A CONTAINER-BASED PLATFORM FOR MULTI-CLOUD APPLICATION ORCHESTRATION

A.M.A.S. Adikari

(179301K)

Degree of MSc in Computer Science specialising in Cloud Computing

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

May 2020

A CONTAINER-BASED PLATFORM FOR MULTI-CLOUD APPLICATION ORCHESTRATION

Adikari Mudiyanselage Akila Srinath Adikari

(179301K)

Thesis submitted in partial fulfillment of the requirements for the
degree Master of Science in Computer Science specialising in Cloud Computing

Department of Computer Science and Engineering

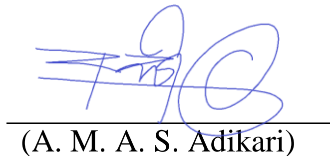
University of Moratuwa
Sri Lanka

May 2020

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgment any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where the acknowledgment is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature:  Date: *21/04/2020*

(A. M. A. S. Adikari)

The above candidate has carried out research for the Master's dissertation under my supervision.

Name of the supervisor: Dr. H. M. N. Dilum Bandara

Signature of the Supervisor: Date: *17/03/2020*

ABSTRACT

Multi-cloud applications are becoming popular, as they can run across multiple public and private cloud platforms while overcoming vendor lock-in, reducing cost, and enhancing flexibility and reliability. Applications hosted on multiple cloud platforms use either libraries or service-based abstraction layers. Application orchestration platforms further simplify the deployment and management of multi-cloud applications by providing auto-scaling, service metering, health monitoring, and a rich set of operational tools. Containerization is particularly useful in multi-cloud applications, as it provides a consistent environment for an application regardless of where it is deployed. However, container orchestration platforms such as Docker Swarm lack support and operational tools to enable seamless application orchestration across multi-cloud resources.

In this research, we developed a container-based platform for application orchestration in a multi-cloud setup as a set of microservices and required operational tools addressing the above limitations. Docker was chosen to demonstrate the proof of concept solution, as it already provides features to orchestrate microservices. Containerized multi-cloud applications can use the proposed application orchestration platform to achieve resource elasticity across multiple cloud platforms. To trigger scale in and out decisions, we used a rule-based approach where we compared the container runtime metrics provided by Docker with preconfigured threshold values. We evaluated the utility of the proposed platform using three web applications that were compute-intensive, memory-intensive, and utilized a RESTful application programming interface integrated with an external cloud service. The proposed container-based application orchestration platform improved the throughput of the three web applications by 180%, 73%, and 46%, respectively, compared to the same web applications deployed in a private cloud. Whereas the response time was reduced by 36%,-232%, and 7%, respectively. Even for cases where latency is increased error rate was reduced.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my research supervisor Dr. Dilum Bandara for providing supervision and resources to enhance my research idea. His expertise in the related field was an immense support for me to initiate this research work in this passion and identify possible technologies to complete this research.

I would also like to thank Dr. Malaka Walpola, Dr. Indika Perera and Dr. Charith Chtraranjan for helping and encouraging us to initiate and continuing this research until the end. Further, I would like to thank all my colleagues for joining together to share knowledge, research material, technology guidance, and experience to make this research success. Specially appreciate their encouragement till the end.

This would not have been a reality without the support and love from my parents throughout my life. I am deeply grateful for them for being ever strong support in every step of my life and heartful of blessings. Special thank goes to my beloved wife, including my family for being with me and sharing time to encourage to complete this research.

I would like to greatly thank Dr. Sankalpa Gamwarige, General Manager, Zone24x7 supporting me by providing necessary guidance to select this degree program and all my colleagues at Zone24x7 for their continuous support to manage my work and MSc research work.

TABLE OF CONTENTS

DECLARATION i
ABSTRACT ii
ACKNOWLEDGMENTS iii
TABLE OF CONTENTS iv
LIST OF FIGURES vi
LIST OF TABLES viii
LIST OF ABBREVIATIONS ix
1 INTRODUCTION 1
1.1 Background 1
1.2 Motivation 2
1.3 Problem Statement 3
1.4 Objectives 4
1.5 Outline 4
2 LITERATURE REVIEW 6
2.1 Cloud Computing 6
2.1.1 Cloud Deployment Models 6
2.1.2 Virtualization Based on VMs 7
2.1.3 Container-based Virtualization 9
2.2 Multi-Cloud 13
2.2.1 Multi-Cloud Software Solutions 14
2.2.2 Multi-Cloud Microservice Architecture 15
2.2.3 PaaS Solutions for Private Clouds 15
3 METHODOLOGY 19
3.1 Solution Approach 19
3.2 High-Level Architecture 21
3.3 Detailed Design 26
3.3.1 AppDock Cluster Admin 27
3.3.2 Command Line Interface 31
3.3.3 AppDock Scaling Service 33
3.3.4 AppDock Monitoring Agent 36

3.3.5 AppDock LogDB 38
3.3.6 AppDock HTTP Proxy Interface 41
3.3.7 Docker HTTP Proxy Interface 41
3.4 Cluster Deployment 41
3.5 Summary 42
4 PERFORMANCE EVALUATION 44
4.1 Workload 44
4.2 Experimental Setup 47
4.3 Performance Evaluation of CPU Intensive Workload 49
4.3.1 Throughput Analysis 49
4.3.2 Response Time Analysis 50
4.3.3 Resource Utilization Analysis 51
4.4 Performance Evaluation of Memory Intensive Workload 52
4.4.1 Throughput Analysis 53
4.4.2 Response Time Analysis 54
4.4.3 Resource Utilization Analysis 54
4.5 Performance Evaluation of RESTful API Workload 56
4.5.1 Throughput Analysis 57
4.5.2 Response Time Analysis 58
4.5.3 Resource Utilization Comparison 58
4.6 Summary 60
5 CONCLUSIONS 61
5.1 Summary 61
5.2 Research Limitations 63
5.3 Future Work 65
References 67
APPENDIX A – Available Methods in Proxy Interfaces 71
APPENDIX B – Commands in AppDock CLI 73

LIST OF FIGURES

Figure 2-1 Virtualization via containers and VMs. 9
Figure 2-2 Docker architecture. Source:. 10
Figure 3-1 Conceptual view of a service that integrates containers across multiple cloud providers. 20
Figure 3-2 High-level deployment diagram of the platform deployed in private-public IaaS infrastructure. 22
Figure 3-3 Detailed view of the orchestration layer 23
Figure 3-4 Service view of the AppDock platform. 25
Figure 3-5 Component view of the application 27
Figure 3-6 Configurations required by cloud providers. 29
Figure 3-7 File storage built using Docker volumes and NFS. 30
Figure 3-8 Properties maintained by local storage. 33
Figure 3-9 Properties of INodeStatAnalysisStatus object 34
Figure 3-10 Properties of IScalingServiceConfig object 34
Figure 3-11 Algorithm for adding and removing nodes. 35
Figure 3-12 Properties of IRuntimeStat object 37
Figure 3-13 Calculation for CPU utilization. 38
Figure 3-14 Calculation for memory utilization. 38
Figure 3-15 Class diagram for the repository. 39
Figure 3-16 Configurations required when deploying an AppDock cluster 42
Figure 4-1 Deployment diagram for the experimental setup. 48
Figure 4-2 Throughput comparison – CPU intensive application. 50
Figure 4-3 Response time comparison - CPU intensive application. 50
Figure 4-4 CPU utilization under CPU intensive workload – private cloud mode. 51
Figure 4-5 CPU utilization under CPU intensive workload – multi-cloud mode. 51
Figure 4-6 Memory utilization under CPU intensive workload – private cloud mode. 52
Figure 4-7 Memory utilization under CPU intensive workload – multi-cloud mode. 52
Figure 4-8 Throughput comparison – memory-intensive workload. 53
Figure 4-9 Response time comparison - memory intensive workload. 54
Figure 4-10 Memory utilization under memory-intensive workload – private cloud mode. 55
Figure 4-11 Memory utilization under memory-intensive workload – multi-cloud mode. 55
Figure 4-12 CPU utilization under memory-intensive workload – private cloud mode. 56
Figure 4-13 CPU utilization under memory-intensive workload – multi-cloud mode. 56
Figure 4-14 Overall throughput comparison – REST API workload. 57

Figure 4-15 Overall response time comparison - REST API workload................... 58
Figure 4-16 CPU utilization under REST API workload - private cloud mode 58
Figure 4-17 CPU utilization under REST API workload- multi-cloud mode.......... 59
Figure 4-18 Memory utilization under REST API workload – private cloud mode.
.... 59
Figure 4-19 Memory utilization under REST API workload – multi-cloud mode.. 60

LIST OF TABLES

Table 2-1 Container technologies used by different PaaS vendors.......................... 13
Table 4-1 Testing parameters for the CPU-intensive workload............................... 45
Table 4-2 Testing parameters for the memory-intensive workload......................... 45
Table 4-3 Testing parameters for the RESTful API application.............................. 46
Table A-1 Available methods in the AppDock HTTP proxy interface 71
Table A-2 Available methods in the Docker HTTP proxy interface 72

LIST OF ABBREVIATIONS

API	Application Programming Interface
ARaaS	Application Runtime as a Service
AWS	Amazon Web Services
CD	Continuous Deployment
CLI	Command Line Interface
CN	Container
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
DB	Database
DoS	Denial of Service
EC2	Elastic Compute Cloud
HW	Hardware
IaaS	Infrastructure as a Service
IT	Information Technology
LXC	Linux Containers
NFS	Network File System
ODM	Object Document Mapper
OS	Operating System
PaaS	Platform as a Service
QoS	Quality of Service
REST	Representational State Transfer
SaaS	Software as a Service
SDK	Software Development Kit
SLA	Service Level Agreements
SOA	Service-Oriented Architecture
SW	Software
vCPU	Virtual Central Processing Unit
VM	Virtual Machine

1 INTRODUCTION

1.1 Background

Cloud computing has emerged as the preferred paradigm for computing where a public or private pool of computing resources are connected to deliver a dynamically scalable infrastructure for applications. Cloud computing provides ubiquitous, convenient, and on-demand network access to a shared pool of compute, network, and storage resources with minimum management or service provider interaction [1]. This significantly reduces the cost of application hosting, computing, storage, and delivery. While organizations can utilize Infrastructure as a Service (IaaS) cloud services for traditional application deployment, the cloud also provides Platform as a Service (PaaS) model for Internet-based applications. PaaS is a layer on top of the existing IaaS where the cloud vendor also manages underlying platforms. This offloads maintenance and upgrades from the developers while minimizing the downtime [2], [3].

Today, PaaS is moving towards containerization and interoperability. Containerization provides interoperable, as well as lightweight and virtualized packaging [4]. It provides a virtualized lightweight process environment as close as possible to a standard Linux distribution. Because containers are more lightweight than Virtual Machines (VMs), the same host can accommodate more containers than VMs. Also, it reduces the start-up time of instances, as well as processing and storage overhead [5].

Orchestration solutions have become essential in effectively managing cloud resources in response to rapidly changing business needs, cost, and service-level agreements. It is a continuous process driven by monitored metrics and the specified Service Level Agreements (SLA) [5]. Solutions such as Open-shift [6] and Cloud Foundry [7] are based on container technologies such as Warden and Docker, which allow organizations to run their own PaaS on-premise datacenters.

Among the contemporary cloud deployment strategies, the multi-cloud strategy has become popular, as it enables the freedom to run applications on any public or private cloud. The multi-cloud market is expected to grow by 30% of the compound annual

growth rate from 2017 to 2022 [8]. Cost-saving, capitalizing on-premise infrastructure Information Technology (IT) resources, flexibility, avoiding vendor lock-in, and increased reliability via robustness towards Denial of Service (DoS) attacks can be identified as significant benefits for cloud consumers who are committing to use multi-cloud architectures [9].

1.2 Motivation

Organizations follow a multi-cloud strategy to better utilize their infrastructures due to optimized cost, as well as technological and regulatory reasons. Organizations commit to multi-cloud architectures to provision infrastructure resources dynamically from multiple public IaaS vendors to cater to time-variant demand for resources. This can significantly reduce the cost incurred for public cloud infrastructure resources because they are released automatically in the absence of demand.

Different public cloud providers showcase a wide variety of cloud offerings, and their product lineup is somewhat differentiated from each other to gain a competitive edge. However, this leads to application designs and implementations that are tightly coupled to a particular cloud provider. Such lack of interoperability leads to vendor lock-in, which is a significant concern for public cloud consumers. For example, switching costs to a new cloud environment is relatively high and includes the system re-design, redeployment, and data migration costs. To overcome interoperability issues with multi-cloud abstraction layers are used to hide the differences between cloud providers [10].

Orchestration enables intelligent cloud resource allocation through dynamic provisioning, coordination, and management of services. This greatly reduces human intervention and cost [11]. These benefits are intensified for applications that are deployed in multi-cloud infrastructure due to increased flexibility of infrastructure resource provisioning from multiple IaaS vendors. We can harness those benefits by developing a multi-cloud application orchestration platform that capitalizes better resource utilization and scalability inherently provided by the containers. Existing

applications running on containers could be easily moved to a multi-cloud application orchestration platform.

OpenShift Origin [6] is a supporting platform providing both developer and operation centric tools to develop multi-cloud applications. It extends Google's Kubernetes platform to leverage container orchestration. Tsuru [12] is another similar multi-cloud PaaS using Docker containers. Neither of these platforms provides automated infrastructure resource expansion for time-varying workloads and simplicity in operational aspects of the platform.

1.3 Problem Statement

Contemporary multi-cloud solutions such as multi-cloud microservices architectures [29] and private PaaS solutions [6], [7] are used either by compromising the multi-cloud nature of the application or orchestration features such as dynamic resource expansion. Moreover, management and deployment overhead of microservices and private PaaSes in a multi-cloud setup are considerably high. Library or service-based multi-cloud software solutions [24] have attempted to solve this problem using an abstraction approach where cloud services from different vendors are aggregated and exposed via a standard interface. However, the application becomes tightly coupled to specific libraries provided by those solutions.

We believe that a container-based approach is more desirable to address the limitations mentioned above in contemporary multi-cloud solutions. This is due to the benefits such as easily movable, lightweight container images across multi-cloud environments and applications that are not required to use additional software libraries just to support multi-cloud (because containers provide the execution context for standard technologies). In such a setup, Service Oriented Architecture (SOA) can guarantee interoperability of application components developed using different standard technologies avoiding platform-specific implementations. APIs provided by different IaaS cloud offerings and the lightweight nature of container images make it possible to allocate and deallocation resources dynamically. In this context, the problem to be addressed by this research can be stated as follows:

How to develop a container-based, dynamic, and readily deployable multi-cloud application orchestration platform that minimizes the coupling between applications and the platform?

The proposed solution should be scalable and supports a combination of private and public cloud IaaS offerings. Because the platform directly coordinates with the IaaS layers of cloud providers, it should greatly minimize the dependency on various cloud offerings of cloud vendors, as well as the applications hosted in this platform should be platform-independent.

1.4 Objectives

Following set of objectives is to be achieved to address the above research problem:

- To conduct a comprehensive literature review on existing virtualization technologies, including containerization and various multi-cloud solutions.
- Design a novel platform that could integrate multiple cloud infrastructures to facilitate resource elasticity and seamless application orchestration by providing multi-cloud integration.
- Integrate web and Command Line Interface (CLI) based operational tools to support continuous delivery.
- To conduct a comprehensive performance analysis of the proposed multi-cloud application orchestration platform.

1.5 Outline

The rest of the dissertation is organized as follows. Chapter 2 presents a literature review on the theoretical aspects of cloud computing and cloud technologies such as virtualization based on VMs and containers. It also covers container orchestration and related technologies. Contemporary multi-cloud solutions and architectures are also covered. Chapter 3 contains the solution approach, high-level architecture of the proposed container-based application platform and details on how the multi-cloud application orchestration platform can be deployed on an infrastructure that is a

combination of a private and two public clouds. Also, it has provided a detailed description of the platform we developed along with all its subcomponents. Performance evaluation for the proposed platform is presented in Chapter 4. It contains workloads along with the test scenarios, experiment setup deployment, and the detailed performance evaluation. Chapter 5 contains conclusion remarks, research limitations, and future work.

2 LITERATURE REVIEW

Section 2.1 presents key concepts associated with cloud computing, such as cloud service models and cloud deployment models. An overall insight into containerization using Docker and container orchestration is presented in Section 2.1.3. Introduction to Multi-cloud and different types of Multi-Cloud solutions, including software solutions, multi-cloud microservice architecture, and deployable PaaS solutions, are presented in Section 2.2.

2.1 Cloud Computing

Cloud computing referrers to the model of enabling ubiquitous, on-demand, and convenient access to a shared pool of computing resources that can be easily provisioned and release with minimum or zero-intervention of the cloud service provider. According to [1], there are five essential characteristics of cloud computing, namely on-demand self-provisioning, location independent ubiquitous access, resource pool with a higher level of abstraction, rapid elasticity, and metered service. Organizations can make use of a cloud computing model under three service models, which are IaaS, PaaS, and Software as a Service (SaaS). The cloud deployment model also an essential factor when deciding the organization’s cloud strategy. Depending on the type of applications and their scalability and availability requirements, they can choose among the public, community, private, or hybrid cloud deployment model which suits them best.

2.1.1 Cloud Deployment Models

Cloud deployment models take four forms as follows:

- *Private Cloud* – The computing environment is operated exclusively for a single organization. Cloud services are not accessible to external consumers. Organizations that have excessive privacy and security concerns over their data follow this deployment model. The datacenter may be inside the

organization’s premises or outside. It may also operate by an external party that has required expertise in operating datacenters. This model provides organizations with greater control over the cloud infrastructure than other models.

- *Community Cloud* – Organizations that have common privacy mostly adopt a community cloud deployment model. Two or more organizations with larger computing resource requirements and common privacy, security, and regulatory considerations (e.g., banks) can jointly operate the shared datacenter infrastructure or some third party. Datacenters may exist on or off the premises.
- *Public Cloud* – The cloud infrastructure is provisioned for the use of the general public over the internet. It may be managed by a business organization or any combination of privately owned or government organizations. By definition, the cloud environment is always external to the cloud consumers [13].
- *Hybrid Cloud* – A combination of the above cloud deployment model (Private, Community, or Public) employed to bound two or more cloud environments together by proprietary or standard technologies to serve cloud consumer’s computational needs. Data portability is an important aspect of a hybrid cloud deployment model among the linked cloud environments.

Deployment models are further categorized based on the management and distribution of computational resources to deliver services to cloud consumers [13].

2.1.2 Virtualization Based on VMs

Virtualization refers to the software-based representation of physical computational resources such as applications, servers, storage, and networks to be used to effectively reduce the IT-related cost and bring agility to business while increasing the efficiency [14]. Among the many benefits discussed under virtualization technology reducing capital expenditure, minimizing or eliminating downtime, simplified datacenter management, quick provisioning, enable business continuity, and disaster recovery can be highlighted.

Cloud computing is mainly benefited by virtualization technology that saves the hardware implementation cost for different operating systems. Virtualization can take multiple forms, such as the following:

- *Full Virtualization* – The hypervisor provides all the services of a physical system so that guest operating systems completely disengaged from the underlying hardware. They are unmodified because Hypervisor completely emulates the devices in the physical server. Full virtualization is categorized into either hardware-assisted or software-assisted. As for drawbacks, full virtualization requires a specific type of hardware to support this or performance loss due to the binary translation.
- *Para Virtualization* – Guest operating systems are modified and aware of them being virtualized, and they share resources with other VMs via API calls to the hypervisor. Instead of making direct hardware calls, they issue the command to the hypervisor from their customized device drivers.
- *OS Level Virtualization* – Also known as “containerization”. This technique is capable of providing virtualized context for guest operating systems without Hypervisors. The shared kernel of the host operating system among the guest OSs is capable of isolating the execution context of each VM.
- *Hardware-Assisted Virtualization* – It depends on explicit support in the host CPU, which is not available in all CPUs. By using the hardware capabilities, this technique achieves accelerated virtualization. This technique was added to x86 CPUs in 2006 [15]. This makes the full virtualization more efficient from hardware capabilities. Nevertheless, this approach involves many traps to CPU. This is mitigated by Para virtualized drivers. This combination is called hybrid virtualization.

Hypervisors are low-level programs that allow single hardware to be shared by multiple guest VMs, providing shared system resource access. Type 1 Hypervisors and Type 2 Hypervisors are the main distinguishers for the way they are operating on a host computer. Type 1 Hypervisor runs directly on the host hardware to manage guest operating systems. Type 2 Hypervisor runs only on the operating system, which means it cannot run until the operating system is running [15].

2.1.3 Container-based Virtualization

A single operating system allowing multiple user-space instances running on its kernel refers to the containerization or operating-system-level virtualization. This is a virtualization technique that is based on user-space instances, which are called containers. Unlike other virtualization techniques such as full virtualization or paravirtualization, which use hypervisors to emulate hardware, container-based virtualization uses container engines to provide a managed environment for deploying containerized applications [16]. The primary duties carried out by the container engine are allocating CPU, memory, space isolation, and providing security. It also provides scalability to the addition of new containers. Figure 2-1 shows how the virtualization is achieved using containers compared to a VM that dedicatedly needs a guest operating system to run the software as opposed to container-based virtualization where multiple containers can be run in the same operating system. The container engine runs natively on Linux and shares the kernel among containers. By contrast, VMs are full of guest OSs with virtual access to host resources.

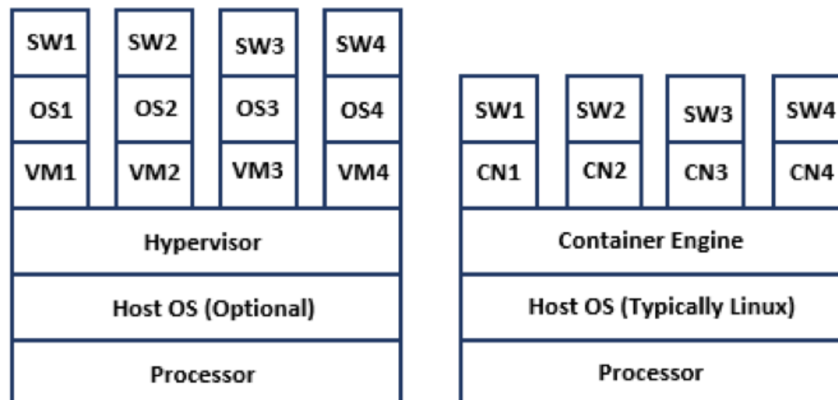


Figure 2-1 Virtualization via containers and VMs.

Containers are becoming popular over the other virtualization technologies because of the isolation benefits it provides without much overhead of space and time. While containers are typically hosted in Linux, Windows and Solaris containers are also available.

Virtualization via containers provides many benefits such as lower hardware cost, improved reliability and robustness, high scalability, efficient storage, and spatial

isolation, high throughput, portability, continuous integration and safety and security. However, container-based virtualization is associated with cons such as increased cost of container management and safety and security.

2.1.3.1 Docker

Docker is a container platform based on Windows and Linux kernels. Even though Docker has gained much popularity among the community, there are other container platforms such as LXD and OpenVZ [17]. Windows were only capable of hosting the Docker engine and carry on Linux workloads initially. Today, it is possible to run Windows containers on the Docker engine. However, container images can only be based on Windows Server Core, Nano server [18].

Docker engine is composed of three main components, Docker daemon, REST API (Application Programming Interface), and CLI (Command Line Interface) for the "docker" command organized in a client-server architecture. Docker daemon directly receives the command from the REST API to control or interact with them through scripting or direct CLI commands. Docker daemon is responsible for creating and managing images, containers, network, and volumes [19].

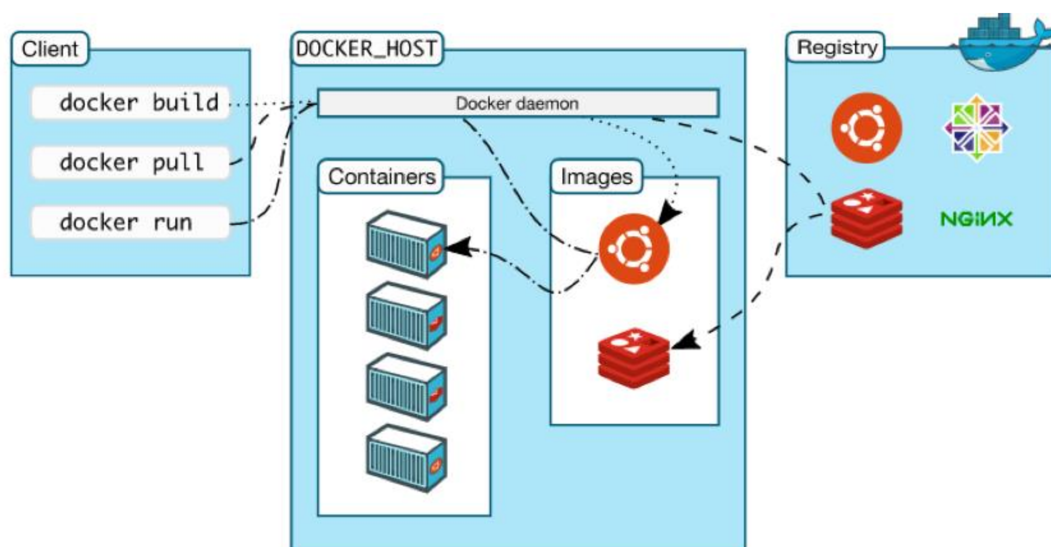


Figure 2-2 Docker architecture. Source: [19].

Docker architecture is presented in Figure 2-2. Docker images are based on other images already containing a supported operating system. We might create our images

and push them to the Docker Hub (i.e., a container repository). The container is a runnable instance of an image. REST API or Docker Client can be used to interact with a container. Any changes maintained in the container state does not preserve if those are not written to an external persisted storage. Docker Service is a collection of Docker daemons running on independent hosts. Services are used to scale applications across multiple hosts by scheduling multiple containers in multiple daemons. Services are load-balanced so that users see the service as a single application.

The underlying technology of Docker is using *namespaces* to provide isolated workspace for containers. Docker engine uses various namespaces to achieve the above purpose. Control groups (cgroups) is another technology used by Docker to limit and control the usage of hardware resources of the host among multiple containers. The Union file system is file systems that operate based on creating layers on top of each other. Container format wraps namespaces, cgroups, UnionFS together to form a container. The default container format is “libcontainer” [19].

2.1.3.2 Linux containers

Linux Containers (LXC) is an operating-system-level virtualization technique for running multiple Linux containers on a control host using a single Linux kernel [18]. It consists set of tools and components that are running on the user space of the operating system for controlling the Linux Kernel containment features. It uses kernel features such as Kernel namespaces (ipc, uts, mount, pid, network, and user), Apparmor and SELinux profiles, Seccomp policies, Chroots (using pivot_root), Kernel capabilities and CGroups (control groups) to contain processes. LXC tries to create an almost close environment similar to the Linux kernel for its containers. Components of LXC can be identified as the liblxc library, language bindings in Python3, Lua, Go, Ruby, Python2, Haskell for the API, standard container controlling tools and distribution container templates [20]. LXC Hiroku provides a containerized way to run technology stacks such as Ruby, Python, NodeJS, Go, and Java using LXC containerization technologies [18].

2.1.3.3 Container Orchestration

Automating the deployment, management, and monitoring of a large-scale container cluster is referred to as *container orchestration*. Container orchestration essential for managing applications that use a large set of containers. Orchestration may include host provisioning, container instantiating, rescheduling in a failure situation, linking containers through interfaces and scaling. Container orchestration provides a single point of access for managing container cluster and monitoring health. It also gives developers and operational teams a holistic view of the container cluster. Container orchestration tools are essential to managing large-scale container clusters effectively. Container orchestration helps to overcome challenges like service discovery, load balancing, auto-scaling, zero-downtime deployments, and configuration management in microservices applications that are spanning in multiple servers. Microservices can get the most benefit from container orchestration. Docker Swarm and Kubernetes are valuable orchestration tools available for Docker containers.

- *Kubernetes* – Is a widely used orchestration tool that is considered to be feature-rich. Kubernetes is designed to operate in a wide range of modes, including bare metal, on-premise VMs, and public cloud environments. Google container engine provides the Docker engine and Kubernetes orchestration tool tightly integrated. Being an open-source platform, it has built up a large community around Kubernetes. Google cloud platform provides the cheapest way to run Kubernetes with free master nodes. This has attracted more Docker customers towards their cloud platform.
- *Docker Swarm* – Docker includes container cluster management and orchestration tool called “Docker Swarm”. Docker CLI is used to interact with Swarm to manage a standalone or distributed Docker cluster. Swarm manages services that can be identified as application components running in a replicated fashion across the cluster. Cluster is composed of nodes, instances of Docker engine in Swarm mode participating the Swarm [21].

With the rapid adoption of containerized applications and Docker becoming an industrial standard for containerization, the need for container orchestration has

become apparent. Table 2-1 lists container technologies used by different PaaS vendors.

Table 2-1 Container technologies used by different PaaS vendors.

PaaS Provider	Container Technology
OpenShift	Docker with LXC
Heroku	LXC
CloudFoundry	Warden Container
Stackato	Docker with Warden Container
AppFog	Warden Container
Virtuozzo	based on OpenVZ
dotCLoud	Docker with LXC

2.1.3.4 Container Registry

Container registry is a server-side application that stores and distributes container images that is part of the container repository, which constitutes different tags and versions of the same container image. Container registry governs container image creation, storing, and accessing process. Container registries are available as hosted services, as well as deployable solutions. For example, Docker Hub, Quay, and major IaaS providers offer their own registries such as Amazon ECR, Azure ECR and Google Cloud's GCR as hosted services and Docker Distribution tool, GitLab container registry can be named as deployable container registries [22].

2.2 Multi-Cloud

The multi-cloud concept can be characterized by the usage of cloud services from diverse cloud providers to run applications. Simultaneous usage of services of multiple clouds that constitutes private and public cloud covers cloud bursting and federated cloud scenarios. Cloud bursting discusses bursting the resource capacity by using the public cloud while application most of the time running in the private cloud. Federated cloud scenario discusses the federation of resources equally among cloud providers, as well as with the private cloud.

The above-discussed scenarios mostly work on the IaaS layer. Interoperability issues have been avoided by using homogeneous environments across cloud providers. Additionally, work-related to interoperability between PaaS services also done, such as semantically interconnecting heterogeneous PaaS offerings across different cloud providers sharing the same technology and service-oriented component-based PaaSes [23].

2.2.1 Multi-Cloud Software Solutions

An important functionality of Multi-Cloud is the management of deployments across various clouds. According to [24], based on the approaches taken to implement multi-cloud solutions, they can be categorized as library-based or service-based solutions. The most known library-based and service-based solutions are identified as follows:

- *jcloud (Library-based)* – This is a Java library that enables the portability of Java applications which allows the unified access of resources in various cloud platforms such as AWS (Amazon Web Services), CloudSigma, Digital Ocean, ElasticHost, Go2Cloud, GoGrid and many other cloud providers [25]. jcloud provides abstractions for Blog storage, Compute services, and Load balancer services in general.
- *libcloud (Library-based)* – This is a Python library that abstracts compute, load balancer, object storage, container, backup, and DNS services from many cloud providers, including leading vendors such as Amazon AWS, Microsoft Azure and Google [26].
- δ-*cloud (Library-based)* – Is REST-based API written in Ruby to the abstract difference between clouds. Once the DeltaCloud server is setup, various clients can be used to communicate with the server directly via HTTP interface or C/C++, Ruby libraries. δ-cloud mainly abstracts the IaaS services from various cloud providers such as Amazone EC2, Eucaliptus, IBM SmartCloud, GoGrid, OpenNebula, Rackspace, Microsoft Azure, Amazone S3, Google Storage and many other cloud providers [27].

- *SimpleCloud (Library-based).* It is another Infrastructure abstraction library for Multi-Cloud that is written in PHP. It provides uniform interfaces for infrastructure services of AWS, Azure, RackSpace, and Nirvanix [24].
- *RightScale (Service-based).* It is a hosted service that provides self-service, cloud management, and cost optimization facilities to Multi-Cloud. RightScale supports IaaS services from Amazone AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Rackspace, Apache CloudStack, OpenStack, and VMware vSphere [28].
- *Kaavo.* It is a hosted service that supports workload and runtime management for the multi-cloud strategy. Its support is expanded across IaaS, PaaS, and SaaS layers [24].

2.2.2 Multi-Cloud Microservice Architecture

Microservices is a software architectural style where the application is composed of smaller processes independently running in a distributed fashion and communicating via language-agnostic APIs. Microservice architecture is mainly driven by micro Linux distributions that support containers, Containers, and Schedulers such as Swarm or Kubernetes [29].

Because microservice architecture is a technique variant of SOA architectural style, loosely coupled smaller services can be distributed among multiple cloud providers. To schedule containers in multiple cloud providers, multiple container schedulers need to run in each environment if nodes in each cloud provider are not in the same network. Otherwise, the single scheduler can be configured with a relatively high configuration effort to schedule containers in all nodes across multiple cloud providers.

2.2.3 PaaS Solutions for Private Clouds

Platform as a Service takes another form when it comes to large organizations. With the possibilities to run their private cloud infrastructure using technologies such as OpenStack, VMware Cloud Foundation, they tend to look for their own private PaaS environments. Applications developed internally are deployed in these private PaaS environments to be used across the organization. They also tend to integrate with

public cloud forming a hybrid cloud deployment model for externally facing high availability applications. Large organizations prefer an internal software portfolio to run behind on-premise firewalls, yet achieving the benefits of cloud computing. It is worth giving attention to private PaaS solutions with higher market share such as OpenShift Origin, Cloud Foundry, AppScale, Microsoft Azure Stack, and Apache Stratos.

OpenShift Origin is considered to be a Kubernetes distribution that is optimized for continuous application development and multi-tenant deployments. OpenShift is backed by RedHat. It mainly runs on a Docker container cluster managed by Kubernetes. Container nodes consist of a set of master nodes and a set of nodes to guarantee high availability by avoiding a single point of failure [6].

Cloud Foundry is a multi-cloud Platform as a Service that can be deployed in private, public or any combination of it. This is a large-scale PaaS software that can run on any IaaS provider. Cloud Foundry internally developed tool BOSH supports provisioning and managing VMs. Kubernetes is also integrated as the container orchestration tool to supports its container-based architecture to operate in multi-cloud infrastructure. Before integrating Kubernetes, the container management system, Diego was used for this purpose. Cloud Foundry supports Docker images also can be integrated with the Docker Registry. While platform VMs running the Cloud Foundry PaaS, platform host VMs are used to run applications deployed in CF. CF uses two types of deployment models. Build packs and Docker images. Source code can be pushed directly via the CF command line to the CF cloud controller. Then Diego will build the application and deploy the artifacts in the container cluster. Names of the Docker images in the registry can be directly pushed into the Cloud Controller via CLI to deploy them in CF [7].

AppScale is an open-source Google App Engine PaaS platform that can be run in any cloud. This does not operate in multi-cloud infrastructure. This has reached the market in China via deploying it in Alibaba cloud-enabling Google App engine apps to run in China [30].

Azure Stack is an extension of Microsoft Azure to operate Azure cloud services within the organization's datacenter integrated with the Microsoft Azure public cloud

allowing organizations to follow a hybrid cloud deployment model. Azure Stack delivers both IaaS and PaaS capabilities to an on-premise datacenter allowing portability of applications. It also allows serverless computing, distributed microservices architectures, and on-premise container management features. Azure Stack is offered as an integrated system of hardware and software known as the Azure Stack Integrated System. It ranges in size from four to 12 nodes [31].

Apache Stratos is a PaaS framework that can be deployed in any IaaS infrastructure that uses VMs and has compatibility with Apache Jclouds a cloud abstraction layer implemented using Java. Kubernetes also can be integrated as the IaaS layer to leverage the use of Docker containers [32]. In the Stratos terminology, a *cartridge* is referred to as a pluggable component from which service can be created in the Stratos PaaS. Cartridge instance can be either a VM or a Docker container that has software components to interact with Stratos PaaS to act as a service. VM cartridges provide OS-level isolation for cloud applications, whereas container cartridges provide software level isolation.

The architecture of Apache Stratos framework can be described as a component-based. A wide range of responsibilities is assigned to components such as Stratos Manager, Cartridge Agent, Artifacts Distribution Coordinator, Complex Event Processor, Cloud Controller, Message Broker, Load Balancer, Identity/Logging/Monitoring/Metering Services, Auto Scaler NS CLI/Web UI.

- Stratos Manager – Responsible for providing various interfaces for managing and interacting. It uses Stratos PaaS in the form of Web UI, REST, and CLI.
- Cartridge Agent – Handles the communication between cartridges and the Stratos framework. Cartridge Agent communicates with various components such as message broker, complex event processor, etc., while residing in the cartridge.
- Artifact Distribution Coordinator – Apache Stratos typically works with remote Git server where users maintain their code repositories. Deployment Synchronization happens when a user upstream their artifact to the Git repository to synchronize them with the relevant to that cartridge instance. This component handles automated artifact updates and deployment tasks.

- Complex Event Processor – CEP module is responsible for real-time monitoring based on the statistics published by Cartridge Agents and various services. Summarized information is sent to Auto Scaler to make the orchestration decisions [33].

3 METHODOLOGY

This chapter presents the high-level design of the proposed multi-cloud application orchestration platform. Section 3.1 presents the solution approach, deployment methods, and services to be offered. High-level architecture is presented in Section 3.2. Section 3.3 describes all the components of the proposed AppDock platform. The deployment process of the AppDock platform is explained in Section 3.4, while the summary is presented in Section 3.5.

3.1 Solution Approach

Dynamic infrastructure resource allocation can only be achieved on multiple IaaS cloud platforms using their public APIs. Containers are more desirable for dynamic resource allocation due to lightweight and easily movable in a multi-cloud setup and also do not impose any platform-specific dependencies on the applications. Therefore, we proposed a container-based application orchestration platform, namely AppDock which could operate on multi-cloud infrastructure. Platform consumers could choose any cloud infrastructure model such as private, hybrid, or multi-cloud to suit their business requirements. Different IaaS vendors can be integrated with the platform. AppDock contains a standard provider interface that can be extended to support multiple private and public IaaS vendors. Docker is chosen as the container technology due to the wide availability of container base images on standard technologies that can be extended as plug-ins. Container orchestration services offered by the cloud vendors cannot be used to develop the proposed platform, as those containers cannot be orchestrated by the proposed platform's base container orchestration system, which is Docker Swarm. AppDock provides tools required to manage and operate the application orchestration platform. They can be CLIs and web-based.

Despite vastly different platform services provided by the cloud providers, we focused only on providing an application runtime. Because moving the application's state across multiple nodes can introduce various complexities, in this work, we do not

support state-full technologies such as relational/No-SQL databases and in-memory data services.

A *service* is a logical grouping of containers running the same logical process. As shown in Figure 3-1 a service spans across multiple container engines (i.e., nodes) running on multiple cloud infrastructures. Any application deployed in the AppDock platform is considered as a service of which tasks are deployed in multiple nodes across multiple cloud infrastructures.

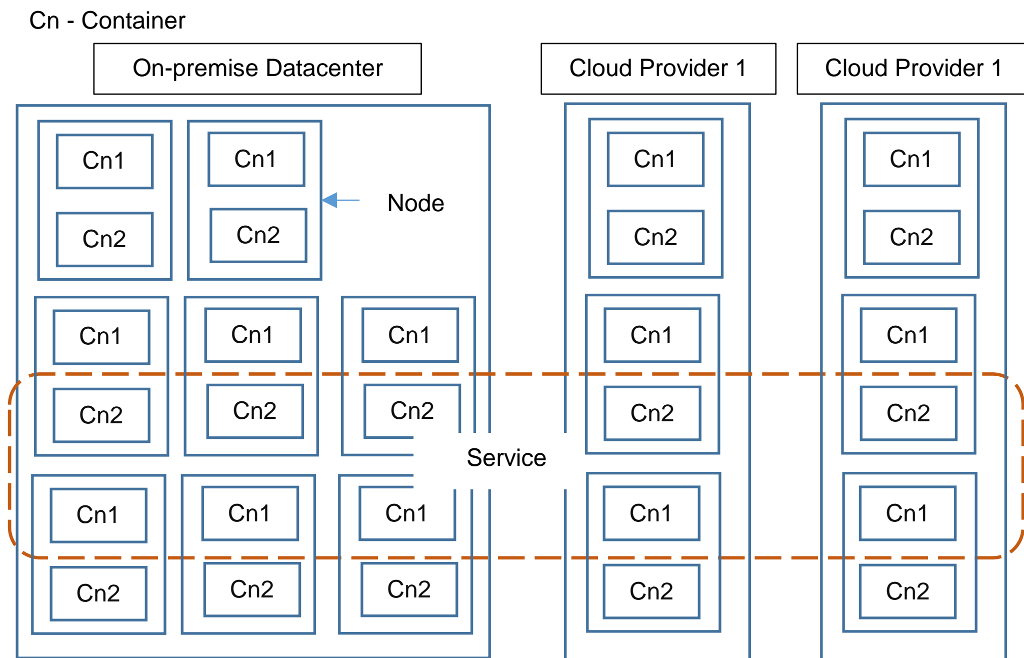


Figure 3-1 Conceptual view of a service that integrates containers across multiple cloud providers.

Multiple IaaS providers could be integrated via vendor-specific APIs with the standard provider interface in the platform. Cloud environments communicate via network layer connectivity allowing dynamic resource allocation. The cloud environment in which the orchestration platform is initially deployed is the primary cloud environment. Other cloud providers connected via the standard provider interface by providing required configurations only provide the additional resource capacity.

When automating dynamic infrastructure resource allocation, resource expansion and contraction decisions are made. To make such decisions, resource utilization metrics reported by the container orchestration platform are compared with preconfigured

resource utilization threshold values globally configured for the cluster. Following resource utilization thresholds are considered:

- *Minimum CPU utilization* – The lower bound of the CPU utilization, which is compared with the CPU utilization of each node within the cluster when node removing decisions are made.
- *Maximum CPU Utilization* – The upper bound of the CPU utilization, which is compared with the CPU utilization of each node within the cluster when deciding to spawn a new node.
- *Minimum memory utilization* – The lower bound of the memory utilization to be compared with the actual memory utilization of each node within the cluster when node removing decisions are made.
- *Maximum memory Utilization* – The upper bound of the memory utilization, which is compared with the actual memory utilization of each node within the cluster when deciding to spawn a new node.

These threshold values have to be derived empirically by simulating the expected loads after applications are deployed within the application orchestration platform. Currently, these values are platform-specific.

3.2 High-Level Architecture

As shown in Figure 3-2, the Docker container cluster contains two types of containers, namely containers that belong to the orchestration layer and containers delivering capacity. The orchestration layer hosts various services that are required for the continuous operation of the platform. Those services include a RESTful API that is used by the CLI and web portal to perform cluster management and operational activities by developers, services that host static content for the web portal in the AppDock platform, Scaling Service, and LogDB. Microservices belong to the orchestration layer primarily run inside the cloud infrastructure on which the platform is installed. For example, the primary cloud infrastructure in Figure 3-2 is the private cloud. The orchestration layer is integrated with two other IaaS providers to increase the resource capacity of the platform to cater to time-varying resource demands.

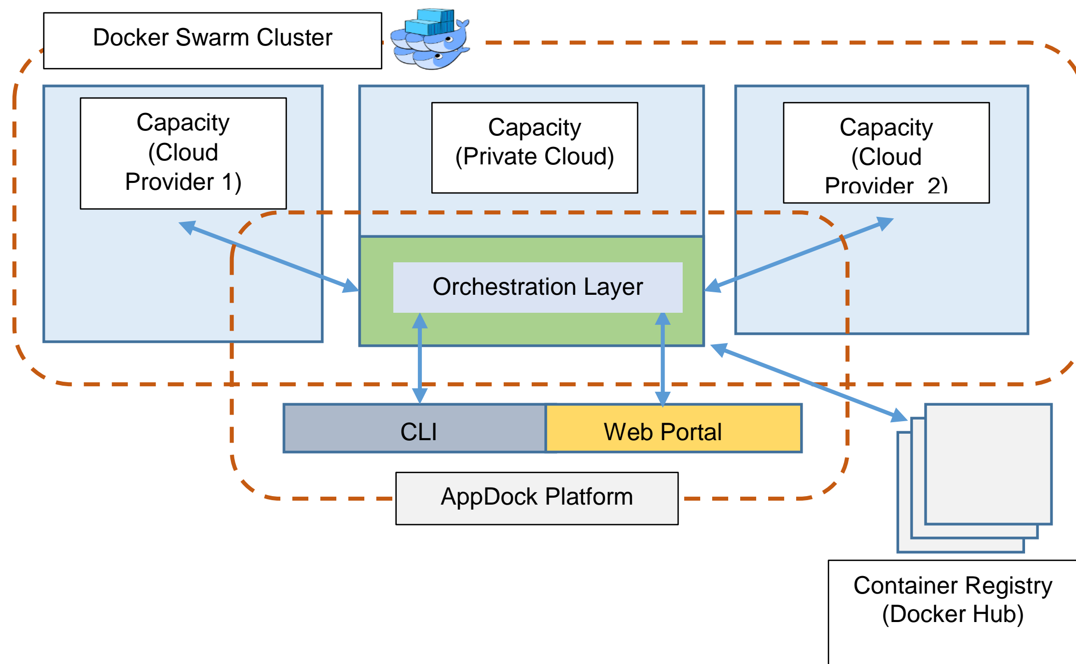


Figure 3-2 High-level deployment diagram of the platform deployed in private-public IaaS infrastructure.

Figure 3-3 shows a detailed view of the orchestration layer where the deployable core components of the platform, namely Cluster Admin, LogDB, and Scaling Service reside. The container cluster is virtually divided into two areas to denote the orchestration layer and the capacity. Containers that run orchestration layer related services are inside the primary infrastructure. Capacity containers are spanned across all IaaS providers, including primary cloud infrastructure. The AppDock platform is integrated with AWS and Microsoft Azure IaaS providers. They provide a rich set of APIs to manage infrastructure resources effectively. Further integrations with other cloud providers can be made by extending the standard provider interface.

We selected Docker Swarm as the baseline technology to implement the solution. Core components that form the solution have to be deployed in a distributed fashion as Docker Swarm is a collection of Docker nodes, and deployable components eventually become Docker Services. SOA lays the foundation to communicate across all the components.

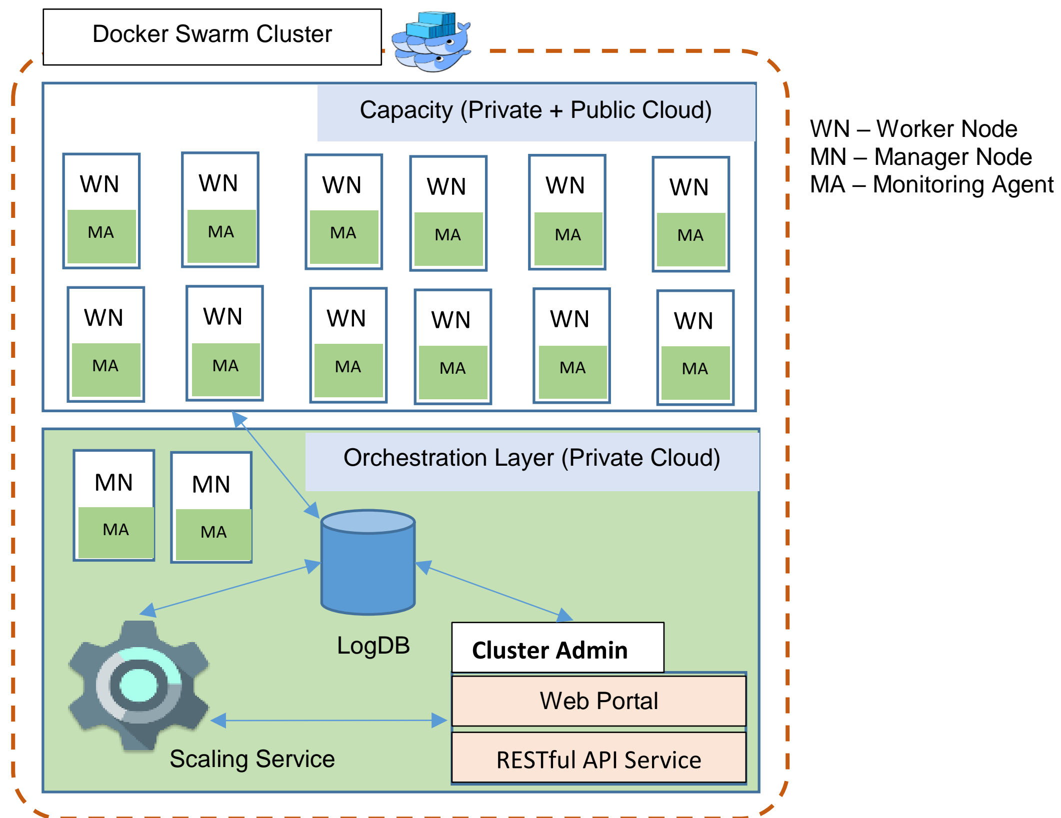


Figure 3-3 Detailed view of the orchestration layer.

The Cluster Admin is responsible for building container images from the application code and deploying it when the developer pushes new code to the platform via CLI. It also hosts the web portal for managing configurations for public cloud vendors and plug-in management services. It consists of a RESTful web API application and a web portal hosted in a lightweight web server. Cluster Admin is connected with the LogDB. Docker Swarm handles failover recovery.

LogDB acts as the data storage for the AppDock platform. This stores data required for the continuous operation of the platform, such as application runtime metrics, application scalability requirements, user information, and application metadata. The data file of this storage is kept in the Network File System (NFS) location, which can be safely accessed even in a failure situation of the database engine instance.

Scaling Service is a continuously running process that periodically analyzes runtime matrices of CPU and memory utilization data pushed to the database by monitoring agents residing in each worker node. New node deployment and removal decisions are

made based on the minimum and maximum thresholds set for CPU and memory utilization. Once the threshold is reached, the API in the Cluster Admin core component is used to expand or shrink the number of nodes and rescale each application deployed on the platform according to the current number of nodes in the cluster.

Monitoring Agent is responsible for reporting health and runtime matrices to LogDB. The monitoring agent is a continuously running process residing in each node.

Other components of the proposed platform are as follows:

- *Command Line Interface (CLI)* – is the only non-deployable core component used to communicate with the Cluster Admins' RESTful API to achieve the functionality requested by the users. CLI is the primary interaction point for the application orchestration platform. It provides commands to carry out tasks such as deploying applications, scaling, and generating boilerplate code and project templates.
- *RESTful API Service* – exposes a rich set of functionalities of the platform to be carried out by the users via CLI. This is running on a light-weight web server inside the Cluster Admin core component.
- *Container Cluster (Capacity)* – is a collection of worker nodes across the multi-cloud infrastructure that participate in delivering the required capacity. Deployed applications become Docker services in the underlying Docker Swarm cluster. The routing mechanism is built into Docker Swarm to route the traffic. Every node in the Swarm participates in an ingress routing mesh. Every node in the swarm is capable of accepting connections for any service via published ports [34].
- *Node managers* – can be configured to contribute to the capacity similar to worker nodes. Worker nodes are coordinated by the node manager and typically behave based on the service definition provided by the docker-compose.yml file [21]. This component already exists in Docker Swarm. This representation shows that core components are deployed along with other services in the Docker Swarm cluster.

Figure 3-4 presents the service view of the proposed platform in which core components prefixed with AppDock are also running along with other services in the Docker Swarm container cluster. This representation shows that core components are deployed along with other services in the Docker Swarm cluster.

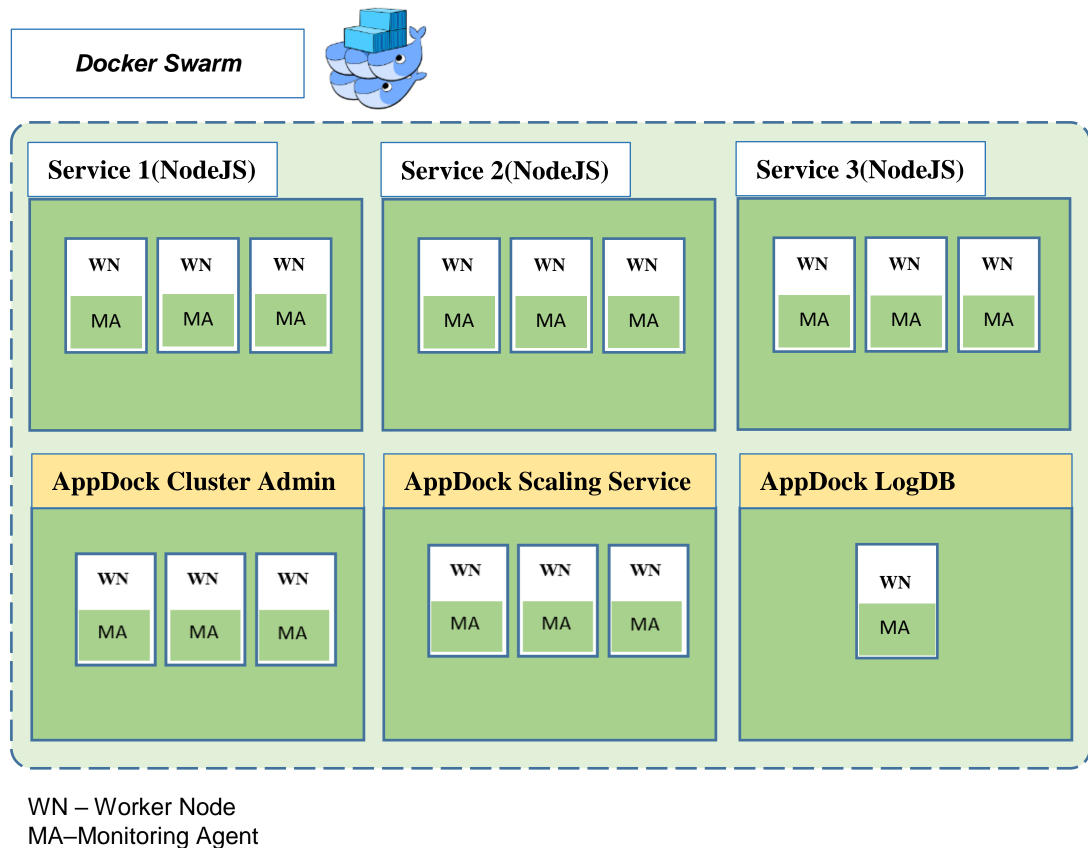


Figure 3-4 Service view of the AppDock platform.

An application deployed on the AppDock platform becomes a service, as it is deployed as a Docker Service. Applications are initially created using service plug-ins, as base Docker images from vendors are extended with required software components to be used as service plug-ins for the AppDock platform. It is possible to manually upload plug-in Docker images to the cluster administrator via the web portal. Applications are pushed directly to the platform via the CLI. The platform takes care of building the Docker images and deploying them in the container cluster.

3.3 Detailed Design

As shown in Figure 3-5, core components LogDB, Cluster Admin, CLI, Scaling Service, and Monitoring agent with libraries such as repository classes, AppDock HTTP interface, and Docker HTTP interface collectively form the AppDock platform. All the components were developed as part of this research. Components deployable in Docker Swarm cluster such as Cluster Admin, Scaling Service, and Monitoring Agent are deployed as Docker services during the initial creation of the AppDock cluster. The replica number of each task is four, two, and global (one replica in each node), respectively.

Every node, including static nodes at the primary cloud environment, participating in the Docker Swarm cluster exposes the Docker API. Cluster Admin, Scaling Service, and Monitoring Agent issue Docker commands via the Docker API to respective nodes to achieve various functionalities such as application orchestration and reading runtime stats of the nodes.

In addition to the deployable components and the non-deployable core component CLI, 2 HTTP proxy libraries and a data access library are also included in the design as they can be re-used across the solution. AppDock HTTP Proxy interface provides access to the RESTful API hosted within the Cluster Admin core component to other core components referenced it. Docker HTTP Proxy Interface provides access to the Docker APIs exposed in each node in the cluster, and it is referenced by all the core components. Data access library, known as "Repository Classes," is referenced by all the deployable core components to read/write data in the LogDB core component. All the core components are explained in detail below.

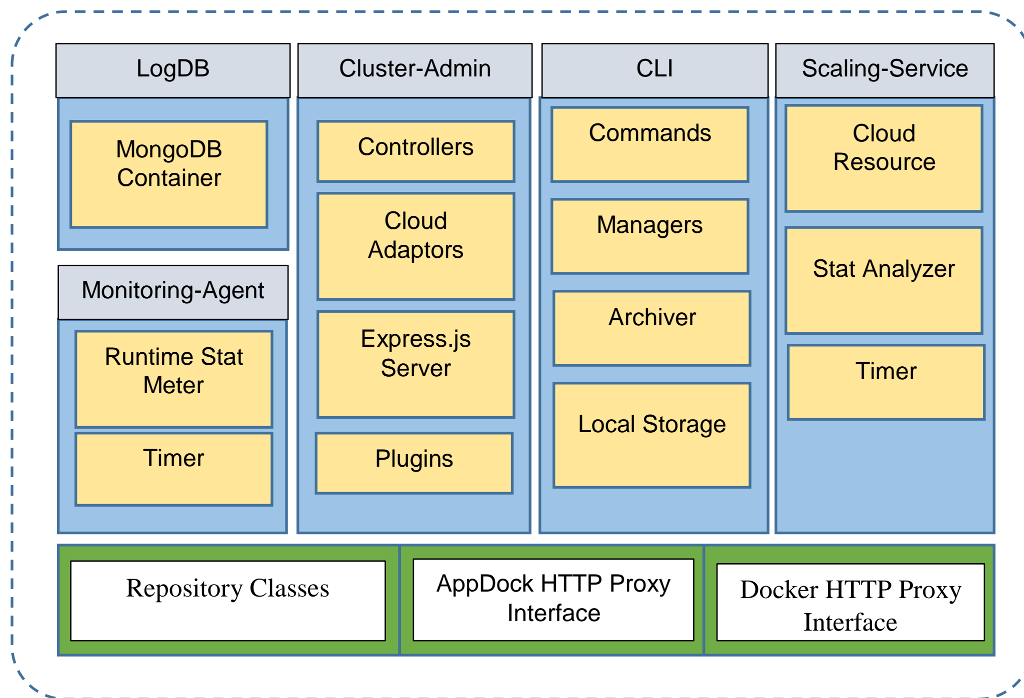


Figure 3-5 Component view of the application.

3.3.1 AppDock Cluster Admin

Cluster Admin component is a RESTful API application running as a Docker service within the Docker Swarm cluster. This component is configured to run multiple replicas to cater to concurrent platform management requests from other core components efficiently. Typically, the Docker services can be accessed via any node participating in the Docker Swarm cluster. Web interface provided with this can be used to configure public cloud providers (discussed in Section 3.3.1.2) and uploading service plugins. Upon deployment of the AppDock platform using the CLI, Cluster Admin service is initially created along with other core components within the cluster. Next, we explain the key sub-components of the Cluster Admin.

3.3.1.1 Controllers

Controllers contain the essential logic to be executed when requests are made to the Cluster Admin. Following are the roles of the controller in the AppDock platform:

- *CloudResourceController* – Creates and removes VMs in public cloud platforms. Based on the given cloud adaptor, new VMs are spawned on the respective public cloud infrastructure. Removal of VMs are requested by providing the node address to this controller, and the same cloud adaptor by which node was created is used

to remove the spawned nodes. Upon spawning of new nodes in the public cloud infrastructures, cloud resources associated with the new VM are recorded in the LogDB.

- *ConfigurationController* – Persistence and retrieval of configurations of public cloud providers are provided by this controller. Currently, AWS and Azure configurations required by respective adaptors are incorporated.
- *NodeController* – Performs operations related to managing nodes such as adding new nodes, removal of nodes from the Swarm, building service images of all services in a given node, and persisting node information in the LogDB. Adding a new node involves joining the node to the Docker Swarm, updating node labels to balance tasks across all the nodes, and building images of all the service and finally persisting the node in LogDB as an INode record. Removal of nodes involves leaving Docker Swarm, removing INode record, and INodeStatAnalysisStatus records from LogDB. *INodeStatAnalysisStatus* record contains the aggregated CPU and memory utilization of all the *IRuntimeStat* records logged by the Monitoring Agent of a particular node in each interval. This record is retained as long as the node remains within the cluster.
- *PluginController* – Create, Read, Update, and Delete (CRUD) operations for type *IPluginDefinition*, which contains plugin content, are carried out by this controller. Plugins are discussed in Section 3.3.1.4.
- *ScalingServiceController* – Retrieve and persist type *IScalingServieConfig* for storing minimum/maximum CPU and memory threshold values.
- *ServiceController* – Handles service-related operations such as creating new services based on plugins, service deployment, and updating service replicas when new VMs are added to the node cluster. Service creation involves downloading the selected plugin to the current working directory and saving it as an *IServiceDefinition* object in LogDB. The first deployment of service will create the service in Docker Swarm with the expected number of replicas (i.e., Replicas per Node × Number of Nodes). Before service creation or update, compressed application content is built as a Docker Image in every node. Then the *IServiceDefinition* record is updated with the latest modifications to the service

template done by the developer. When new nodes are added by automated resource expansion or CLI, the latest Docker images of all the services will be built in the new node.

Controller instances are instantiated injecting required repositories described in Section 3.3.5.1. Request router of the Cluster Admin component has used controllers in requests served by the Cluster Admin component. Each controller has access to the NFS path configured during the platform deployment phase. File artifacts created during management operations are stored in the NFS path accessible by all core components.

3.3.1.2 Cloud Adaptors

Cloud adaptors are used to connecting with public cloud infrastructures. Currently, adaptors for AWS and Microsoft Azure public cloud infrastructures are developed. These cloud adaptors implement interface *ICloudAdaptor,* which defines methods for creating and removing VMs. Cloud adaptors connect with their respective cloud environments with APIs provided by public cloud providers. Creating VMs in the public cloud may involve creating multiple other cloud resources such as storage, public IPs, and network interfaces. These resources, including the VM information, are persisted in the LogDB as an *ICloudResourceBatch* to refer back when excess VMs are removed from the AppDock platform. Each cloud adaptor requires a set of configurations to connect and spawn VMs in respective public cloud environments. Figure 3-6 shows configurations required by AWS and Azure public cloud environments.

```
AWSConfigurations = {
   VMImageReferenceID: string;
   Location:string;
   AccessKeyID: string;
   SecretAccessKey: string;
   SecurityGroupID:string;
   VPC:string;
   SubnetID:string;
   Active:boolean;
   AvailabilityZone:string;
   InstanceType:string;
}
```

```
AzureConfiguration = {
   Location?:String;
   TenantID?:String;
   ClientID?: String;
   ClientSecret?: String;
   SubscriptionID?: String;
   ResourceGroupName?: String;
   VMImageReferenceID?: String;
   PublicIPPrefix?:String;
   NetworkSecurityGroupID?:String;
   VNet?:String;
   SubnetID?:String;
   SSHKey?:String;
   Active:boolean;
}
```

Figure 3-6 Configurations required by cloud providers.

3.3.1.3 Server

The server is the execution environment for the Cluster Admin component. The web server hosts both the Cluster Admin web interface at the root resource location and API. As shown in Figure 3-7 each task of the Cluster Admin component is mounted with a Docker Volume mapped to /mount local path in each container as the file storage. Every file stored in /mount is synchronized with the NFS path configured at the creation of the cluster. This will guarantee the accessibility of files to all tasks/containers regardless of which task created the file. Also, files are persisted even with task failures.

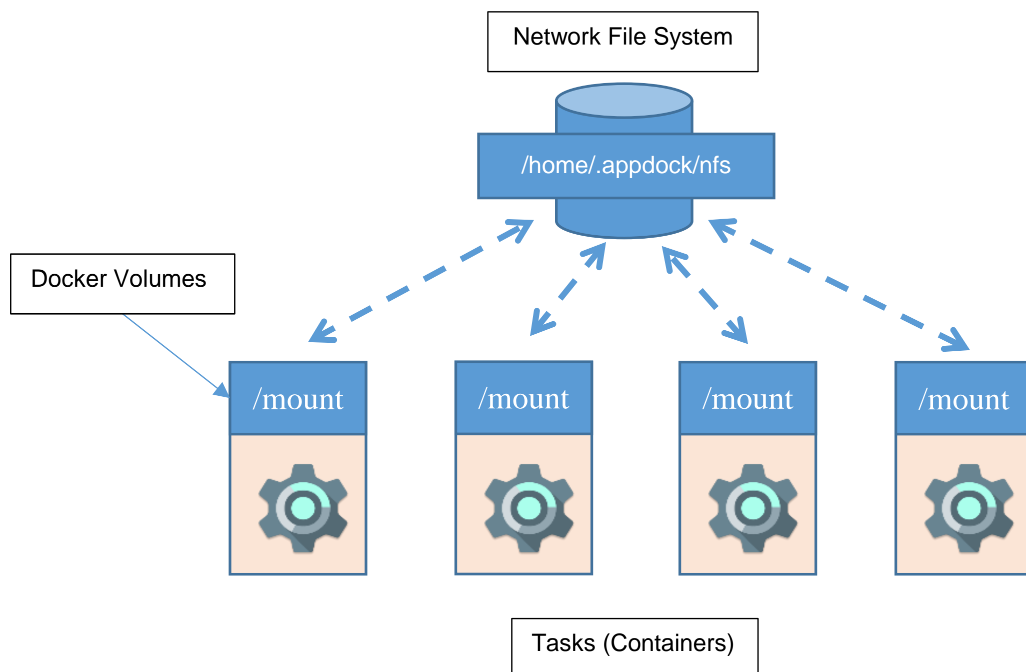


Figure 3-7 File storage built using Docker volumes and NFS.

3.3.1.4 Plugins

Applications deployed in the AppDock platform need to be created using a plugin. This happens at the service creation step. The plugin provides minimum required files to deploy an application in selected technology within the AppDock platform. A plugin is a .zip archive containing required files for service in selected technology to be deployed in the AppDock platform. Plugins can be uploaded to the web portal of the Cluster Admin module. All plugins need to have *Dockerfile* and *appdockservice.config* files. *Dockerfile* is required to build the Docker image from the artifacts from which

the service is created. It contains a sequence of commands native to Docker terminology. The *appdockservice.config* file contains the Docker service template in JSON format [35]. Other files in the .zip archive are specific to the service technology.

Ideally, applications that are deployed in the AppDock platform should be stateless because when Docker services are scaled, Docker Swarm built-in routing mechanism does not guarantee related requests are routed to the same task instance. Hence, plugins targeting particular technology should also be stateless.

When a new service is created, the selected plugin content is downloaded and extracted to the working directory. Plugin content just downloaded will have minimum required files to create a service in the AppDock platform. The first deployment by using the "appdock deploy" command will create a Docker Service in the underlying Docker Swarm cluster. Afterward, the developer could make incremental changes to the source code and push the application to the AppDock platform.

3.3.2 Command Line Interface

The command-line utility is provided for developers to interact with the platform for Continuous Deployment (CD) of the applications. It provides commands required to administer and develop applications with the platform. CLI is expected to be installed in developer workstations to push the application to the AppDock cluster as each developer does the incremental developments.

Node.js library *Commander* is used to re-use basic CLI functions. Each command is defined within a command class in which the command template and the action are defined. Each command class is injected with the required manager class, along with its dependencies (refer to APPENDIX B – Commands in AppDock CLI for supported commands).

3.3.2.1 Managers

Each command is injected with manager classes that are required for carrying out underlying operations. Manager classes communicate with Docker API in cluster nodes and API endpoints of the Cluster Admin core component. The responsibilities of each manager class are described below:

- *ClusterManager* – Operations related to managing clusters are carried out by this class. Currently, operations associated with cluster creation and calling related APIs for managing scaling service configuration via AppDock proxy interface are carried out by this manager class.
- *NodeManager* – Adding/removing new nodes to the AppDock cluster, including deploying new nodes in public cloud environments, is carried out by this class.
- *ServiceManager* – This class handles new service creation and deployment of services. Archiver class is injected into this class additionally to extract plugin content upon new service creation and archive service artifacts to be sent to deployment.

3.3.2.2 Archiver

Archiver is a class integrated into the CLI core component to perform compression and decompression of files and folders. Service plugins are in the form of .zip archives. Extracting plugin content upon service creation is done via Archiver. This is used by service deployment command to send application artifacts to Cluster Admin to be deployed as a Docker service.

3.3.2.3 Local Storage

Local storage is a helper class that provides read and write access to the data stored in file "settings.json". This file stores data required by CLI, such as Cluster Admin URL, NFS information, and possible data required by CLI in future enhancements. This file is located in the path where CLI-installed artifacts reside. Initially, *settings.json* file is created in the local node that was used to create the AppDock cluster. Upon successful creation of the AppDock cluster, the file content is stored in the LogDB. Subsequent installations of CLI application in other developers' computers will download the "settings.json" file to their computers in the path where the CLI application is installed. Figure 3-8 shows the properties used by the CLI component. *ClusterAdminURL* contains the location Cluster Admin component can be accessed. *NFSConfig.Addr* and

NFSConfig.Device contains the address of the NFS contains and remote path of the shared directory, respectively.

```
{
    "ClusterAdminURL": "<Cluster Admin Server URL>",
    "NFSConfig": {
        "Addr": "<NFS Server Address>",
        "Device": "<NFS Path>"
    }
}
```

Figure 3-8 Properties maintained by local storage.

3.3.3 AppDock Scaling Service

Scaling Service provides automated resource expansion capability to the application. This runs periodically in the manager nodes and calculates average CPU and Memory utilization of all containers of the selected node. For this, it relies on the Monitoring Agent residing in each node. Each replica selects a node to analyze and prevent other replicas from selecting the same node by marking *INodeStatAnalysisStatus.Status* (see Figure 3-9) property as “STARTED” in LogDB during the analysis period. Based on the calculated average values, resource allocation, or removal decision is made. To ensure system stability, we set a minimum time gap between two consecutive resource allocation or deallocation decisions. This time gap is decided based on the time needed to stabilize the services in the new nodes added to the cluster, and it is proportional to the number of services deployed in the cluster.

3.3.3.1 Stat Analyzer

Stat Analyzer summarizes the runtime statistics (see Figure 3-12) reported by the Monitoring Agent. It calculates the average resource utilization values to denote the current state of a particular node. Stat Analyzer selects a node, where resource utilization analysis has been not started, completed, or staled due to runtime errors during the analysis. Selection is made by creating or updating an *INodeStatAnalysisStatus* object in LogDB with Status value as “STARTED” and priority is given to the node with the oldest analysis result. A new node is selected in

each execution of the Scaling Service. Stat Analyzer output and store average CPU and Memory utilization of the node and every service of which tasks are running within the node in the *INodeStatAnalysisStatus* object in the LogDB.

```
INodeStatAnalysisStatus = {
   Status:number;
   Node: INode;
   LastAnalysisResult:{
      AvgCPUUtilization: number;
      AvgMemoryUtilization: number;
      DateTime:string,
      Services:Array<{
         ServiceID:string;
         AvgCPUUtilization: number;
         AvgMemoryUtilization: number;
      }>
   }
```

Figure 3-9 Properties of INodeStatAnalysisStatus object

3.3.3.2 Cloud Resource

"CloudResource" class depends on Cluster Admin core component and Scaling Service configurations stored in LogDB. As shown in Figure 3-10, Scaling service configuration (*IScalingServiceConfig*) contains maximum and minimum threshold values for CPU and memory utilization. Upon completion of the node stat analysis by the Stat Analyzer mentioned above, the algorithm in Figure 3-11 is used to make new node deployment and removal decisions by comparing a nodes resource utilization with the configured CPU and memory utilization threshold.

```
IScalingServiceConfig = {
   MaxCPUUtilization:number;
   MinCPUUtilization: INode;
   MaxMemoryUtilization:number;
   MinCPUUtilization:number;
   NodeUpMinimumDuration:number;
   LastScaledAt:datetime;
   ScaleStatus:STARTED|COMPLETED|ERROR;
   ResourceScalingTimeGap:timespan
   }
```

Figure 3-10 Properties of IScalingServiceConfig object

According to the *CanScaleResources* procedure, if the cluster is in a state where resource scaling is possible, scaling status is marked as "STARTED" in the LogDB before starting any scaling procedures. "AddNewNode" procedure spawns a new node in a randomly selected cloud environment and joins the node with the AppDock cluster using the node's private IP address. "RemoveNode" procedure removes the current node that owns the *INodeStatAnalysisStatus.LastAnalysisResult* object both from the cloud environment and the AppDock cluster. *ScaleAllServices* procedure increases or decreases the replicas by the *replicas per node* number set in the service creation phase for all the services other than services that belong to core components. *CanScaleResource* procedure guarantees no overlapping scaling procedures are performed by multiple instances of the Scaling Service component by maintaining a global scaling status. Also, it will wait for *IScalingServiceConfig.ResourceScalingTimeGap* before attempting to balance the cluster after a previous addition or removal of nodes.

```
Algorithm 3.1: Scale Resources

procedure Scale;
if LastAnalysisResult.AvgCPUUtilization
>ScalingServiceCon_g.MaxCPUUtilization OR
LastAnalysisResult.AvgMemoryUtilization
>ScalingServiceCon_g.MaxMemoryUtilization then
    if CanScaleResources() then
        UpdateScalingStatus(STARTED);
        AddNewNode();
        ScaleAllServices();
        UpdateScalingStatus(COMPLETED);
else if LastAnalysisResult.AvgCPUUtilization
<ScalingServiceCon_g.MinCPUUtilization AND
LastAnalysisResult.AvgMemoryUtilization
<ScalingServiceCon_g.MinMemoryUtilization AND (NodeStartedTime+
ScalingServiceCon_g.MinNodeUpTime) <CurrentTime then
    if CanScaleResources() then
        UpdateScalingStatus(STARTED);
        RemoveNode();
        ScallAllServices();
        UpdateScalingStatus(COMPLETED);

procedure CanScaleResources;
        return ScalingStatus != STARTED AND CurrentTime >=
LastAnalysisTime + ResourceScalingTimeGap;
```

Figure 3-11 Algorithm for adding and removing nodes.

When multiple public cloud environments are connected, new nodes are deployed on randomly selected public cloud environments. All the nodes deployed via Scaling Service in the public cloud environments are assigned with label “CloudNode”. Only nodes with label “CloudNode” are removed by the Scaling Service when they are not utilized up to the minimum limits to avoid removing permanent nodes, especially in the primary cloud infrastructure.

The number of replicas in each service is increased by the *replicas per node* when a new node is deployed. When a node is removed, *replicas per node* number will be reduced from the total number of replicas of each service. All the services are deployed in a balanced strategy where each node will get an equal number of replicas of each service.

Minimum and maximum CPU and memory threshold values are decided based on the type of application and typical workload on-peak and off-peak times. These thresholds are shared across the cluster and can be decided by simulating the expected load and monitoring the CPU and memory utilization values of the *INodeStatAnalysisStatus* (Figure 3-9) reported in the LogDB by the Stat Analyzer. Initial CPU utilization and memory utilization values at which the static set of primary cloud resources are fully utilized can be set an arbitrary value like 90% each and keep optimizing thresholds according to the workload and primary cloud infrastructure resources available. If there are plenty of primary cloud resources available or a single node should not reach maximum utilization of resources, lower values can be set.

3.3.4 AppDock Monitoring Agent

This is a Docker service deployed in a “global” mode where a single task is running in every node. When new nodes are deployed, a task of this stared in each new node. It periodically retrieves runtime stats via the Docker API of the local node. The Docker engine does not provide a valid place holder for the IP address of the node at the service creation time. Therefore, before reporting runtime stats to the LogDB, the IP address of the local node is searched through all the nodes by the hostname of the local node.

This is achieved by listing to all the nodes in the cluster by sending a request to one of the manager nodes and matching the hostname of the local node supplied via an environment variable. Runtime statistics (see Figure 3-12) are retained in the LogDB only for a certain period that will represent the most current state of the node in terms of resource utilization. Older stat records are overridden by the subsequent execution cycles. As shown in Figure 3-12, runtime stat record contains CPU and memory utilization of each container running in the node identified by the "NodeAddr" field at a point of time represented by "DateTime" field. Reference to the service to which each container belongs is maintained in the "ServiceID" field in each array object.

```
{
   NodeAddr:string;
   DateTime:string;
   Metrix: Array<{
      ContainerID:string;
      ServiceID:string;
      CPUUtilization:number;
      MemoryUtilization:number;
   }>;
}
```

Figure 3-12 Properties of IRuntimeStat object

3.3.4.1 Runtime Stat Meter

Runtime Stat Meter handles Reading and writing resource utilization runtime stats to LogDB, removing older stats, and calculating CPU memory utilization. To read the runtime stats for the node, all the tasks running in the node are retrieved via Docker API. Then runtime statistics for each task are retrieved by the container ID. CPU and memory utilization is calculated using the algorithms listed in Figure 3-13 and Figure 3-14 respectively. These algorithms are recommended by Docker to calculate the CPU and memory utilization of a single Docker container. Because Docker API does not provide actual utilization values for the entire node, Runtime Stat Meter uses derived values by calculating the averages of the resource utilization for all the containers running in the node. This will closely resemble the actual resource utilization of the node returned by the operating system.

Algorithm 3.2: CPU Utilization at a Given Time

procedure GetCPUUtilization(containerStats);
if *CurrentContainerCPUUsage == 0* **then**
  return 0;
CPUUtilization = NumofCPUs * 100 * (CurrentContainerCPUUsage - PreviousContainerCPUUsage)/(CurrentSystemCPUUsage - PreviousSystemCPUUsage);
  return CPUUtilization;

Figure 3-13 Calculation for CPU utilization.

Algorithm 3.3: Memory Utilization at a Given Time

procedure GetMemoryUtilization(containerStats);
if *CurrentMemoryUsage* == 0 **then**
  return 0;
MemoryUtilization = 100 * (CurrentMemoryUsage - MemoryCache)/SystemMemoryLimit;
  return MemoryUtilization;

Figure 3-14 Calculation for memory utilization.

3.3.5 AppDock LogDB

LogDB is the main data storage for the AppDock platform. Core components such as Cluster Admin, Scaling Service, and Monitoring Agent directly communicate with the data storage. CLI core component communicates with this data storage via the AppDock HTTP proxy interface. All the data schema objects currently used in AppDock platform are AWSConfiguration, AzureConfiguration, CloudResource, CloudResourceBatch, Node, NodeStatAnalysisStatus, NodeStatsLog, PluginDefinition, RuntimeStat, ScalingServiceConfig and ServiceDefinition.

3.3.5.1 LogDB Implementation

This is a MongoDB service deployed like any other core component. Even though this has been deployed as a Docker service, the presented implementation is limited to one task. This is due to a limitation in MongoDB Docker images where many DB engines cannot share single persistent storage. Objects in non-relational schema have been

modeled via Mongoose Object Document Mapper (ODM). Mongoose provides a schema-based solution to application data for MongoDB and Node.js applications.

Repository classes following the repository design pattern provide an abstraction layer over Mongoose implementation for MongoDB to the application. By following the repository pattern, we could avoid writing repetitive CRUD operations for each schema object while developing the application since the repository pattern provide those in RepositoryBase class from which all other repositories are inherited. Classes of which name prefixed as X and interfaces with IX, as shown in Figure 3-15 represent all the classes and interfaces required by each schema object. Further, Classes and Interfaces in Figure 3-15 is described below.

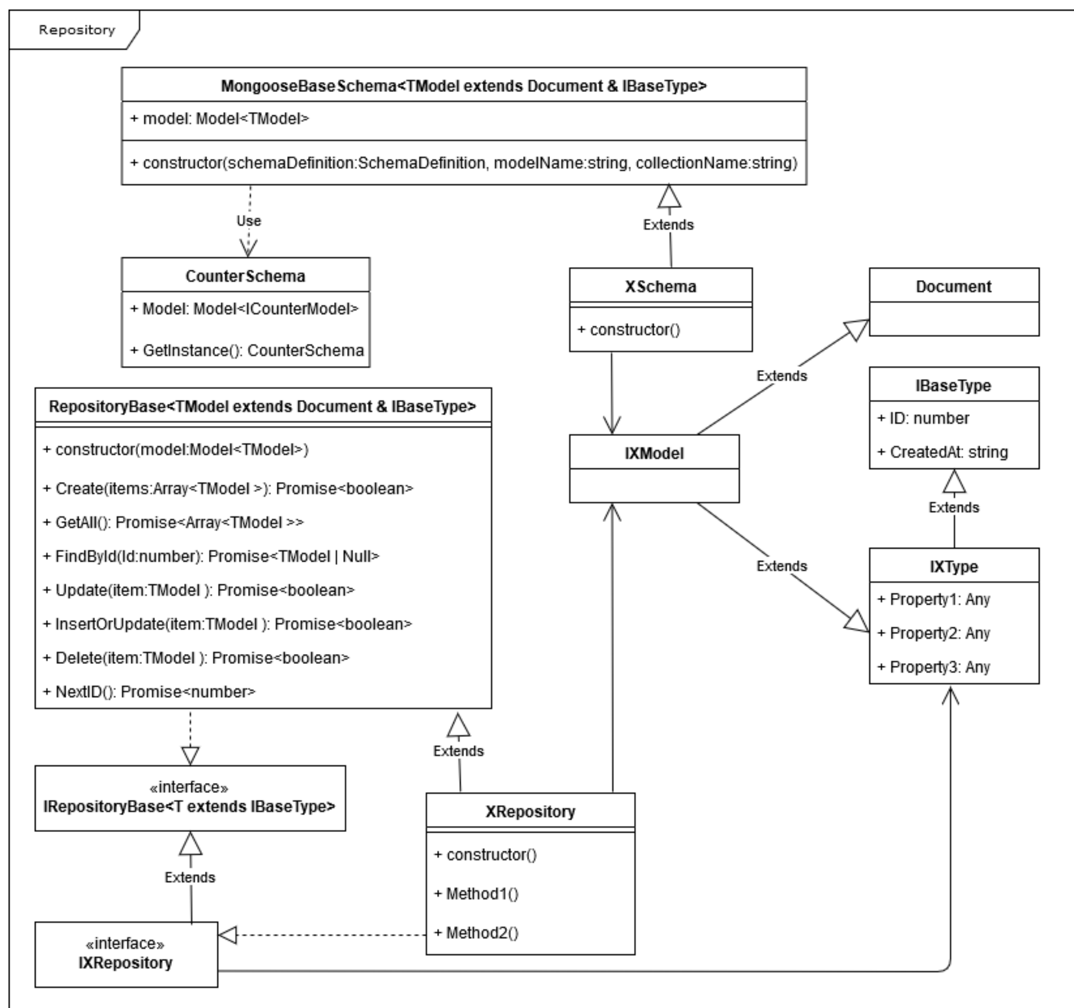


Figure 3-15 Class diagram for the repository.

- *MongooseBaseSchema* – Share the common logic for all the schema objects that inherit from this class. Common logic includes initializing the model to be used in the corresponding repository for that schema object and initializing a trigger that populates auto-increment field ID and CreatedAt with current date and time.
- *CounterSchema* – This class represents the schema object that maintains the counters for all other schema objects. The counter is incremented and used when adding a new object to a particular collection to populate the ID field of that object.
- *XSchema* – The constructor of each schema class contains the model definition of the schema object also the name of the schema object to be created in MongoDB. This is inherited from MongooseBaseSchema generic class.
- *RepositoryBase* – The generic parent class for all the repositories where database operations are defined. When an object of a child class, i.e., XRepository is instantiated, the connection to the database is made. This is a Mongoose specific implementation.
- *XRepository* – In the repository pattern, all the schema objects in the database have a corresponding Repository class where all the database operations required are defined. This class is inherited from the generic class RepositoryBase of which type (XModel) has been defined in each XRepository implementation.
- *IXRepository* – Interface that defines required methods specific to the schema object.
- *IRepositoryBase* – Interface that defines standard methods required by repositories.
- *IXType* – Interface for the entity that represents the required properties of the schema object.
- *IBaseType* – Parent entity interface that defines common properties of all schema objects.

- *IXModel* – Entity type that aggregates properties in schema objects and Document class in Mongoose.

3.3.6 AppDock HTTP Proxy Interface

AppDock HTTP acts as a proxy interface for the Cluster Admin core component. This has been referenced by other core components that require sending requests to the Cluster Admin core component. Possible requests can be made to the Cluster Admin core component via the proxy interface, which has been listed in Table A-1.

3.3.7 Docker HTTP Proxy Interface

Docker HTTP is a proxy interface for Docker HTTP API in each node. Core components that use the Docker API, reference the Docker HTTP Proxy Interface as a local NPM package. Table A-2 lists possible requests to Docker HTTP API via the Docker HTTP proxy interface.

3.4 Cluster Deployment

The AppDock node cluster consists of two types of nodes Worker Nodes and Manager Nodes. Worker Nodes and Manager Nodes ultimately required by the underlying Docker Swarm cluster. It is recommended to have at least 3 manager nodes where one node failure can be tolerated since always there should be more than half of the manager nodes running to run the Docker Swarm cluster. Even with 4 manager nodes, only 1 manager node failure is tolerated. *CreateClusterCommand* (see the command reference at APPENDIX B – Commands in AppDock CLI) is issued along with the cluster configurations explained below.

Figure 3-16 shows the minimum required configurations when creating the cluster. Nodes array specifies the list of permanent nodes participating AppDock cluster. While nodes are distinctly identified by “NodeAddr”, “AdvertiseAddr” should be specified too for Swam API access. Manager nodes are denoted by NodeType property with value “m” while worker nodes are denoted by value “w”. Docker daemon in every node should be configured to listen on TCP port 2375 and IP address specified in

"NodeAddr" to accept requests from remote hosts [36]. "ClusterAdmin" and "LogDB" objects contain configurations required by respective core components.

Network File System (NFS) allows users and programs to access files stored in a remote system over a network [37]. Configurations required in "NFS" are used to mount Docker Volumes in each service with a remote NFS path. This will allow each service to store its files in a central location accessible by each task of that service. Known hosts are mentioned in the "Hosts" section of the configuration. This will help the core components of AppDock to reach nodes by its name. Hostnames are only for the private nodes as hostnames of the nodes in public cloud environments are not known in advance. Specified DNS servers in the "DNSServers" section are used to resolve hostnames of the nodes in public cloud environments.

```
{
   Nodes: Array<{
      NodeAddr:string;
      AdvertiseAddr:string;
      NodeType:string;
   }>;
   CusterAdmin:{
      Port:number;
   };
   LogDB:{
      Port:number;
   };
   NFS: :{
      Addr:string;
      Device:string;
   };
   DNSServers: Array<string>;
   Hosts: Array<string string>;
}
```

Figure 3-16 Configurations required when deploying an AppDock cluster

3.5 Summary

We came up with a solution that integrates multiple IaaS environments with network layer connectivity to address dynamic infrastructure resource allocation and remove dependency towards cloud vendor-specific Software Development Kit (SDK)s or any third-party libraries when applications are deployed in multi-cloud setup. The solution

is a container-based platform for application orchestration named “AppDock”. The application orchestration platform can be deployed in any cloud infrastructure. Afterward, multiple IaaS providers can be integrated to expand the cluster nodes across those IaaS platforms when dynamic infrastructure resource allocation is enabled. All the IaaS cloud environments need to be connected via network layer connectivity.

The AppDock platform is well modularized and comprised of a collection of microservices deployed on a Docker Swarm cluster and a CLI tool. Those are identified as core components. Docker services include Node.js Docker services such as Cluster Admin, Scaling Service, and Monitoring Agent and LogDB a MongoDB service. Additionally, to the aforementioned core components, we developed three software libraries to be reused by those core components. Those are LogDB Repository, AppDock HTTP proxy interface, and Docker HTTP interface. We developed the AppDock application orchestration platform using Docker, Node.js, and MongoDB as main technologies. Typescript was selected over pure JavaScript because we get the type support which will be beneficial to the maintainability of the code base when the AppDock platform evolves. The AppDock platform can be deployed in a few nodes, i.e., a fixed set of infrastructure resources identified as the primary infrastructure using the *CreateClusterCommand* (see Appendix B). Multiple public IaaS platforms connected with the primary cloud infrastructure via network layer connectivity can then be integrated with the AppDock platform via the Cluster Admin web interface.

4 PERFORMANCE EVALUATION

Extensive evaluation of performance was done to demonstrate the practical applicability of the AppDock platform to host applications categorized as CPU intensive, memory-intensive, and RESTful APIs integrated into external cloud services. Each test scenario was delivered with distinctive workloads matching the compute capacity of private nodes enough to keep the private resources busy.

Section 4.1 describes the test scenarios from which the AppDock platform will be tested. The test environment and performance metrics are described in Section 4.2. Section 4.3, 4.4, and 4.5 present detailed performance evaluations for CPU intensive application, memory-intensive application, and Danveem RESTful API, respectively. The performance evaluation summary is presented in Section 4.6.

4.1 Workload

Automated resource expansion and contraction, the primary feature of the AppDock platform has to be evaluated under the workload only the private cloud nodes alone cannot handle efficiently. We designed the following three test cases to stress all the private cloud resources:

- CPU intensive workload – is a Node.js web application exposing single HTTP/GET endpoint, which triggers a CPU intensive logic. CPU intensive application consists of a single GET endpoint, which triggers a CPU intensive mathematical function that calculates the tangent and the arctangent in radians iteratively 8^7 times [38]. Even though this function not practically useful to an end-user, it is capable of keeping the CPU busy from a single request. 20 users (threads) were simulated using JMeter for this application. Refer Table 4-1 for the testing parameters.

Table 4-1 Testing parameters for the CPU-intensive workload.

Parameter	
Maximum CPU utilization threshold	90%
Maximum memory utilization threshold	20%
Minimum CPU utilization threshold	15%
Minimum memory utilization threshold	15%
Minimum node uptime	15 minutes
Replicas per node	2
Number of private nodes	2
Number of threads (users)	20
Ramp-up time	10 minutes
Test duration	40 minutes
Think time	0 seconds
Arrival rate	0.03 users per second

- Memory-intensive workload – is a Node.js web application exposing single HTTP/GET endpoint, which triggers a memory-intensive logic. Refer to Table 4-2 for the test parameters. Memory intensive application is a Node.JS web application that exposes a single GET endpoint from which memory intensive function is triggered. This function does not do anything useful but declares a Float64Array with 2^{20} elements. This will consume a considerable amount of memory in the container from a single request. However, Docker will stop any containers to consume memory to the detriment of the node [39]. Since this is possible with real applications too, 150 users were simulated with low compute capacity. i.e., three primary cloud nodes. This caused some containers to be failed by throwing this error.

Table 4-2 Testing parameters for the memory-intensive workload.

Parameter	Value
Maximum CPU utilization threshold	40%
Maximum memory utilization threshold	50%
Minimum CPU utilization threshold	5%
Minimum memory utilization threshold	15%
Minimum node uptime	15 minutes
Replicas per node	2
Number of private nodes	3
Number of threads (users)	150
Ramp-up time	10 minutes
Test duration	40 minutes
Think time	0 seconds
Arrival rate	0.25 users per second

- RESTful API workload – is a Node.js web application integrated with AWS DynamoDB as the backend exposing 6 HTTP/GET and POST endpoints. Refer Table 4-3 for test parameters. “Danveem” is a RESTful web service developed using Node.js and integrated with Amazon DynamoDB, which is a key-value store and a document database. This application was designed to be deployed in AWS Elastic Beanstalk. However, later application was adopted to be containerized by making the required modifications. JMeter test was created simulating 1200 users to request endpoints of the Danveem API mentioned below.
  - Create Board (POST Request)
  - Create Notice (POST Request)
  - Create User Invitation (POST Request)
  - Get Invitations for an Email (GET Request)
  - Get Notices for a Board (GET Request)
  - Get User (GET Request)

Table 4-3 Testing parameters for the RESTful API application.

Parameter	Value
Maximum CPU utilization threshold	95%
Maximum memory utilization threshold	30%
Minimum CPU utilization threshold	5%
Minimum memory utilization threshold	15%
Minimum node uptime	15 minutes
Replicas per node	2
Number of private nodes	1
Number of threads (users)	1200
Ramp-up time	10 minutes
Test duration	40 minutes
Think time	0 seconds
Arrival rate	2 users per second

Maximum CPU and memory thresholds indicate the upper limit of the respective resource utilization values to which private nodes can handle the load. Once this limit is reached, new nodes need to be created in connected public clouds. When the minimum CPU and memory threshold is reached, nodes in public cloud environments could be released once the minimum node uptime is exceeded. The minimum node up time setting allows any public nodes to retain in the cluster even though resource

utilization values of that node are below the minimum threshold values. By holding on to such resources until the minimum node up time is reached, we can prevent the system from becoming unstable due to the rapid reconfiguration of the system. Further, this does not introduce additional costs as cloud resources are already paid for some minimum usage. Replicas per node indicate the number of tasks (containers) a service can run in a single node.

Maximum and minimum threshold values were decided by observing resource utilization behavior during the test run in private cloud mode, where automated resource expansion is disabled. We kept every cloud node running a minimum of 15 minutes regardless of reaching the minimum resource utilization threshold values giving enough time to spawning services within them. Duration for all the tests was 40 minutes, including the ramp-up period of 10 minutes. Think time was set to 0 in each test case to measure the maximum throughput. The arrival rate is calculated by dividing the number of users by the ramp-up period in seconds. Replicas per node and the Number of private nodes were decided based on the minimum resource requirements for the type of workload running without any task failures.

4.2 Experimental Setup

Workloads in the test cases mentioned above were deployed in both private cloud mode and multi-cloud mode. Workload traces were played using Apache JMeter [32]. As shown in Figure 4-1, Subnet 1 in the AWS VPC acted as the private cloud. Nodes in the Subnet 1 were connected with Subnet 2, which acts as the public cloud via inter-subnet connectivity in AWS VPC. The node running the JMeter workload had a client VPN connection to the private cloud through which Subnet 2 can also be reached.

CPU intensive workload and memory-intensive workloads are not integrated with any backend. User requests are also independent. RESTful API workload is a Node.js application which consists of GET and POST endpoints integrated with an AWS DynamoDB backend. Our objective with these experiments is to demonstrate the effectiveness of infrastructure resource elasticity and scaling applications in the multi-cloud setting by evaluating the performance of the three workloads deployed in private

cloud mode with a static set of infrastructure resources and multi-cloud mode with infrastructure resource elasticity.

In the setup, three types of workloads (Memory intensive, CPU intensive and RESTful API) were deployed in the private cloud mode one at a time with a pre-defined number of private nodes as listed in Table 4-1 and Table 4-3. Then response time and throughput were measured. Again, we enabled the multi-cloud mode adding AWS public cloud where new nodes were spawned and measured the same. In the multi-cloud mode, the application spawned new nodes in the AWS cloud, and it was steady until the JMeter load was completed. Then they were removed automatically when the load was removed.

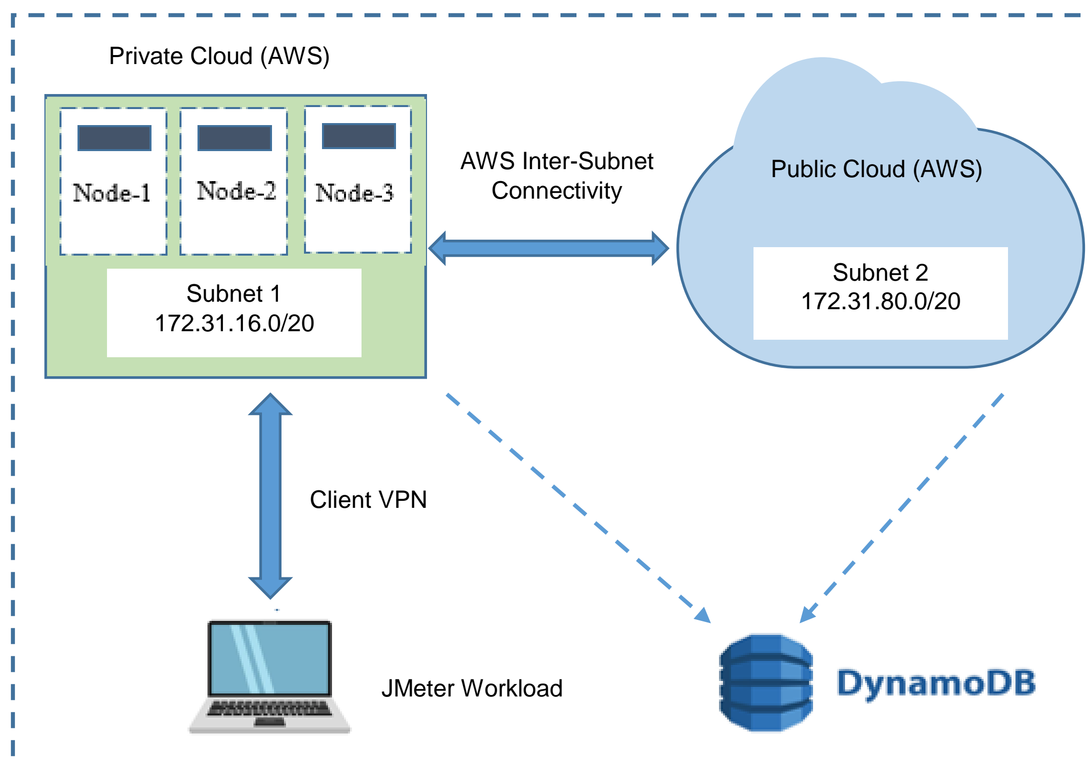


Figure 4-1 Deployment diagram for the experimental setup.

All the nodes in the AppDock cluster (private nodes and public nodes) were AWS EC2 t3a.small Ubuntu instances with AMD EPYC 7000 series processors with an all-core turbo clock speed of 2.5 GHz (2vCPUs), 2 GB of RAM, and 20GB of SSD persistence storage. JMeter was deployed on a laptop with Intel Core i7-8750H CPU @ 2.20GHz (12 CPUs), 16GB memory, 512 GB SSD storage. JMeter workload generator was

connected through a VPN connection, upload and download bandwidth were set to 50Mbps and 100Mbps, respectively.

For each of the test cases, response time and throughput data were calculated for both private cloud and multi-cloud modes. We configured JMeter to log each request for a particular endpoint to a CSV file, and we recalculated JMeter metrics for every 10-seconds.

4.3 Performance Evaluation of CPU Intensive Workload

Initially, we tested the performance under the CPU intensive workload, and the targeted system consisted of only in the private cloud. Next, we ran the same test in a multi-cloud mode connecting both private and public cloud nodes. In the multi-cloud mode, after completing the 40-minutes of testing, including the ramp-up period, we stopped the workload and let the system remain idle to release accumulated public cloud resources.

4.3.1 Throughput Analysis

Figure 4-2 shows the throughput of the CPU intensive application with time. It can be seen that the multi-cloud mode results in higher throughput compared to the only having the private cloud nodes. While private cloud nodes saturate during the workload ramp-up, the multi-cloud mode can increase the throughput until the steady-state is reached. Then the multi-cloud mode retains the throughput, which is 180% higher than the private cloud mode. This is because the private cloud mode does not have sufficient resources to meet the workload demand; hence, require to acquire resources from the public cloud to meet the workload. This confirms that the AppDock platform can dynamically provision resources from the public cloud nodes to meet the workload demand effectively.

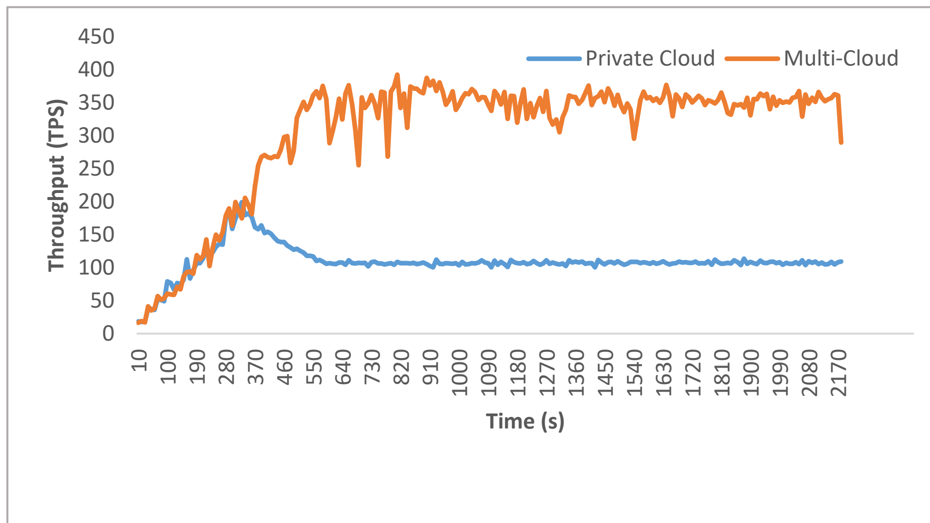


Figure 4-2 Throughput comparison – CPU intensive application.

4.3.2 Response Time Analysis

Figure 4-3 shows the average response time for CPU intensive workload when deployed on both the private cloud and multi-cloud modes. When the workload is deployed in the multi-cloud mode, a clear improvement in the response time (36%) is visible. It can be seen that AppDock can maintain steady response time by acquiring resources from the public cloud as the workload increases. Whereas in the private cloud mode, response time becomes stable with a higher value (1400ms) under the full workload after the ramp-up period.

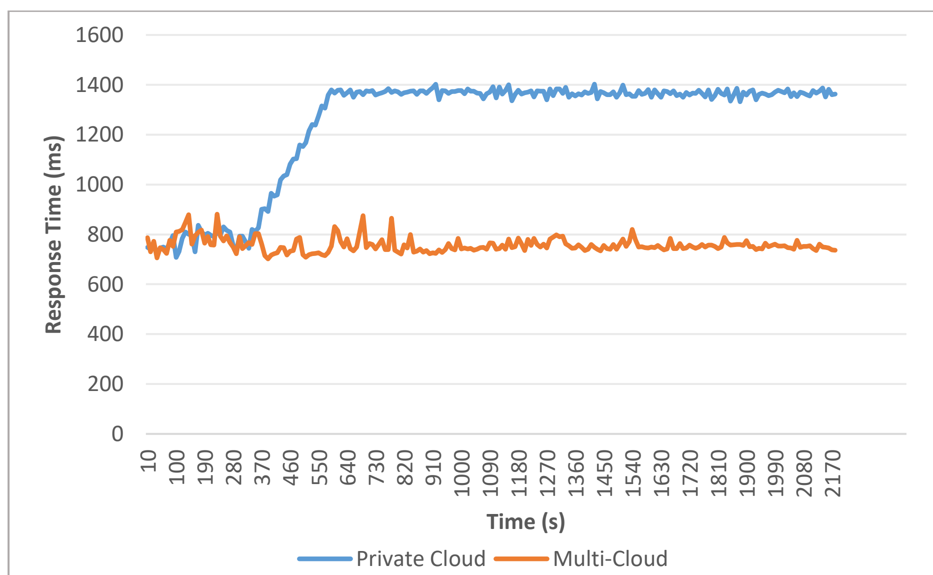


Figure 4-3 Response time comparison - CPU intensive application.

4.3.3 Resource Utilization Analysis

Because the AWS EC2 *t3a.small* instances consist of 2 vCPUs CPU utilization could reach up to 200%. That can be seen in Figure 4-4, which shows CPU utilization of each VCPU (Virtual CPU) in the private cloud node. In Figure 4-5 we can see that that the CPU utilization is reduced in each node as extra public nodes deployed in the multi-cloud mode. In this case, the node 172.31.88.28 is the manager node that does not contribute to the application compute capacity. Also, in Figure 4-5 deployments of new nodes have caused new series to begin in the middle of the time axis. The CPU utilization of both the nodes became stable after the ramp-up period, which is the first 600 seconds.

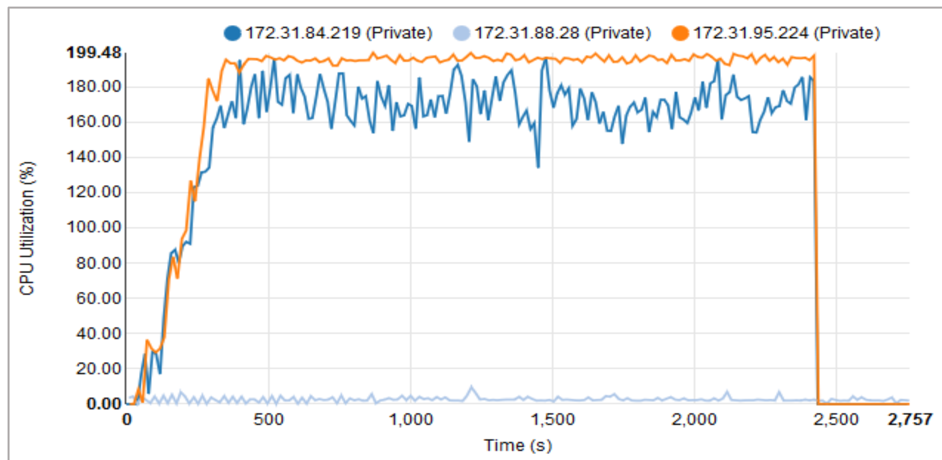


Figure 4-4 CPU utilization under CPU intensive workload – private cloud mode.

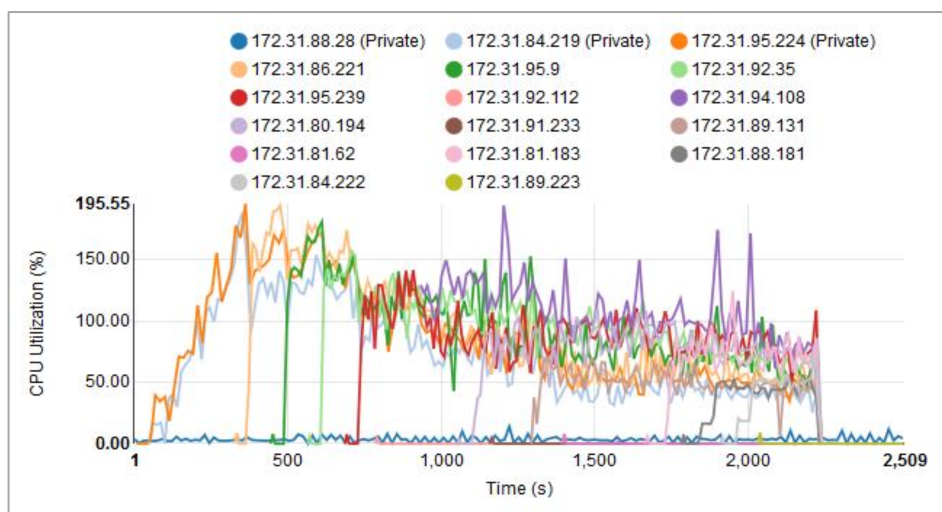


Figure 4-5 CPU utilization under CPU intensive workload – multi-cloud mode.

By observing Figure 4-6 and Figure 4-7 it can be seen that there is no noticeable difference in memory utilization in the nodes for this application in two deployment modes because it consumes less memory compared to CPU.

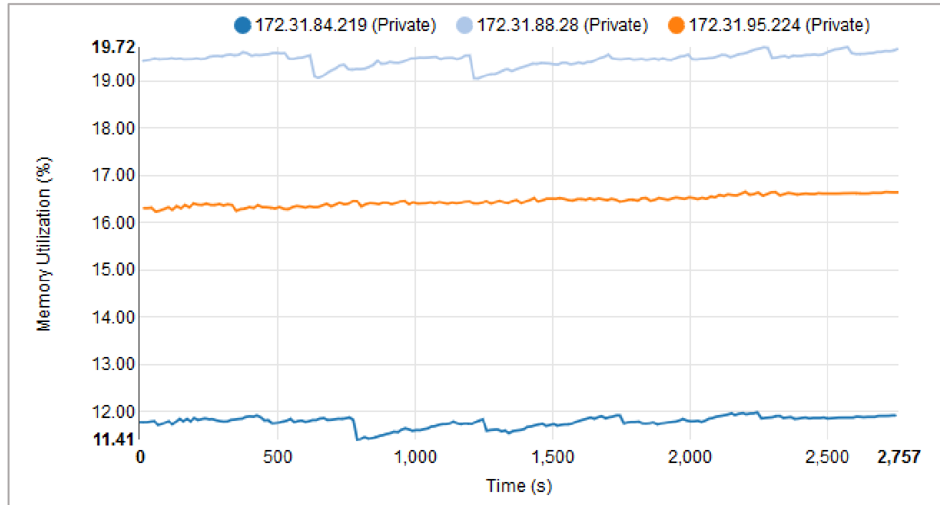


Figure 4-6 Memory utilization under CPU intensive workload – private cloud mode.

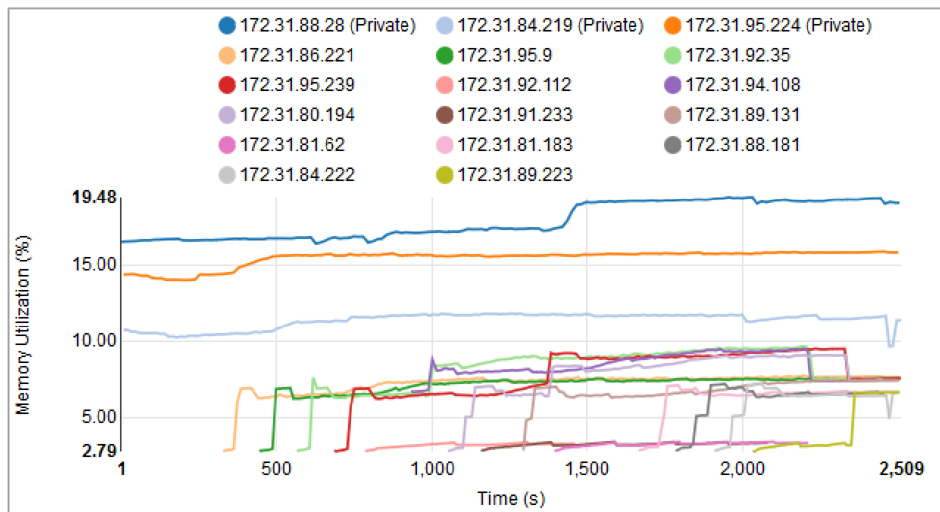


Figure 4-7 Memory utilization under CPU intensive workload – multi-cloud mode.

4.4 Performance Evaluation of Memory Intensive Workload

The automated resource expansion feature of the AppDock platform was effective in this scenario as it spawned new nodes in the public cloud environment by keeping memory utilization bellow the maximum threshold value. However, in the initial

stages of the test, we experienced task failures, which have been explained under the memory-intensive workload in Section 4.1. The impact of this has been reflected in the following sections.

4.4.1 Throughput Analysis

In Figure 4-8, throughputs in private cloud mode have been affected by task failures. After each drop recorded in the private cloud mode, there is a slight improvement in the throughput since a new task is stared with enough memory. Even though the workload became steady after the ramp-up period (i.e., 600s), to overcome the task failures and cluster to become stable without further failures by spawning nodes giving enough memory, it has taken up to the 1400s to reach a steady throughput. When comparing overall performance throughput has been better in multi-cloud mode. Overall throughput gain is 73% when compared to the private cloud mode.

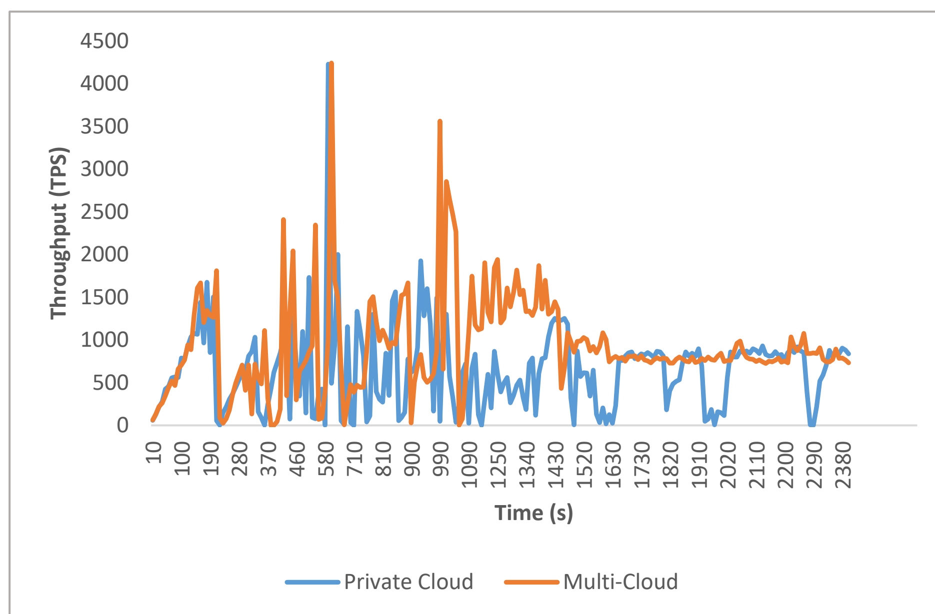


Figure 4-8 Throughput comparison – memory-intensive workload.

4.4.2 Response Time Analysis

We experienced staled requests due to random task failures and accumulated memory with tasks. Consequently, the response time for some of the requests was much longer. In Figure 4-9 we can observe this behavior during the first 1400s in the test. However, conditions have been improved after the first 1400s where we see a steady but higher response time. This is a result of the dynamic infrastructure expansion to the connected public cloud providing sufficient memory to run the workload. Afterward, there are no task failures recorded due to overutilization of memory. Response time has been higher in multi-cloud mode due to accumulated memory in tasks that are always running. In contrast, in the private cloud mode, new tasks due to task failures have delivered lower response times for a smaller number of successful requests, which can be seen in lower throughput in the throughput analysis. Overall response time drop is 232% compared to the private cloud mode with lower throughput.

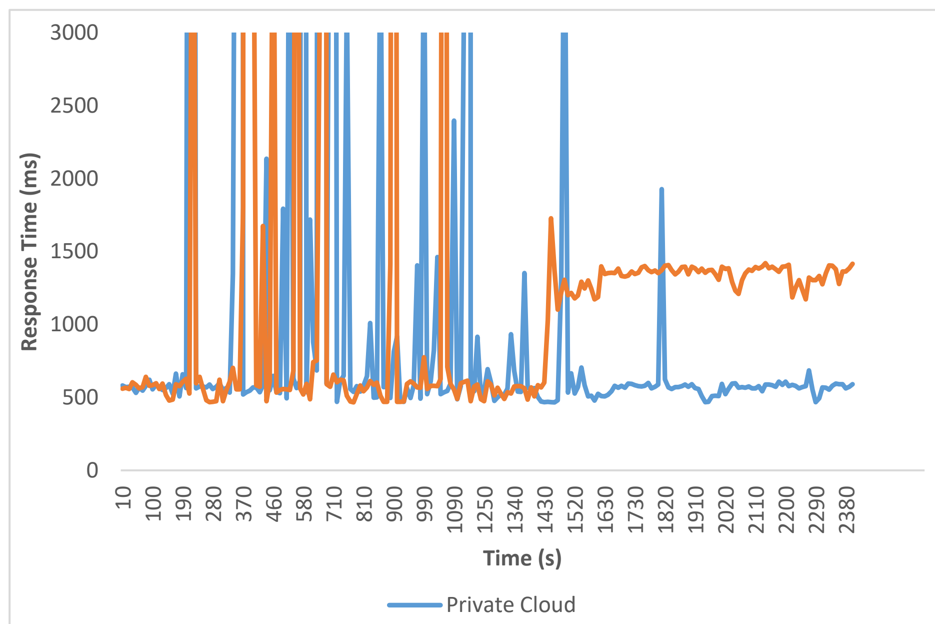


Figure 4-9 Response time comparison - memory intensive workload.

4.4.3 Resource Utilization Analysis

Figure 4-10 and Figure 4-11 show the memory utilization of nodes when the memory-intensive workload is applied to private cloud mode and public cloud mode,

respectively. Node 172.31.87.50 and 172.31.86.222 had been the manager nodes in the two deployment modes. In the multi-cloud mode (Figure 4-11), rapid fluctuations of memory utilization values in the first half of the test duration was caused by the aforementioned task failures. It can be seen that with the new deployment of nodes in the public cloud, memory utilization value has become stable and memory utilization values are maintained below 50%, which is the maximum memory utilization limit.

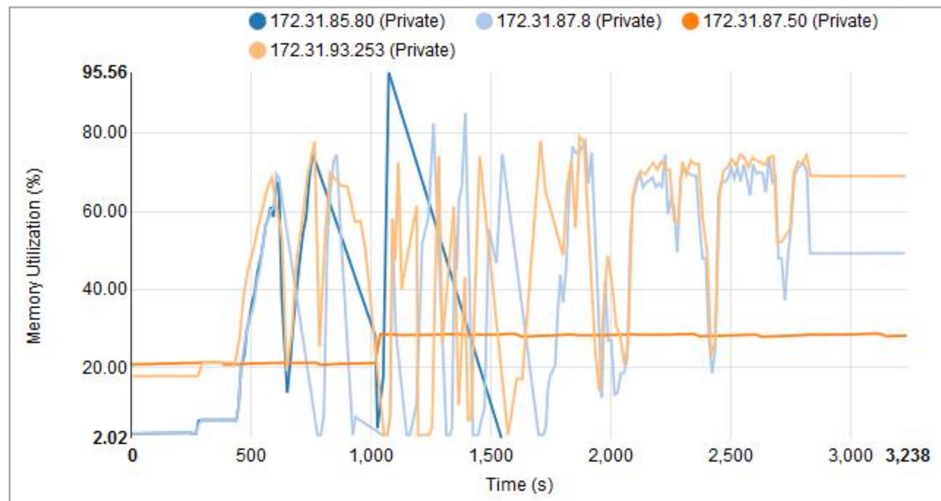


Figure 4-10 Memory utilization under memory-intensive workload – private cloud mode.

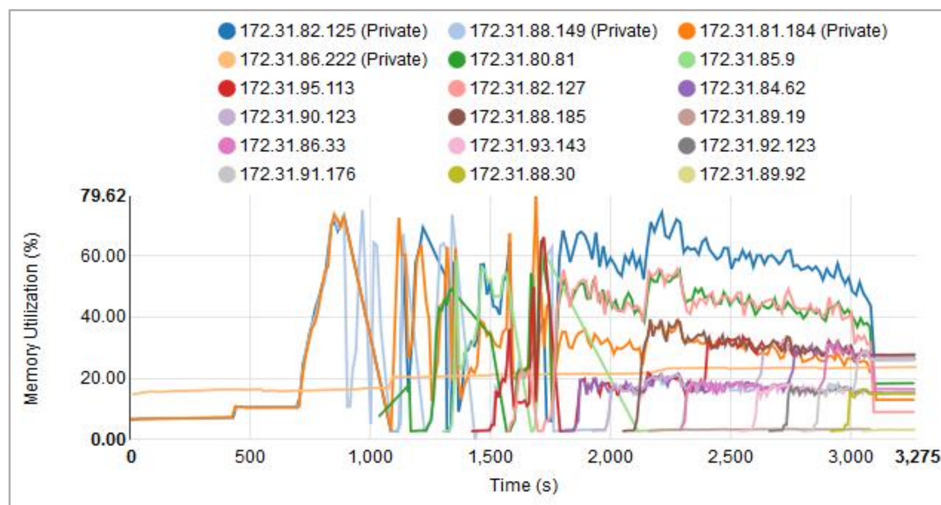


Figure 4-11 Memory utilization under memory-intensive workload – multi-cloud mode.

Figure 4-12 and Figure 4-13 show the CPU utilization of each node when this application is deployed in private cloud mode and multi-cloud mode, respectively. Exceeding maximum CPU utilization threshold (40%) in multi-cloud mode may or

may not had been caused new nodes to be deployed because those recordings may have occurred during a no-scaling period of 2 minutes after each new node deployment to stabilize the load distribution.

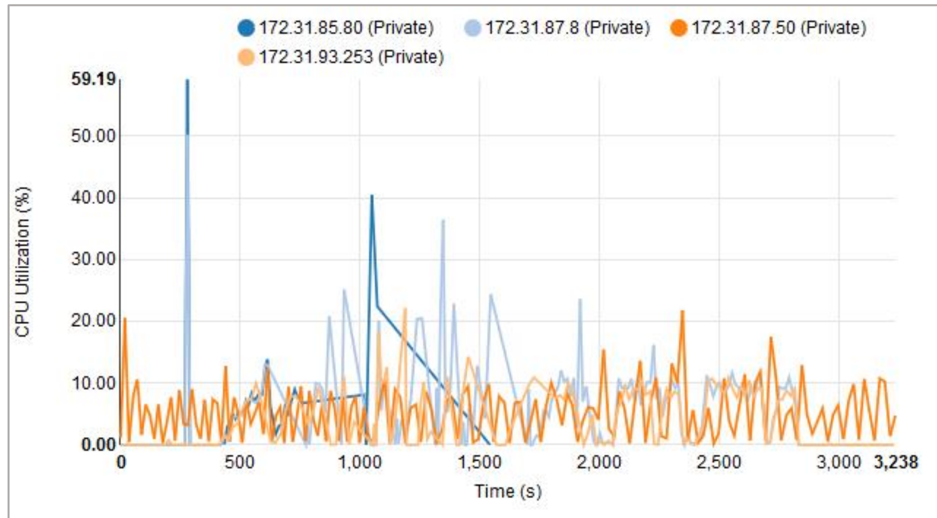


Figure 4-12 CPU utilization under memory-intensive workload – private cloud mode.

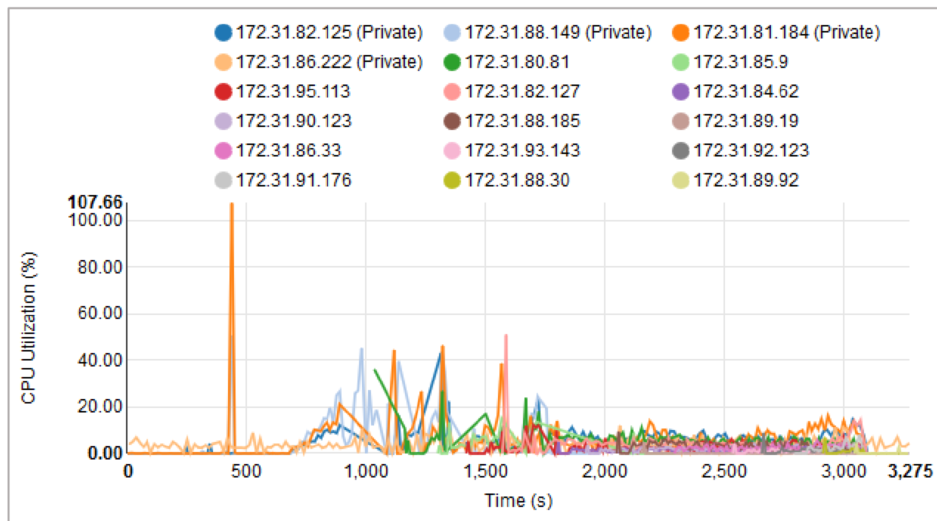


Figure 4-13 CPU utilization under memory-intensive workload – multi-cloud mode.

4.5 Performance Evaluation of RESTful API Workload

Private cloud mode deployment of this application was done using a single worker node. Before deciding on the test parameters of this test, we tested the same application with a higher number of nodes and a smaller number of users. We were able to observe

that there is no visible improvement in multi-cloud mode with automated resource expansion is enabled because private nodes alone had been able to handle the node without any performance loss. Therefore, we increased the threads up to 1200 with a single private cloud worker node to get visible improvement in the multi-cloud mode. Throughput and response time were calculated on the aggregated test results of the six requests even though we collected performance test results for the individual request.

4.5.1 Throughput Analysis

Figure 4-14 shows overall throughput comparison data of the Danveem API. Request data of all 6 endpoints were used to calculate the overall throughput. We could see that the overall throughput is improved by 46% in multi-cloud mode. Significant improvement in throughput can be gained by increasing the number of users (threads) further. Initial elevation of throughput was observed due to accumulated data introduced by API requests making the application delivering lower throughput for subsequent requests.

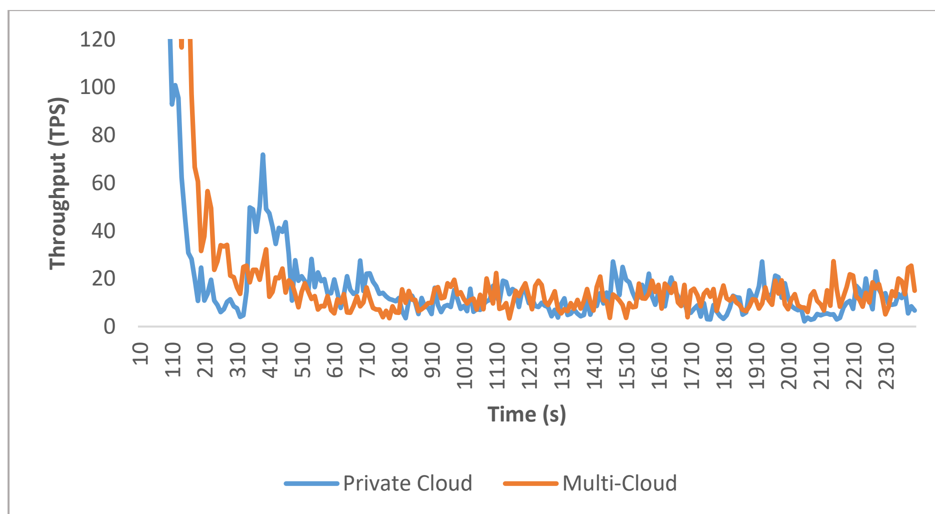


Figure 4-14 Overall throughput comparison – REST API workload.

4.5.2 Response Time Analysis

As we can see in Figure 4-15, the overall response time has been improved in multi-cloud mode. Response time data for individual endpoints were aggregated to calculate the overall response time. The overall reduction in response time is 7%.

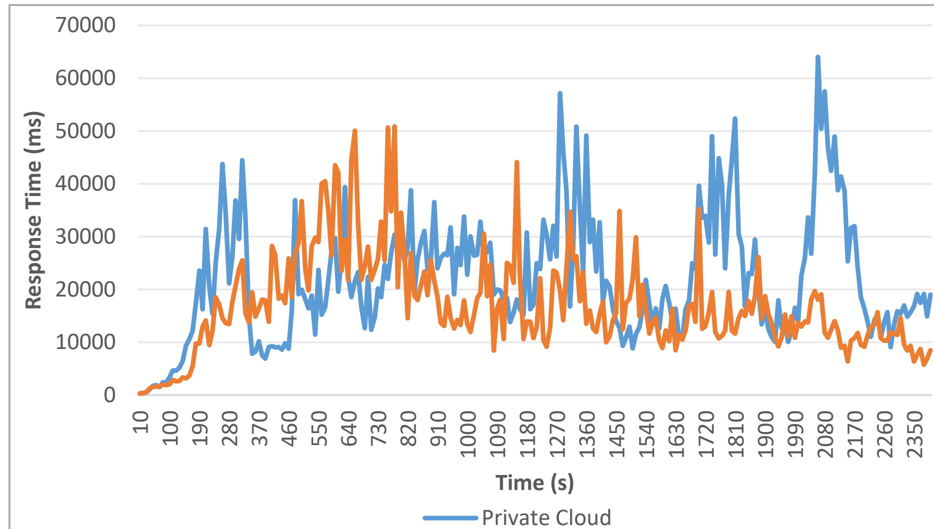


Figure 4-15 Overall response time comparison - REST API workload.

4.5.3 Resource Utilization Comparison

The node named as 172.31.94.177 shows steady CPU utilization throughout the test in Figure 4-16 is the manager node. We can observe that 2 vCPUs have been almost fully utilized during the test. Hence, based on these stats maximum CPU threshold value was decided as 95%.

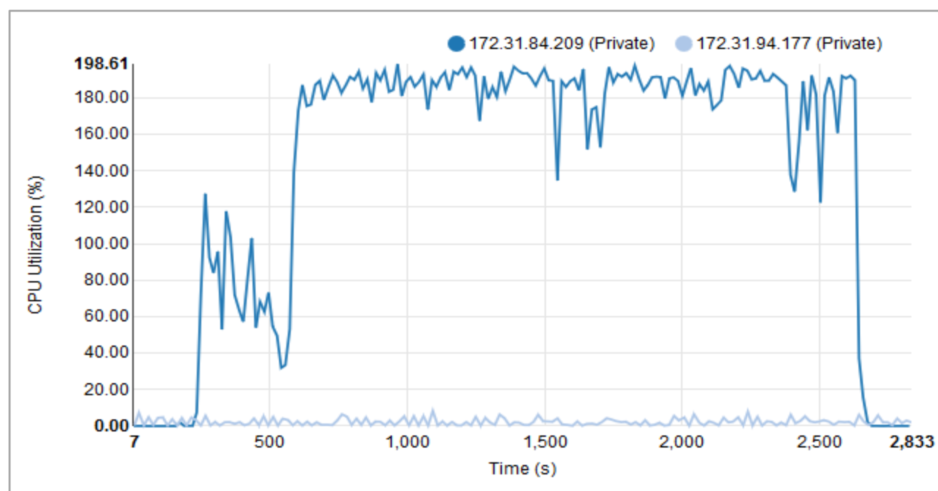


Figure 4-16 CPU utilization under REST API workload - private cloud mode

As shown in Figure 4-17, 2 nodes were automatically deployed by the AppDock automated resource expansion functionality. Node with IP address 172.31.90.251 had been removed after 15 minutes as it did not contribute to the capacity to meet the minimum CPU or memory utilization threshold values. We can see that the two nodes 172.31.84.209 and 172.31.82.250 have been able to maintain the CPU utilization value below the maximum CPU utilization threshold of 95%. The maximum memory utilization threshold value was decided as 30% to keep the memory utilization of all the nodes lower that amount by observing the statistics shown in Figure 4-18.

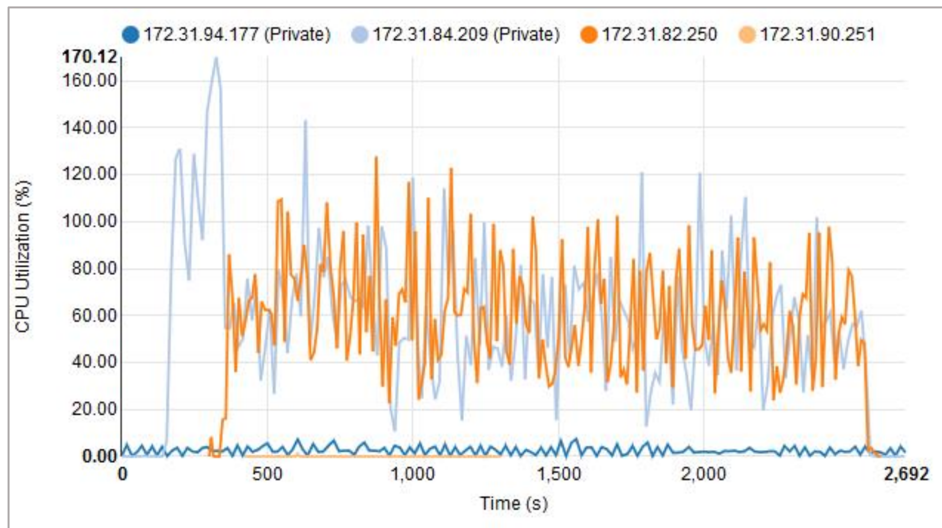


Figure 4-17 CPU utilization under REST API workload- multi-cloud mode.

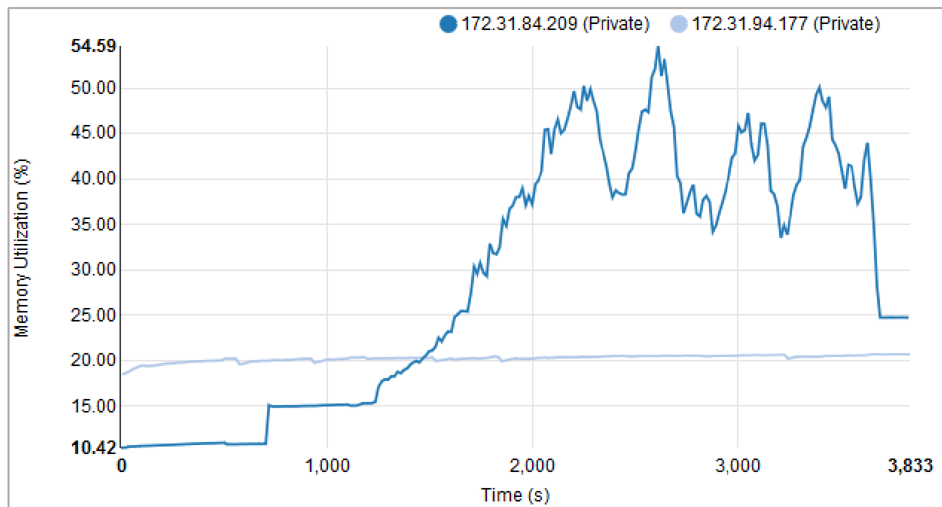


Figure 4-18 Memory utilization under REST API workload – private cloud mode.

Figure 4-19 shows that any node has not exceeded the maximum memory utilization threshold value of 30%. This result shows that automated infrastructure resource expansion can keep the resource utilization of the nodes in the cluster within the minimum and maximum boundaries.

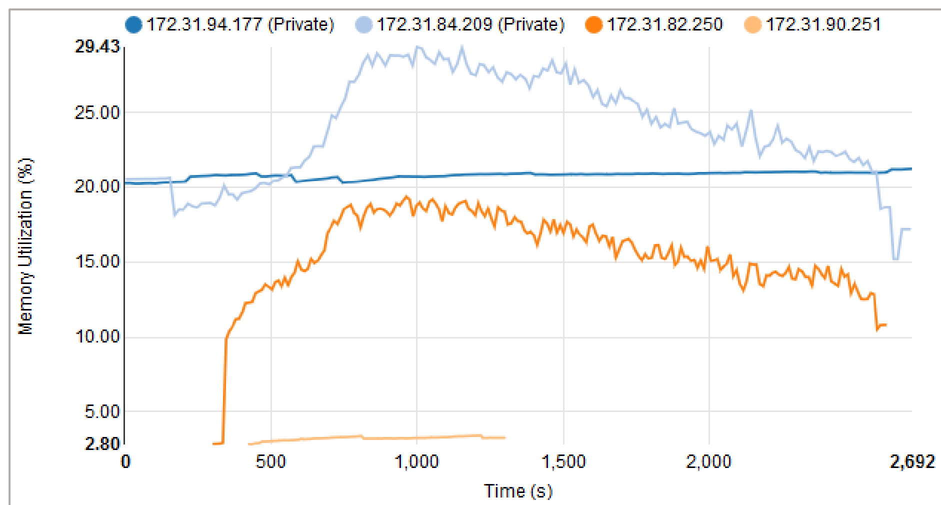


Figure 4-19 Memory utilization under REST API workload – multi-cloud mode.

4.6 Summary

We evaluated the performance of the AppDock platform for the possible type of workloads that could be deployable in this platform. Those are CPU intensive workloads, Memory intensive workloads, and applications integrated with external services with balanced CPU and memory utilization. Almost all the applications that could be containerized can be categorized into any of these types gaining real cost benefits with performance. In the evaluation, we could see that the AppDock platform performed well, handling extreme CPU and memory workloads.

This performance gain is further extended when the number of users is increased. This performance evaluation was able to prove that a multi-cloud approach with dynamic resource allocation can be used successfully with improved performance for any type of application. As we observed increased response time in the multi-cloud mode for the memory-intensive application, it is advisable to release the memory after the request has been served. We saw that restarting tasks due to out of memory failures lowered the response time. This phenomenon has caused releasing the memory and eventually led to lowering the response time in private cloud mode.

5 CONCLUSIONS

5.1 Summary

Contemporary multi-cloud solutions have drawbacks such as tight coupling with the applications, deployment complexities, or confined set of fixed infrastructure resources, making it challenging to serve time-variant demands. We proposed a container-based platform for multi-cloud application orchestration that can provision infrastructure resources automatically. This leads to a significantly reduced cost for time-variant resource demand due to resource efficiency. Combined with simplicity, the proposed solution best fit for multi-cloud applications over other multi-cloud software solutions and deployable PaaS solutions. Because the solution is based on containerization, applications do not have any dependency on the platform. We based the proposed platform, namely "AppDock," on Docker container orchestration platform so that any application deployable as a Docker service can be deployed within the proposed platform too. The proposed platform has been designed to simplify deployment and application migration in mind. AppDock platform can be deployed in any number of identical nodes that act as the primary cloud. The primary cloud can be established in any public or private cloud infrastructure. Nodes can be prepared for the deployment by using the initial script provided with the platform. Once deployed, multiple public cloud providers connected with the primary cloud via network layer connectivity can be integrated by providing the required configurations through the web interface of the Cluster Admin core component. Although the current system can only be integrated with AWS and Azure, other public cloud providers can also be integrated as future enhancements as the proposed architecture is extensible. Applications that are deployed in this platform are created as plugins. Even though the platform has only provided Node.js plugin, developing plugins for other technologies when those technologies are ready to be deployed in the Docker Swarm architecture is easily enabled by the plugin architecture of the AppDock platform. Because this is only a hosting platform, AppDock does not provide any SDKs for the applications. This will guarantee that applications will not tightly be coupled with the hosting platform. Hence, the AppDock platform avoids vendor lock-in syndrome.

All the core components of the AppDock platform are developed as Docker services. Apart from the Monitoring-Agent core component, all other services are runs only inside the manager nodes. Monitoring-Agent resides in all the nodes within the cluster and reports the runtime metrics to LogDB to be analyzed by the Scaling-Service core component. This platform is developed using Node.js. Since all the core components are running as Node.js containers deploying applications in other technologies is allowed natively by Docker. As all the core components provide their service through a REST API AppDock's architecture reflects the SOA. This makes it easier to extend the platform. Even though only the core functionalities have been implemented within the scope of this research, applications with time-variant resource demand can still be benefited as all the minimum required utility tools are developed. CLI tool is used for deploying the AppDock platform within the primary cloud infrastructure and continuously push applications to AppDock. The Scaling-Service core component is mainly responsible for providing automated resource expansion features in this platform. In collaboration with the Monitoring-Agent core component by utilizing runtime metrics data, it analyzes CPU and memory utilization of each node in every configured interval. Resource expansion or removal decisions are made if calculated utilization values are not within the configured maximum and minimum threshold values. After each scaling decision made, 2 minutes of the no-scaling period is given to cluster balance the workload with newly spawned nodes or remaining nodes after the removal of idle nodes.

We tested the effectiveness of AppDock by deploying it in different cloud providers such as AWS EC2 and UpCloud. In such cases, cloud environments were connected via site-to-site VPN connections. In addition to evaluating the effectiveness of deploying the platform in multiple deployment scenarios, we performed a comprehensive performance test covering three workload types, which include CPU intensive, memory-intensive, and web applications integrated with external services. Performance tests confirmed the performance improvement when the automated resource expansion is enabled. CPU intensive workload exhibited significant improvements in performance by improved throughput and response time. When it comes to the memory-intensive workload, even though it displayed improvement in

performance with throughput, average response time increased. We assumed that accumulated memory in multi-cloud mode had been the reason for this behavior since we noticed that multiple task failures had been occurred due to out of memory conditions in the nodes when deployed in the private cloud mode. This has led to improved response time for successful responses, while many requests had been failed. The RESTful API displayed expected performance improvement in both overall throughput and response time in multi-cloud mode. Throughput improvement in CPU intensive, memory-intensive, and REST API workloads were 180%,73%, and 46%, respectively. While CPU intensive and REST API workloads reported improved performance of response time as 36% and 7%, memory-intensive workload reported a performance loss in response time 232% due to task failures.

5.2 Research Limitations

Providing simple to deploy a hosted solution in a multi-cloud environment has its limitations while maintaining simplicity. When the solution is evolving with new features added, it is inevitable to introduce further configurations for deployment and maintenance. Since the application was intended to be considering simplicity as its top priorities possibilities of adding sophisticated features like in OpenShift [6] and Cloud Foundry [7] would be limited.

Since the AppDock platform has heavily adopted the Docker echo system, only applications that can be containerized can be deployed in this platform since ultimately, these applications are deployed as Docker services within the cluster. Most of the persistence storage technology vendors such as MongoDB and MySQL have not designed their applications automated scaling in mind. Adopting their containers to be scaled in a Docker Swarm cluster is challenging. This will require significant deployment effort additionally to AppDock cluster management effort. Some of the widely adopted applications cannot be deployed on this platform. Thus, we have to look for alternative applications that are Docker Swarm friendly.

Multi-Cloud approach under the IaaS model depends on the network layer connectivity, which is only achievable via VPN or costly direct connections like AWS

Direct Connect [40]. Both options have their drawbacks. The reliability of VPN connections cannot be guaranteed for critical applications. Even though we could maintain a secondary VPN connection to improve the reliability, network communication over VPN has a performance impact on the application. Indented cost-effectiveness cannot be gained if multiple cloud providers are connected via direct network connections. Because of these reasons having to connect multiple cloud providers via VPN is a limitation in this approach. Placing the private cloud within the public cloud environment itself is a viable alternative solution for this problem as we have already tested successfully in this deployment mode doing our performance test.

The threshold values used by the resource scaling logic is the same for all the services deployed within the platform. This means all the services deployed in the platform will have to be scaled as new nodes are added or removed from the node cluster. We have introduced replicas per node number parameter at the service level to gain granular control over services to be scaled by this number when new nodes are added or removed. Hence, maintaining common resource utilization threshold values for all the services has led to scaling all the services by its replicas per node parameter value when infrastructure resources are scaled when resource utilization of a particular node has mostly affected by a single service.

We have not considered overhead when collecting resource utilization metrics from the nodes. Monitoring Agent running inside each node will communicate with the Docker engine in its machine periodically. Even though we do not think this is posing a major impact on the performance of containers running the applications, it is worth looking deeper into Docker's implementation of how these stats are reported.

The AppDock platform is capable of hosting multiple services simultaneously. Our performance evaluation has not covered this. Since our categorization of workloads was mainly CPU intensive, memory-intensive, and REST web API integrated with a PaaS backend service, this does not cover a workload that is a mix of these and with multiple services.

Performance results may vary based on the test parameters, especially threshold values, the number of primary cloud nodes, and the number of users. We decided these

values empirically based on the initial run in each test case in the private cloud mode. Currently, we do not have a systematic methodology or equation to build a relationship among these parameters. Further performance gains could be made with the right set of values for the parameters above.

The selection of the public cloud provider when the infrastructure resource scaling decisions are made follows a random approach. We think it is more desirable to follow a priority-based approach based on the user-defined criteria. We suggest providing configurations to the user set the priorities by considering high availability requirements of individual service to run in a mix of cloud environments in case network connectivity drops between cloud providers. Preferred cloud environments could also be set based on the available regions of individual service for improved QoS (Quality of Service). While integrated with multiple cloud environments, it is important to configure the workload distribution cost-effectively. These factors can be considered when calculating the priority of a cloud environment.

5.3 Future Work

Even though there may be many new features possible with this kind of platform, the following functional areas were identified as immediate future work:

1. Enabling multiple instances for LogDB core component – Currently, the LogDB core component in this solution uses MongoDB as the persistence storage. Even though we initially planned running MongoDB multiple instances as a Docker Service for a single storage file located in the NFS folder, we were able to run a single container in standalone mode. This has hit the application with a single point of failure. We need to further work on extending MongoDB into multiple instances in Docker Swarm mode or changing the persistence storage technology to any other technology. LogDB Repository developed using a repository pattern to deal with this kind of LogDB technology changes. Upon selecting viable technology, a new implementation for the existing set of interfaces has to be added in addition to the Mongoose implementation for MongoDB.

2. Adding multiple manager nodes – With the current implementation, multiple manager nodes can be added when the platform deployment and when new nodes are added via CLI command "*appdock addnode*". The current solution is not capable of adding new manager nodes with automated resource expansion. It is essential to come up with a logic to increase the number of manager nodes to maintain the quorum of managers [41].
3. Enabling user authentication for managing the AppDock cluster – Upon deployment of the AppDock cluster, anybody can access the cluster-admin module within the network on port 3000. This behavior is not desirable. We have identified that any user authentication mechanism should be incorporated into the user interaction points of the application. User interaction points are CLI that each developer is installing on their development environments and Cluster Admin web interface. CLI application is already using a local storage file to store user data.
4. Incorporating monitoring and metering features – The current implementation of the AppDock platform lacks resource monitoring dashboards and usage metering features. Resource monitoring dashboards can be integrated with the existing Cluster Admin web interface. Resource utilization real-time data is already available in the LogDB. Usage metering will be a complicated task to accomplish where new core components might be needed, with proper usage listeners added into it.
5. Threshold values for individual service – It will be more successful if individual service can have minimum/maximum resource utilization threshold values, as infrastructure resources can be allocated more effectively among services.
6. Testing the AppDock with a mix of workloads – As our performance evaluation did not cover a mix of CPU intensive and memory-intensive workloads in a real-world application with multiple services, it is worth evaluating the suitability of the AppDock platform for such real-world extreme workloads to prove the suitability of the platform for any workload.
7. Dynamically threshold value estimation – Following ad hoc heuristics to determine threshold values does not guarantee the resource allocation meeting expected QoS and the SLA while efficiently using the cloud resources. A dynamic threshold value estimation method could be introduced by considering the above factors.

REFERENCES

[1] P. Mell and T. Grance, “The NIST definition of cloud computing,” National Institute of Standards and Technology, September 2011. [Online]. Available: http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf. Accessed: Dec. 31, 2019.

[2] Amazon Web Services, “Deploying applications to AWS Elastic Beanstalk environments,” [Online]. Available: https://docs.aws.amazon.com /elasticbeanstalk/latest/dg/using-features.deploy-existing-version.html. (accessed Sep. 7, 2019).

[3] Red Hat, Inc., “OpenShift deployments,” [Online]. Available: https://docs.openshift.com/enterprise/3.2/dev_guide/deployments.html. (accessed Sep. 07, 2019].

[4] C. Pahl, “Containerization and the PaaS cloud,” *IEEE Cloud Computing,* vol. 2, pp. 24-31, May-June 2015.

[5] A. Tosatto, P. Ruiu and A. Attanasio, “Container-based orchestration in cloud:state of the art and challenges,” in *2015 9th Int. Conf. on Complex, Intelligent, and Software Intensive Systems*, Blumenau, 2015.

[6] Openshift Origin, “Openshift Origin documentation,” RedHat, [Online]. Available: https://docs.openshift.org/latest/minishift/getting-started/index.html. (accessed Jun. 1, 2019).

[7] Cloud Foundry Foundation, “Cloud Foundry overview,” Cloud Foundry Foundation, [Online]. Available: http://docs.cloudfoundry.org/concepts/ overview.html. (accessed Jul. 1, 2019).

[8] MarketsandMarkets Research Private Ltd, “Multi-cloud management market,” 2020. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/multi-cloud-management-market-18600020.html. (accessed Feb. 18, 2020).

[9] P. Fretty , “Understanding the benefits of a multi-cloud strategy,” BMC Software, Inc., 15 05 2018. [Online]. Available: https://www.cio.com/article/3273108/cloud-computing/understanding-the-benefits-of-a-multi-cloud-strategy.html. (accessed Jul. 30, 2018].

[10] M. Toivonen, “Cloud provider interoperability and customer lock-in,” in *Univ. of Helsinki,Cloud-Based Software Eng.*, Helsinki, 2013.

[11] Intel Corp., “The case for orchestration of cloud infrastructure,” May 1, 2015. [Online]. Available: https://www.intel.com/content/www/us/en/cloud-computing/cloud-orchestration-for-business-agility-paper.html. (accessed Aug. 8, 2019).

[12] tsuru authors, “Tsuru documentation,” Tsuru, [Online]. Available: https://tsuru.io. (accessed Sep. 07, 2019).

[13] W. Jansen and T. Grance, “Guidelines on security and privacy in public cloud computing,” *NIST Special Publication 800-144,* December 2011.

[14] vmware, “Virtualization,” VMware, Inc, 2018. [Online]. Available: https://www.vmware.com/solutions/virtualization.html. (accessed 01 01 2019).

[15] M. Saleem, S. and B. G. Shah, “Cloud computing virtualization,” Int. *Journal of Computer Applications Technology and Research,* vol. 6, no. 7, pp. 290-292, 2017.

[16] D. Firesmith, “Virtualization via containers,” Software Engineering Institute, Carnegie Mellon Univ., 25 Sep. 2017. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-containers.html. Accessed: Mar. 1, 2019.

[17] R. Wadsworth, “Beyond Docker: Other types of containers,” Contino, 11 2016. [Online]. Available: https://www.contino.io/insights/beyond-docker-other-types-of-containers. (accessed Apr. 1, 2019).

[18] A. Hawkins, “Container technologies: more than just Docker,” Cloudacademy, 25 08 2016. [Online]. Available: https://cloudacademy.com/blog/container-technologies-more-than-dockers/. (accessed May 1, 2019).

[19] “Docker overview,” Docker Inc, 2017. [Online]. Available: https://docs.docker.com/engine/docker-overview/.

[20] linuxcontainers.org, “LXC introduction,” Canonical Ltd , [Online]. Available: https://linuxcontainers.org/lxc/introduction/. (accessed Oct. 7, 2019].

[21] Docker Inc., “Swarm mode overview,” Docker Inc., 2018. [Online]. Available: https://docs.docker.com/engine/swarm/. (accessed Aug. 1, 2019).

[22] T. Taylor, “The ultimate guide to container registries,” Oracle, 21 04 2017. [Online]. Available: http://blog.wercker.com/ultimate-guide-to-container-registries. (accessed Aug. 19, 2019).

[23] A. J. Ferrer, D. G. Pérez and R. S. González, "Multi-cloud platform-as-a-service model, Functionalities and," in *Procedia Computer Science*, Madrid, 2016.

[24] D. Petcu, "Multi-cloud: Expectations and current approaches," in *Proc. 2013 Int. workshop on Multi-cloud applications and federated clouds*, Prague, 2013.

[25] The Apache Software Foundation, "Apache jclouds," The Apache Software Foundation, [Online]. Available: http://jclouds.apache.org/. (accessed Mar. 2, 2020).

[26] The Apache Software Foundation, "Apache LibCloud," The Apache Software Foundation, [Online]. Available: http://libcloud.apache.org/.(accessed Mar. 2, 2020).

[27] The Apache Software Foundation, "&.Cloud," The Apache Software Foundation, [Online]. Available: https://deltacloud.apache.org/. (accessed Mar. 2, 2020).

[28] RightScale, Inc., "Welcome to RightScale Docs," [Online]. Available: https://docs.rightscale.com/. (accessed Mar. 2, 2020).

[29] R. Mitchell, "The 8 best open-source tools for building microservice apps," TechBeacon, [Online]. Available: https://techbeacon.com/8-best-open-source-tools-building-microservice-apps. (accessed Jul. 31, 2019).

[30] AppScale Systems, Inc, "AppScale," AppScale Systems, Inc, 2018. [Online]. Available: https://www.appscale.com/. (accessed Jul. 1, 2019].

[31] Microsoft Azure, "Azure stack," Microsoft Corporation, [Online]. Available: https://azure.microsoft.com/en-us/overview/azure-stack/. (accessed Jul 1, 2019).

[32] Apache Stratos, "4.1.x about Apache Stratos," Apache, 18 08 2015. [Online]. Available: https://cwiki.apache.org/confluence/display/STRATOS/4.1.x+About+Apache+Stratos. (accessed Jun. 27, 2019).

[33] Apache Stratos, "Apache Stratos 4.1.x architecture," Apache, 15 08 2018. [Online]. Available: https://cwiki.apache.org/confluence/display/STRATOS/4.1.x+Architecture. (accessed Jun. 27, 2019).

[34] Docker, Inc., "Use swarm mode routing mesh," [Online]. Available: https://docs.docker.com/engine/swarm/ingress/. (accessed Jul. 15, 2019].

[35] Docker Inc., “Docker engine API (v1.40),” [Online]. Available: https://docs.docker.com/engine/api/v1.40/#operation/ServiceCreate. (accessed Oct. 23, 2019].

[36] Docker Inc., “Post-installation steps for Linux,” [Online]. Available: https://docs.docker.com/v17.12/install/linux/linux-postinstall/. (accessed Oct. 25, 2019).

[37] Canonical Ltd. , “SettingUpNFSHowTo,” [Online]. Available: https://help.ubuntu.com/community/SettingUpNFSHowTo. (accessed Oct. 10, 2019).

[38] S. Louv-Jansen, *A CPU intensive operation,* Copenhagen, 2019.

[39] J. Carroll, “Container exits with non-zero exit code 137,” Docker Inc., [Online]. Available: https://success.docker.com/article/what-causes-a-container-to-exit-with-code-137. (accessed Dec. 15 2019).

[40] Amazon Web Services, Inc., “AWS Direct Connect,” [Online]. Available: https://aws.amazon.com/directconnect/. (accessed Dec. 16, 2019).

[41] Docker Inc., “Administer and maintain a swarm of Docker engines,” [Online]. Available: https://docs.docker.com/engine/swarm/admin_guide/. (accessed Dec. 16, 2019).

APPENDIX A – AVAILABLE METHODS IN PROXY INTERFACES

Table A-1 Available methods in the AppDock HTTP proxy interface.

Method Name	HTTP Verb	URL	Description
AddNewNodeToSwarm	POST	/api/swarm/node	Adds a new node to the platform
RemoveNodeFromSwarm	DELETE	/api/swarm/node	Remove a node from the platform.
CreateNewNode	POST	/api/cloud/node	Creates a new node in the requested public cloud environment.
RemoveCloudNode	DELETE	/api/cloud/node	Remove the node from the public cloud environment.
Deploy	POST	/api/deploy	Deploys an application to the AppDock cluster.
SaveNodes	POST	/api/nodes	Persist list of nodes in the LogDB.
CreateService	POST	/api/service	Create a service definition and download the plugin content.
UpdateAllServices	POST	/api/service/update	Update all services to be scaled in newly added nodes.
GetAllPlugins	GET	/api/plugin	List all plugins.
DownloadPluginContent	GET	/api/plugin/download	Downloads the selected plugin content.
SaveScalingServiceConfig	POST	/api/scalingservice/config	Save scaling service configurations.
GetScalingServiceConfig	GET	/api/scalingservice/config	Get the scaling service configurations.

Table A-2 Available methods in the Docker HTTP proxy interface.

Method Name	HTTP Verb	URL	Description
JoinSwarm	POST	/swarm/join	Join the node with an existing swarm.
GetTokens	GET	/swarm	Gets join tokens of the Docker swarm.
InitSwarm	POST	/swarm/init	Initializes a new swarm.
CreateSwarm	POST	/services/create	Creates a new Docker service.
GetContainerStat	GET	/containers/<container Id>/stats?stream=false	Gets container runtime statistics.
GetAllTasks	GET	/tasks	Gets all the tasks running in the node.
GetSystemInfo	GET	/info	Gets system information of the node.
LeaveSwarm	POST	/swarm/leave	Leave a node form Docker swarm.
BuildImage	POST	/build	Build a Docker image in the selected node.
InspectService	GET	/services/<service_name>	Inspect a Docker service.
UpdateService	POST	/services/<service_name>/update	Update a Docker service with the given service template.
DeleteService	DELETE	/services/<service_name>	Remove a service.
TagImage	POST	/images/<image_name>/tag	Tags an existing Docker image.
InspectAnImage	GET	/images/<image_name>/json	Inspect an image that exists in the selected node.
ListAllNodes	GET	/nodes	List all the nodes of a Docker swarm.

APPENDIX B – COMMANDS IN APPDOCK CLI

- *CreateClusterCommand – appdock create -c ,--config <configFile>*
  This command initializes the AppDock platform on the provided nodes via a configuration file. Cluster creation involves configuration file validation, initializing Docker Swarm cluster on the first manager node, initializing core services and persisting nodes in the LogDB.
- *AddNodeCommand – appdock addnode -c, --config <config file> | --aws -t <w/m> | --azure -t <w/m> | --addr <node address> [--advaddr <node advertise address>] -t <w/m>*
  This command is used for adding nodes after the AppDock cluster has been initialized. A list of nodes can be added via --*config* option and a configuration file. New nodes in AWS and Azure cloud environments can be added via --*aws* and --*azure* options. A single node in the private cloud environment can be added via --*addr* option. In each option -*t* is passed to indicate whether the node is a worker (w) or a manager (n).
- *CreateServiceCommand – appdock crtsvc*
  Initial service creation is done via this command. Series of 5 inputs will be taken from the user for service name, service description and plugin number to be used, replicas for a node and NFS folder name to be used as the file storage exclusively for the service. Afterward, files can be accessed on the path /mount/<folder name given> within the application. This will extract plugin content from the current working directory. Plugin content alone is sufficient to create an application of intended technology within the AppDock platform. The application is incrementally developed by the developer.
- *DeployCommand – appdock deploy*
  Running this command in the directory containing application source code along with the extracted and modified plugin content suite to the application's need will deploy the application as a service in the AppDock platform. Service will have the number of replicas in each node specified in the service creation phase.

- *ScalingServiceConfigCommand – appdock ssconfig --maxcpu <max cpu> --mincpu <min cpu> --maxmemory <max memory> --minmemory <min memory> --minuptime <minimum node up time>*

  This command is used to alter threshold values used by the Scaling Service core component to deploy new nodes when nodes maximum limits and to remove nodes when they reach minimum limits. Setting *--minuptime* guarantee node is not removed even minimum resource utilization is not met until the specified time is passed.