of the application.

If anything goes wrong, either it is found by a user or the operation team, developers have to start troubleshooting. If they can not fix the problem, it usually means that the upgrade had failed. The application needs to be rollbacked. Because of all these factors, the downtime window normally takes a few hours for a middle size application. In order to minimize this business impact, the upgrade activity is normally scheduled at midnight. Having said that, in the globalization century, customers can be physically located everywhere. It is really difficult to find a time slot that suits for all customers. Hence, by using Docker, the downtime can be minimized or even avoided. DevOps teams are keeping the old container running, while the new version of service is started in a new container instance. Once the new instance has started running, then, all the traffic is diverted from the old container to the new one. If the new container can handle requests without problem, the old container can be terminated. Otherwise, all the traffic will be routed back to the old container during problem debug phase. In this way, there is no downtime during the software upgrading activity. More importantly, this architecture gives a great flexibility to deploy containers based on each single service's workload.

Figure 4.4 describes how DevOps practice helps make the process automatically.

4.4 Highlight and Future Goal

TestKitchen is an experimental project. In consequence, we are facing a huge burden of operation cost if we rent VMs by using traditional way. So the pay-as-you-go model is really helpful. Cloud users do not manage servers anymore, cloud providers take over this responsibility. As consumers, they only need to upload their code to cloud platform and make configuration. This is so-called Serverless. In Serverless mode, users only pay for actual API execution. If there is no request coming into a system, users do not spend any money. It is a raising star in recent years. Amazon and other

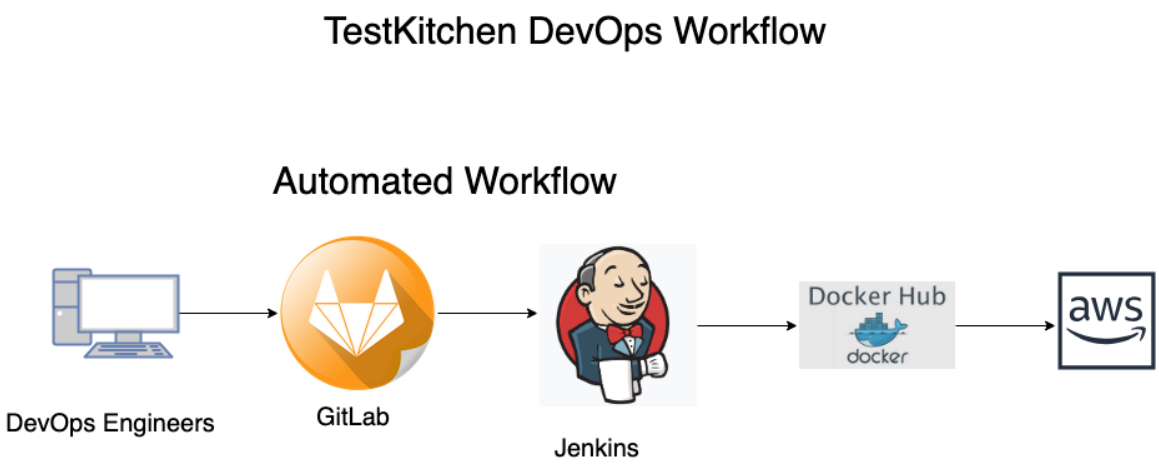

Figure 4.4: TestKitchen DevOps Work Flow

mainstream cloud providers all have their product line. To give an instance, AWS has Lambda [5], it allows user to create so called function in AWS. When a request comes to AWS, based on the configuration, AWS gateway routes the request to a user's function, and AWS creates a instance for the user if there is no existing one. The instance will handle the upcoming request. When Lambda instances finishes, AWS will try to keep it alive for a while, in case there is a new request coming. After time out, AWS Lambda will terminate the Lambda instance. AWS only charges a user for active Lambda instances running time. If there is no request at all, not a penny will be charged to the user. This is a typical Serverless user case and it is a perfect fit for our scenarios. In short term, this is the objective we want to achieve for TestKitchen.

4.5 Summary

Follow our design principles, we have implemented a cloud-native application, named TestKitchen. It is a successor of GIPA. The difference between them is that TestKitchen allows users to write their own algorithms, while, users need to use a common algorithm to pre-process all images in GIPA. TestKitchen is a GUI-based satellite imagery processing tool. It was running on AWS cluster, managed by Mesos. Users just put their algorithms into the TestKitchen code window, and then click the run button to execute their algorithms. Comparing to the command line model, this is a

huge improvement.

Chapter 5

Evaluation, Analysis and Comparisons

It is easy to compare containers versus Virtual Machines (VMs) or Mesos versus Kubernetes. However, Microservices and DevOps are hard to evaluate due to the reason that they are design patterns. In order to perform the assessment of Microservices and DevOps, I leverage two applications: the Spectral Geospatial Images Processing Application (GIPA) and TestKitchen. These two applications are similar to each other at some extend. First of all, they both focus on processing satellite images. Secondly, they both run on cloud platforms. The differences is GIPA runs on a batch processing system named Compute Canada, while, TestKitchen is a GUI-based satellite imagery processing tool runs on public cloud Amazon Web Services (AWS).

During the evaluation process, I compare the start up time of Docker container and VMs, then, check Kubernetes' ability of handling Docker container. The most important part of my evaluation is running GIPA jobs on Compute Canada platform. GIPA is constructed as a containerized application and deployed on Compute Canada. The Spectral Lab uses GIPA to process a data set that was obtained from European Space Agency (ESA) Sentinel-3 satellite.

In the rest of this chapter, I will discuss how we setup a testbed to compare VM and Docker. This work was done by a group students under my supervisor. Then I will talk about the performance of GIPA on Compute Canada. At last, I will share the observation from running TestKitchen on AWS.

5.1 Lightweight Versus Full Virtualization

The first question we have encountered while designing our applications is using VM or Container. I use bench-marking to verify which is better. Bench-marking is an effective way to evaluate VM or Container performance. There are many metrics in bench-marking a system, such as: response time, CPU usage and start up time. Different users may focus on various aspects. Bench-marking gives an overall summarization of a system for potential users, which can help users make a better decision.

In the next sections, I focus on the procedures to setup testbed environments, and detailed test scenarios related to VM and Docker container. Meanwhile, the results of these experiments are presented and discussed.

5.2 Testbed Architecture

Testbed environments are set on Compute Canada and AWS cloud platform. Single desktop machine is excluded here. The hardware details of the two testbeds environments are listed in Table 5.1 and Table 5.2 . All the content can be referred from the websites of Compute Canada [17] and Amazon AWS [23].

Table 5.1: Compute Canada Machine Type Specifications

Machine type	**Processor family**	**Virtual CPUs**	**Memory**
base	two Intel E5-2683 v4 "Broadwell"	32	128-184G

Table 5.2: Amazon AWS Machine Type Specifications

Machine type	**Processor family**	**Virtual CPUs**	**Memory**
t2.medium	Intel Xeon Family	2	4G

5.3 Kubernetes VM on VMware vSphere

First of all, we test the start up speed of VM. In order to do that, a cloud platform is needed. However, because of the one node limit for free users on Amazon AWS, we decide to build a private cloud platform that allows us to create a multi-nodes Kubernetes cluster and to preserve our privileges for full control over deployment and metric evaluation. This cloud uses VMware vSphere as its hypervisor. VMware vSphere is an enterprise-class, type-1 hypervisor developed by VMware for deploying and serving virtual computers. We then deploy Kubernetes VM on this cloud platform.

Our cloud platform is built from the scratch [64]. This means VMware vSphere is installed on some bare metal hosts, with one of the hosts designated to be the hypervisor by installing VMware vCenter Server on it. Figure 5.1 shows the architecture of our deployed VMware cloud platform. All the physical hosts are added to one resource pool in which VMs can be created.

A Kubernetes cluster is created on this cloud platform with the help of "Kubernetes Anywhere", which is an official tool to set up Kubernetes clusters on VMware vSphere platforms. Figure 5.2 shows a sample architecture of a 4-Node Kubernetes cluster. There is only one master node which is the controller and has "APIServer" running on it. The remaining three work nodes can only be used to run containers.

To ensure a systematic and reproducible approach, it is important to note that a DHCP server is required in the cloud platform to successfully create a Kubernetes cluster. In our experience, a physical router with DHCP service in the VMware platform is effective for this purpose. Figure 5.3 shows a dashboard result of a successful deployment of a Kubernetes cluster.

We test the VM start up time by using Kubernetes Anywhere images. All these nodes are managed by the hypervisor and we deploy VM on top of it.

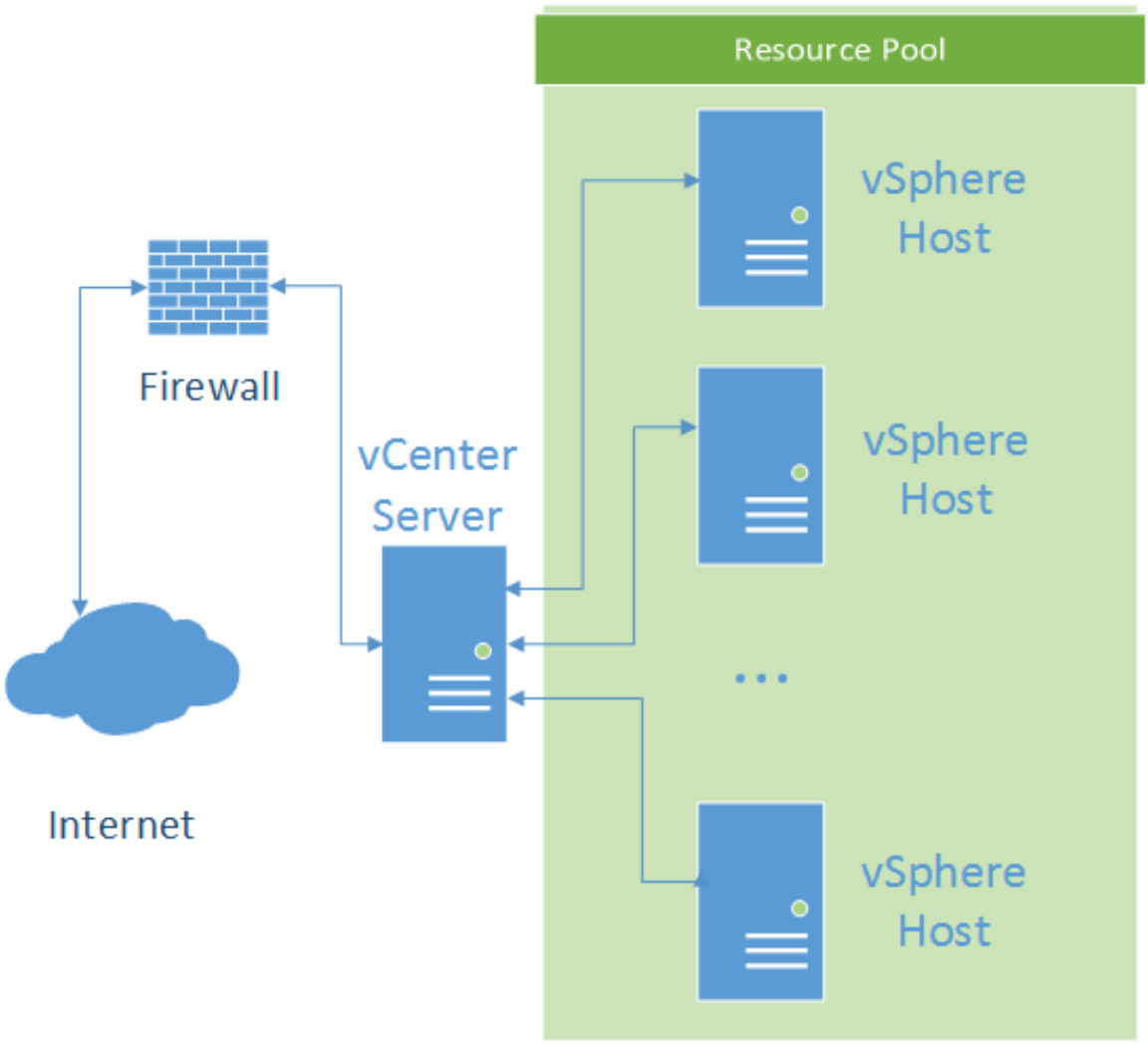

Figure 5.1: VMware vSphere Cloud Platform

5.4 Systematic Evaluation: VM Launch Times

As the nodes in the Kubernetes cluster are VMs in the cloud platform, it requires approximately 3 minutes to create a Kubernetes cluster from scratch. Figure 5.5 confirms that the launch time of a VM or a node in the cluster is on the order of minutes. This is because VMs are created using a Kubernetes Node Template image which is based on "Photon OS" [58], with minimum package installed on it. The template image itself is more than 150MB in size. Though, in order to streamline the process, developers can customize the packages to be loaded during the launch of containers. It is also not always necessary to load the whole Operating System (OS). Secondly, the experiments exploring the launch time of containers are conducted, and they are

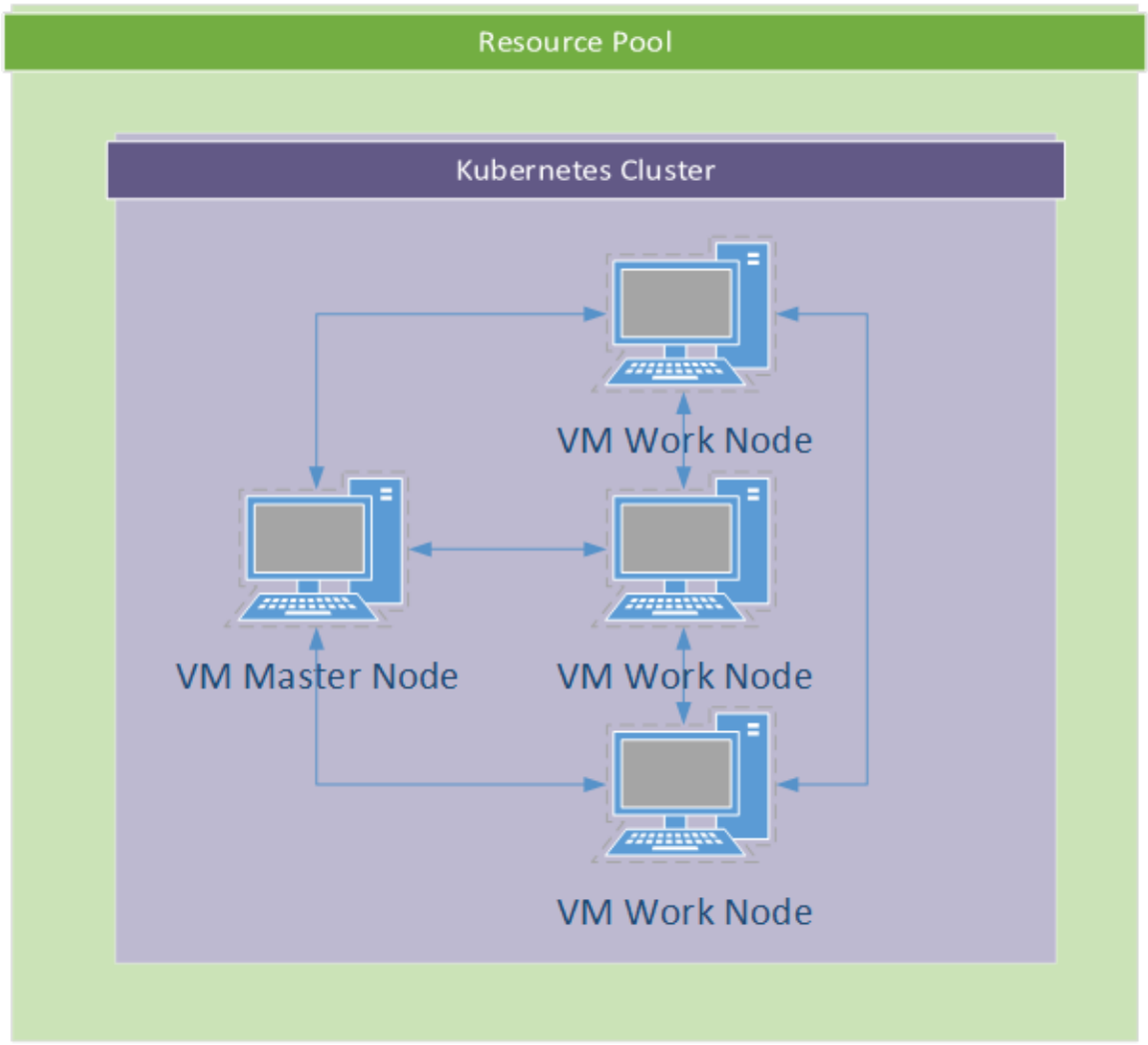

Figure 5.2: A Simple 4-Nodes Kubernetes Cluster

described in the next section.

5.5 Kubernetes with Minikube

After the testbed environment is ready, we check the containers' start up performance. In this case, we run containers on Kubernetes with Minikube [29]. Unlike traditional VMs, the launch of containers is faster and can be in the order of seconds, and it is validated through our experiments on Kubernetes.

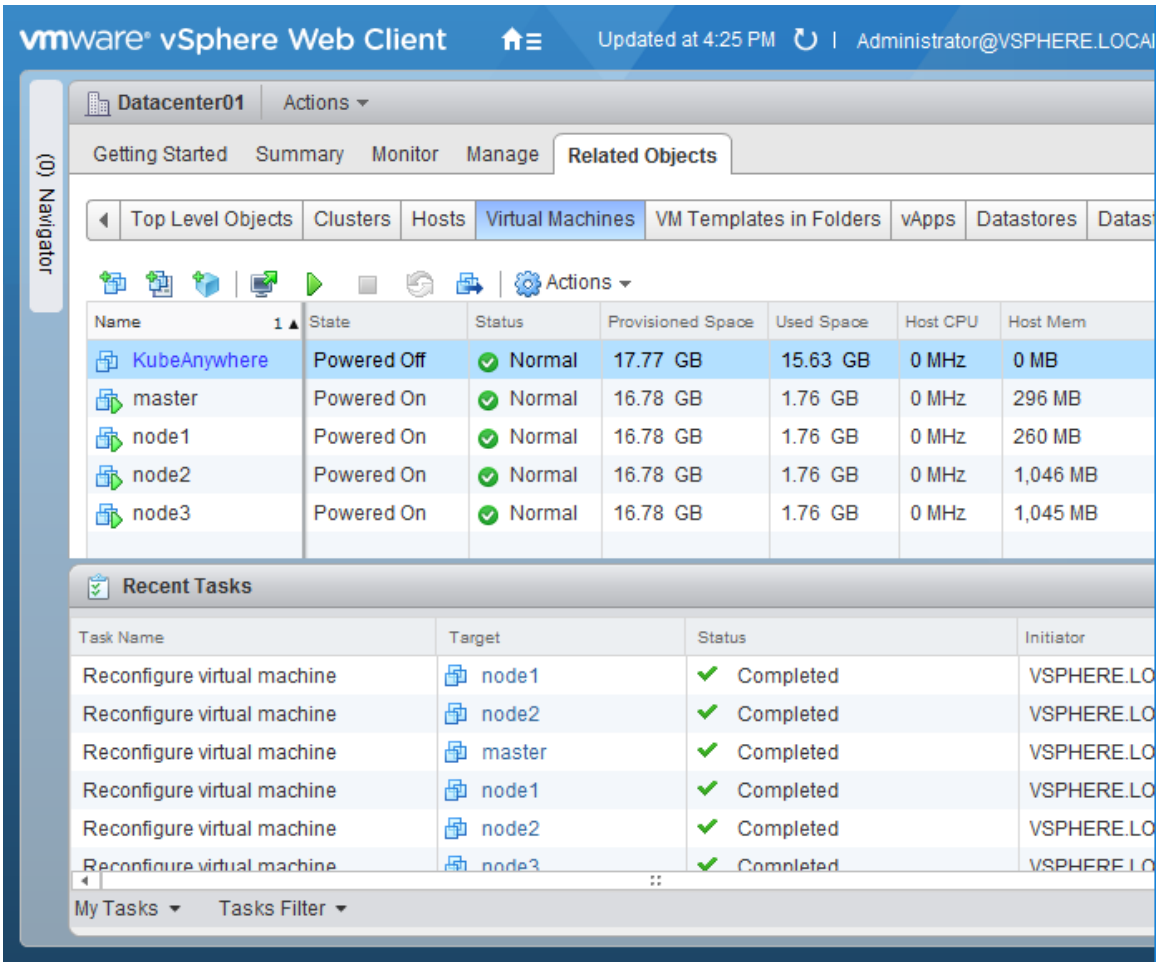

Figure 5.3: Dashboard on VMware vSphere Cloud Platform for Kubernetes Cluster

5.5.1 Setup: Kubernetes with Minikube

The first step is to set up the test environment for testing containers' start up speed. In keeping up our private cloud with VMware vSphere, we also set up a local deployment of Kubernetes with "Minikube" on a Windows 10 platform. In order to perform our tests, several additional packages need to be installed before deploying "Minikube". These packages include "PowerShell", "GoogleCloudAPI", "curl", and "Docker". More detailed instructions can be found in [29]. Figure 5.4 shows the results of a successful deployment of "Minikube". The Kubernetes cluster is created with a single master node.

In order to systematically deploy and test a container-based application, the steps listed below need to be followed:

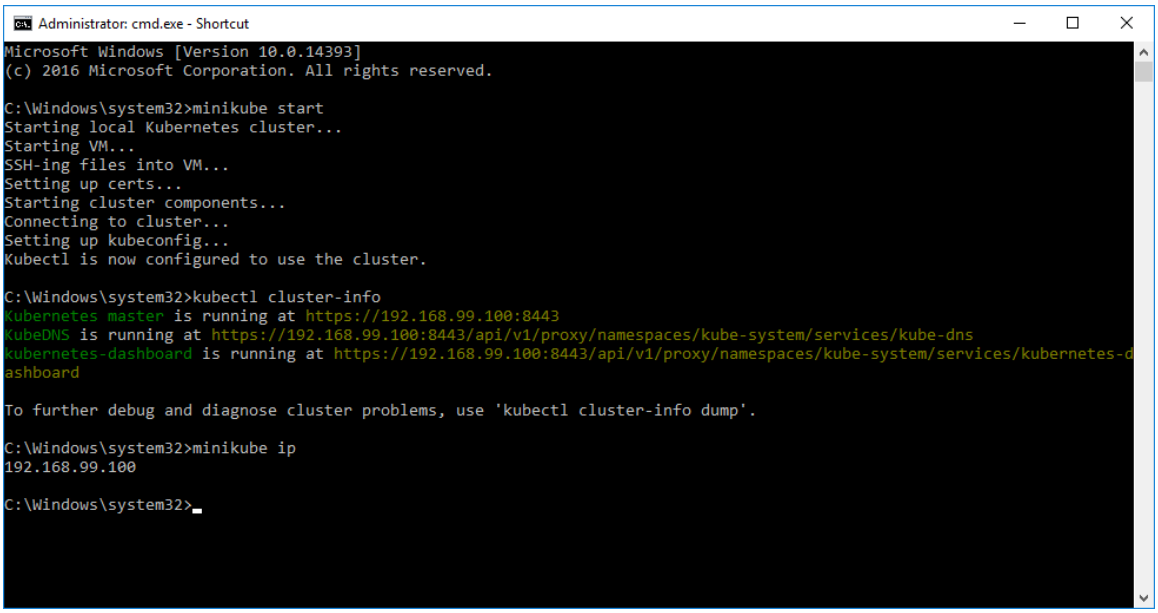

Figure 5.4: Setup for Minikube

1. Create a representative application. In this paper we use the typical "HelloNode" example with "Node.js" [61], in which a simple HTTP server is created and listening on port 8080. Any requests sent to this port can get a text response of "HelloWorld".

2. Create a Docker Container Image. A specific file named "Dockerfile" is required under the same folder as the application file. This "Dockerfile" describes the image. For more details about how to write a Dockerfile, please see [41].

3. Create a deployment on Kubernetes. A Kubernetes pod, which is a group of one or more containers, will be created for a "Deployment". The pod in the "HelloNode" example has only one container and the Kubernetes deployment will check the health of the pod, and restart the pod's container if it terminates [57].

4. Create a service. In order to make the "HelloNode" container accessible from the outside of the Kubernetes cluster, the pod needs to be exposed as a Kubernetes service. Once the pod is exposed as a service, a browser window is opened up with a local IP address that serves the "HelloNode" application. The browser shows the "Hello World" message.

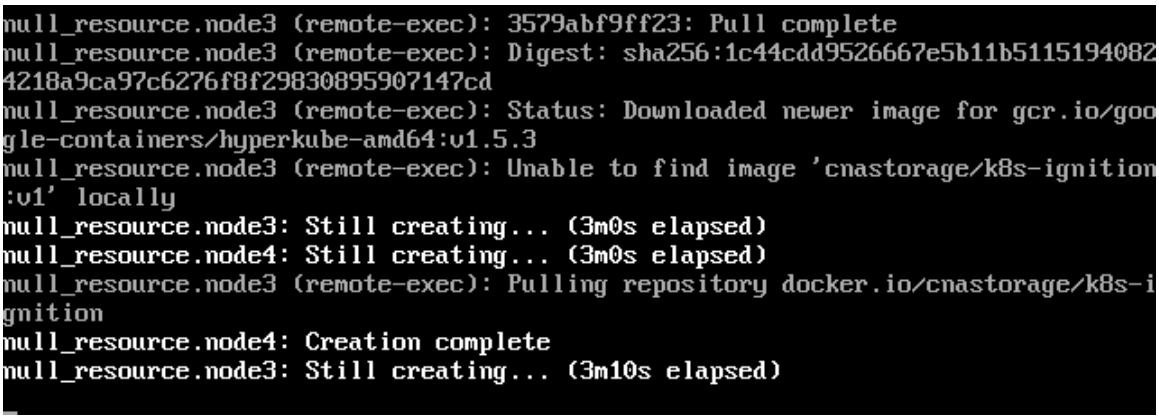

Figure 5.5: Virtual Machine Launch Time in a VMware vSphere Cloud Platform

5.6 Systematic Evaluation: Container Launch Times

A successful deployment can be verified if a similar result is achieved as shown in Figure 5.6. In this example, two pods are created and launched in about 25 seconds. As shown in Figure 5.7, a dashboard can be used to manage and collect the statistics of running containers.

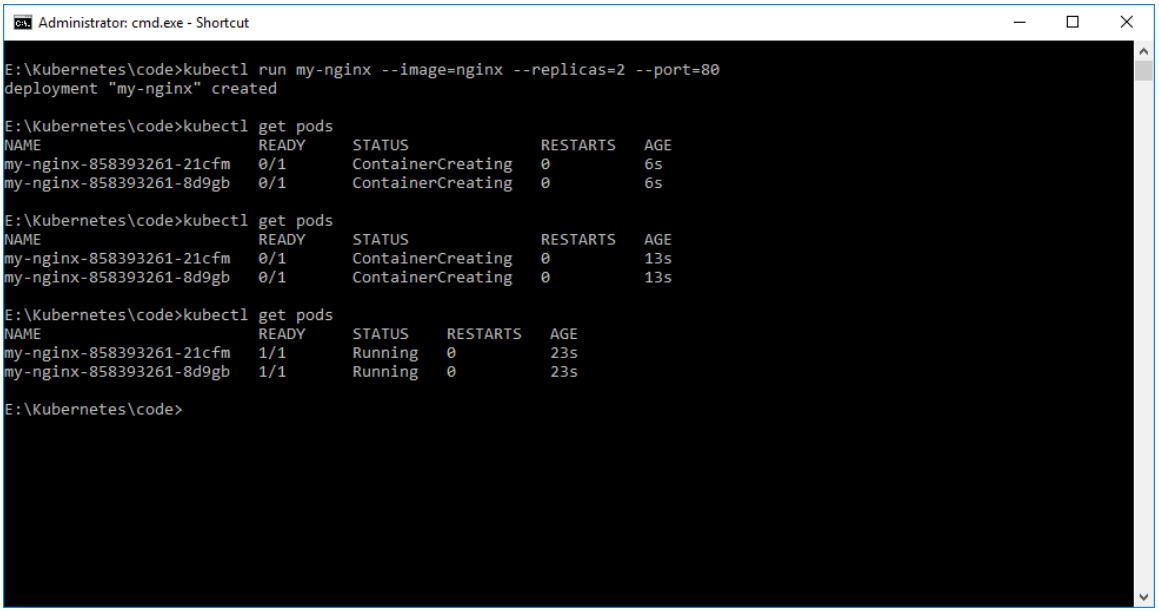

Figure 5.6: Starting Two Nginx Pods

Furthermore, in order to systematically evaluate containers' launch time in an accurate and reproducible way, we develop a python-based utility for the bench-

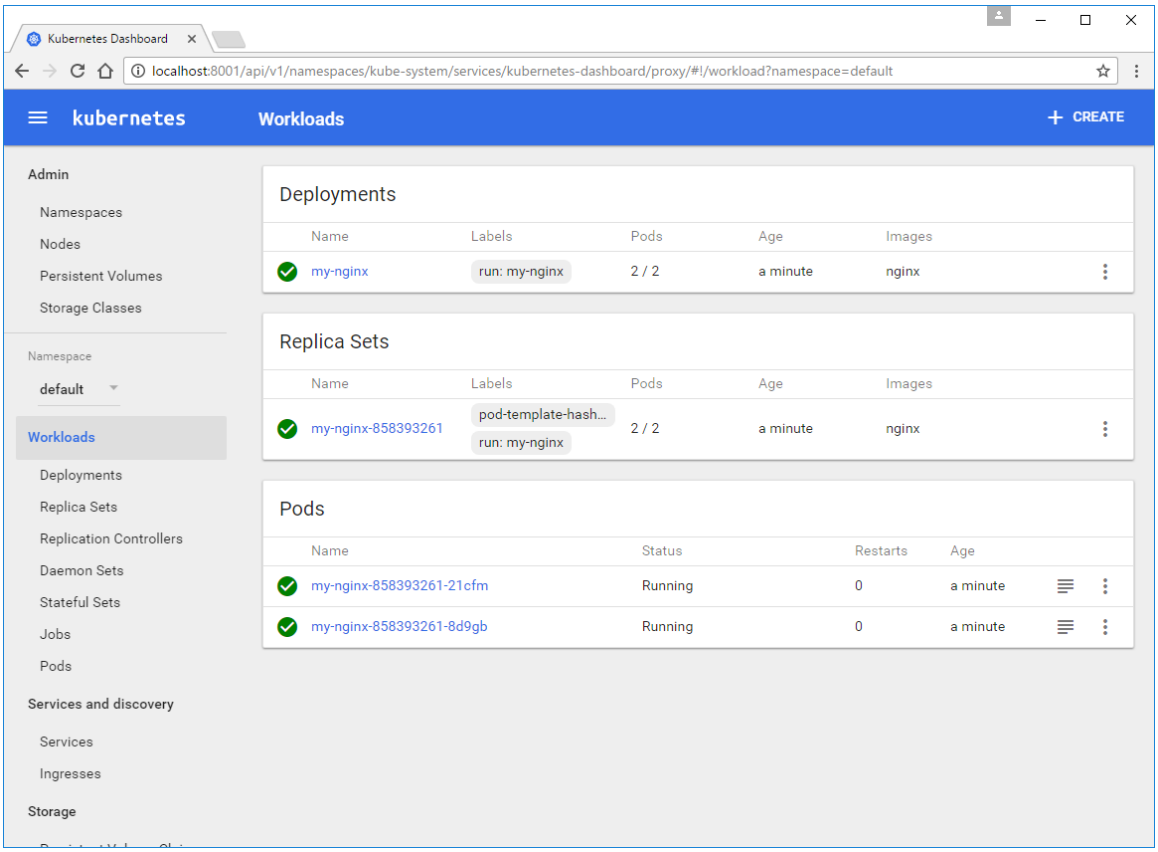

Figure 5.7: Kubernetes Dashboard for Two Nginx Pods

marking. Basically, this utility can launch a system at scale, create containers, run tasks and communicate with Kubernetes cluster on Minikube. The utility works in two modes, *singleton* and *batch*. The utility uses python-kube library to communicate with Minikube system, by making http based RESTful calls to Minikube API. The API exposes various methods for running and checking the status of clusters, nodes and pods.

In *singleton* mode, the utility fills the cluster with a number of nodes in the container that user specifies. The utility establishes the elapsed time and starts one, single, new container and collects the elapsed time again. In *batch* mode, the utility starts up and tests from 10 percent to 100 percent of the data load and reports the timing. Figure 5.8 shows the result of one successful launching process of the container.

In our experiments, each pod only has one container running. The total number

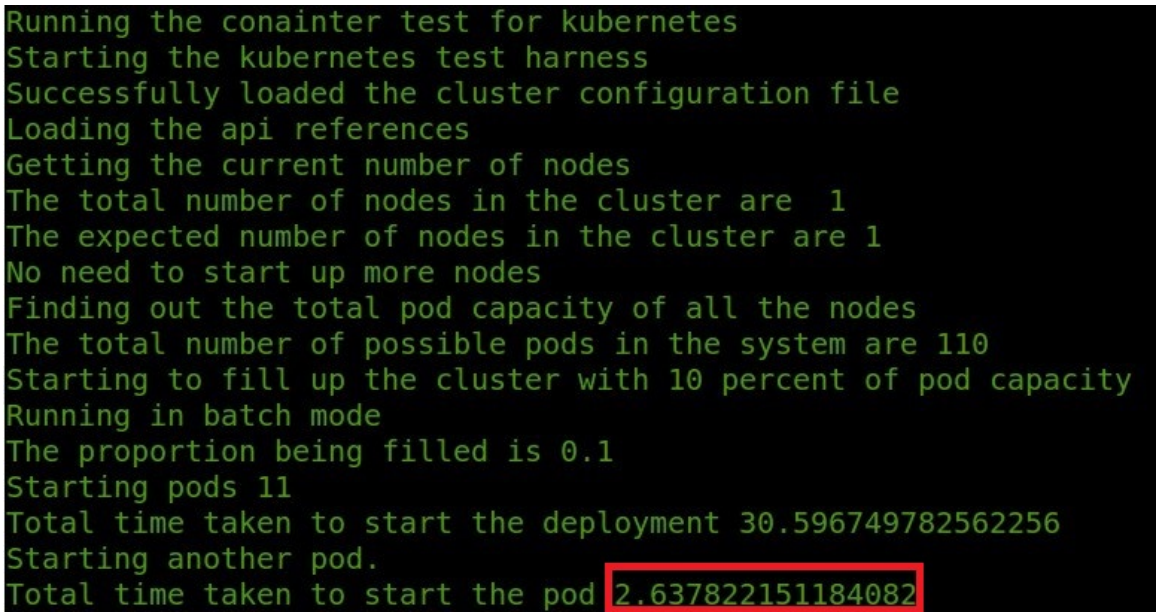

Figure 5.8: Container Launch Time

of the available pods is 110. We gradually scale up the number of pods in the cluster, from 10% to 100% of all the available pods. We then record the launch time for each new container under the corresponding full capacity of the system.

Table 5.3 illustrates a summary of our results. These metrics reveal that, in our specific case study, the launch for a new container increases 8x when the cluster goes from 10% to 80%. However, starts to drop when it reaches the peak. The take away here is that the launch times do not scale linearly. Though potentially not prohibitively, the nature of this unforeseen outcome demonstrates the need for the staged, systematic evaluation proposed in this paper. The cause of this outcome can be the result of a configuration option that could be tweaked in the configuration of the scheduling algorithm, or other management rules in Kubernetes. However, uncovering the default configuration options may not be necessary for small IT teams who lacks of time to do so. Moreover, the initial setup time for loading the containers is higher than the rest, this primarily reason due to how the infrastructure is setting up.

It is important to note that container images do not always need to be built from scratch. This is another benefit of using containers. There are plenty of container images existing in Docker community. They are ready to be used as a base image.

Table 5.3: Measuring Container Launch Time

Percentage Filled	Time to Fill (sec)	New Container Launch (sec)
10%	27.11	1.03
20%	13.707	1.952
30%	15.200	2.021
40%	14.924	2.361
50%	16.039	1.277
60%	18.710	2.906
70%	19.78	3.495
80%	**36.406**	**5.15**
90%	38.721	6.847

For example, we can create two nginx pods that are listening on port 80 with only one line of "kubectl" command:

```
[language=bash]
  $ kubectl  run my-nginx --image=nginx --replicas=2 --port=80
```

In the subsequent stage of the experiment, another local deployment with Minikube allows us to evaluate the average launch time of containers at different scales. Comparing to traditional VMs, these times are on the order of seconds. On top of that, the launch time of a new container does not scale linearly with an increasing load in a Minikube pod. In our case, we are able to establish a 8x increase in launch times between running at 10% to 80% capacity of pods available in the cluster. A virtualization with faster start up speed is the main factor we consider when building our Microservices style application.

5.7 Test Plan For Example Applications

In the first step, we run GIPA on a personal computer. In the second step, we test GIPA on Compute Canada. At the last step, we run TestKitchen on AWS cloud.

5.7.1 Spectral GIPA On Compute Canada

We use Compute Canada cloud and Singularity container to build GIPA and to process its huge amount of data. Here I demonstrate how the tests are conducted and in the next subsection the results will be shown.

In Compute Canada, the test cases are conducted following the requirements from the Spectral Lab. The Spectral Lab wants to remove noisy data from the images and bin the data, so that they can analyze these so called level 3 files.

We build a pipeline to perform the procedures, in order to run these experiments on Compute Canada. The pipeline script submits job to a Compute Canada scheduler. When the scheduler grants our resources request, the script will initialize container instances and run image process script inside the container. We wrap the polymer tool inside the Singularity container because Compute Canada software stack does not provide it.

The first task is to check how fast Compute Canada can processes these data. The process is finished in two steps. The first step generates level 2 output from the raw satellite images. This step mainly removes noisy data. The second step uses the level 2 file to generate level 3 files. When we get level 2 file after applying Polymer algorithm to the raw data, it is not yet finished. The Spectral Lab does not have time to analyze ten thousands of files. Beside, it's meaningless to do so, as many of the files are quite similar or useless. So the Spectral Lab introduces a new method called binning, this method can combine many level 2 files together to generate one level 3 file. To give an example, we do a monthly binning. On an average number, there are 150 files per month which generate 12 level 3 files after binning.

5.8 GIPA Result Analysis

In the first step, as mentioned in Chapter 3, we build GIPA application using Singularity container. The start up time of a Singularity container is almost instant, compared to VM instances in Compute Canada whose start up time can go up to a few minutes. Additionally, VMs are less flexible to use in a HPC environment. In

Compute Canada, web GUI needs to be used to create and lunch VM instances [7].

On a single machine, depending on the size of each image, it takes roughly 5 to 15 minutes to process an image. For a large size dataset, it is too dawdling. If something goes wrong during the process, the whole thing needs to be started all over again. After moving the tasks to Compute Canada cloud, the processing time is dramatically reduced. Now that we can request 100 nodes within a three hours slot to process the data set. Based on our test results, Cedar scheduler system grants 100 nodes within three days on average after all. This is because we run a large number of test cases synchronously on Compute Canada. As a consequence of this high frequent activity, our job priority is lower than other jobs in Compute Canada. Normally users get requested resources in one day. Nevertheless, we decrease the total processing time from months to days.

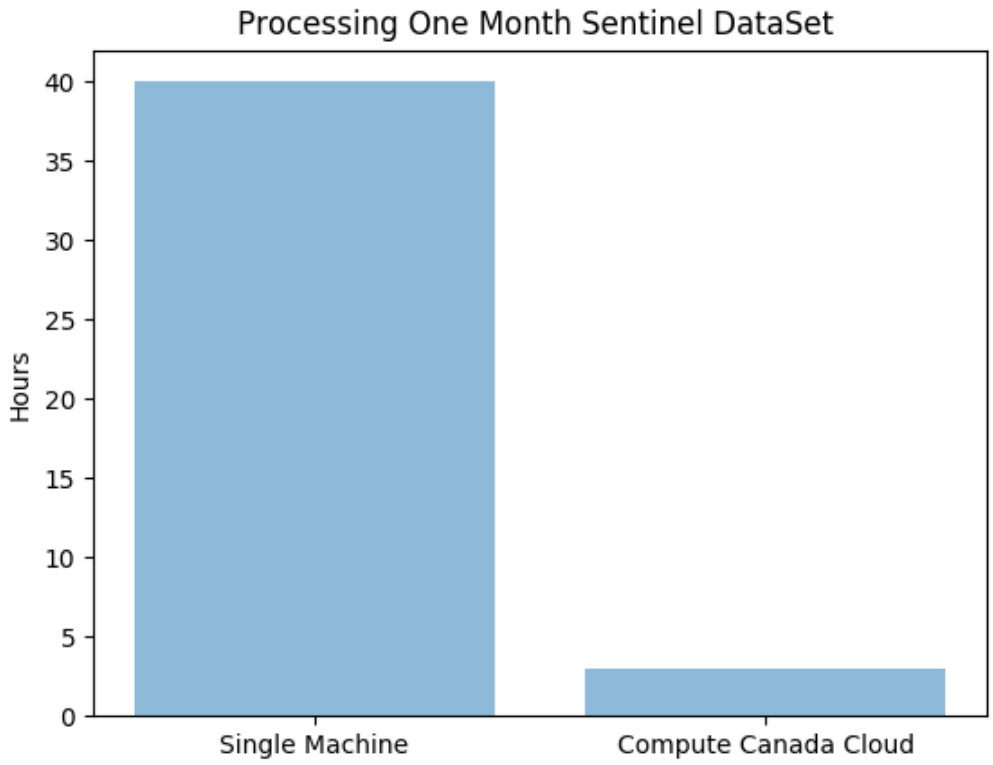

Figure 5.9: Single Machine vs Cluster

Figure 5.9 shows the result for one month data processed by a single machine and Compute Canada cluster.

Considering we run our jobs in a distributed way on Compute Canada, it faces an issue that multiple nodes may try to access one file at the same time. Even we have implemented a mechanism to prevent it from happening. It still occurs due to the design of the HPC scheduler system. When a node reaches its time limitation, the scheduler will stop the running instance, then all the ongoing jobs will fail. Consequently, these image files need to be re-processed anyway. The good news is that we manage to make the duplication rate significantly low.

As Table 5.4 indicates we run a job on more than 200 nodes to process over 1000 images. The total number of the files that are processed more than once is about 12 to 20. The total duplicating rate is less than 2%. This result demonstrates that GIPA almost fully utilized the cloud resources without significant waste.

Table 5.4: Measuring Duplication Rate on Compute Canada

Number of File	**Number of Nodes**	**Duplication Count**	**Rate**
1086	200	12	1.1%
1632	260	20	1.2%

5.8.1 Binning Analysis

We actually split the data on monthly base in order to bin level 2 images. It takes roughly seven and half hours to process one month's images. This time we process one year's level 2 images, and it is finished in one day now instead of twelve days. In fact, processing three years' data consumes the same amount of time. This is because we request 36 nodes for the whole data set at the same time. Besides, there is no competing as multiple nodes will not try to access the same file. Every node reads a specific folder only.

The results show Singularity container has overcame VMs' disadvantages. The

Singularity container starts almost instantly. If there is any issue in the container, we can shutdown the problematic one and restart a new one. It is better to utilize the resources in Compute Canada platform. Singularity's architecture design for HPC can spread to many nodes in a short time. The container CPU and memory can be updated by changing the configuration file. If you want to apply the same configuration change in a VM image, the re-build of the VM instance is required in Compute Canada platform.

Overall, the results meet the expectation of the Spectral Lab. Although Compute Canada has a complicated scheduling system, it gives every job a priority. Then based on the job priority, the scheduler determines when a job will be run. Still this mechanism introduces some troubles to GIPA. The binning job requires longer time to finish. Compute Canada only reserves a small percentage of nodes for long running jobs. Therefore, public cloud may be a good alternative for such situation.

5.9 TestKitchen On AWS

5.9.1 Docker on Mesos

Our partner is an AWS cloud customer, therefore, TestKitchen is running on AWS EC2. Running TestKitchen on AWS cloud gives us some considerable benefits, such as, avoiding a lot of setup time. The only work needed to do is initializing clusters on AWS platform. After servers are up, we can start our TestKitchen. Originally TestKitchen is running on Mesos.

5.9.2 Mesos On AWS

Public cloud platforms are different from Compute Canada. Most of them are not batch processing systems, which means users do not share resources. AWS cloud is one of them. Our partner deploys their applications on AWS. As a consequence, TestKitchen and other systems share infrastructure, but not the compute resources. When we ran TestKitchen application on our partner's AWS cluster, Mesos raised some problems.

The main issue of Mesos is the lack of horizontal scaling. Individual containers are grouped in a single server and orchestrated by Mesos. Even though this is a rare case in our test, there is no automated way to add a server when needed. Additionally, the container sizes (memory and disk resources) are statically allocated as part of the container definition. Consequently, the container sizes must be sized to the largest expected job. This is challenging, especially when we don't know how big the jobs might be. We may end up having a limited amount of memory on the server instance. Therefore, a more dynamical and scalable system would be ideal. Last but not the lease, the run-time of the jobs is currently defined by an http timeout interval and the API is designed for synchronous calls to a server. This design is much out of date and needs to be replaced with an asynchronous API.

Based on our investigation, Kubernetes is a better choice as it naively supports horizontal scaling. The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller based on pre-defined metrics. For instance, observed CPU utilization, or some other application-provided metrics etc, which can totally free our Ops staff from manually scaling the cluster.

Kubernetes supports vertical scaling as well. Nevertheless, vertical scaling is more difficult than horizontal one, especially, when dealing with stateful container application. Stateful applications, like database, require keeping the internal state of the server. After a vertical scale, containers need to be restarted. This situation makes vertical scaling become a hard choice. If we restart a stateful container, all the current connections will lost and need to reconnect after the restart process is finished. Overall, we believe this is not a critical issue for the TestKitchen application. Even we do not know the largest image size in advance, we can do various evaluation to find a suitable value. Beside we use Microservices pattern, changing one container's specification does not impact other container. We can rapidly change the impacted container's configuration and push the changes to code repository, then CI/CD tool can take over and do the rest tasks automatically.

5.10 Development and Maintenance Experiences

It is challenging to compare the development and maintenance processes among the different approaches we have distinguished so far. We do not have a similar application that is developed following traditional software development flow. Our partner company has such experiences, as they have been doing business in software industry for decades. They have both the monolithic and Microservices software development experiences. I have got some feedback from TestKitchen project coordinator. Including my own industrial experience, Microservices and DevOps have the following listed benefits.

Firstly, it can shorten software release time. In our case, it only took us less than a year for 3 developer to build the TestKitchen. In addition, we released our first version only in 4 weeks. This rapid iteration cycle allows team to develop new features and get end users feedback quickly. As a result, we can change our application quiet often without worrying the time is wasted to develop some features that end users may not be interested.

Secondly, it reduces bugs and defects. Yet, controversially, most of the bugs and defeats are reported by ourselves.

Thirdly, development team has the freedom to choose the right technical stack to do the job. In our case, the front-end of TestKitchen is a JavaScript application. The back-end is a Python application. The Microservices architecture allows the development team to choose the best tools from the existing tool set, like programming language, build tool etc.

Last but not the least, with CI/CD tool Jenkins, we automated the whole process. When a developer submits his/her code to a repository, it triggers a build process to compile the code and run the unit tests. Then Jenkins executes integration tests and builds Docker image. Further, Jenkins pushes the image to the project's Docker repository. Finally, Jenkins runs add-hoc job to login to the server and deploy the latest Docker image.

At last, I would like to summarize the different cloud platforms advantages and

disadvantages. Table 5.5 displays the results of comparison.

Table 5.5: Comparison of Different Platforms

	Single Machine	**Compute Canada**	**AWS**
Availability	single failure	High available	Always on
Opportunity for DevOps	low	middle	high
Scalability	no support	support but hard to automatic	easy to scale
Usability	hard to use	hard to use	user-friendly

5.11 Summary

In this chapter, I list test cases that we used to verify Docker and Kubernetes performance. Then, I shown how GIPA processed huge amount of satellite imagery data on Compute Canada. From the test results, we can see the container do better job in the initialization phase. It is much faster to initialize a container instance than a VM instance. What's more, the launch time of containers can be improved if caching is enabled. Also in the same platform, the Docker container can initialize more instances than the VM, because it does not hold resources exclusively. Only when it requires the resources then the container will ask from the hosted system. In contrast to container, a VM takes all resources when it starts. After that, even if the VM is in idle status, other VMs can not use these resources.

The GIPA result shows a significant process time reduction on Compute Canada platform. Still, the scheduler algorithm makes our job logically complicated. Based on our experience learned from GIPA, we design and build a new application named TestKitchen. At the end, we ran TestKitchen Application on AWS platform. On a public cloud, like AWS, we do not need to consider schedule policy. We have observed some issues when running TestKitchen on Mesos. Therefore. we proposed to use Kubernetes in the future.

The container image size is also extremely small. A production VM image is normally several Gigabytes, a container image can be as small as few hundred Megabytes.

The TestKitchen Front-end container image is only about 200 Megabytes.

On one hand, Docker container does have security issues because it shares host. On the other hand, the VM totally isolates each other. If a VM is hacked by a hacker, the others remain safe. Unfortunately, if hackers break a container's host, they may get all containers' information that running on this host. The good news is that Docker community is working on this issue. In addition, Singularity container's special design can bypass this problem. The Singularity container does not require root privilege most of time. This secures the use of Linux containers on Linux multi-user clusters. Further more, Singularity container has solved a serious problem in HPC environment that is how to manage the software stack we need in our task.

Compute Canada helps to process a massive data set in a short time. As a HPC batch processing system, all tasks must go through a command line style scheduler. This makes Compute Canada that is not a user friendly environment. Users face a steep learning curve. As an improvement, we have developed a GUI-based application that aims to process satellite images. Clients can use their browser instead of the terminal window to execute their algorithms.

Chapter 6

Conclusions and Future Work

The emerging of Microservices architecture and DevOps has empowered agility in software communities. Furthermore, containerization is promising to make a difference in the IT in a more profound way than full virtualization, especially in high performance computing area. Containers offer users more flexibilities than Virtual Machines (VMs). A successful monolithic application eventually will grow into a gigantic block that very few, if any, developers understand. The application becomes more and more difficult to scale and developers cannot do quality control. As a result, agile development and continuous delivery will be impossible. Therefore, development teams split a project into several small sub-projects. A sub-project means that a compacted development team can handle it, and they can adopt agile more efficiently. The Microservices architecture enables each sub-project to be developed independently by a small team.

I believe that Microservices and DevOps are able to provide benefits. Firstly, Microservices tackle the problem of complexity. Next, the Microservices architecture enables each service to be deployed independently. Finally, Microservices architecture pattern allows each service to be scaled separately. DevOps supports development and deployment too. In this work, I demonstrate how to design and implement applications using the Microservices and DevOps paradigm. The container technology offers a great opportunity to implement software companies goals. Container technology is a new attempt to reduce the virtualization overhead in cloud platforms. Software communities are working on improving software development efficiency, and making software easy to maintain. In this work, I have developed two applications. The results show that Microservices and DevOps help to reduce development time

and keep applications in good quality.

Choosing a methodology is the first step. Right tools can let practicers go further and achieve their goals. In order to manage Docker containers, companies choose container orchestration tools, for instance, Mesos and Kubernetes to assist DevOps engineers to manage the containers, and Continuous Integration (CI)/Continuous Delivery (CD) tools like Jenkins to build and deploy applications.

6.1 Contributions

In this work, I did not solely containerize an application with Docker. From a computer science student perspective, the practice specifically targets cloud environments and optimizes applications for better performance. Because the software communities realize if software teams just move an application from VM to container, the benefit is small. A cloud-native application can be a better solution. The development phase is part of the evaluation results. From the beginning, the development teams have adapted DevOps and Microservices in this work.

First, an application is developed to process a large volume of geospatial image data using Singularity container. The rough processing time was reduced from several months to days. After the Spectral Lab run the scripts on Compute Canada instead of a lab computer, the performance of Spectral Lab GIPA in Compute Canada cloud platform is expected as estimated. The overhead of the duplication is outstandingly small. Also Singularity container makes installation and maintenance software on Compute Canada less complex and error-prone.

Secondly, a prototype of GUI-based satellite images application was delivered in a short time. There were more features added to TestKitchen compared to the one run on Compute Canada. TestKitchen was deployed on Amazon Web Services (AWS) cloud. This product follows the popular practices of the current software communities. TestKitchen runs Git as its version control tool, and CI/CD tool Jenkins together with Mesos to fully explore DevOps. Both applications were designed and implemented by applying Microservices framework and DevOps paradigm. Developing these applications gives the development teams first hand experience of these

frameworks. Docker container and container orchestration tool, such like Mesos and Kubernetes, are running and managing these applications.

6.2 Limitations

The first problem is that TestKitchen cannot run test cases on AWS cloud due to the cost issue. Secondly, Compute Canada does not support Docker container because of some technical reasons. Instead Singularity container is chosen to replace Docker. Even though Singularity container declares 100% compatibility with Docker container, there are still some subtle differences that may affect the accuracy of test results. Furthermore, our observation results collected from our partner's infrastructure is based on their backend architecture which they use for their own business. Hence, the infrastructure is not specifically optimized for TestKitchen application.

The last problem is that Compute Canada is a batch processing system, which means the resources are shared by every user. Every user must submit their request to a scheduler. The scheduler will determine priority for a job. Commonly, the longer a user request is, the lower priority is assigned to the job. This mechanism makes job management really hard. It is not like a public cloud platform, because users have dedicated resources under their complete control.

6.3 Microservices and DevOps Successful Factors

By developing TestKitchen and Spectral Lab GIPA on Compute Canada using Microservices pattern, I have summarized some tips that can help to reach a successful end.

First of all, development teams and companies must embrace virtualization. It is the essential element to success in this era. Then, automation makes everything easier than manual activity. Reducing manual operation means fewer errors. Meanwhile, it saves team members' time to do more valuable tasks. One indispensable measure for automating the software development environment for Microservices architecture is to employ CI/CD. By integrating with CI/CD, developers can detect

bugs or any other code quality issues as early as possible. Every time they commit code to a repository, the CI/CD system will automatically build the code, run unit tests and integration tests, build a Docker image using the latest code, and finally deploy the new version to a system by pulling a new Docker image from the repository. It can save highly valuable time for engineers to solve real problems instead of repeating repetitive tasks. This is the most important lesson the development teams have learned from these projects.

6.4 Summary

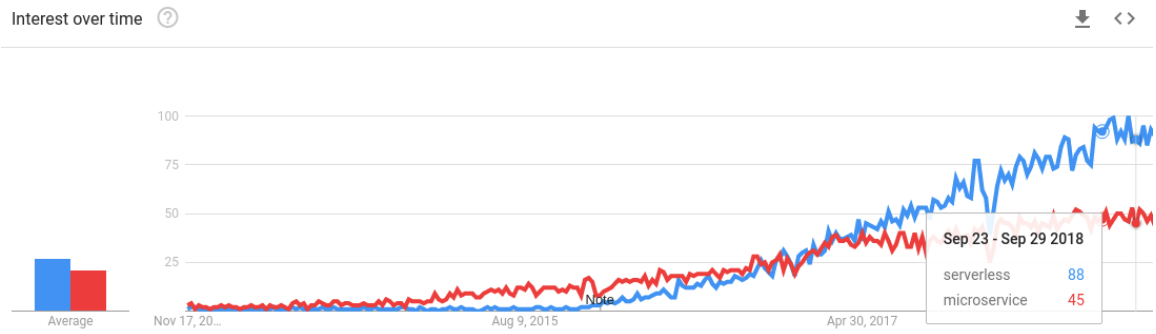

Figure 6.1: Google Trends for Serverless and Microservices

The evolution of software development practice is continuing. Just as Microservices stemmed from Service-oriented architecture (SOA), the new breakthrough is now Serverless framework. Figure 6.1 shows the Serverless search trend is sharply increasing in the past few years. Serverless framework significantly eases Ops task. Virtualization technology is under development as well, from full virtualization to Operating System (OS) level lightweight virtualization. Unlike traditional VMs, the containers are faster to launch due to their lightweight properties and low overhead.

Based on experience learned from this thesis, new participants can avoid the same flaws that we were faced with. Microservices and DevOps positively help software development teams to quickly release their prototype and rapidly push new features to customers. Together with CI/CD tools, developers can deploy code to cloud platforms after developers commit their code to project repositories.

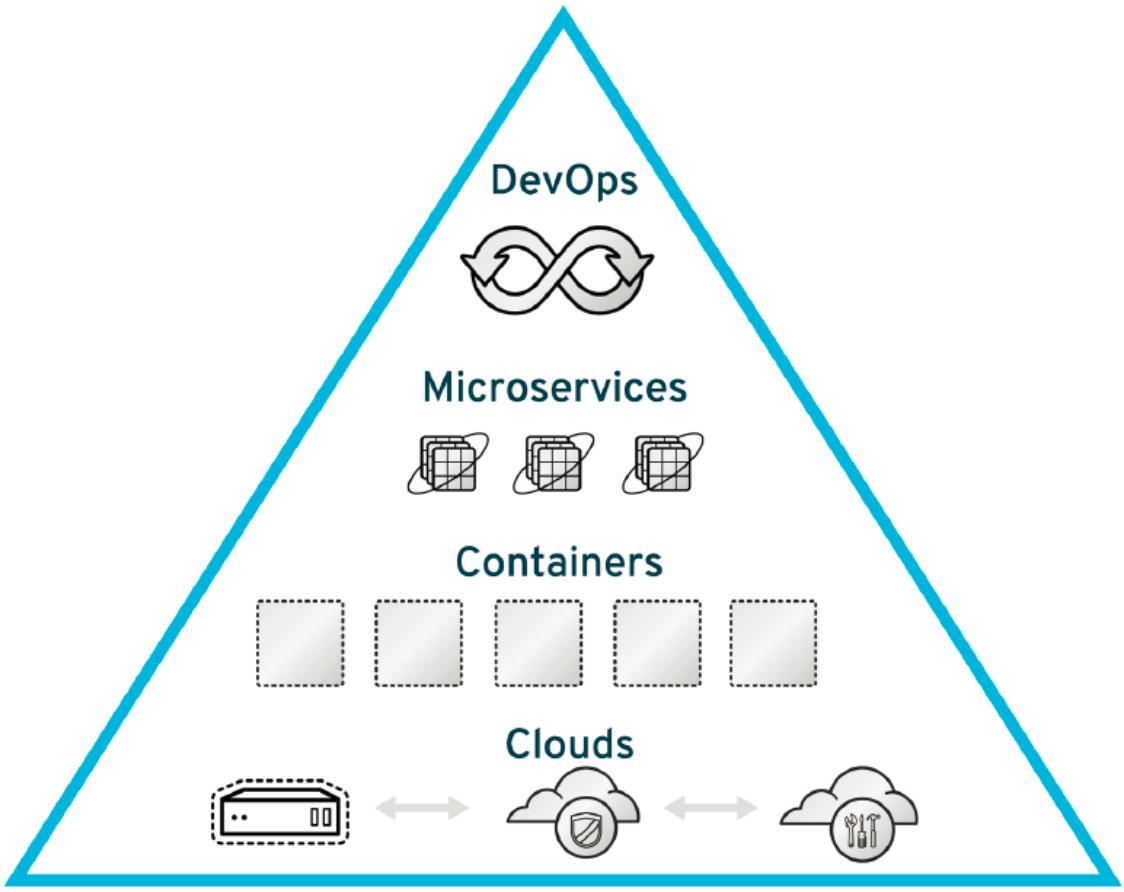

Figure 6.2: From DevOps to Clouds
Source: [2]

In public clouds, there are tools like Kubernetes and Mesos. The container cluster management tools help users control their container fleet. Development teams significantly reduce maintenance cost, so DevOps staff can spend more time on development tasks instead of being stuck in a cloud server terminal and trying to debug. Kubernetes can detect problematic containers and automatically terminate them and initialize new instances. Even better, Kubernetes can do autoscaling automatically. Thus users do not monitor log systems to figure out if a container is overloaded. Figure 6.2 displays a pyramid that describes my work. The fundamental pillar is clouds platforms. Using the container to deploy our application. Microservices help us to implement the continuous delivery and DevOps make the fast iteration possible.

Even in a High Performance Computing (HPC) environment, users have also

found that container technology improves efficiency. A Big Data imagery processing application was developed to process geospatial satellite images. By using Singularity container, HPC users avoid installing software in the HPC environment. Users can completely control software tool by themselves in the HPC environment. Thus users use Compute Canada computing resources to process a huge data set in a short time.

Software communities are facing many open questions. We share our lessons in this paper. The Microservices architecture plus DevOps give the software team an advantage in the competition. With the help of container technology, we accelerate software time to market, quickly solve issue resolution and increase IT operational efficiency.

As always, Microservices also brings some drawbacks. It faces the same challenge as a monolithic software, along with a project growth, development team receives more requirements, which means development team have to create more microservices. In our experiment, we have six microservices, and it is easy to manage. Imagine if there are fifty or even hundreds of microservices in a project, it will be hard to manage them. More services means increasing of complexity of an application. There are more barriers between each service. These barriers include communication between each service, it can be even more complicated compared to the function call in a monolithic software. Further, when there are more services in application, testing a task becomes more difficult, because developers may not have upstream and downstream systems ready for test. This situation makes test plans harder than a monolithic application. But I believe that these issues are new research areas for other researchers.

6.5 Future Work

As a pilot project, TestKitchen is immature. There are still many places that can be improved. For instance, the users should be allowed to save their algorithms to cloud storage and to support version control of these algorithms. This feature can let user easily manage their algorithms history.

Moreover, TestKitchen can apply Serverless technique to improve service effi-

ciency. By using new virtualization technologies, such as gVisor and Firecracker, TestKitchen can get good isolation, low overhead, and high performance. This can make TestKitchen easy to use Serverless framework.

Cloud users can use API Gateway service proxy that route requests to their application. Or cloud users can use Serverless framework, such as AWS Lambda, to handle requests. Cloud users also can put their application into Docker container that runs on EC2 instance. It is not easy to distinguish which one is overall the best one for an application. It could be an interesting question from the perspective of performance.

The ultimate goal is applying machine learning algorithms to analyze satellite images. Letting machine learning technology help us to distinguish clouds from images is a classic example. While this is a tedious task for human beings, it is relatively easy for ML algorithms. Distinguishing cloud from images is a typical classification problem. We teach our algorithm what a cloud looks like, then leave ML algorithm to learn and finally tell us if an image contains fire or not.

Beside that, TestKitchen can become a general purpose platform in the future. It can be extended to deal with other type of images. This will be a good research topic in the future.

Appendix A

Appendix

A.1 Playground Dockerfile

```
FROM docker.urthecast.com/ds-sandbox/alpine-gdal

LABEL ImageBaseName=playground

ENV PYTHONUNBUFFERED=1

# Install Python
RUN apk add --update \
    python \
    py-pip \
  && rm -rf /var/cache/apk/*

# Install PROJ.4
RUN apk add --update \
    build-base \
  && cd /usr/src \
  && curl -SL "http://download.osgeo.org/proj/proj-4.9.2.tar.gz" -o
   ↪  proj-4.9.2.tar.gz \
  && tar -xzvf proj-4.9.2.tar.gz \
  && cd proj-4.9.2 \
  && ./configure \
```

```
21    && make \
22    && make install \
23    && cd / \
24    && rm -f /usr/src/proj-4.9.2.tar.gz \
25    && rm -rf /usr/src/proj-4.9.2 \
26    && apk del \
27      build-base \
28    && rm -rf /var/cache/apk/*
29
30
31  ENV NUMPY_VERSION="1.11.1" SCIPY_VERSION="0.17.0"
32
33  # Install System Dependencies
34  RUN apk add --update \
35      build-base \
36      gfortran \
37      python-dev \
38      lapack-dev@edge-community \
39      lapack@edge-community \
40      hdf5@edge-testing \
41      bash \
42      zlib \
43      freetype \
44      libpng \
45      libjpeg-turbo \
46    && ln -s /usr/include/locale.h /usr/include/xlocale.h \
47    && pip --no-cache-dir install numpy==$NUMPY_VERSION \
48    && pip --no-cache-dir install scipy==$SCIPY_VERSION \
49    && apk del \
50      build-base \
51      gfortran \
52      python-dev \
53      lapack-dev \
54    && rm -rf /var/cache/apk/*
55
```

```
56 RUN mkdir /code
57
58 WORKDIR /code
59
60 ADD requirements.txt /code/
61
62 # Install App Dependencies
63 RUN apk add --update \
64     build-base \
65     gfortran \
66     python-dev \
67     hdf5-dev@edge-testing \
68     zlib-dev \
69     freetype-dev \
70     libpng-dev \
71     libjpeg-turbo-dev \
72   && pip --no-cache-dir install --upgrade pip \
73   && pip --no-cache-dir install --upgrade setuptools \
74   && pip --no-cache-dir install -r requirements.txt \
75   && apk del \
76     build-base \
77     gfortran \
78     python-dev \
79     hdf5-dev \
80     zlib-dev \
81     freetype-dev \
82     libpng-dev \
83     libjpeg-turbo-dev \
84   && rm -rf /var/cache/apk/*
85
86 ADD . /code/
87
88 # Image Build Metadata
89 ARG GIT_COMMIT=none
90
```

```
LABEL GitCommit=$GIT_COMMIT
```

A.2 Docker-Compose File

```
 webserver:
image: nginx:1.9
environment:
  TERM: "xterm-256color"
volumes:
  - ./docker/nginx/www.conf:/etc/nginx/conf.d/www.conf:ro
  - .:/code
ports:
  - "8080:8080"
links:
  - tile
  - collector
le:
build: .
environment:
  TERM: "xterm-256color"
  PYTHONDONTWRITEBYTECODE: "1"
  PYTHONPATH: "/code/playground"
command: gunicorn -c python:gunicorn_config --reload app:app
volumes:
  - .:/code
links:
  - db
  - cache
llector:
build: .
environment:
  TERM: "xterm-256color"
  PYTHONDONTWRITEBYTECODE: "1"
  PYTHONPATH: "/code/playground"
```

```
31 command: gunicorn -c python:gunicorn_config --reload app:app
32 volumes:
33   - .:/code
34 ports:
35     - "8000:8000"
36 links:
37   - db
38   - cache
39 :
40 image: mongo:3
41 environment:
42   TERM: "xterm-256color"
43 ports:
44   - "27017:27017"
45 che:
46 image: redis:3
47 environment:
48   TERM: "xterm-256color"
49 command: redis-server /usr/local/etc/redis/redis.conf
50 volumes:
51   - ./docker/redis/cache.conf:/usr/local/etc/redis/redis.conf
52 ild:
53 image: node:4-onbuild
54 environment:
55   TERM: "xterm-256color"
56 command: bash /code/viewer2/bld.sh
57 volumes:
58   - .:/code
```

A.3 Singularity Recipe

A.3.1 Polymer Image

```
BootStrap: debootstrap
OSVersion: xenial
MirrorURL: http://us.archive.ubuntu.com/ubuntu/

%runscript


%files
    polymer-v4.8.tar.gz
    test1.py
    test2.py

%post
    sed -i 's/$/ universe/' /etc/apt/sources.list

    apt-get -y install software-properties-common build-essential

    apt-get -y update
    apt-get -y install python3 wget python3-pip
    add-apt-repository -y ppa:ubuntugis/ppa

    apt-get -y update
    apt-get -y install gdal-bin python-gdal python3-gdal
     ↪ python-pyproj libhdf4-dev python libgrib-api-dev
     ↪ libgrib2c-dev libnetcdf-dev netcdf-bin
    apt-get -y update

    pip3 install jupyter matplotlib numpy pandas scipy
    pip3 install pyepr
    pip3 install cython pyproj
    pip3 install python-hdf4 glymur lxml
```

```
    pip3 install netcdf4
    wget -nc https://hdfeos.org/software/pyhdf/pyhdf-0.9.0.tar.gz
    tar zxvf pyhdf-0.9.0.tar.gz
    cd pyhdf-0.9.0
    python3 setup.py install
    python3 -m wheel convert
     ↪  /pyhdf-0.9.0/dist/pyhdf-0.9.0-py3.5-linux-x86_64.egg
    python3 -V
    pip3 install /pyhdf-0.9.0/pyhdf-0.9.0-cp35-none-linux_x86_64.whl
    cd /
    tar zxvf polymer-v4.8.tar.gz
    cd /polymer-v4.8
    make auxdata_all
    python3 setup.py build_ext --inplace
    make ancillary
    pwd
    cp /test1.py test1.py
```

A.3.2 SNAP GPT Command Line Utility Image

```
BootStrap: debootstrap
OSVersion: xenial
MirrorURL: http://us.archive.ubuntu.com/ubuntu/


%runscript
    exec /usr/local/snap/bin/gpt "-h"

%files
    esa-snap_sentinel_unix_6_0.sh

%post
#change file execution rights for snap installer
    chmod +x esa-snap_sentinel_unix_6_0.sh

```

```
# install snap with gpt
    ./esa-snap_sentinel_unix_6_0.sh -q

# link gpt so it can be used systemwide
    ln -s /usr/local/snap/bin/gpt /usr/bin/gpt

# set gpt max memory to 4GB
    sed -i -e 's/-Xmx1G/-Xmx4G/g' /usr/local/snap/bin/gpt.vmoptions
```

A.4 Polymer Docker Dockerfile

```
FROM ubuntu:18.04
MAINTAINER gaobing@uvic.ca

COPY polymer-v4.9.tar.gz /
    #here you need put the script name here.

RUN    apt-get -y update
RUN    apt-get -y install software-properties-common build-essential

RUN    apt-get -y update
RUN    apt-get -y install python3 wget python3-pip

RUN    apt-get -y install gdal-bin python-gdal python3-gdal
 ↪  python-pyproj libhdf4-dev python3-h5py libgrib-api-dev
 ↪  libgrib2c-dev libnetcdf-dev netcdf-bin
RUN    apt-get -y update

RUN    pip3 install jupyter matplotlib numpy pandas scipy
RUN    pip3 install pyepr
RUN    pip3 install cython pyproj
RUN   pip3 install python-hdf4 glymur lxml
RUN    pip3 install netcdf4
RUN      pip3 install h5netcdf
```

```
22 RUN    wget -nc https://hdfeos.org/software/pyhdf/pyhdf-0.9.0.tar.gz
23 RUN    tar zxvf pyhdf-0.9.0.tar.gz
24 WORKDIR pyhdf-0.9.0
25 RUN    python3 setup.py install
26 RUN    python3 -m wheel convert
    ↪ /pyhdf-0.9.0/dist/pyhdf-0.9.0-py3.6-linux-x86_64.egg
27 RUN    python3 -V
28 RUN    pip3 install
    ↪ /pyhdf-0.9.0/pyhdf-0.9.0-cp36-none-linux_x86_64.whl
29 WORKDIR ../
30 RUN    tar zxvf polymer-v4.9.tar.gz
31 WORKDIR  polymer-v4.9
32 RUN    make auxdata_all
33 RUN    python3 setup.py build_ext --inplace
34 RUN    make ancillary
35 RUN   pip3 install filelock
36 COPY test1.py /polymer-v4.9
37
38 ENTRYPOINT [ "sh", "-c", "echo $HOME" ]
```

Bibliography

[1] Devops diagram, (accessed Apr 12, 2019). https://www.slideshare.net/barton808/devops-microservices-and-containers-a-high-level-overview.

[2] From devops to clouds, (accessed Apr 12, 2019). https://www.infoq.com/news/2015/11/eisele-microservices.

[3] Microservices vs monolithic, (accessed Apr 12, 2019). https://medium.com/devopslinks/microservice-architecture-is-it-right-for-your-software-developmer

[4] Container vs vm architecture, (accessed Apr 13, 2019). https://pawseysupercomputing.github.io/container-workflows/02-about-containers/index.html.

[5] Aws lambda official site, (accessed Dec 17, 2018). https://aws.amazon.com/lambda/.

[6] Cloud platform catalog, (accessed Dec 17, 2018). https://dzone.com/articles/iaas-vs-caas-vs-paas-vs-faas-choosing-the-right-platform/.

[7] Compute canada vm guide, (accessed Dec 17, 2018). https://docs.computecanada.ca/wiki/Cloud_Quick_Start.

[8] Infrastructure as a service, (accessed Dec 17, 2018). https://en.wikipedia.org/wiki/Infrastructure_as_a_service.

[9] Polymer official site, (accessed Dec 17, 2018). https://www.hygeos.com/polymer/.

[10] React official site, (accessed Dec 17, 2018). https://reactjs.org/.

[11] Redis official site, (accessed Dec 17, 2018). https://redis.io/.

[12] Singularity official site, (accessed Dec 17, 2018). https://www.sylabs.io/docs/.

[13] Snap official site, (accessed Dec 17, 2018). https://senbox.atlassian.net/wiki/spaces/SNAP/overview.

[14] Software as a service, (accessed Dec 17, 2018). https://en.wikipedia.org/wiki/Software_as_a_service.

[15] Firecracker github site, (accessed Feb 17, 2019). https://github.com/firecracker-microvm/firecracker.

[16] gvisor github page, (accessed Feb 17, 2019). https://github.com/google/gvisor.

[17] Compute canada official site, (accessed Mar 17, 2019). https://www.computecanada.ca/.

[18] Docker website, (accessed Mar 17, 2019). https://www.docker.com/.

[19] Google cloud platform homepage, (accessed Mar 17, 2019). https://cloud.google.com/.

[20] Kubernetes structure, (accessed Mar 17, 2019). https://platform9.com/blog/kubernetes-vs-mesos-marathon/.

[21] Platform as a service, (accessed Mar 17, 2019). https://en.wikipedia.org/wiki/Platform_as_a_service.

[22] Virtualbox, (accessed Mar 17, 2019). https://www.virtualbox.org/.

[23] Amazon aws homepage, (accessed Nov 30, 2018). https://aws.amazon.com/.

[24] Heba Ali, Mohammed Elmogy, and Shereif Barakat. A big data processing framework based on mapreduce with application to internet of things. volume 31. 07 2016.

[25] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. volume 33, pages 42–52. IEEE, 2016.

[26] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[27] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. 2003.

[28] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[29] D. Bernstein. Containers and cloud: From LXC to docker to kubernetes. volume 1, pages 81–84. September 2014.

[30] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. volume 1, pages 81–84. IEEE, 2014.

[31] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.", 2016.

[32] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. volume 59, pages 50–57. ACM, 2016.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. volume 51, pages 107–113. ACM, 2008.

[34] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. volume 33, pages 94–100. IEEE, 2016.

[35] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. 2015.

[36] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, USA, 2000.

[37] Martin Fowler and Matthew Foemmel. Continuous integration. volume 122, page 14. 2006.

[38] Martin Fowler and James Lewis. Microservices. volume 28, page 2015. 2014.

[39] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. volume 60, page 80. 2016.

[40] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22. 2011.

[41] K. Hirokuni. Gotchas in Writing Dockerfile. 2014. [Online]. Posted at: 2016-05-27.

[42] Paco Hope. Using jails in freebsd for fun and profit. volume 27. 2002.

[43] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Boston, 2011.

[44] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. 2010.

[45] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. 2019.

[46] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116. 2000.

[47] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure devops. In *Cloud Engineering (IC2E), 2016 IEEE International Conference on*, pages 202–211. 2016.

[48] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. 2007.

[49] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011.

[50] Lubos Mercl and Jakub Pavlik. The comparison of container orchestrators. In *Third International Congress on Information and Communication Technology*, pages 677–685. 2019.

[51] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. volume 2014, page 2. Belltown Media, 2014.

[52] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. In *CLOSER (1)*, pages 137–146. 2016.

[53] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. volume 12, pages 2825–2830. 2011.

[54] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. volume 17, pages 412–421. ACM, 1974.

[55] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. volume 10, pages 76–82. IEEE, 1990.

[56] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. volume 38, pages 39–47. IEEE, 2005.

[57] J. Rufino, M. Alam, J. Ferreira, A. Rehman, and K. F. Tsang. Orchestration of containerized microservices for iiot using docker. In *2017 IEEE International Conference on Industrial Technology (ICIT)*, pages 1532–1536. March 2017.

[58] S. Singh and N. Singh. Containers & docker: Emerging roles & future of cloud technology. In *2016 IEEE International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pages 804–807. 2016.

[59] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. 2007.

[60] François Steinmetz, Pierre-Yves Deschamps, and Didier Ramon. Atmospheric correction in presence of sun glint: application to meris. volume 19, pages 9783–9800. Optical Society of America, 2011.

[61] S. Tilkov and S. Vinoski. Node. js: Using javascript to build high-performance network programs. volume 14, pages 80–83. IEEE, 2010.

[62] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. Self-managing cloud-native applications: Design, implementation, and experience. volume 72, pages 165–179. Elsevier, 2017.

[63] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. 2015.

[64] Tianming Wei, Madhav Malhotra, Bing Gao, Tomas Bednar, Derek Jacoby, and Yvonne Coady. No such thing as a free launch? systematic benchmarking of containers. In *Communications, Computers and Signal Processing (PACRIM), 2017 IEEE Pacific Rim Conference on*, pages 1–6. 2017.