QATAR UNIVERSITY

COLLEGE OF ENGINEERING

# QUEUEING THEORY BASED KUBERNETES AUTOSCALER

BY

KHALED NABIL KADOURA

A Project Submitted to

the Faculty of the College of

Engineering

in Partial Fulfillment

of the Requirements

for the Degree of

Masters of Science in Computing

January 2018

© 2018 Khaled Nabil Kadoura. All Rights Reserved.

# COMMITTEE PAGE

The members of the Committee approve the Project of Khaled Nabil Kadoura defended on 27/12/2017.

______________________________

Dr. Abdelkarim Erradi
Thesis/Dissertation Supervisor

______________________________

Dr. Qutaibah Malluhi
Committee Member

______________________________

Dr. Osama Halabi
Committee Member

______________________________

Dr. Khaled Khan
Committee Member

# Abstract

KADOURA, KHALED, K., Masters : January : 2018, Masters of Science in Computing

Title:_QUEUEING THEORY BASED KUBERNETES AUTOSCALER

Supervisor of Project: Abdelkarim, Erradi.

The microservices architecture is emerging as a new architectural style for designing and developing applications by composing loosely coupled services that exchange standard messages using standard interfaces and protocols. Docker provides a platform to automate microservices deployment into isolated containers. Kubernetes automates the deployment, scaling and management of Docker containers. Unlike current virtual machines (VM) based deployment, containerization allows more effective scaling of resources to meet the requirements of varying workloads. Benefiting from the research advances in VMs consolidation, placement and auto-scaling approaches, as well as the queueing theory, our work provides a custom queueing theory based auto-scaler for Kubernetes, which dynamically make vertical and horizontal scaling decisions. The auto-scaler goal is to achieve the desired Quality of Service (QoS) while optimizing the cloud resources usage.

Keywords- Microservices; Docker; Container; Kubernetes; Auto-scaling; Queueing Theory

# Table of Contents

List of Figures ..... vii

List of Tables ..... viii

List of Templates ..... viii

List of Algorithms ..... viii

Introduction ..... 1

Problem Statement ..... 4

Contributions ..... 4

Background ..... 5

Docker ..... 5

Kubernetes ..... 7

Pods ..... 8

Deployment ..... 10

Services ..... 11

Components and Architecture ..... 12

Cluster Monitoring ..... 13

Horizontal Pod Auto-Scaler ..... 15

Kubernetes API ..... 17

Related Work ..... 21

Kubernetes Testbed ..... 23

Testbed Architecture ..... 23

Testbed Deployment ........ 25
Kubernetes Evaluation ........ 26
Results Discussion ........ 27
Problem Formulation ........ 34
Queuing Theory ........ 34
Problem Definition ........ 36
Virtual Machines Auto-Scaling ........ 39
Pods Auto-Scaling ........ 40
Solution Architecture ........ 41
QAS Internal Components ........ 43
QAS External Components ........ 44
Solution Implementation ........ 44
Pod Configuration ........ 45
Heapster and InfluxDB ........ 45
Reverse Proxy Agent ........ 46
Heapster Metrics Agent ........ 47
Decision Maker and Communicator ........ 48
Experiments and Evaluation ........ 51
Evaluation Setup ........ 51
Evaluation ........ 52

Conclusion and Future Work .............................................................................. 54

References ........................................................................................................ 55

## List of Figures

*Figure 1* Virtual Machine vs Containers virtualization ..................................................... 3

*Figure* 2 Container-based cluster architecture ..................................................... 6

*Figure* 3 Sidecar, Ambassador, and Adapter containers in a pod ..................................... 8

*Figure* 4 Kubernets architecture diagram. ..................................................... 13

*Figure* 5 Heapster Monitoring within Kubernetes ..................................................... 15

*Figure* 6 Kubernets Horizontal Pod Auto-Scaler ..................................................... 16

*Figure 7* AcmeAir basic Architecture ..................................................... 24

*Figure 8* Response Time for different deployments ..................................................... 28

*Figure 9* CPU utilization of Nodes running single application Pod ................................ 29

*Figure 10* CPU usage of Pods ..................................................... 30

*Figure 11* Response time for optimal Pod size ..................................................... 30

*Figure 12* High resource availability cluster at 150 and 180 request threads .................. 31

*Figure 13* Cumulative CPU usage in millicores by frontend application ......................... 32

*Figure 14* Network throughput for 40, 100, 150 and 180 request threads ....................... 33

*Figure* 15 Response time as a function of utilization ..................................................... 36

*Figure 16* Simple Multi-Client to Multi-Server Architecture ........................................ 37

*Figure 17* Queueing Theory Based-Auto Scaler Low Level Architecture ....................... 41

*Figure 18* Queueing Theory Based-Auto Scaler High Level Architecture ...................... 42

*Figure 19* Response time comparison between HPA and QAS ...................................... 52

*Figure 20* Number of Pods started by different auto scalers ......................................... 53

## List of Tables

*Table 1* Brief list of Kubernets Commands .......................................................... 18

*Table 2* List of HTTP verbs available on Kubernetes Resful API.................................... 20

## List of Templates

*Template 1* Kubernetes Deployment descriptor file ........................................................ 11

## List of Algorithms

*Algorithm 1* Reverse Proxy Agent ........................................................................ 47

*Algorithm 2* Heapster Metrics Agent .................................................................... 48

*Algorithm 3* Pods Migration.................................................................................. 49

*Algorithm 4* Scale Up Algorithm ........................................................................... 50

# Introduction

Cloud computing is the product of rapid development trend that is accompanied by advancements in storage, networking and processing power. Cloud providers are able to lease resources such as CPU, and storage in on demand fashion. Virtualization forms the foundation of the cloud computing by providing virtualized resources that can be dynamically acquired and releasing on demand [1].

The benefits of virtualization lie in the ability to dynamically map physical resources to virtual applications, this allows multi-tenancy of virtual applications within a single physical machine. Such consolidation reduces the operational and managerial costs on cloud providers and leasing prices on cloud consumers [2]. One type of virtualization that is common today on the cloud is Virtual Machines (VM), VMs runs hardware level virtualization [2] where every VM acts as a Guest OS. A system hypervisor (running either bare-metal or a part of an OS) allows multiple Guest OS to run a single physical machine, by virtualizing hardware for each VM, VMs are able to run independently and in isolation [3]. While VMs have long proved to be successful in optimizing the physical resources, in practical cases the workload required from a VM cluster fluctuate, leading to either under-provisioning or over-provisioning [4]. To accommodate such fluctuations in demand, different cloud providers such as Amazon Web Services AWS provide a cluster auto-scaler (e.g. Amazon Auto-Scaling AAS), which is a reactive auto-scaler that adjusts VM count according to current workload demands. While auto-scaling models do improve system availability, and optimizes cost, such models do not adequately address several challenges. In many cases, cloud users would experience variable traffic patterns, rapid demand spikes and outages with corresponding retry storms, such behavior is a burden on the Quality of

Service QoS, especially as initiating and running new instances of VM is a heavy task and would requires several minutes, according to a Netflix study, new instances require 10-45 minutes to run [5]. Many researches and solutions suggest predictive and reactive-predictive models to account for such fluctuations, such as Scryer [5] and Elastisys which provides algorithms for recurring workloads, irregular workloads and reactive provisioning [6], while such models lessens the burden, workload in different applications can be unpredictable, thus QoS would still suffer from the startup delay of VM instances.

In addition to the delay caused by VMs startup time, Virtual Machines suffer greatly from an overhead as it must run a complete copy of an OS, this overhead, which in turn affects the startup time, degrades performance as machine instructions has to be translated for the VM to the Host OS, and require a relatively a big storage space. The isolation between different VMs within the same physical machine means that inter-VMs communication is feasible through networking only (e.g. Ethernet devices) [3].

The challenges and limitations of Virtual Machines has paved a way for further virtualization research areas. Operating system (OS) level virtualization creates a standard encapsulated OS processes and manages them through the OS kernel [2]. Virtualized OS containers has been gaining popularity recently due to their better performance and much lower overhead in comparison to their VM counterpart. Containers are meant to deliver a level of security and isolation similar to VMs while being tightly integrated with the host OS, such integration eliminates the need for hardware emulation, which enhances performance [3]. As shown in *Figure 1*(b), OS virtualization, namely container virtualization, greatly reduces overhead by eliminating the need to run multiple OS and creating virtual hardware.

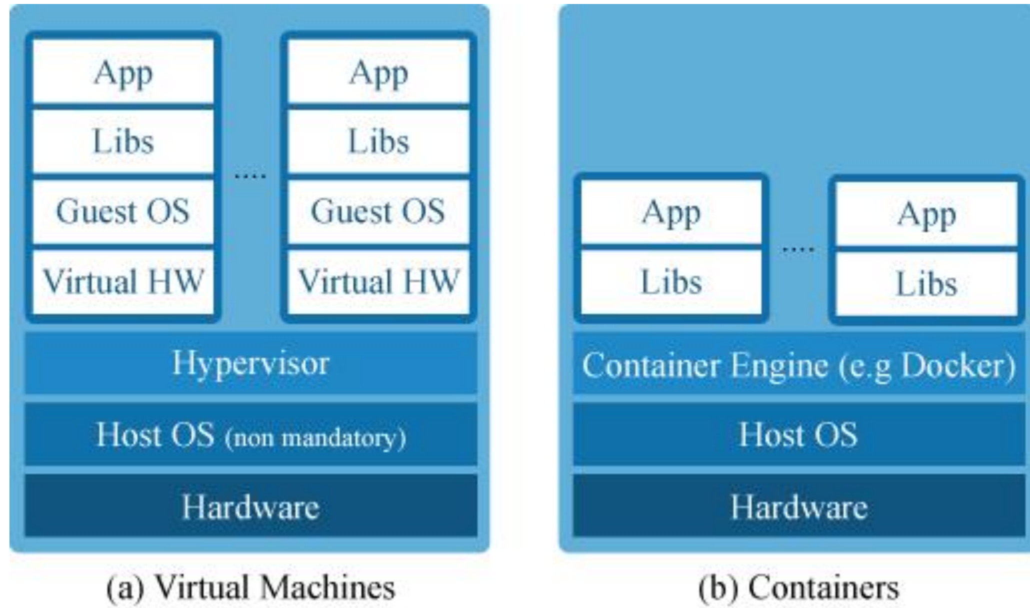


*Figure 1* Virtual Machine vs Containers virtualization

The promising opportunities of OS virtualization brings a whole new set of challenges. Due to the similarity between the two types of virtualization, some hardware virtualization solutions can be applicable on OS virtualization.

The reminder of the paper is organized as follows. In section 2 we provide an overview of Docker and its containerization technology and describe Kubernets and its components in depth. In section 3 we briefly introduce related work in VM consolidation and auto scaling approaches. In section 4 and 5 we implement a testbed on Kubernets which is used to evaluate Kubernetes system for a better understanding of its behavior. Later in section 6, we introduce queueing theory, define the paper's problem and how our solution is applied. Lastly, sections 7 and 8 the solution is evaluated and compared with Kubernets native auto scaler, then we briefly conclude this paper.

## Problem Statement

The varying workload on cloud hosted services is a crucial to tackle issue in order to maintain a reliable Quality of Service QoS that meets clients' Service Level Agreements SLA. The emergence of containerization platforms, poses new research areas including containers' auto scaling. Using Virtual Machine Auto Scaling techniques, we investigate if such models are applicable to Container-based deployments.

## Contributions

The contributions of this paper are 1) present two open-source related evolving technologies, Dockers and Kubernetes, 2) present a Kubernets Auto-Scaler that, based on Queueing Theory dynamically scales a cluster vertically and horizontally, 3) Through evaluation of different versions of the Auto-Scaler, present a thorough discussion and possible future work.

## Background

## Docker

In OS virtualization, it's the OS kernel's responsibility to implement the container abstraction and allocate CPU, memory and network shares. Such shares follow allocation strategies similar to hardware virtualization, such as dedicated, shared and best effort. The containers reply rely on OS kernel for service instead of their own, in some cases a different OS kernel may be emulated to processes in a container in order to support backward compatibility or support different OS APIs [2].

Linux provides a lightweight Linux Containers (LXC) implementation offering an operating-system-level virtualization method for running multiple isolated Linux containers on a host using a single Linux kernel. Such implementation runs a single process inside each container that is assigned a unique PID. It uses Linux kernel cgroups functionality for limiting and prioritizing resources (CPU, memory, I/O, network) allocated to a container. Additionally, LXC uses Linux namespace isolation functionality to provide complete isolation of a container's view of the operating environment, including process trees, networking, user IDs and mounted file systems [3]. This enables providing a private IP address for each container and ensures resource isolation. While LXC provides a container environment that runs at native speed through a lightweight implementation, it is limited to Linux environments, and sufferers in terms of secure containment environment [3].

Docker is a daemon that manages Linux containers as self-contained images [3] and provides deployment services. By extending LXC, Docker provides an application-level unified API that runs processes in isolation [7]. To run processes in isolation, Docker

utilizes LXC namespaces concept [3], which uses a container specific user namespace to ensure that host root privileges are not permitted to a container's root user. Furthermore, using LXC cgroups, Docker limits a container's resource pool and monitors it [8]. A Docker container can be saved and created using a base image, such image can contain a prebuilt application or just OS fundamentals [7]. To speed up the deployment of new Docker images, simple text files (Dockerfiles) containing build commands are created to automate the build process [8].

Dockerfiles has many advantages in comparison to Virtual Machine images in terms of reproducibility, as a Dockerfile is a tiny text file, in comparison to a huge VM image, making transferring a Docker image build file more feasible. Dockerfiles are well documented, providing both instructions and human readable summary of software dependencies thus minimizing build errors. Version management systems (git and subversion) can easily exploit the benefits Dockerfiles by tracking changes and pushing versions [9].

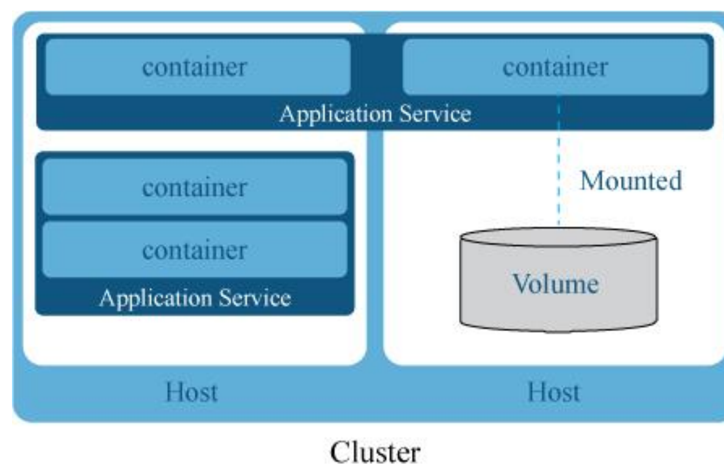


*Figure* 2 Container-based cluster architecture

As shown in *Figure* 2, the interoperability of containers facilitates running applications over a cluster of container hosts. Such architecture could consist of a combination of several physical bare-metal and virtual machine host servers. A host, running several application containers, would also run common services such as load balancing and scheduling. Such application containers could run on different hosts for scaling purposes, where a logical grouping of Application services would allow the aforementioned scaling capability. Different hosts may mount different containers to volumes for data persistence, such volumes persist data post container termination [10]. Such container cluster management are beyond the capabilities of Docker, a cluster management solution should have the ability to deploy distributed applications automatically and enable a management API to manage container life cycles [10].

## Kubernetes

Kubernetes is an open-source cluster management platform for Docker containers [11]. Kubernetes enables Docker images deployment, scheduling, and containers management across machine clusters. Networking is one of the most important aspect of Kubernetes as it allows the discovery, communication and synchronization of containers in the cluster [11]. Kubernetes Control Plane consists of two types of nodes that run on a cluster, master and worker nodes [12]. A cluster is a collection of physical or virtual machines defined as nodes, within a cluster there exists typically a single master node and zero or more worker nodes; each node contains several Kubernetes pods [13]. A worker node runs services essential to run Pods, such services are managed by the master component. Node services include Docker, Kubelet, and Kube-proxy. The role of Docker service is to run the docker

images inside the pods, while Kubelet service handles the node's communication with the master node, finally, a Kube-proxy service responsible for the networking services, which creates a virtual which clients can access [12].

## Pods

Docker containers run inside a Pod, which is the smallest deployable unit that Kubernetes creates and manages [14]. While the rule is not enforced, typically coexisting application containers within a Pod should serve different applications (two completely different application, or different services for the same application), to allow efficient scaling of cluster [13]. However, the golden rule is, only place tightly coupled application containers within the same Pod [11].

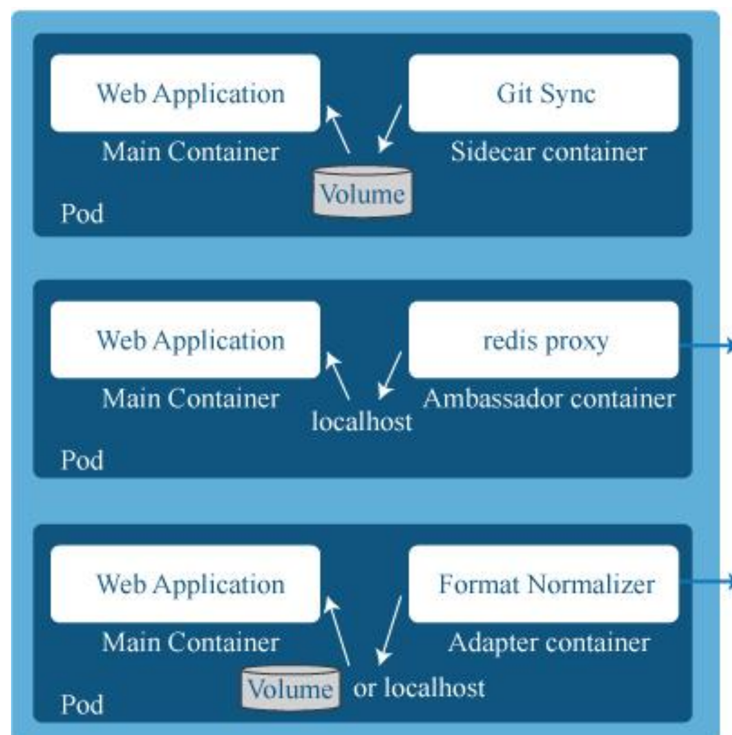


*Figure* 3 Sidecar, Ambassador, and Adapter containers in a pod

Grouping multiple containers in a single pod can be an advantage if done correctly, as containers would utilize the local communication using localhost within the same Pod; the decision-making can few general patterns of modular application development. Burns [15] identified three different patterns as shown in *Figure* 3: Sidecar containers, Ambassador Containers, and Adapter containers. Sidecar containers are deployed to enhance the main container, such as a web server container extended by container that synchronizes the file system with a git repository. Ambassador containers proxy local connection publicly, and acts as a separation of concern by abstracting the connection of the main application to a single localhost connection, while the ambassador takes control of exposing the requests publicly. As an example, the ambassador would handle sending the request to appropriate read or write servers in a cluster. Adapter containers enables an abstraction of the main application for normalizing and standardizing the output. An example use-case would be a monitoring application that would expect a standardized input of data from different servers.

Docker containers within a pod would typically run a single microservice. The motivation behind the single microservice per container concept is to establish transparency that enables Kubernetes to provide container services such as process management and resource monitoring. Moreover, such architecture would efficiently decouple software dependencies allowing single containers to be updated and rebuilt independently [12].

Pods follows similar concept to Docker to provide isolation, which is through Linux namespaces, and cgroups. Within a pod, containers may communicate through standard inter-process communications. However, containers in different pods run on distinct IP addresses [12]. Pods provide a high-level abstraction for a group of containers that share

volumes and network namespace, at which the scheduling and replication are performed in Kubernetes rather than at the level of individual containers. Thus, it is essential to carefully group only tightly coupled containers within a pod [13].

## Deployment

In Kubernetes, container images are deployed into Pods using Deployments. A Deployment is a template that provides declarative description for Pods, and Replica Sets. Once created, Kubernetes master node schedules Pod replicas into worker nodes based on the template specifications. A Deployment controller manages, and monitors Deployment objects and assures that related Pods are meeting the desired state, by creating, and deleting Pods or rolling updates [12, 14].

Within the Deployment template, specification of the desired behavior of the Deployment are set in terms of labeling the Pods, setting number of replicas, the Pod description template and other Pod related values. The Pod template defines specification of the desired behavior of the Pod, and its containers.

Below, is a Deployment template creating a Deployment name deployment-example that requests creating three Pods using the container image nginx: 1.10.

*Template 1* Kubernetes Deployment descriptor file

```
01: apiVersion: apps/v1beta1
02: kind: Deployment
03: metadata:
04:   name: deployment-example
05:   spec:
06:   replicas: 3
07:   template:
08:     metadata:
09:   labels:
10:     app: nginx
11:   spec:
12:     containers:
13:     - name: nginx
14:       image: nginx:1.10
```

The replica value defined in the template defines the required number of running pods at a time; a Replication Controller manages the replication level of Pods [13]. By monitoring the Pods, the Replication Controller will kill extra pods, replace failed pods, or start new pods [12]. The Replication Controller provides an interface to manually scale a cluster of Pods easily, or use an auto-scaler to adjust the size of the cluster horizontally.

Pods are vulnerable for failure, failed Pods are deleted and replaced by new Pods by the Replication Controller, and furthermore scaling the cluster creates new pods or delete existing ones.

## Services

In Kubernets, Pods are assigned IP addresses dynamically within the cluster; Services abstracts the access to the pods by logically grouping a set of pods under a single static IP address [12]. A Service provides the external interface for one or more pods, those pods are matched using their label selectors, which are a key-value pair that identify a group or

subset of resources in Kubernets. Such abstraction allows an external client to connect to a Pod using only the Service name and the port an application is exposed at. Internally, a Service directs the request to a Pod in a round-robin fashion or can further be used as a load-balancer [13].

## Components and Architecture

*Figure* 4 demonstrates the high-level architecture of Kubernetes, at the top, a master node (or more if high-availability master is used). Each master node includes three main components, the Controller manager is responsible for cluster level operations such as nodes management, discovery and monitoring, and deployments scaling and updates. The scheduler assigns Pods across the Nodes cluster. Etcd stores the master configuration data and its persistent state. Finally, Kubectl API server provides the REST API for different Kubernets objects [16].

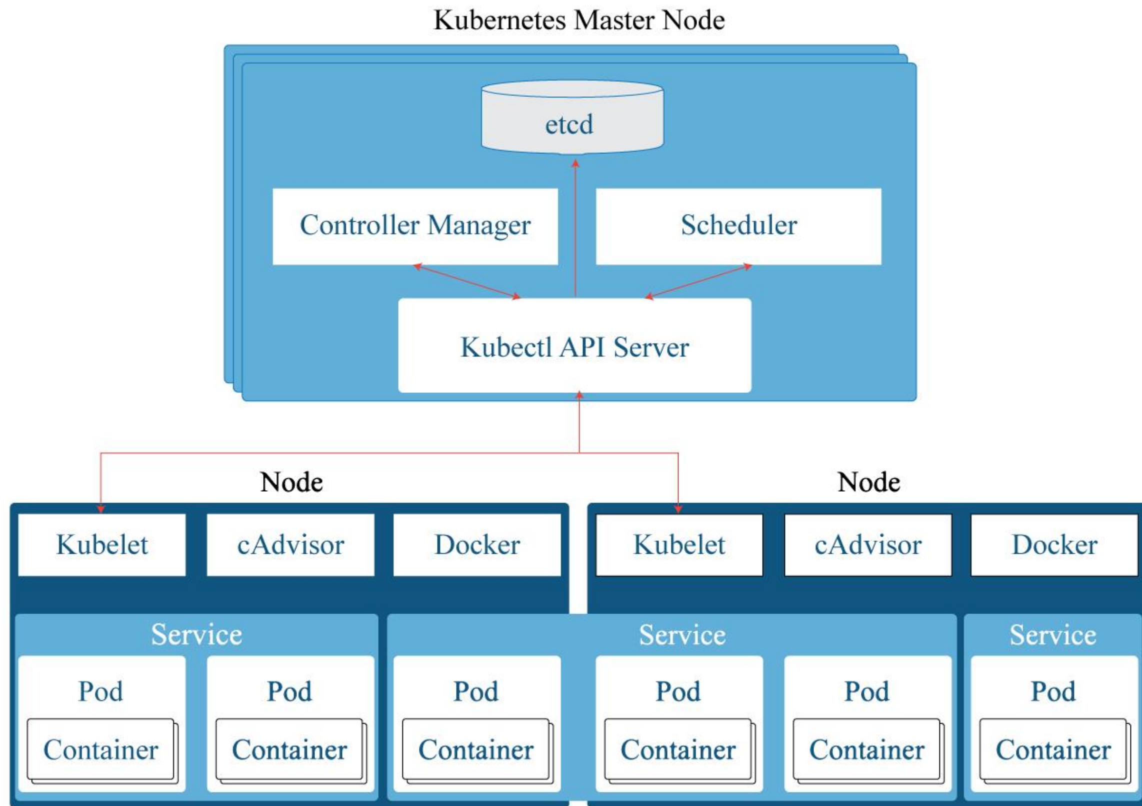


*Figure* 4 Kubernets architecture diagram.

The worker nodes contain a Kubectl agent responsible for interaction with the Master node; inside the nodes, zero or more Pods are located, where each Pod runs one or more containers. Services binds one or more Pods logically by creating an interface that abstracts and distributes client requests though a single internal IP address and optionally an external IP address.

## Cluster Monitoring

In order for applications to provide a reliable quality of service, it is crucial to monitor an applications cluster and make scaling decisions based on the performance metrics. In a

cluster, it is essential to be able to monitor the resource usage at different levels, this provides deep performance insight and assists in discovering potential application bottlenecks; thus, Kubernetes allows resource monitoring at containers, Pods, Services and clusters levels [12].

Heapster is a monitoring metrics and events processing tool that enables container cluster monitoring and performance analysis is integrated within Kubernetes to work with its clusters. Heapster consists of two components: (i) Eventer which records Kubernetes master events. (ii) Heapster core, which reads and records metrics from different nodes in a Kubernets cluster, and provides Heapster metric model through a REST API. The Heapster model allows extraction of historic metrics up to 15 minutes; the metrics are provided at different levels (Cluster, Node, Namespace, Pod, and Container levels) and are updated every 60 seconds by default [17].

In a Kubernets cluster, as shown in *Figure* 5, Heapster runs as a Pod, and discovers all nodes in the cluster to query their usage information through node's specific Kubernetes agent Kubelets. By managing the Pods and containers on a node, Kubelets fetches each containers usage statistics from cAdvisor, aggregates them for each Pod and then exposes them via a REST API [12]. cAdvisor supports Dockers natively, it collects, aggregates, processes and exports resource usage and performance information about each running container [18]. Heapster stores all the different cluster usage and performance information in a configurable backend to allow visualization [17, 12].

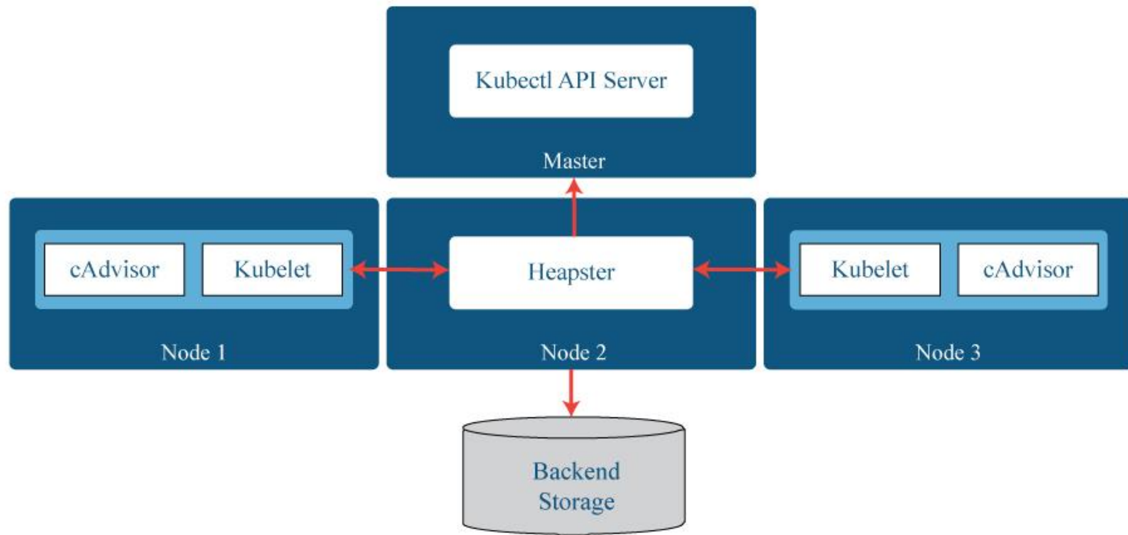


*Figure* 5 Heapster Monitoring within Kubernetes

## Horizontal Pod Auto-Scaler

The variability of a cluster's load emphasizes the importance of scaling up or down the number of Pods of a deployment, and using Heapster's metrics scaling decisions can be made, however user-intervened scaling model is unsuitable for a production-level cluster where high availability is key under a non-predictable load; thus, Kubernets provides a Horizontal Pod Auto-Scaler (HPA) [16].

Kubernetes Horizontal Pod Auto-scaler is implemented as a Kubernetes API resource and a controller. The HPA defines a CPU utilization threshold; by observing Heapster's metrics, the Auto-Scaler is able to communicate scaling decisions through to the Replication Controllers, Deployments or Replica Sets that meets user specific criteria [12], as shown in Figure 6.

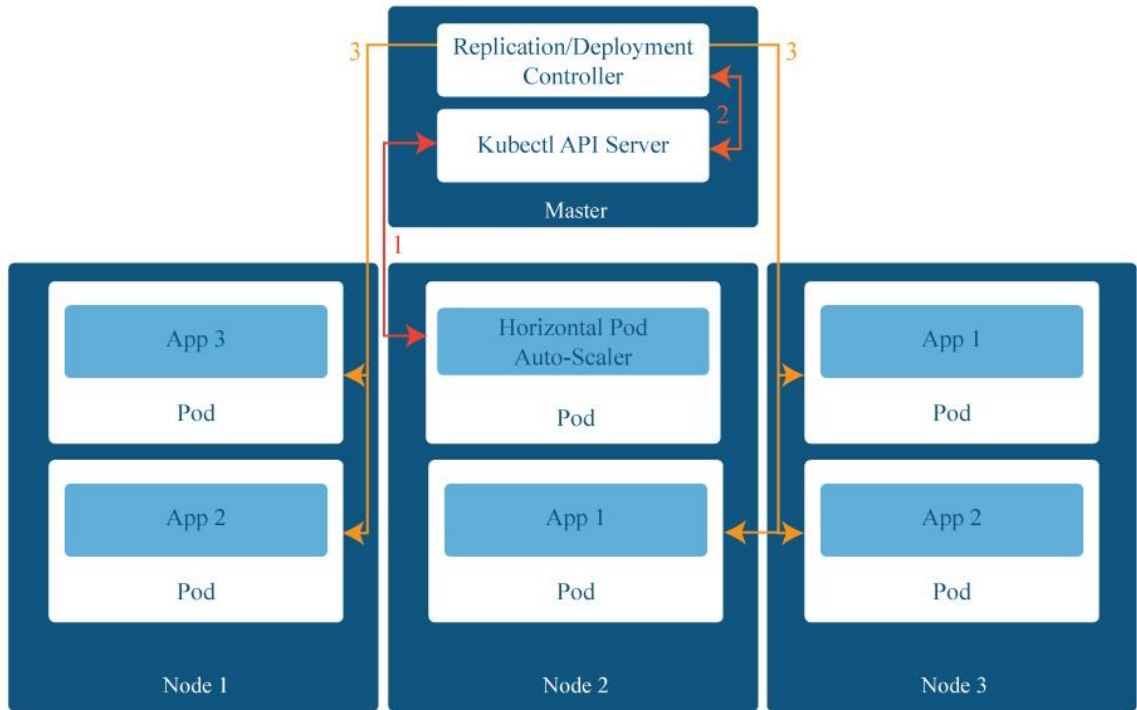


*Figure* 6 Kubernets Horizontal Pod Auto-Scaler

The Auto-Scaler queries Pods periodically in a controlled loop fashion. The query collects per Pod metrics which are later aggregated to calculate the arithmetic mean of CPU utilization, the mean value is then compared to the user defined utilization target, and if needed the cluster is scaled accordingly while preserving the number of replicas within the user defined limits (MinReplicas <= Replicas <= MaxReplicas) [12, 19].

Scaling the cluster often is likely to introduce noise, thus the Auto-Scaler implements an algorithm with predefined values to minimize such noise. By default, the Auto-Scaler queries Pods every 30 seconds for utilization values, the per Pod utilization metric is defined by dividing the last 1 minute of CPU usage by CPU requested. The per Pod CPU, utilization value is then used to calculate the target number of Pods using this formula:

TargetNumOfPods = ceil(sum(perPodCPUUtil) / Target) (1)

Once a scale decision is made the Auto-Scaler would allow a grace period of 3 minutes or 5 minutes before scaling-up or scaling down again, respectively. Moreover, a change in CPU utilization must exceed a 10% relative tolerance ratio before any scaling decision will be made [19].

## Kubernetes API

As illustrated in *Figure* 4, the API server carries out actions to worker nodes and facilitates communication with other master components. Additionally, the API Server in Kubernetes allows end users to manage the cluster by providing two interfaces, RESTful, and Command-Line APIs through *kubectl* agent. Typically, the RESTful API is more suitable for requests through code and remote users, while *kubectl* CLI is appropriate for manual configurations directly on the server.

### Command-Line API

Also known as *kubectl,* is the command-line interface to interact with Kubernetes cluster through terminal. The general syntax of kubectl is.

kubectl [command] [TYPE] [NAME] [flags]

The **command** indicates the operation type required, Kubernets provides a long list of different management operations, below is a subset of the most common operations:

*Table 1* Brief list of Kubernets Commands

| Operation | Description |
|---|---|
| run | Run a specified image |
| create | Create one or more resources from a file |
| delete | Delete resources |
| apply | Apply a configuration change to a resource |
| get | List one or more resources |
| label | Add or update the labels of one or more resources |
| expose | Expose a replication controller, service, or pod as a new Kubernetes service |
| rolling-update | Perform a rolling update by gradually replacing the specified replication controller and its pods |
| scale | Update the size of the replication controller |
| autoscale | Define or set an auto-scaler |
| describe | Display the detailed state of resource(s) |
| logs | Print the logs for a container in a pod |

Further, **TYPE** specifies the resource type to run the command against, some of the more comment resource types in Kubernetes are deployments, nodes, pods, services, jobs, replicasets, and replicationcontrollers, however there are many other resource types. **NAME** attribute specifies the resource target, however, for some operations the name can be omitted, instead the operation is performed on all the resource type, and many names

can be used to select different resources in the same command. Finally, the last part is optional for flags [12].

**REST API**

Kubernetes RESTful API enables managing objects via standard HTTP verbs such as POST, PUT, DELETE, and GET. Typically, the API accepts and returns JSON schemas defined by "kind" and "apiVersion" fields.

The JSON schema consists of three terms, 1) Kind, the name of the object schema belonging to Kind categories, 2) API Group, being the set of resources exposed and the apiVersion, and 3) Resource, which can be either a single resource entity or a list of the homogeneous resources. There are three types of categories, 1) Object, which is defines a single resource such as a Pod, Service, or Namespace. 2) Lists, which is a list of resources of a single kind such as PodLists, ServiceLists, NodeLists, and 3) Simple, which is an action on a resource such as scale, or status [19].

Using standard HTTP verbs, the RESTful API allows different actions on resources:

*Table 2* List of HTTP verbs available on Kubernetes Resful API

| Verb | URI | Description |
|---|---|---|
| GET | /ResourceName | Retrieve a list of resource |
| | /ResourceName/EntityName | Retrieve a resource entity by name |
| POST | /ResourceName | Create a resource from JSON in body |
| PUT | /ResourceName/EntityName | Update or create the resource by name |
| PATCH | /ResourceName/EntityName | Modify specific field(s) of an entity |
| DELETE | /ResourceName/EntityName | Delete specific resource entity |

## Related Work

Resource management on the cloud is an area undergoing heavy research, however, most researches focus on Virtual Machines resource provisioning.

While predictive models predict patterns ahead of time, wrong predictions can lead to over or under provisioning which may lead to serious drawbacks. Moreover, prediction requires samples of data to learn from and work effectively, deeming it unsuitable for new applications [22].

Different researches describe models either by using only queueing theory, which serving mostly as real-time reactive model, or in a hybrid providing further predictive capabilities. A simple queue model based on web applications is experimented in [22], the models decides on scaling VMs up and down while avoiding live migration. The model works by allocating an equal amount of resources for all VMS, arguably, by doing so, the system will avoid wasting resources while guaranteeing easy placement VMs. A Cloud Controller which acts as the entry point for all requests, records the response time of requests and length of the waiting queue. Periodically, the aforementioned Cloud Controller will check system status and decide on changing the number of VMs for a select application on the cloud. The dynamic reallocation is based on calculating the average length of the waiting queue, average waiting time, and average arrival time. Using user defined error grace range and number of consequent error repetitions, if the queue metrics doesn't meet Service Level Agreement values, a scale up or down of number of VMs is triggered.

In an older paper, a file transfer web server is deployed in a queueing network model. The network consisted of two nodes for the web server and two more for the internet communication, each with its single queue. Jackson's Network Model was used to

determine the response time using eight variables were used, namely, Network Arrival Rate, Average File Size, Buffer Size, Initialization Time, Static and Dynamic Server Times, and Server and Client Network Bandwidths. Different scenarios were evaluated in the study, increasing server power, increasing bandwidth and adding additional servers. When the network bandwidth was the bottleneck, only increasing the bandwidth was found beneficial, however, when servers are experiencing high arrival rates, increasing server power performed best. In the latter scenario, increasing the network's bandwidth performed as second best up until a certain arrival rate where it increases exponentially, giving the second best to adding additional servers [21]. It is worth noting that while extra servers where added the network bandwidth remained constant.

In the cloud, many researches propose models to improve live VM migration, however, VM migration causes problems related to resource sharing; as VMs compete for shared resources. Thus, an optimal model must consider the migration's impact on VMs located at the source and destination Physical Machines. In [23], queueing theory is employed to evaluate the relationship between the residual resource bandwidth and performance of a VM, later, a bandwidth allocation algorithm improves live migration and balances the impact on VMs. Similarly, in [24] an algorithm based on queueing theory calculates an optimal distribution probability vector which proposes a scheme that minimizes and controls the number of active servers, from which the algorithms consolidate VMs into optimal Physical Machines.

## Kubernetes Testbed

In order to evaluate Kubernetes behavior when experiencing high-load of requests, as well as to develop and evaluate the Queueing-Theory based Auto-Scaler, a testbed is required. The testbed used a Docker image AcmeAir. AcmeAir is a Nodejs based web application that provides Airline querying and booking service. The developers provided the application in two different architectures, monolithic and microservices.

For our purpose, the microservices version is deployed to fit Kubernetes deployment architecture.

## Testbed Architecture

The AcmeAir microservices version consist of three components: 1) AcmeAir frontend, which provides the web interface for user interaction through port 9080. 2) AcmeAir Authentication service, which handles users' authentication (login and logout). 3) and MongoDB Database, which stores all the records of users and flights.

Through creating YAML Deployment files that deploys each component in a separate Pod, and then exposing each Pod through a Service to facilitate communication between Pods.

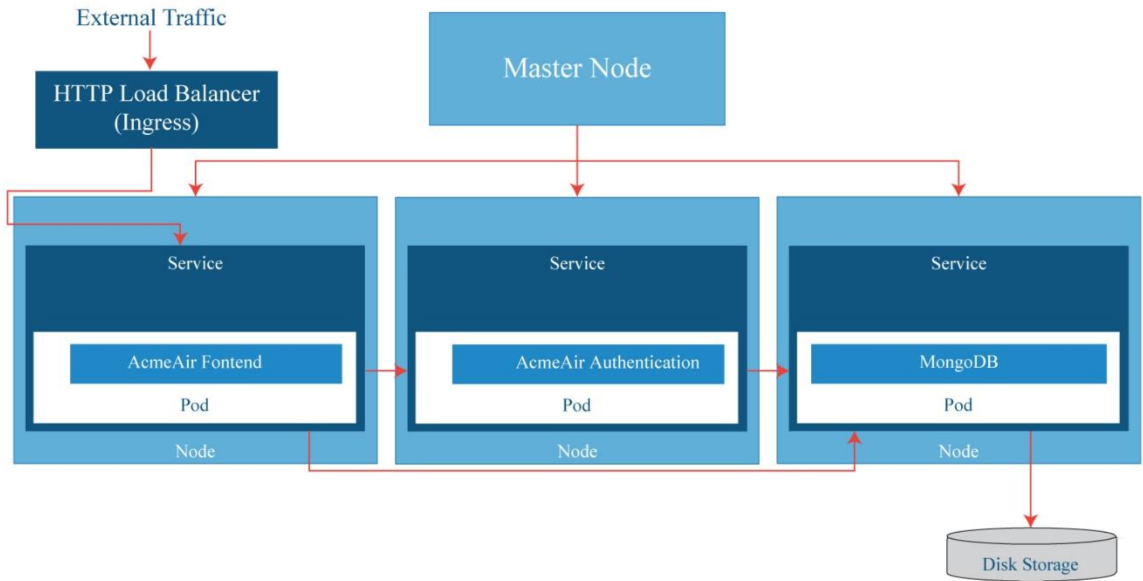


*Figure 7* AcmeAir basic Architecture

As shown in *Figure 7*, each Pod is exposed via a Service (load-balancer endpoint), however except AcmeAir Front-end; all Pods are only exposed locally for Pod-to-Pod communication. Furthermore, Mongo Service Pod is connected to a persistent storage disk for storing the database. The moFrontend Pods are responsible for receiving incoming requests and then either respond directly (for static requests e.g. webpage, image, css or JS files), create a user session through Authentication Pods, or query MongoDB and retrieve results.

## Testbed Deployment

Google Cloud Platform allows developers to create Kubernetes clusters through Kubernetes Engine (GKE) which creates VMs preconfigured with Kubernets (Nodes). The GKE provides developers with many VM options, from shared VMs providing little memory and CPU resources, to VMs with huge pool of resources.

For experimental purposes, the testbed is deployed upon shared VMs, where each Node is a single core vCPU, and 0.6GB of memory. The testbed has a minimum of three Nodes in all experiments. Additionally, a second testbed with higher CPU and memory capacities' is experimented, this testbed uses 2 vCPU, and 3.75GB of memory, with a minimum of three Node.

AcmeAir Deployments didn't specify any resource limits on its Pods, thus every Pod is free to utilize its Node's resources.

## Kubernetes Evaluation

Before designing an Auto-Scaler for Kubernetes, it is essential to understand how Kubernetes utilizes resources (CPU, Memory, and Network). Such understanding will draw the roadmap towards designing an effective Auto-Scaler by exploiting bottlenecks, and default behaviors experimentally.

Treating Kubernetes clusters similarly to Virtual Machine clusters by mapping a Pod to VM behavior will result in an inadequate resource and cluster management. Thus, in this section, a series of load testing experiments are conducted to better evaluate and understand Kubernetes Pod behavior.

In purpose to exploit all different possibilities, the AcmeAir cluster is deployed in different architecture derived from the basic architecture described in *Figure 7*, and is exposed to heavy load using Apache JMeter. Each architecture is evaluated over 10 cycles, 10 minutes each, where JMeter logs each request's response time. Response time is later aggregated and averaged for each period of 30 seconds, lastly the results of each cycle are aggregated over the 30 seconds period to normalize results and neglect any outliers. Moreover, to better understand the bottlenecks, Heapster metrics are recorded during the load-testing cycles, representing Node (CPU and Memory Utilization, and Network throughput) and Pod (CPU, Memory and Networking usage) related metrics.

Initially, dozens of different architectures are evaluated, however only eight are represented due to their key differences, and insights they represent about the cluster. The eight architectures are varied in terms of number of Pods for each microservice (frontend, authentication, and MongoDB), their placement, and the Nodes' resources capacity. Most

of the experiments were exposed to 100 JMeter threads that send requests continuously over the 10 minutes period, unless stated otherwise.

**Nodes with small resources pool (1 vCPU, 0.6 Memory)**

1. Single Pod for each microservice, one application Pod per Node (1.1f 1.1a 1.1db).
2. Two frontend Pods, two authentication Pods and single database, each two application Pods share one Node (2.1f 2.1a 1.1db).
3. Two frontend Pods, two authentication Pods and two databases, each Pod runs on single Node (2.2f 2.2a 1.1db).
4. Two Pods of each application, each on a single Node (2.2f 2.2a 2.2db).
5. Three of each frontend and authentication and single database, on single Nodes (3.3f 3.3a 1.1db)
6. Three of each frontend and authentication and two databases, on single Nodes (3.3f 3.3a 2.2db)

**Nodes with big resources pool (2 vCPU, 3.75 Memory)**

7. Single Pod on a Node of each application, 4 runs at different number of threads (40, 100, 150, 180). (1.1f 1.1a 1.1db - big).
8. Two of each frontend and authentication Pods on a single Node each, and single database, 4 runs at different number of threads (40, 100, 150, 180). (2.1f 2.1a 1.1db - big ).

## Results Discussion

It is essential to evaluate how high loads affect different deployment architectures, from the JMeter results, the response time averages are described in the following graph.

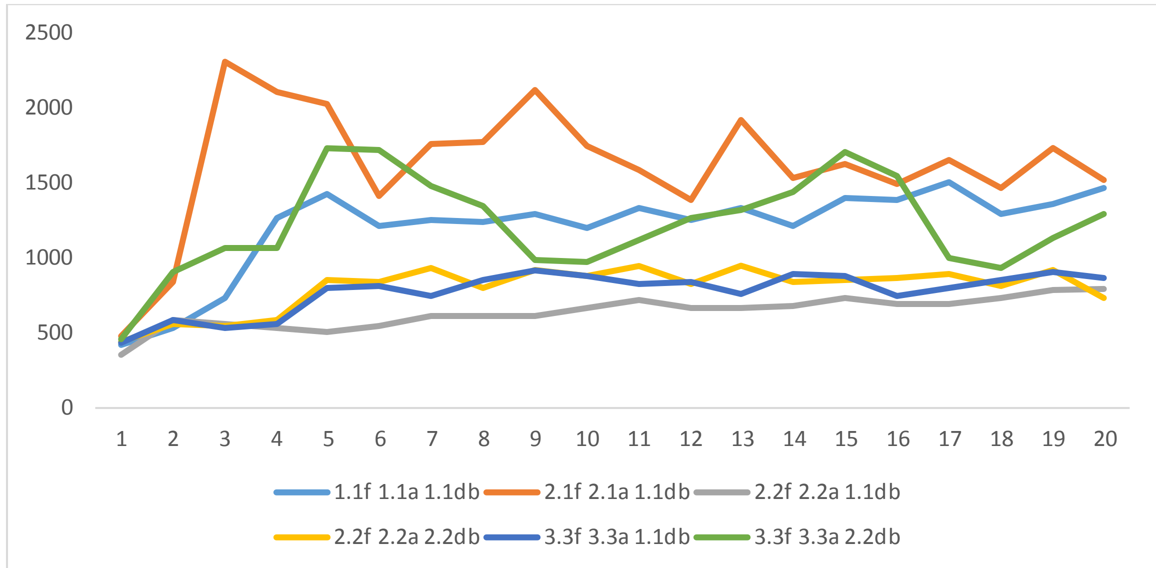


*Figure 8* Response Time for different deployments

From the graph, it's clear that the single Pod architecture (1.1f 1.1a 1.1db) show's a steady growth in response time, moreover using a replication set of databases does increase the response time and introduce unstable response time; this can be explained by the need of database synchronization between the different replicas, and in our application, a single database is capable enough to handle all requests, thus using a replication set is a drawback.

From *Figure 8* it's evident that using an appropriate amount of Pods results in the smallest response time (2.2f 2.2a 1.1db), even in comparison to overprovisioning. However, collocating two application Pods in the same Node dramatically worsen performance, that is also indicated in the amount of error responses and timeouts received by such architecture (2.1f 2.1fl 1.1db). The latter behavior is explained by considering resources utilization.

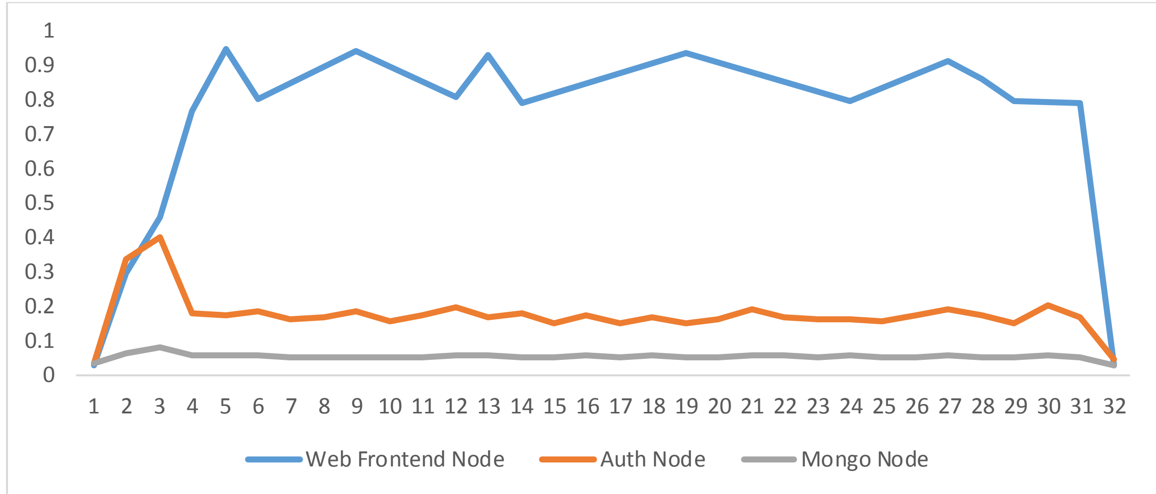


*Figure 9* CPU utilization of Nodes running single application Pod

The CPU utilization indicated in *Figure 9* shows that a single Pod of Web frontend along with Kubernets system components under a high load yield high utilization, this is further cleared when looking at *Figure 10* indicating the CPU usage (in millicores of total 1000 millicores). Noticeably, the frontend Pod is struggling for CPU resources, and thus placing two Pods on a single Node results in throttling CPU usage, which causes Pod failure that results in errors and timeouts.

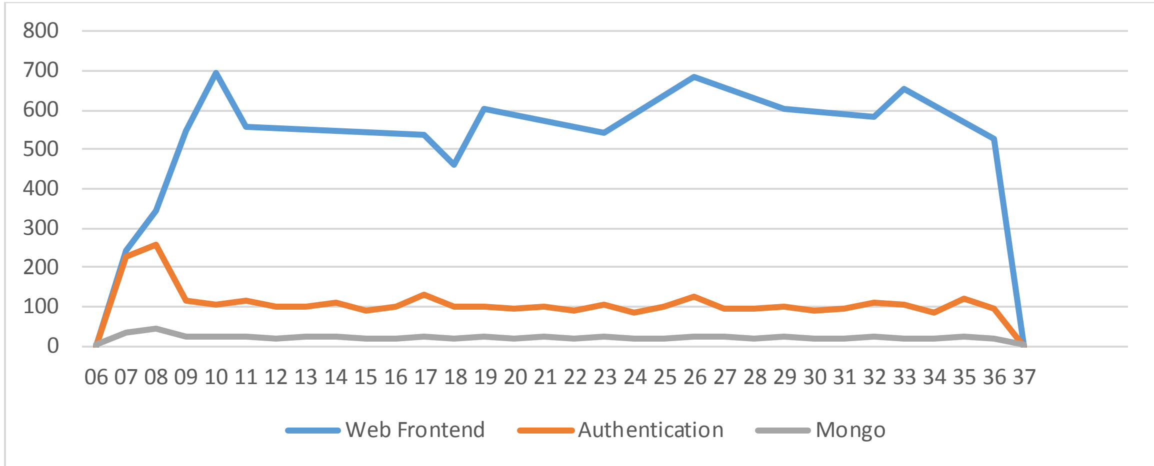


*Figure 10* CPU usage of Pods

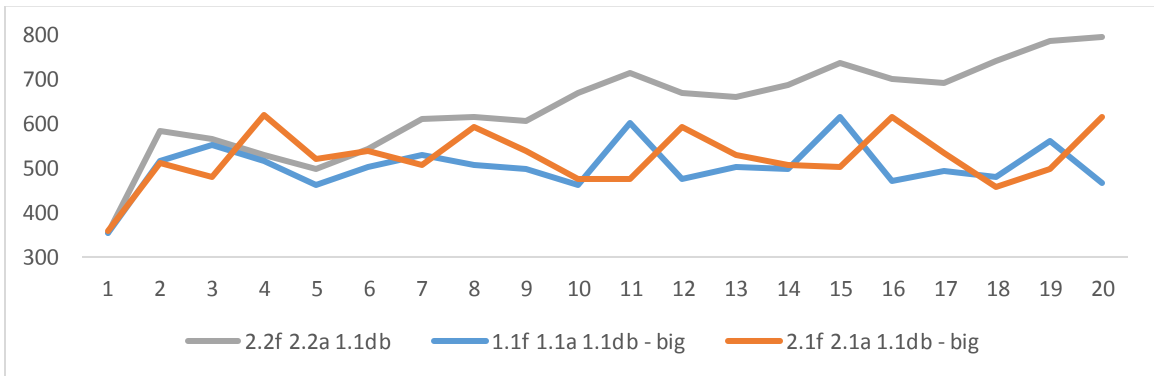


*Figure 11* Response time for optimal Pod size

The results present the importance of understanding a Pod's CPU requirements when scheduling it on a Node. *Figure 11* further compares the difference in response time between the case where CPU is throttled (2.2f 2.2a 1.1db) due to insufficient Node resources and when there are sufficient allocable CPU resources. Moreover, collocating Pods in a Node with sufficient resources produced the same response time as a single Pod.

To further understand the behavior, the cluster is exposed to different number of JMeter request threads.

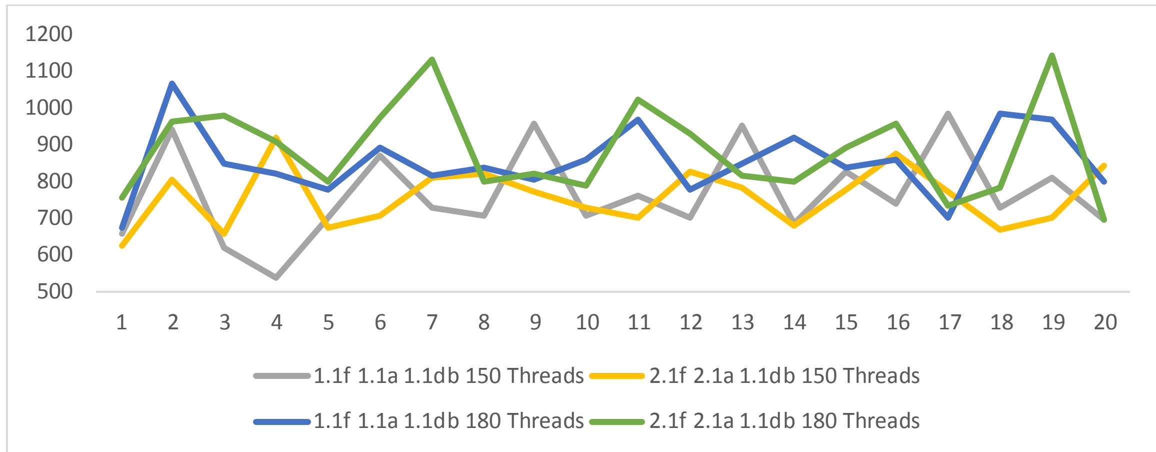


*Figure 12* High resource availability cluster at 150 and 180 request threads

When the number of request threads increased to 150 instead of 100, the response time increased by roughly 50% correspondingly. While the average response time between the single Pod and two Pod architectures is marginally neglectable (765ms vs 755ms), the latter appears more stable, however when the load increased by using 180 request threads, using two Pods did destabilize response time and marginally increased it (850ms vs 890ms), as shown in *Figure 12*.

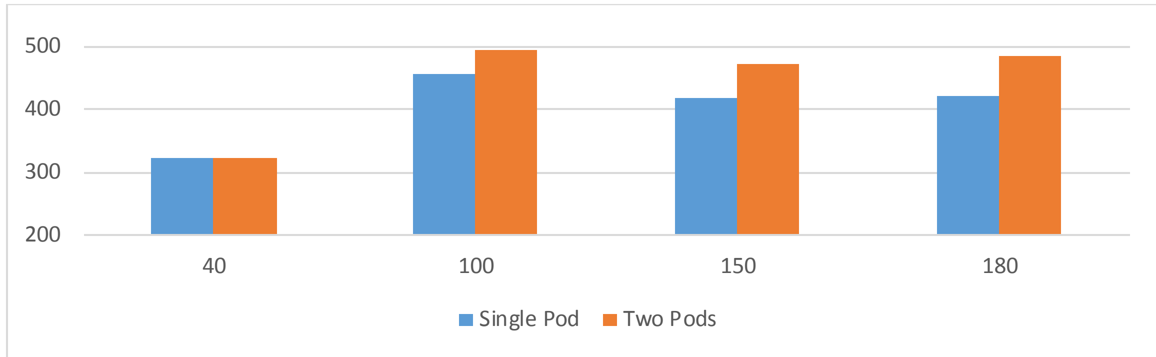


*Figure 13* Cumulative CPU usage in millicores by frontend application

Unlike the earlier case with small resource pool, the collocated Pods had low CPU utilization even at highest load, *Figure 13* indicated the CPU usage (millicores) of a single Pod compared to two Pods at different number of request threads. As shown, the two Pods combined has roughly utilized the same amount of CPU as a single Pod. Moreover, the CPU usage didn't represent the increasing number of requests, but rather declined due to the high number of requests failures.

The data in *Figure 13* indicate that CPU utilization is not effective enough to represent the high load for web applications, nor that allocable CPU is the bottleneck. Network throughput provides a deeper insight about the amount of requests, as shown in *Figure 14*, at 40 request threads the network throughput was lower and steadier than higher number of requests, at 100 threads, the single Pod had was steadier, conforming with *Figure 11*, however we can see a throughput decrease at fluctuation on 150 threads and above, indicating the failures.

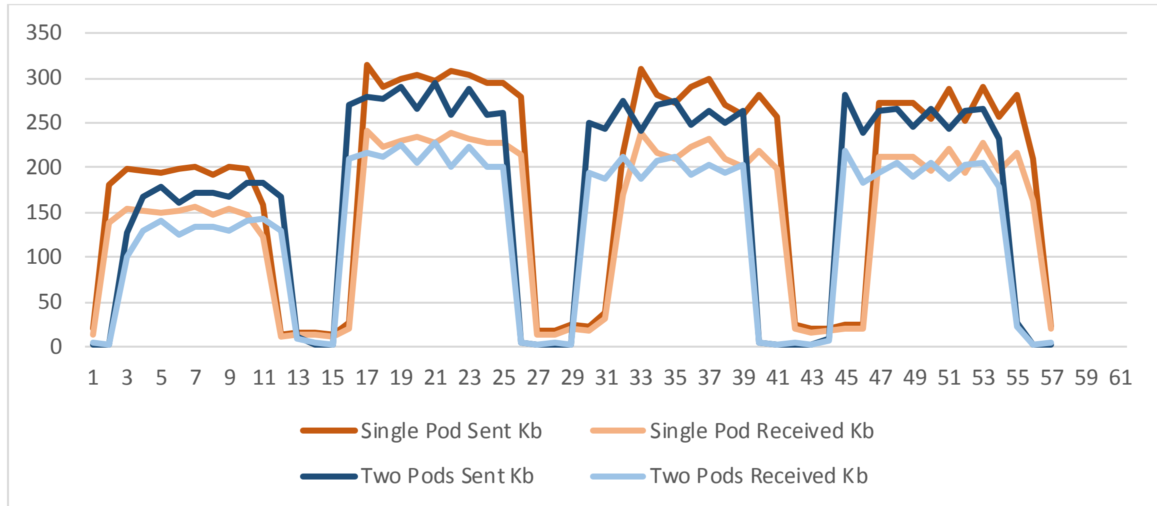


*Figure 14* Network throughput for 40, 100, 150 and 180 request threads

Typically, CPU and Memory resources are definite for each Node, however Network Bandwidth is not stated nor customizable. Moreover, there exists a virtual limit on the Network throughput either on the OS level, or the application level, thus for web applications, it is important to consider the incoming number of requests before considering other resources as the bottleneck. From the evaluation, it's understandable that increasing CPU or memory resources doesn't always improve the performance of an application where the network is the bottleneck, thus it is of high importance to detect the source of throttling and optimize based on such factors.

## Problem Formulation

Kubernetes offers a built-in Horizontal Pods Auto-Scaler (HPA) that is reactive, and threshold driven based on CPU utilization. However, the HPA has different drawbacks, as setting a fixed threshold can be application ignorant, in addition to CPU utilization not being an effective metrics for all applications, specifically when other constrains exist, moreover, HPA calculates Utilization as Pod's request versus usage, which doesn't indicate if a Pod is being throttled as indicated in our evaluation and require users to manually define resource request which is a task requiring expertise.

Queueing theory based Auto-Scaler provide a mean for a more accurate scaling decisions that are determined based on queue performance metrics. Queueing metrics such as queue length, arrival rate, service time and waiting time are effective in creating decisions based on short-term estimates [25].

Queueing Theory promises a great potential for Pods Auto-Scaling in Kubernets, thus we derived a Queueing Theory Auto-Scaler based on model described in [25, 26] which creates scaling decisions for Virtual Machines on a cloud system, the model is adjusted to meet our Kubernetes evaluations in section 0 as an improvement of the existing built-in Auto-Scaler.

## Queuing Theory

Queueing Theory focuses on analyzing mathematical models of systems that experiences random request patterns. Fundamentally, three components make up the queueing model. *Input process,* which generally describes the distribution of inter-arrival times of requests. *Service mechanism* is concerned with the duration an ongoing request being served blocks other requests from the service, this is defined as service time. *Queue discipline* describes

the behavior of incoming requests at a blocked system, which is a system busy serving other requests at the time a new request arrives. A request arriving at a blocked system may leave unserved, or get pushed into a waiting queue. Selecting requests from the waiting queue can be done through different algorithms, however first-in first-out algorithm (FIFO) provides a simple yet sufficient solution to systems subject to random request patterns [20]. Our model is based on web servers, which inhibits queueing-based scheme. Typically, in a web server architecture, multiple requests compete over shared resources; such shared resources are only accessible by one request at a time. As more requests arrive simultaneously, a request queue is formed, from which each request is responded to and then removed from the queue. Different heuristics based on the queue give an idea of a server load and ability to satisfy such requests.

From Queueing Theory's perspective, there exists a single queue feeding requests to a set of service system, namely the web servers (a Kubernetes service). Queueing Theory model will monitor each queue in terms of average arrival rate ($\lambda$), average service time ($T_s$) and average queuing time ($T_q$).

The queue metrics determines the server's utilization, utilization ($U$) is defined as a product of average arrival rate and average service time.

$$Utilization\ (U) = \lambda * T_s$$

Utilization value is bound between 0 and 1, where a utilized system approaches 1. According to Queueing Theory, a stable system experiences an average arrival rate less than the service rate. A system which experiences random arrival rate of requests exhibits an Poisson distribution, such system is known as an M/M/c queue, where M and c represents the memoryless fashion of arrival rates and number of servers, respectively.

Such system's response time grows sharply as it approaches utilization value of 1, as demonstrated in *Figure* 15**Error! Reference source not found.**. Using Little's Law, an average number of requests in the waiting queue ($N$) is calculated $N = \lambda * T$, where $T = T_s + T_q$ or $T = \frac{T_s}{1-U}$ for M/M/1 queue [21, 20, 22].

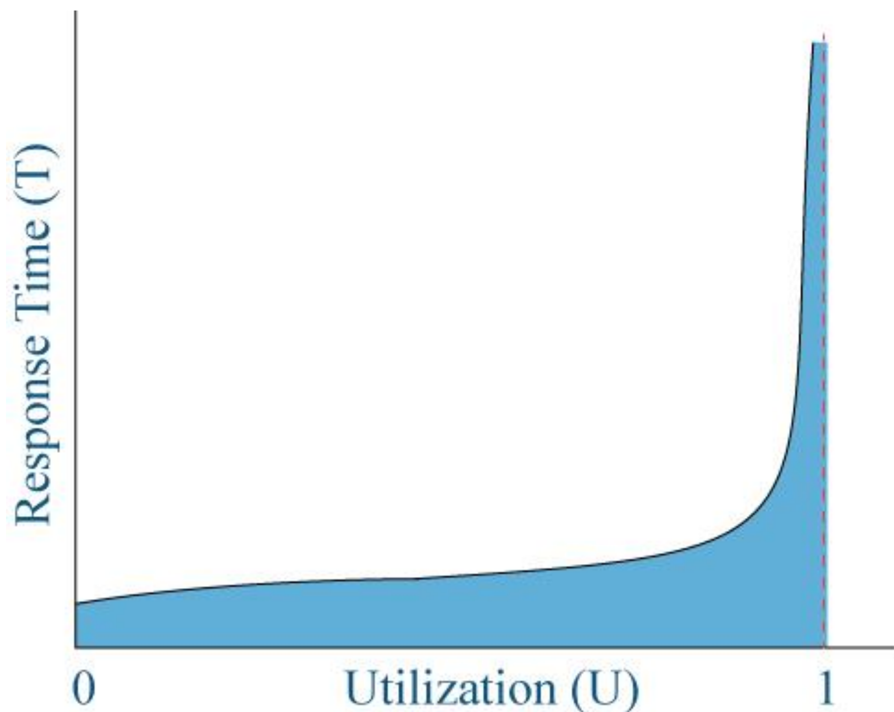


*Figure* 15 Response time as a function of utilization

## Problem Definition

Consider a system using M/M/c queue, which consists of $c$ VMs experiencing requests arrival at Passion distribution of at rate $\lambda$ and service rate $\mu$. At such systems, clients' requests are received at the Load Balancer as an entry point, from which the load is

distributed across the servers as showing in *Figure 16*. In such system, server $i$ at time $t$ has a $\mu_{i,t}$ service time, with $c$ servers in the system.

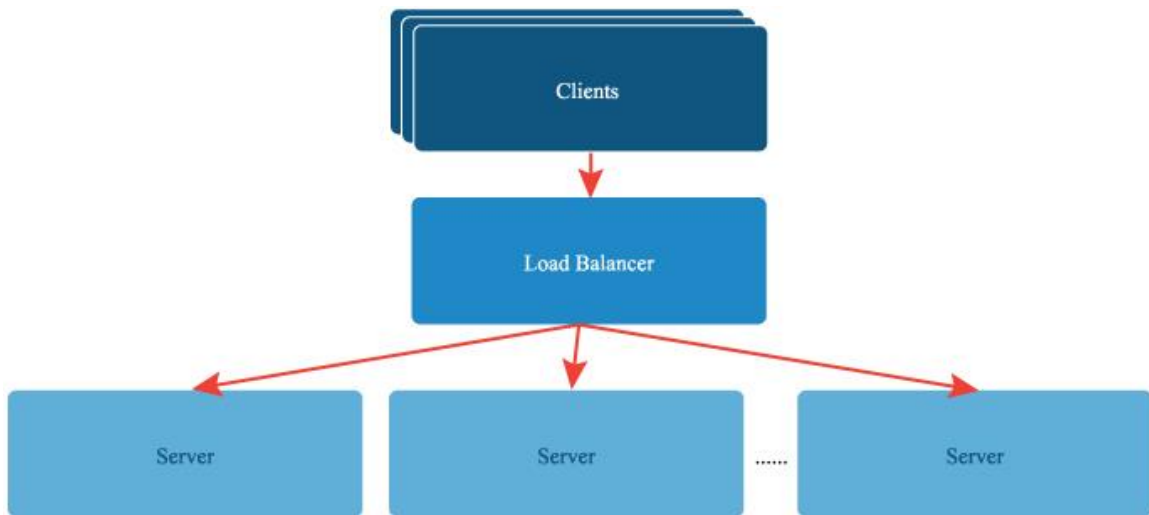


*Figure 16* Simple Multi-Client to Multi-Server Architecture

In total, $\mu_t$ describes the total amount service time of the system at time $t$, such that

$$\mu_t = \sum_{t=1}^{n} \mu_{i,t}$$

*Equation 1 System service time*

In a steady-state system with $c$ independent and identical servers, service rate $\mu$ is dependent on number of servers such that for $i$ number of requests at moment time

$$\mu = \begin{cases} i(1/T_s) & 1 \le i \le c \\ c(1/T_s) & i \ge c \end{cases}$$

*Equation 2 Service Time*

From Equation 2 we can draw system utilization value as a factor of arrival rate $\lambda$, which is a calculated from interarrival time $\lambda = 1/X(t)$, in addition to service time $\mu$ and number of servers $c$ as

$$\rho_t = \frac{\lambda_t}{c_t \mu_t}$$

*Equation 3 System Utilization*

The latency $L$ of such system is a factor of execution time $T_s$ and waiting time $T_q$. $T_s$ is given from Pod monitoring values, however to draw waiting time in the queue $T_q$ we calculate the size of the queue $L_q$ which a factor of idle probably of the system $P_0$ and system utilization, then calculate $T_q$ from the queue size $L_q$ and arrival rate $\lambda$ [27, 28]

$$P_0 = 1 \Big/ [\sum_{n=0}^{c_t - 1} \frac{(c_t \rho_t)^n}{n!} + \frac{(c_t \rho_t)^{c_t}}{c_t! \ (1 - \rho_t)}]$$

*Equation 4 Idle system probability*

$$L_q = \frac{c_t^c \rho_t^{c+1}}{c! \, (1 - \rho_t)^2} P_0$$

*Equation 5 Expected Queue Size*

$$T_q = \frac{L_q}{\lambda}$$

*Equation 6 Queue Waiting Time*

Resultantly, latency $L$ is factor of arrival rate, service rate, and number of server

$$L(\lambda, \mu, c) = T_q + T_s$$

*Equation 7 System latency*

The objective is to minimize response time while maximizing utilization by minimizing the cluster size, thus, our goal is to find $c$ servers where $L < T$, where $T$ is the desired response time. Firstly, to discover if the system requires scaling, by using Equation 7 calculate the system's current latency under $c$ servers, then we normalize latency

$$G = {}^{L}/_{T}$$

*Equation 8 Latency normalization*

Where an under provisioned system exhibits $G > 1$, and an overprovisioned server exhibits $G < 1$, in order to stabilize the system, a threshold $\theta$ is defined to determine if latency is out of acceptable range, in which case an exhaustive search algorithm calculate $L$ for an increasing $m$ number of servers, where $m$ starts at $m = c$ if $G > 1$, or $m = 1$ where $G < 1$.

## Virtual Machines Auto-Scaling

Considering the number of VMs $c$ at time $t$, using the queueing theory models, the Auto-Scaler can define resource requirements at $c_{t+1}$ such that, $c_t > c_{t+1}$ invokes Scaling-Down while $c_t < c_{t+1}$ invokes Scaling-Up.

Generally, Virtual Machines have a hard-predefined limit of resources on a Physical Machine (PM), and since initiating and destroying VMs is a time and resource consuming task, an Auto-Scaler is to preferably consider Vertically Scaling decisions, by adding more resources to current nodes if feasible. Thus, for Scaling-Up the Auto-Scaler considers

calculating the remaining resources at $PM_i$ for each $VM_i$, while setting a maximum limit of resources a VM make benefit of. The Up-Scaling algorithm works in three stages:

1) Defining resource requirements as $V_{t+1}^{`} = V_{t+1} - V_t$, where $V$ is amount of resources required, derived from $c$.
2) Vertical Up-Scaling by add $V_{t+1}^{`}$ resources to zero or more VMs.
3) If Vertical Up-Scaling fails to accommodate $V_{t+1}^{`}$, then start one or more VMs.

On the other hand, Down-Scaling was only defined horizontally as Vertical Down-Scaling may cause VM failure. The Down-Scaling algorithm seeks to select one or more VMs to destroy, where the total of closed resources is less or equal to minimum resources.

## Pods Auto-Scaling

The model proposed by [25, 26] does offer an improvement over HPA, however due to key differences between VMs and Pods, and according to our Kubernetes evaluation, the algorithm needs redefining.

VMs have well-defined resource limits, however in Kubernetes, setting resource limits for Pods is optional, and while such limits assist Kubernets' Scheduler in Pod placement, the QAS will be responsible for optimally placing Pods within available Nodes. Limit-free Pods in Kubernets compete for available resources on a Node, thus; it is of high importance to detect resource throttling before making scaling decisions, once throttling is detected, QAS will seek freeing up resources by migrating Pods, if no migration is possible then scaling will occur.

QAS will utilize two key variables that affects the model, $\Gamma$ and $\tau$ will define the queue latency monitoring cycle and resource throttle monitoring cycle in seconds, respectively,

where typically $\Gamma > \tau$, while $\emptyset$ is the Node utilization threshold which invokes possible migration.

## Solution Architecture

The Queueing Theory Based Auto-Scaler (QAS) will use Heapster metrics through REST API to query current cluster status and enable decision-making. The QAS will run on the cluster as a Pod that queries Pods service time and retrieve resource allocation of a Pod through Heapster.

Currently, Kubernetes' Services do not provide any metrics about incoming requests through the API, and Heapster is only able to provide metrics regarding resource allocation. Thus, by default the cluster does not log or provide any metrics regarding incoming requests. A workaround is to deploy a reverse proxy as an ambassador container co-existing at each Pod we wish to auto scale.

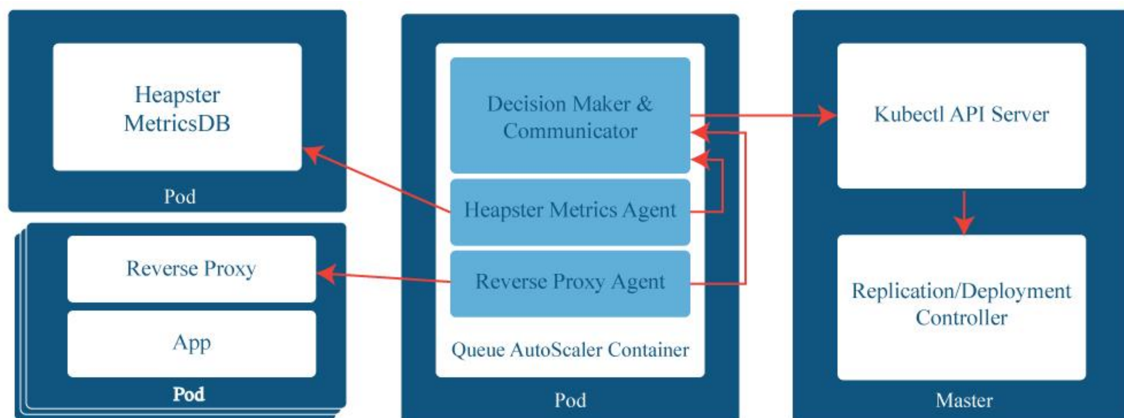


*Figure 17* Queueing Theory Based-Auto Scaler Low Level Architecture

As illustrated in *Figure 17*, QAS itself consists of three main components that work together to retrieve and aggregate metrics, and then update the cluster. The three main components of QAS are a reverse proxy agent, Heapster metrics agent and the main decision-making component.

Moreover, QAS will require three external components that provides important metrics, namely Heapster, a query-able metrics database, and a reverse proxy container on each Pod that requires auto scaling as showing in *Figure 18*.

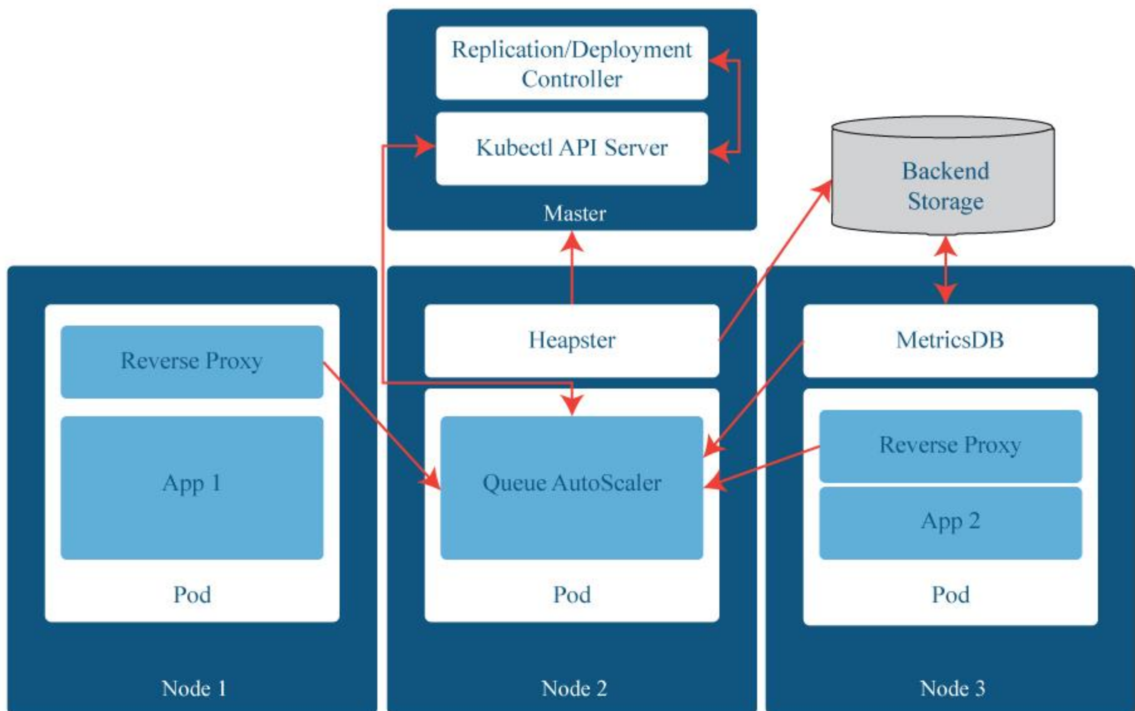


*Figure 18* Queueing Theory Based-Auto Scaler High Level Architecture

## QAS Internal Components

QAS is implemented in NodeJS, and while the three internal components are running under one container, it is possible for such components to be deployed on different containers or Pods while communicating through localhost.

### Decision Maker and Communicator

DMC is the main component of QAS, as described in *Figure 17*, it receives different cluster metrices from both the Heapster and Reverse Proxy agents. The DMC then uses the Queueing Theory Algorithm described in the previous section to create scaling or migration decisions. Once a decision is made, the DMC will communicate cluster updates to Kubernets API Agent through REST API. The API agent in turn communicates with Kubernets Replication/Deployment Controller to apply such updates on the selected Deployments or Replication Set.

### Heapster Metrics Agent

Despite the name, HMA does not communicate directly with Heapster to retrieve resource utilization metrics, but rather with a metrics datastore agent that provides a REST service to query Heapster's recorded metrices in the Backend Storage. Moreover, HMA will calculate residual resources at each Node and each application's Pod average resources.

### Reverse Proxy Agent

RPA will communicate with the reverse proxy container on all the Pods selected for auto scaling. Additionally, it will compute the average response time (service time) within the specific cycle period for an application. Lastly, RPA will communicate those metrics to the DMC for decision making.

## QAS External Components

### Heapster Metrics Database

While Heapster provides a REST API for recorded metrics, it however rotates the logs every 15 minutes, thus limiting the Auto-Scaler cycle to no more than that time; to overcome such limits a metrics datastore is deployed to use Heapster's backend storage showing in *Figure* 5 .

InfluxDB is an open-source time-series database, that is capable of recording Heapster metrics and exposing them over a HTTP API in a SQL-like query language. InfluxDB will return resource metrics to HMA in a JSON format.

### Reverse Proxy

Traefik is a HTTP reverse proxy and load balancer that supports several microservices frameworks such as Docker, and Kubernets [29]. Traefik provides metrics through a REST API, and it logs different requests and responses.

While Traefik is deployable in the Kubernets environment as a Pod that works on top of ingress Services, currently it provides service time metrics for all associated Pods within the cluster in an aggregated form, thus to get more detailed metrics about each single Pod, Traefik is deployed as an ambassador container within a Pod, and exposing the REST API at port 8080 that provides number of requests, total response time, and average response time. However, the RPA will query such data and calculate metrics only related to the defined auto-scaling cycle.

## Solution Implementation

In this section we will describe the algorithm implementation of the different components, and Pod configurations to enable auto scaling.

## Pod Configuration

In Kubernetes Pods are deployed using a Deployment or a Replication Set as described in section 0. A Deployment describes the template of a Pod by configuring different attributes and most importantly specifying container images.

To retrieve a container's service time, Traefik reverse proxy is deployed as an ambassador container as mentioned in section 0. Traefik requires special configuration to enable routing to a specific container and its exposed port. There are different ways to configure Traefik routing, however File Backend is the most suitable configuration for our use case, File Backend allows Traefik to establish routing configurations using a *configure.toml* file [30]. To maintain modularity and keep the Traefik Docker image unchanged, the configuration file is passed to the container image through a *ConfigMap* attribute in the Deployment template. The *ConfigMap* will specify the content Traefik configuration file by specifying Traefik entry point, a backend (i.e. application container's address and port), and a frontend which specifies the routing rule. By default, the backend should point to "localhost" or "127.0.01" address, since both containers exist in the same Pod, and the frontend rule will be routing from the root path "Path:/".

## Heapster and InfluxDB

As described on section 0, Heapster metrics will be retrieved through a metrics database, namely InfluxDB. The current version of Kubernetes deploys a Heapster Deployment by default, thus the default Deployment must be configured by defining InfluxDB as the storage sink at Heapster container.

## Reverse Proxy Agent

The goal of this component is to estimate the arrival rate, and system service time, this is achieved by retrieving the request rate and service time for each Pod of a Deployment. Querying Traefik, the RPA retrieves the total service time in seconds $\mu_i^T$ and the total requests count $\lambda_i^T$. However, we're only interested in a specific period, thus RPA will keep track of the previous total service time and total requests count, then calculate the difference. By aggregate, all the Deployment's Pods service time and requests count values, the RPA outputs the system's average service time and requests arrival rate.

*Algorithm 1* Reverse Proxy Agent

**Input:** $\mu_i^T, \lambda_i^T, \Gamma, \mu_{i,t-1}^T, \lambda_{i,t-1}^T$

**Output:** $array < \mu_t, \lambda_t >$

1: $i \leftarrow 1, \lambda_t^T \leftarrow 0, u_t^T \leftarrow 0$

2: **while** $i <= c_t$ **do**

3: **if** $\mu_{i,t-1}^T == undefined$ **then**

4: $\mu_{i,t-1}^T \leftarrow 0, \lambda_{i.t-1}^T \leftarrow 0$

5: **endif**

6: $u_t^T \leftarrow u_t^T + (\mu_i^T - \mu_{i,t-1}^T)$

7: $\lambda_t^T \leftarrow \lambda_t^T + (\lambda_i^T - \lambda_{i,t-1}^T)$

8: **end while**

9: **return** $array < {u_t^T}/{\lambda_t^T}, {\lambda_t^T}/{\Gamma} >$

## Heapster Metrics Agent

The HMA is composed of three parts, the first part queries Node resource utilization at the defined Γ cycle at each Node deployment the monitored Pod, while second part queries the every Pod resource allocation at certain Node where utilization passed threshold $\emptyset$, the third parts queries every Node allocable resources for the nominated list of Pods to be migrated.

Definitions: $V_{ij}$ is the resource allocation of Pod $i$ placed on Node $j$, $U_j$ is the utilization of Node $j$ hosting the monitored Pod, $p_j$ is number of Pods other than monitored on Node $j$, $np_j$ are nominated Pod for migration, $N$ is all Nodes on the system, $\chi$ is a list of Pods with candidate Node to migrate to, all metrics are for cycle Γ.

*Algorithm 2* Heapster Metrics Agent

**Input:** $\chi$
**Output:** $Nominated\ Node - Nominated\ Pod\ array$
01: **if** $(U_j > \phi)$ **then**
02: $i \leftarrow 0$
03: **for** $(i\ ..\ p_j)$ **do** //find $np_j$ with most resources
04: $np_j = 0$
05: **if** $resource(np_j) < V_{ij}$ && $not_in(np_j, \chi)$ **then**
06: $np_j = i$
07: **endif**
08: **endfor**
09: $n \leftarrow 1$
10: **for** $(n\ ..\ N)$ **do** //find Node with sufficient resources
11: **if** $(allocable(n) \geq resource(np_j)$ && $not_in(n, \chi))$
12: $append(\chi, array < np_j, n >)$
13: bearkforloop()
14: **endif**
15: **endfor**
16: **return** $\chi$
17: **endif**

## Decision Maker and Communicator

The main component will receive the custom metrics defined by the RPA and HMA, then run an algorithm defined in section 0. The DMC will run at two different cycles, at Γ for scaling, and at τ for migration.

The following algorithm receives a list of Pods and Nodes to migrate at cycle $\tau$ from Heapster Metrics Agent described in section 0.

*Algorithm 3* Pods Migration

**Input:** $\chi$
**Output:** the migration process
01: **if** $not_empty(\chi)$ **then**
02: **foreach** $(x\ of\ \chi)$ **do**
03: $start(np_j, n)$
04: $destroy(np_j, j)$
05: **endfor**
06: **endif**

The start and destroy algorithms are a logical grouping of micro actions to define the cluster. Kubernets allows selecting a set of Nodes on which to deploy specific Pods using labels, in order to sustain QAS of migrated Deployment's Pod, the new Node will be labeled according to the Deployment Node selector, then the Deployment will be scaled up, Kubernetes Scheduler prefers handles starting the new Pod on the most suitable Node, being the new labeled Node. Once the new Pod is up and running, QAS will scale down the Deployment after removing the label from the old Node to ensure that it's the one being destroyed.

The second algorithm of DMC will handle scaling up or down the cluster based on latency, this algorithm will run less often that the preceding to ensure that the system is at its most feasible steady state, where all Pod's throttling is eliminated, as possible.

*Algorithm 4* Scale Up Algorithm

**Input:** $\lambda - arrival\ rate, \mu - service\ time,$
$T - expected\ latency, n - number\ of\ Pods, \theta - latency\ threshold$

**Output:** //scaling the cluster

01: $L = get_latency(\lambda, \mu, n)$ //using equation (7)
02: $G = normalize_latency(L, T)$ //using equation (8)
03: **if** $(G > 1 + \theta)$ **then** //scale up
04: $m \leftarrow n$
05: **for** $(m..N)$ **do**
06: $L_m = get_latency(\lambda, \mu, m)$
07: **if** $(L_m < T)$ **then**
08: $scale(m)$
09: **endif**
10: **endfor**
11: **else if** $(G < 1 - \theta)$ **then** //scale down
12: $m \leftarrow 1$
13: **for** $(m..n)$ **do**
14: $L_m = get_latency(\lambda, \mu, m)$
15: **if** $(L_m < T)$ **then**
16: $scale(m)$
17: **endif**
18: **endfor**
19: **endif**

## Experiments and Evaluation

In order to evaluate the effectiveness of the Queuing Based Auto-Scaler (QAS), we have conducted a series of tests a high workload, in this section we draw out the outcome such results, further we compare QAS with HPA as a baseline Auto-Scaler to compare such performances.

## Evaluation Setup

Just like the elementary evaluation, such workload is carried by Apache JMeter which produces flight queries, user's login, logout, information change, and flight check-in tests that keeps the three different components of AcmeAir application closely coupled.

The evaluation used big servers' setup (2 vCPU, 3.75 Memory), as QAS was unable to pull metrics from Heapster agent on low settings servers due to high utilization values, which deemed Heapster unable to communicate with Pods effectively.

As discussed, HPA requires defining a value of requested resources for a Pod to draw Pod's utilization, to get realistic data, a set of initial experiments were set to draw Pod's CPU usage under high load. From the results, two HPA experiments are defined, using the upper and lower mean values, namely (300CPU millicores) and (400CPU millicores).

On the other hand, QAS metrics (expected latency and threshold) were defined based on response time of the system at a normal work load. Apache JMeter provides response time at client end of 350ms, however QAS measures latency internally (neglecting network latency between the client and servers), thus the desired latency was defined at 200ms, however two tests were run with different grace threshold value (0.6 and 0.4). Grace threshold allows response rate deviation at such value before provoking the auto scaler.

## Evaluation

Horizontal Pod Autoscaler HPA provides an effective autoscaling capability upon CPU utilization metrics, from the results indicated in *Figure 19*, while HPA-300 performs better than HPA-400, the difference is barely noticeable, and the system indicates peaks in response time reflecting a rather unsteady behaviour. QAS at a threshold of 0.6 performed only a little better than both HPA, however it also indicates peaks in response time as the cluster latency must diverge substantially (60%) before auto scaling is invoked. However, when QAS threshold is minimized to 40% divergence, the cluster provides a steadier performance and lower response times.

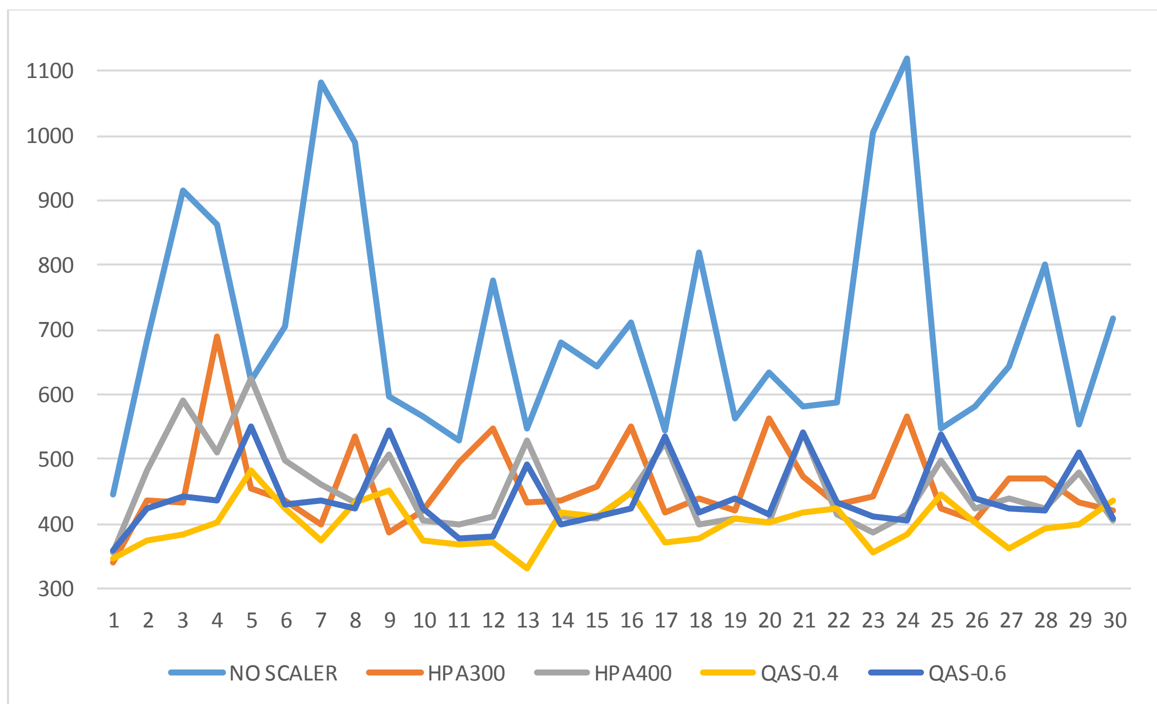


*Figure 19* Response time comparison between HPA and QAS

To better understand the results, *Figure 20* plots the number of Pods each auto scaler setup has started to correspond to the workload, expectedly, QAS with low latency threshold started the most number of Pods when needed as per high load of incoming requests. This indicates the importance of defining the threshold adequately to reflect cost to performance ratios.

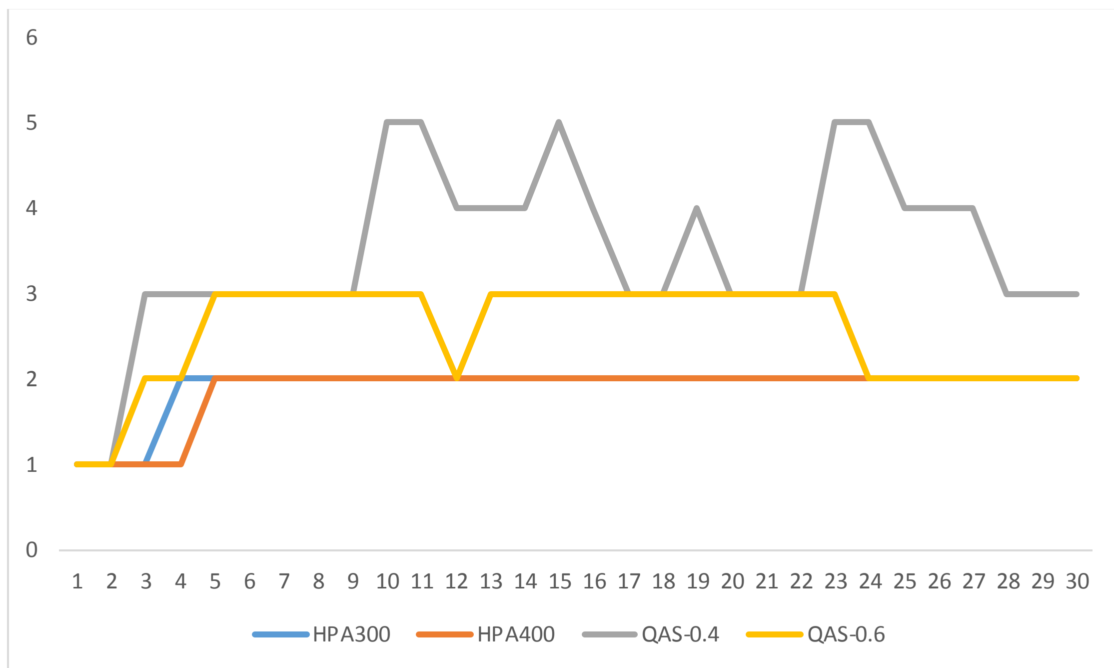


*Figure 20* Number of Pods started by different auto scalers

## Conclusion and Future Work

The microservice architecture reflects more accurately on the separation of micro-application concerns by providing means to discover bottlenecks and correctly adjust applications on the cloud based on microservice metrics as an independent system.

Kubernetes provides a promising platform for microservice deployment architecture, however more research and development are required in the field. Microservice auto scaling is a growing field that while shares similarities with Virtual Machines, which provides the basic concepts for microservices auto scaling while requiring adjustments to meet architecture, while considering the application underlined.

Queueing theory is a well established and researched model that translates well into the web application architecture by providing a more context aware metrics, yielding more accurate decisions.

In this work, we've proven that using queueing theory provides a great mean for scaling application that requires little resources (CPU or Memory), however move thorough research and evaluation is required to, with a great extinct, define the uses cases of such a model. Moreover, in our implementation, the auto scaler doesn't provide an actual live migration due to Kubernets current limitation. Benefiting from advances in other areas within Kubernets, an Auto Scaler should be able to lively migrate Pods by sharing status and affinity if any. Lastly, a more in-depth research is required to set migration policies in relation to CPU and Memory utilization.

## References

[1] Q. C. L. &. B. R. Zhang, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications,* vol. 1, no. 1, pp. 7-18, 2010.

[2] P. C. L. S. P. &. T. Y. C. Sharma, "Containers and Virtual Machines at Scale: A Comparative Study," in *In Proceedings of the 17th International Middleware Conference*, 2016.

[3] R. R. A. R. &. K. D. Dua, "Virtualization vs containerization to support paas," in *Cloud Engineering (IC2E), 2014 IEEE International Conference*, 2014.

[4] A. A. &. A. S. A. Bankole, "Cloud client prediction models for cloud resource provisioning in a multitier web application environment," in *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium*, 2013.

[5] J. Daniel , Y. Danny and J. Neeraj , "Scryer: Netflix's Predictive Auto Scaling Engine," http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html, 5 November 2013. [Online]. Available: http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html. [Accessed 21 March 2017].

[6] "Predictive auto-scaling in elastisys cloud platform," Elastisys, [Online]. Available: https://elastisys.com/cloud-platform-features/predictive-auto-scaling/. [Accessed 21 March 2017].

[7] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," in *IEEE Cloud Computing*, IEEE, 2014, pp. 81-84.

[8] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal,* vol. 2014, no. 1075-3583, p. 239, 2014.

[9] C. Boettiger, "An introduction to Docker for reproducible research," *SIGOPS Oper. Syst. Rev.,* vol. 49, no. 0163-5980, pp. 71-79, jan 2015.

[10] C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Computing,* vol. 2, no. 3, pp. 24-31, 2015.

[11] V. Marmol, R. Jnagal and T. Hockin, "Networking in Containers and Container Clusters," in *netdev 0.1*, Ottawa, 2015.

[12] "Kubernetes Concepts," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/. [Accessed 25 March 2017].

[13] D. Vohra, Kubernetes microservices with Docker, British Columbia: Apress, 2016.

[14] "Kubernetes Tutorials," Kubernetes, [Online]. Available: https://kubernetes.io/docs/tutorials/. [Accessed 26 March 2017].

[15] B. Burns, "The Distributed System ToolKit: Patterns for Composite Containers," Google, 29 June 2015. [Online]. Available: http://blog.kubernetes.io/2015/06/the-distributed-system-toolkit-patterns.html. [Accessed 4 April 2017].

[16] D. Vohra, Kubernetes Management Design Patterns, Springer, 2017, pp. 299--308.

[17] V. Kannan, "Compute Resource Usage Analysis and Monitoring of Container Clusters," Heapster, [Online]. Available: https://doi.org/10.5281/zenodo.574110. [Accessed 10 May 2017].

[18] V. Marmol, "Analyzes resource usage and performance characteristics of running containers.," cAdvisor, [Online]. Available: https://doi.org/10.5281/zenodo.574111. [Accessed 10 May 2017].

[19] M. Noorali, "Kubernetes," Google, [Online]. Available: https://doi.org/10.5281/zenodo.803408. [Accessed 6 June 2017].

[20] R. Cooper, "Queueing theory," in *Handbooks in Operations Research and Management Science*, vol. 2, Elsevier, 1990, pp. 469-518.

[21] K. Elleithy and A. KOMARALINGAM, "Using a queuing model to analyze the performance of web servers," in *International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medecine on the Internet*, Rome, Italy, 2002.

[22] H.-p. Chen and S.-c. Li, "A queueing-based model for performance management on cloud," in *Advanced Information Management and Service (IMS), 2010 6th International Conference on IEEE*, 2010.

[23] C. Zhu, B. Han, Y. Zhao and B. Liu, "A Queueing-Theory-Based Bandwidth Allocation Algorithm for Live Virtual Machine Migration," in *Smart City/SocialCom/SustainCom (SmartCity), 2015 IEEE International Conference*, 2015.

[24] C. Pham, N. H.Tran, C. T. Do and C. S. Hong, "Live consolidation for data centers in cloud environment," in *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, 2015.

[25] G. Huang, S. Wang, M. Zhang, Y. Li, Z. Qian, Y. Chen and S. Zhang, "Auto scaling virtual machines for web applications with queueing theory," in *Systems and Informatics (ICSAI), 2016 3rd International Conference*, 2016.

[26] J. Jiang, J. Lu, G. Zhang and G. Long, "Optimal Cloud Resource Auto-Scaling for Web Applications," in *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, Delft, 2013.

[27] R. B. Cooper, Introduction to queueing theory, Second Edition ed., Boca Raton, Florida: North Holland, 1981.

[28] S. &. C. Y. Nahmias, Production and operations analysis, New York: McGraw-Hill/Irwin, 2009.

[29] E. Vauge, "Træfik, a modern reverse proxy," traefik.io, 13 September 2015. [Online]. Available: https://doi.org/10.5281/zenodo.1058976. [Accessed 18 November 2017].

[30] "Configuration Backend: File," Containous, [Online]. Available: https://docs.traefik.io/configuration/backends/file/. [Accessed 19 November 2017].

[31] Y. Lu, T. Abdelzaher, C. Lu, L. Sha and X. Liu, "Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers," in *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings*, 2003.