BUILDING APPLICATIONS FOR LARGE SCALE WITH CONTAINER ORCHESTRATION

Pradeep Pai T, BEL, Jalahalli, Bangalore - 560054

N.1. ABSTRACT

Application that needs to handle large workloads catering for millions of requests requires being highly scalable, available, fault tolerant, easily manageable and secure. Traditional approach in catering for such high workloads involves increasing capacity of individual machines hosting such workloads which can include increasing system RAM, utilizing a better performing CPU or by increasing the storage capacity of the server. This process is well known as vertical scaling. It is common knowledge that there is a practical limitation to the amount to which individual systems can be vertically scaled to cater for such higher workloads. Therefore there is a need to address such a problem through alternate means.

An alternate approach at building systems that can handle such large workloads would be by using horizontal scaling; where-in the actual workload is distributed over multiple machines. Horizontal scaling of applications come with their own set of challenges which include handling application failures in certain machines, machine failures in it-self, load balancing across the machines, mechanism to perform large scale updates, mechanism to scale-out applications on to more machines on need basis and so forth. Addressing such challenges require usage of software infrastructure that can provide for a platform which is configurable and also provides for a means to perform and maintain the horizontally scaled software application eco-system. Container orchestration is one such means that provides a mechanism of achieving the same.

N.2. INDEX TERMS

Containers, orchestration, Container orchestration, Kubernetes, Docker swarm, apache mesos, master-worker configuration, Cloud computing, Resource allocation, virtual machines, predictive scaling, system reliability, auto-scaling mechanism.

N.3. INTRODUCTION

Container orchestration as a concept has gained prominence in recent times. It primarily deals with automating, controlling and provisioning containerized applications on large scale on multiple machines. Container orchestration caters for large workloads by ensuring scalability, availability and ease of maintenance. Container orchestration maintains and controls the complete eco-system of horizontally scaled application.

Three prominent software platforms that provide for container orchestration are as follows
1) Kubernetes
2) Docker Swarm
3) Apache Mesos

While Docker Swarm and Apache Mesos have added a lot of new features in the recent times, Kubernetes has been the clear leader over the years in the choice for container orchestration platform. Kubernetes over the years has grown into a robust container orchestration platform, which is widely accepted and used in horizontally scaling applications. The rest of the paper focuses on features, components and architecture of Kubernetes.

N.4. KUBERNETES

Kubernetes is a Production-Grade Container Orchestration platform which builds upon containerized applications using Container platform like Docker. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes has been in use handling production workloads at Google for over 15 years, and has matured into a production grade system in such time. Kubernetes has been open-sourced and has a very active community that makes significant contributions in developing new features and maintaining Kubernetes

N.4.1. KEY FEATURES OF KUBERNETES

Kubernetes has been developed to handle deployments over potentially infinite scale without necessarily increasing the operations team. Kubernetes has evolved over the years through the learning's which have been obtained in running humungous workloads at Google. Kubernetes over the years has been proven to handle complex workloads with ease. Kubernetes draws its features from the open source community and has wide acceptance in the open industry. Kubernetes setup can be easily established on various kinds of cloud infrastructure which includes public infrastructure, hybrid infrastructure or even on-premises infrastructure, which in turn gives a lot of flexibility in deployment of applications, as the same can effortlessly, can be moved around the various infrastructures of Kubernetes.

Kubernetes has inbuilt algorithms that manage how the workloads are distributed over machines based on the resource requirements of applications. Kubernetes intelligently manages resource requirements of applications and distributes them on hardware which can satisfy the resource requirements, while ensuring the availability parameters of these applications are not violated. Kubernetes ensures optimum utilization of system resources by scheduling workloads in effective manner on machines. Kubernetes has capabilities to perform regular health checkups on running applications to ensure that they are healthy and delivering the configured minimum availability parameters. If containers have been found to be unhealthy Kubernetes performs necessary action to ensure availability by killing and restarting such containers. Kubernetes also monitors node health, if
nodes are to fail; Kubernetes ensures availability by rescheduling all containers running on such nodes onto other healthy nodes. Kubernetes also does not expose the applications until the containers are healthy and in ready state.
Kubernetes can scale the number of instances of applications on nodes automatically by monitoring system resources like the CPU, or on manual inputs issued either through command or by using Kubernetes dashboard. Kubernetes has mechanisms to incrementally deliver updates to applications; Kubernetes incrementally kills older instances replacing them with newer versions of the application without affecting the availability of such applications. If the updates are to fail for any reason Kubernetes gracefully roles-back all changes and re-instates the older version of the application. All such actions are performed in a transparent manner to the consumers of such applications. Kubernetes assigns unique IP addresses to all containers of the application and assigns each such container with an unique DNS name, which can be used by other applications without the need for explicitly knowing the IP addresses of the containers themselves. Kubernetes effectively manages load by distributing incoming requests among all the available healthy instances of the application.

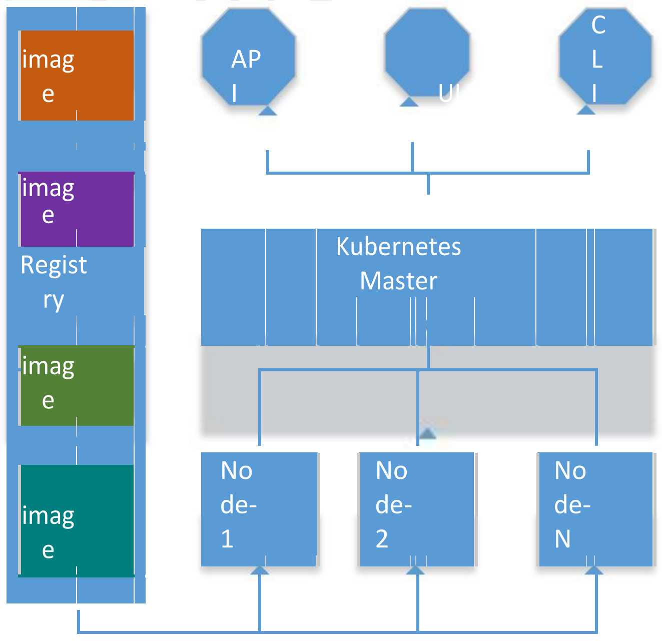

Fig.1. Kubernetes high level architecture

Kubernetes allows for mounting of wide variety of storage options available in the open world. Kubernetes has capabilities to use storage from local storage or from cloud providers such as GCP or AWS or network storage systems such as NFS, iSCSI, Gluster, Ceph, Cinder or Flocker. Kubernetes also provides for batch execution as it can easily manage batch and CI workloads, replacing containers that fail, if desired.

N.4.2. KUBERNETES HIGH LEVEL ARCHITECTURE

Fig. 1 depicts the high level architecture of Kubernetes, which contains two primary components, the Kubernetes master and the worker nodes or more simply nodes. Kubernetes master is responsible for maintaining and controlling the entire Kubernetes infrastructure, while the nodes host the applications. A node may be a VM or physical machine, depending on the cluster. Each node contains the services necessary to run pods and is managed by the master components.

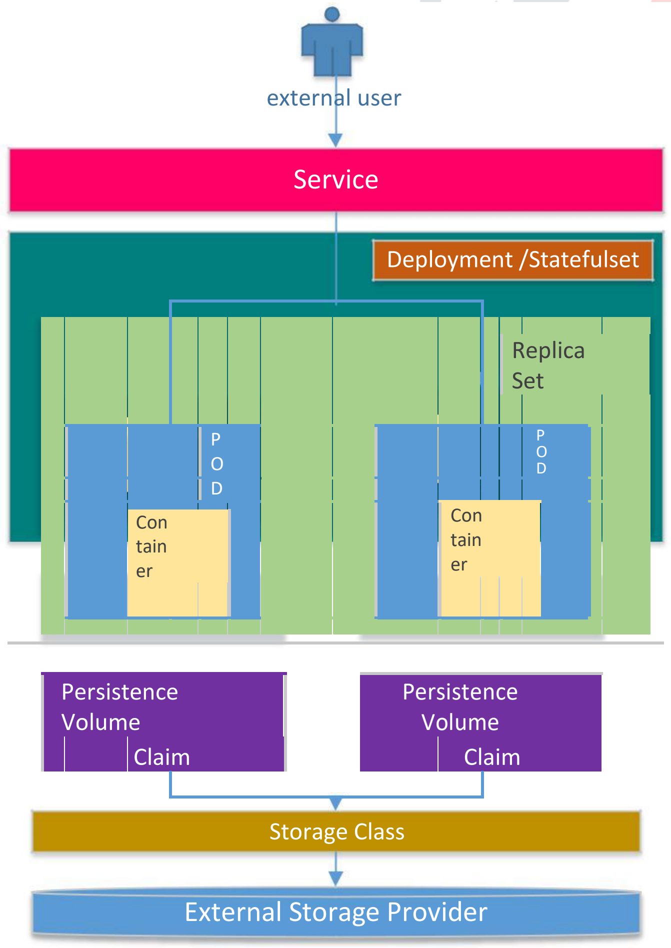

Fig.2. Kubernetes logical components

Containerized application images may be hosted in the registries like the docker local or cloud registries and the same may be configured with Kubernetes. Deployment objects may refer to such containerized application images and their versions with tags for deployment of the applications.

Kubernetes master exposes services which can be read and configured either by using API or User Interface or Command Line Interface.

N.4.3. KUBERNETES LOGICAL COMPONENTS

Fig.2. displays all the various logical components of the Kubernetes infrastructure. Kubernetes effectively manages a set of machines, called nodes (as represented in fig.1.) that run containerized applications. The set of machines is in-itself called a cluster in Kubernetes. A cluster has several worker nodes and at least one master node.

The smallest and simplest entity in Kubernetes is a pod. A Pod is a typically a set of inter-related containers. Kubernetes ensures that all containers of a given Pod are scheduled and run on a single node and all such containers share a common network and storage. A pod is typically self-sufficient in-itself and can be typically considered to be a single instance of the application with all necessary components. All containers of the pod can logically discover each other over the network by localhost, which gives the all the participating containers of the pod as though they are co-located on a single machine. Kubernetes supports different types of container runtimes the prominent of such container runtime that is widely used in the open is Docker container runtime. Containers on the other hand can themselves be considered to be light weight self-sufficient atomic template images of applications. Containers typically contain all necessary dependencies required for their runtime and generally perform a specific/unique functionality. In Kubernetes pods are managed through objects known as Deployments. Deployments are created in Kubernetes by providing YAML files that describe the containers that make up the pod and other related parameters necessary for the pod to perform their functionality. Deployments in Kubernetes are managed by entities known as deployment controllers. Deployment controllers are responsible to ensure that the pods are initiated and are provided with all necessary components necessary for their

run and also that the pods are in the described state as defined in deployment descriptor YAML files. Deployment controllers are primarily responsible in achieving the desired state from the current state.

While deployments are more suitable for workloads that are stateless, the same may not be suitable for workloads that are stateful. Stateful applications are better managed by Kubernetes objects known as StatefulSets. StatefulSets in Kubernetes ensure the pods are rightly ordered and are unique. StatefulSet function in manner very similar to deployments where-in the application pods are specified through YAML files and are managed by controllers known as the StatefulSet controllers. StatefulSet controllers like the deployment controllers ensure that pods are in the described state as defined in deployment descriptor YAML files, and are primarily responsible in achieving the desired state from the current state. StatefulSet controllers additionally ensure that all the pods managed by StatefulSet controller have sticky and unique identity. StatefulSet controller ensures this uniqueness and sticky identity of pods are retained even while it reschedules such pods on different nodes owing to failed health check or any such event which triggers rescheduling.

Kubernetes has capabilities to automatically scale applications based on defined metrics through means of an API resource known as Horizontal Pod Autoscaler. Horizontal Pod Autoscaler or more simply the HPA performs automatic scaling of pods by continuously monitoring the CPU utilization of target pods on a periodic basis. The period at which the HPA performs probe on CPU utilization is configurable and generally the time interval is application dependent. HPA performs both scale up and scale down of pods based on load and detected CPU usage of pods. Although HPA generally performs scaling operation based on CPU usage it can also be configured to perform scaling based on other resource metrics like the memory usage and network load as well. HPA intelligently performs scaling by considering CPU usage statistics across all pods that are active, against the minimum or maximum target values set.

Many of the applications depend on storage and or retrieval of data from disks that are made available to it. When such applications are run in containers on Kubernetes, such dependency on on-disk files creates a problem of loss of data, as Kubernetes may restart or reschedule such containers on different nodes because of health check failures or related issue. On the other hand all running containers of the pod may require sharing of data among each other, which brings in the requirement of shared storage. These issues are solved in Kubernetes by the use of Volume abstraction.

Volumes are nothing but storage devices that can be used to store/retrieve disk data. Kubernetes ensures same volume is made available to all containers of a pod, thereby all containers have access to the same data. Also, as the lifecycle of the volume is linked to the lifecycle of the pods themselves, data in the volumes are persevered and made available to containers, even if they are restarted or rescheduled due to any reason.

Kubernetes allows defining of different kind of volumes that can be used by containers through the definition of StorageClass. Individual StorageClass can define different combination of storage parameters like the provisioner, how the volume shall be reclaimed, mount options and so on. Kubernetes supports various kinds of provisioners like AWS block storage, FC, Azure disk and so on.

Kubernetes further provides another level of abstraction to the administrators through the usage of API resources known as PersistentVolume and PersistentVolumeClaim. PersistentVolume differ from volumes in terms that their lifecycle is independent of the pods themselves and they can effectively outlive the lifecycle of the pods. Pods that require usage of such volume storage make a request to Kubernetes through claims known as PersistentVolumeClaims. The type of volume claim request made by a pod essentially also involves the parameter of the desired storage class and also how such volume shall be mounted. Volumes can be mounted in read only, read/write mode as the need may be.

Applications running in containers that make up a pod are exposed for external usage through abstractions known as Service. As many instances of application may be running on Kubernetes based on the requirement of scaling, the request from external user is transparently handled without the user getting to know which pod processed the incoming request through the abstraction of Service. Applications/pods are exposed through IPs and specific ports through the definition of a service. External users access the processing of the backend applications only through the IP address and ports exposed through this abstraction of Service.

N.4.4. KUBERNETES MASTER ARCHITECTURE

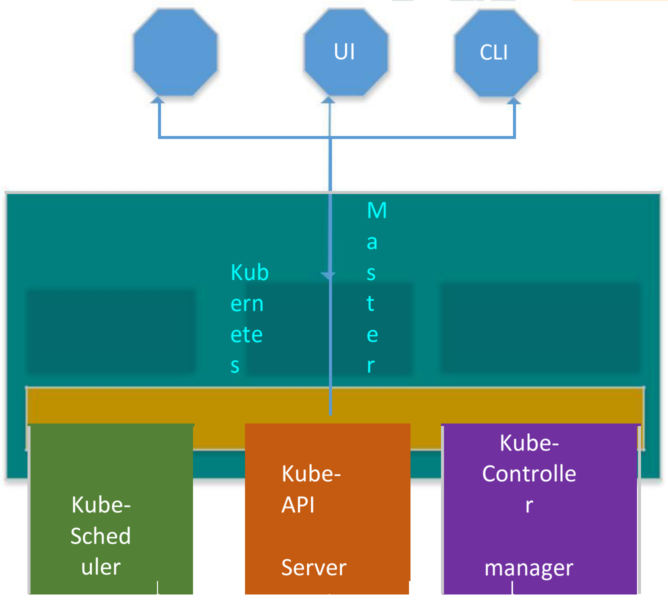

Fig. 3. Kubernetes master architecture

Fig. 3 depicts the architecture of Kubernetes master. The Kubernetes master acts as a primary node that hosts the api server, scheduler and the controller and acts as the primary interface for deployment and management of applications.

Kube-apiserver component on the master acts as the primary interface between the Kubernetes master and the Kubernetes nodes, all communication directed towards the master from the Kubernetes nodes/ pods is handled by the Kube-apiserver. Kube-apiserver has capability to scale horizontally by design.

Kube-scheduler component on the master plays a major role in watching for any freshly created nodes and is primarily responsible in ensuring that the pods are scheduled on nodes that meet the constraints specified in the YAML file descriptor of the pod. Kube-scheduler takes into consideration like the resource requirement specified in the descriptor and tries to find a node in the cluster that has such resource available and schedules the pod to run on such node. Kube-scheduler also considers the affinity factors in scheduling the pods, ensuring that pods are scheduled on nodes labeled with affinity label avoiding nodes labeled in anit-affinity labels.

Kube-controller-manager is the component on the master that is responsible to create and manage controllers. Controllers are like watchdogs that run continuously and monitor the current state of the cluster runs controllers. There are three prominent controllers namely Node controller, Replication Controller, Endpoints Controller and the Service Account & Token Controller. Each controller makes is made aware of the desired state if the cluster and these controllers are primarily responsible in the transition of the cluster state from current to the desired state. Although these controllers run independently as separate processes, for ease of management they are generally bundled together.

The Node controller continuously monitors the health of every individual node of the cluster. The interval at which such health check probe is made is configurable. If the node controller determines any node to be unhealthy, it initiates actions to address loss of functionality due to outage on node. The Replication Controller is made available the number of containers that are to be available at any given point in time through the pod spec, hence this controller continuously monitors the number of replicas of the containers/pods that are available and also ensures that the descried numbers of pods/containers are created and are running in a healthy state. The Endpoints Controller is primarily responsible in joining services with the pods in accordance with the Endpoints object. The Service Account & Token Controllers are primarily responsible in ensuring authentication and authorization of various components of the cluster. The Service Account & Token Controllers ensures the availability of default accounts and API access tokens for all namespaces.

All these components on the master make use of a consistent highly-available key value store known as Etcd. Etcd is used as Kubernetes' backing store for all cluster data. It is advisable to maintain a backup plan for etcd's data for the Kubernetes cluster.

N.4.5. KUBERNETES NODE ARCHITECTURE

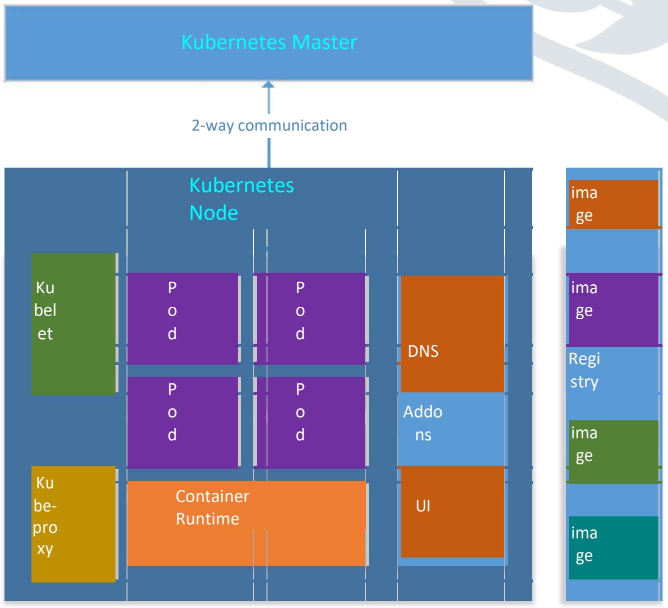

Fig. 4. Kubernetes node architecture

Fig.4. depicts the architecture of Kubernetes node. Kubernetes Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment. Major components that make up a node include Kubelet, Kube-proxy, container runtime and any add-ons, as necessary.

Container Runtime is the software that is responsible for running containers. Kubernetes supports several runtimes: Docker, rkt, runc and any OCI runtime-spec implementation. The most prominent container runtime used in the open industry is docker container runtime and hence most of the container specs in Kuberenetes find great similarity in the docker world.

Kubelet is the primary agent available on all of the Kubernetes nodes and is responsible in ensuring containers provided in the descriptor YAML files of the pod are running on the node. The Kubelet interacts with the container runtime available on the node and manages containers through the container runtime. Container runtime on the node may host other containers apart from the the ones managed by the kubelet. Kube-proxy, as the name indicates acts as a proxy for communications between the node and the master. It is primarily responsible in enforcing network rules and ensuring the communication reaches the right components on the node and vice-versa.

In Kubernetes Add-ons are themselves deployed as pods and Services. These add-ons implement certain cluster features of which some may be considered to be very essential and others to enhance ease of use. The Add-on pods have a lifecycle similar to that of any normal pod residing on Kubernetes and are hence generally managed by Deployments, Replication Controllers.

Some of the prominent add-ons are DNS add-on and the Web UI dashboard add-on. While other add-ons are not strictly required, all Kubernetes clusters should have cluster DNS. Cluster DNS is essentially a DNS server, which enables search through DNS names and

generally many deployments rely on availability of such DNS
access. Kubernetes service also relies on the availability of such DNS services. Containers started by Kubernetes automatically include this DNS server in their DNS searches. The Web UI Dashboard is an aesthetic addition to the Kubernetes environment and provides for a web based UI, which can be used to monitor the Kubernetes cluster. The Dashboard also allows for creation/deletion/modification of most of the Kubernetes objects.

N.4.6. KUBERNETES COMMUNICATION

As depicted in Fig.4 Kubernetes works by two way communications between the master and the cluster of nodes.

N.4.6.1. CLUSTER TO MASTER COMMUNICATION

Apiserver component of the master plays a major role in communication between the cluster and the master as all communication originating from the cluster end at the apiserver. None of the other components are designed to handle communication. In order to ensure maximum security all communication between cluster and master are encrypted over the network by the use of HTTPS and also clients are required to authenticate themselves with the apiserver in order for them to initiate communication with the Apiserver. Apiserver also has functionality to enable authorization checks and it is advisable to enable the same for maximum security.

Nodes are generally provisioned with X.509 certificates and also client credentials are provisioned. Nodes in this scenario are required to produce their certificated along with client credentials in order to authenticate themselves and initiate a communication channel with the master.

Pods that need to communicate with the Kubernetes masters apiserver perform such communication securely by leveraging a service account so that Kubernetes will automatically inject the public root certificate and a valid bearer token into the pod when it is instantiated. The Kubernetes service in all namespaces is configured with a virtual IP address that is redirected via the kube-proxy to the HTTPS endpoint on the apiserver.

N.4.6.2. MASTER TO CLUSTER COMMUNICATION

There are two primary communication paths from the master (apiserver) to the cluster. The first is from the apiserver to the kubelet process which runs on each node in the cluster.

The second is from the apiserver to any node, pod, or service through the apiserver's proxy functionality. The connections from the apiserver to the kubelet are used for fetching logs of pods, attaching (through kubectl) to running pods and for providing the kubelet's port-forwarding functionality. These connections terminate at the kubelet's HTTPS endpoint. By default, the apiserver does not verify the kubelet's serving certificate, which makes the connection subject to man-in-the-middle attacks, and unsafe to run over untrusted and/or public networks. The same can be overcome in a manner similar to cluster to master communication, by the use of certificates.

N.4.7. HIGH-AVAILABILITY KUBERNETES MASTERS

In Kubernetes, most of the interaction and administration is done through the Kubernetes master. Therefore a cluster with single instance of Kubernetes master can be visualized to be vulnerable to single point of failure. To overcome the same, Kubernetes provides mechanisms for deploying multiple masters in different zones, where-in the load between the masters can be load balanced through an external load balancer and the data is synchronized among all the masters by ensuring deployment of clustered etcd.

It is recommended to deploy a master replica containing at least three masters in three different zones to ensure high-availability.

N.5. CONCLUSION

Kubernetes container orchestration platform provides for building robust production grade highly scalable systems by using techniques to scale applications horizontally across multiple machines. Kubernetes is widely used in the industry for scaling of applications. Kubernetes is open source and largely community driven and the platform can also be established on an on-premises cloud.

Kubernetes application configurations necessary for deployment of applications on the Kubernetes platform is flexible - as the same configuration can be seamlessly used on any Kubernetes platform which could be an on-premises cloud or SAAS exposed by cloud providers like Google or Amazon, thus providing for abundance of deployment options.

N.6. ACKNOWLEDGMENT

I would like to express my gratitude to Ms. Durga G K (AGM), Mr. Kodandarama K (AGM), Mr. Paraneetharan S (Sr. DGM) and Ms. Shashikala K L (DGM) from C-D&E Software, BEL for providing sufficient infrastructure and support for this work.

N.7. REFERENCES

[1] Kubernetes, October 2018, [online] Available: https://www.kubernetes.io/.

[2] Docker, October 2018, [online] Available: https://www.docker.com/.

[3] P E. Casalicchio, L. Silvestri, "Architectures for autonomic service management in cloud-based systems", Computers and Communications (ISCC) 2011 IEEE Symposium on, pp. 161-166, June 2011.

[4] Docker Swarm, October 2018, [online] Available: https://docs.docker.com/engine/swarm/.

[5] Apache Mesos , October 2018, [online] Available: http://mesos.apache.org/.

[6] Edureka blog on Kubernetes architecture [online] Available: https://www.edureka.co/blog/kubernetes-architecture/

[7] Kublr blog on Kubernetes architecture [online] Available: https://kublr.com/blog/under-the-hood-an-introduction-to-kubernetes-architecture/

[8] Tothenew blog blog on Kubernetes architecture and setup [online] Available: http://www.tothenew.com/blog/understanding-kubernetes-architecture-and-setting-up-a-cluster-on-ubuntu/

[9] Aquasec blog on Kubernetes architecture [online] Available: https://www.aquasec.com/wiki/display/containe rs/Kubernetes+Architecture+101

[10] Wikipedia information on Kubernetes[online] Available: https://en.wikipedia.org/wiki/Kubernetes

[11] Github Kubernetes repository [online] Available: https://github.com/kubernetes/kubernetes