Thesis

Preservation of Low Latency Service Request Processing in Dockerized Microservice Architectures

Submitted by

Leo Vigneshwaran Sudalaikkan

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2016

Master's Committee:

Advisor: Shrideep Pallickara

Sangmi Lee Pallickara
Leo Vijayasarathy

Abstract

Preservation of Low Latency Service Request Processing in Dockerized Microservice Architectures

Organizations are increasingly transitioning from monolithic architectures to microservices based architectures. Software built as microservices can be broken into multiple components that are easily deployable and scalable while providing good utilization of resources. A popular approach to building microservices is through containers. Docker is an open source technology for building, deploying, and executing distributed applications within containers that are referred to as pods in Docker orchestrator terminology. The objective of this thesis is the dynamic and targeted scaling of the pods comprising an application to ensure low latency servicing of requests. Our methodology targets the identification of impending latency constraint violations and performs targeted scaling maneuvers to alleviate load at a particular pod. Empirical benchmarks demonstrate the suitability of our approach.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Organizations are increasingly transitioning from monolithic architecture to micro services architecture. A popular approach to building microservices is through containers. In the last decade the use of virtualization has increased dramatically, and has proven to cut operational costs and provide greater utilization of hardware resources. Containers, on the other hand, are an abstraction that occurs at the Operating System(OS) level, and they are light weight. Containers provide many benefits compared to Virtual Machines(VMs) in terms of CPU and Memory consumption. Docker is an open source technology for building, deploying, and executing distributed applications within containers. Docker provides many advantages over container technologies including portability of containers from one machine to another, rapid application deployment, simplified maintenance, and easy sharing of images with others.

Kubernetes [1] is a popular container orchestrator which is built based on Google's Borg [2], a large-scale cluster management system. Kubernetes scheduler schedules the pods (sets of Docker containers) based on the user input given when creating the pods. Scaling decisions are currently done manually through the user input. Replication Controller in Kubernetes ensures that a specified number of pods are running in the cluster at any given time.

As the complexity of software increases, more and more containers will be created and deployed. Currently, there is no way to scale out the containers based on the current system load, and there is a need to find out an optimal way to scale out based on the system parameters. The objective of this thesis is to accomplish the dynamic and targeted scaling

of the pods comprising an application to ensure low latency servicing of requests. The model generated was able to do the scaling based on the system parameters more efficiently, compared to the existing mechanism, and the results demonstrated a significant reduction in latency for requests.

1.1. Challenges

Scaling of pods based on the system load introduces a set of unique challenges:

- **Latencies:** Latencies are higher for compute intensive application in production.
- **Quality of Service:** Addressing scaling in pods without comprising the quality of service (which includes latency and response time).
- **Metrics:** Limited metrics are currently available in the latest version of the container monitoring solution.

1.2. Research Questions

Research Questions that we explored in this thesis are listed below:

- **RQ-1:** *Which features of the system contribute to an increase in latency of the system?*
- **RQ-2:** *How can we improve the scaling decisions based on the features of the system?*
- **RQ-3:** *How can we scale out the pods to ensure low latency servicing of requests?*

1.3. Overview of Approach

The approach described in this thesis is designed to ensure high quality of service with out manually expanding or over-provisioning the pods in a production environment. The approach here is to analyze the system resources and come up with a model that will be used

to schedule the pods. The model generated will predict 99^{th} Percentile latency depending on which decision is made to (scale out or scale in pods). To ensure sufficient load is given to the system, an HTTP load testing tool was used. Also, the application under test is a webserver written in golang, and it consumes both CPU and Memory when invoked. Number of requests sent can be altered to ensure that sufficient load can be generated in the system. A monitoring solution is deployed as a container along with the webserver to get the system statistics.

1.4. Thesis Contributions

Thesis contributions include:

- Scaling out and scaling in decisions are based on 99^{th} Percentile latency predicted by the Model. The threshold for scaling out or scaling in can be modified based on specified requirements.
- Creating a model using machine language techniques (like Random Forest, Linear Regression and Gradient Boosting) and selecting the best model.
- Container metrics are directly obtained from cAdvisor's API, aggregated and fed into the algorithm.

1.5. Thesis Organization

The rest of this thesis is organized according to how the setup was done, experiments were performed, and results obtained. In Chapter 2, the background about the tools used in this thesis is described, and Methodology is presented in Chapter 3. An evaluation of the results obtained and a discussion of the models applied to the data are presented in Chapter 4. Chapter 5 includes a review and discussion about the related work, which mainly focuses

on the previous work done with the placement of VM's. Conclusions and a discussion of future work are presented in Chapter 6.

CHAPTER 2

Background

In this chapter, we describe the tools used for generating the dataset, and the parameters used for creating the model.

2.1. Dockers

Docker is an open-source platform for the deployment of applications inside software containers. Docker containers wrap up a piece of software in a complete file system that contains everything necessary to run: code, run time, system tools, and system libraries [3]. The traditional way of deploying the applications entangles the applications' executable and, configuration, libraries with each other and with the host OS. Figure 2.1 shows the comparison between a Virtual Machine with Hypervisor and Docker Engine with Containers.

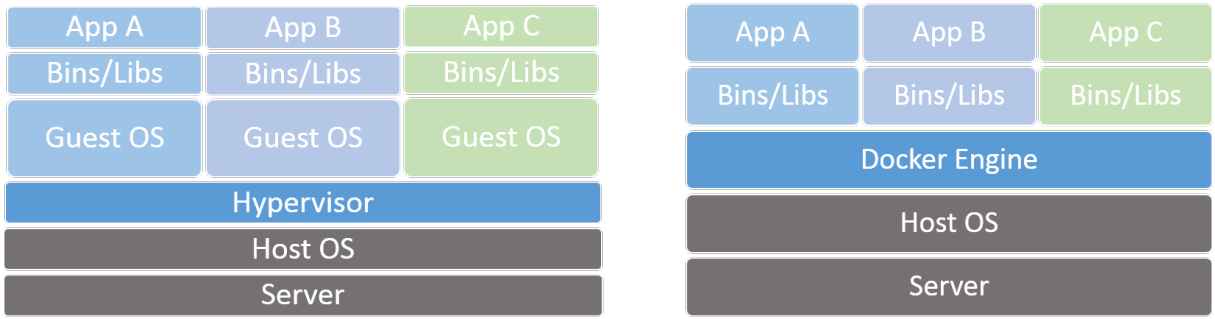

Figure 2.1. Virtual Machine vs Containers.

The Hypervisor does a good job in isolating different Virtual Machines in a single physical server, but VM's are heavyweight. Each VM should have a guest OS installed, and the whole image occupies more space in the disk. Containers on the other hand are light weight, and a single host can have numerous containers deployed and running. Docker uses the resource isolation feature of cgroups, namespace, and aufs to allow independent Docker containers to

run within a single Linux instance. For our thesis, we have installed three hosts with Docker Engine.

2.2. Kubernetes

Kubernetes is an open-source platform which provides a mechanism for deploying, maintaining, and scaling applications [1]. Currently the only supported container framework in Kubernetes is Dockers. Kubernetes and Dockers are becoming popular in cloud environments. Kubernetes provides various types of API, and it is highly configurable, which makes it easy to use.

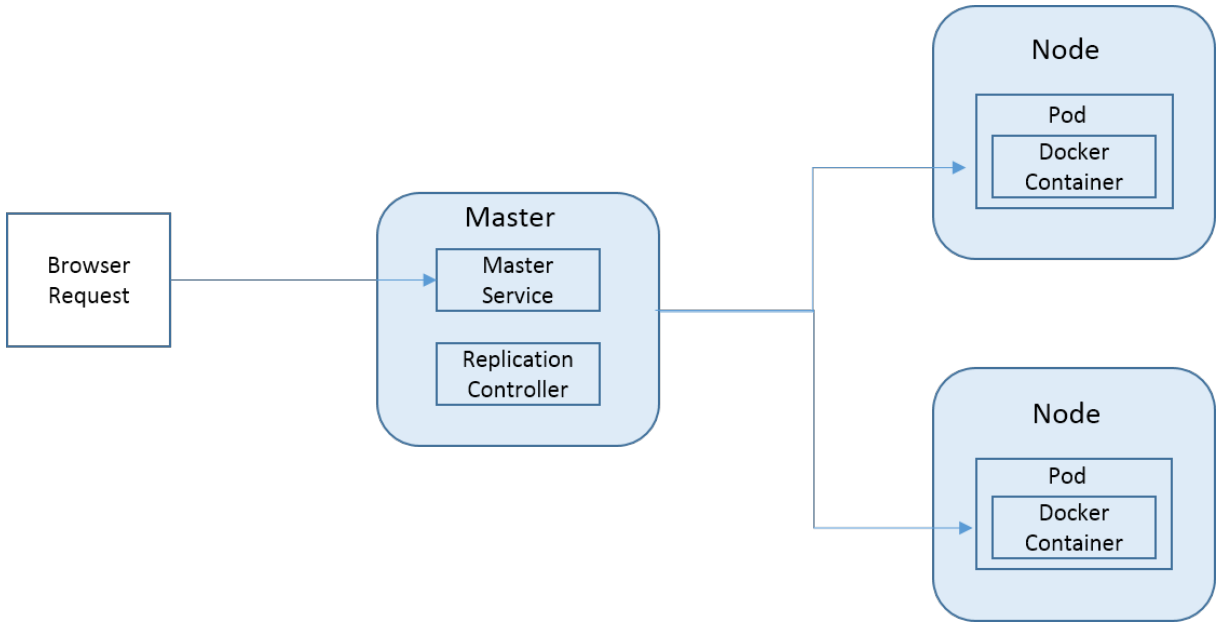

Figure 2.2. Kubernetes Master with Nodes.

Figure 2.2 shows the Kubernetes setup with one master and two nodes. The master runs the replication controller and different kinds of service. The basic scheduling unit in Kubernetes is a pod, which is a collection of one or more containers. Labels in Kubernetes are key-value pairs for any API objects in the system, such as pods or nodes. The replication

controller handles replication of pods and runs a specified number of copies across a Kubernetes cluster. The service consists of a set of pods that work together like a multi-tiered application. The service can be exposed internally or externally so that other pods or clients can access it. The replication controller and services are specified in a yaml file which also specifies the images that have to be deployed. The Figure 2.2 shows a simple Kubernetes setup that has two nodes associated with it. When a web service request comes from a browser, the service in the master node, which acts as a load balancer, redirects the traffic between pods.

2.3. Docker Monitoring

Currently, only a handful of Docker Monitoring solutions are available. Prometheus along with Grafana provides a solution for viewing Docker container stats. Another solution commonly discussed is InfluxDB along with Grafana. InfluxDB which is a NoSQL database, stores the trending data of the containers, and the data can be visualized using Grafana. Both of these solutions can be Dockerized, and they can be deployed as containers using Kubernetes.

All of the monitoring solutions mentioned above use the data given by cAdvisor, which in the current release is natively supported in Dockers, so they should support any containers out of the box. cAdvisor provides information about the resource usage and performance of the containers running. It aggregates, processes, and exports the information about the containers. It also exposes raw and processed statistics for both containers and hosts using REST API. All of the solutions mentioned above add to overhead in terms of load consumed, and this factor will influence the system evaluation.

cAdvisor provides system resource statistics and Network statistics like CPU, Memory, Bytes In/Out, and Packets In/Out. For predicting latencies, we selected nine features from cAdvisor to consider for creating the model (See Table 2.1). Most of the features obtained from cAdvisor are counters. There are other features like Container Page Fault and Container Page Fault Major; these are not considered because they remain at zero most of the time. The current version of cAdvisor supports only a few attributes, and most of the network attributes are empty.

Table 2.1. Features Monitored using cAdvisor.

Features	Description
cpu_usage_total	Total usage of the System
cpu_user	Total usage of the Container
cpu_system	Container Kernal usage
memory_usage	Total RAM usage
memory_working_set	Container RAM usage
rx_bytes_in	Number of bytes incoming
rx_packets_in	Number of Packets incoming
tx_bytes_in	Number of bytes outgoing
tx_packets_in	Number of Packets outgoing

2.4. Load Generator

The HTTP load generator used for this thesis is Vegeta. This tool can be accessed using CLI as well as API. The tool can send http requests at a constant rate, and it gives out various latencies, averaging over time. We have written a wrapper on top of this tool to send requests every 10 secs and to collect the results.Table 2.2 shows the different latencies given by the load generator. It also provides information about the success rate of the requests that are sent.

TABLE 2.2. Latency Attributes provided by HTTP Load Generator.

Features	Description
99^{th} Percentile	Time Taken for 99 Percent of requests
Max Percentile	Maximum Time Taken for the requests
Mean Percentile	Mean Time Taken for all requests
90^{th} Percentile	Time taken for 90 Percent of requests
50^{th} Percentile	Time taken for 50 Percent of requests

Our model evaluates the latency using regression technique based on the system load to predict a latency, which is used to scale out or scale in pods. In most applications, we want to minimize the tail latencies, which correspond to the worst user experience. The 99^{th} Percentile latency gives us a representation of practically the worst case latencies [4].

To summarize, our setup will have one master with two other nodes. Docker Engine will be installed in all of the nodes. The parameters monitored are shown in Table 2.1, and the model will be evaluating 99^{th} percentile latency.

CHAPTER 3

Methodology

The objective of this thesis was to scale out or scale in the pods based on system load. This chapter describes the experiment done using the tools mentioned in Chapter 2. We performed a validation process on the setup mentioned below, obtained the results, and applied them to the model.

The setup used here involves three servers with Kubernetes and Dockers installed. Following is the Kubernetes setup used in this evaluation:

- **Master:** Where all the services and replication controller reside.
- **Node 1:** Initial Pod used for evaluation.
- **Node 2:** Pod which is scaled out or scaled in.

3.1. Data Collection

The application used for the evaluation will be a set of web services that are dockerized as a container and deployed as pods in the Kubernetes cluster. This system is in a controlled environment and will not have any other deployments. All the web services are written in golang programming language. Based on a web service request, a response will be sent to the client. Different types of web services are invoked based on the request type and are described below. The webservice can be accessed through the URL ($http://serviceip:8000/PerfMonitoring$).

- **PerfMonitoring:** Calculates the Fibonacci series for numbers ranging from 25 to 35. Each request is assigned a random number between 25 and 35, and then requests are serviced. In golang the approximate time required to calculate the Fibonacci series for the number 35 is around 100ms, calculation time for the number

30 is around 7ms.The time taken for each request varies depending on the number selected. Each request is allocated a temporary variable, which is killed once the request is complete.

- **BaselineMonitoring:** Calculates the Fibonacci series for requests only for the number 33. This is to give a constant load to the system so that scaling in or scaling out validations can be done.

The load testing tool sends requests to the PerfMonitoring webservice at a constant rate and returns the result in terms of latency, which is averaged over 10 secs. The requests are sent from 100 workers in parallel from the load testing tool. One hundred connections are maintained from the client to the server. The monitoring tool's API is invoked every 10 seconds to get the system stats. Load testing tools' request rates vary from 1000 to 1500 requests per second. The average latency within a 10-second period is obtained from the load testing tool. This value is used as a predictor in our model. Similarly, for every 10 seconds, statistics from cAdvisor are obtained. Load testing is run for 30 hours, and the data is collected every 10 secs.

3.2. Machine Learning Models Applied

Data collected from the system is normalized, shuffled, and three models are trained using 70-30 testing/training method. The following machine learning techniques were used for the evaluation:

- **Random Forest:** Ensemble learning method for classification and regression, which operates by constructing a multitude of decision trees.
- **Gradient Boosting:** Ensemble of weak prediction models, typically decision trees.

- **Linear Regression:** Approach for modeling the relationship between a dependent variable and an independent variable.

Models obtained using these methods are compared, and the best model is chosen based on mean absolute error and mean squared error. The model chosen will be getting the system status for the pod from cAdvisor at a specific interval. Also the model obtained will connect to the Kubernetes API if the latency obtained crosses the specified threshold.

3.3. Finding Steady State of the System

For validating scale out and scale in decisions, we have to bring the system to a steady state by sending a fixed number of requests to the container and maintaining the number of connections to the server. Each request consumes a connection to the server and is alive until the request is processed. Both the server and the client are configurable in terms of number of requests as well as the computation and memory utilization by the server. The processing done from the server side remains constant for all the requests. Web service BaselineMonitoring is accessed, which calculates the Fibonacci series for number 33. This includes creating some objects during the process and killing these objects after the request is completed. Figure 3.1 shows the system latency with various loads.

In Figure 3.1, we can see that the latency increases as the load increases. Here the load is generated using the number of requests that are sent using a fixed number of TCP connections from the client. Since scaling out or scaling in a pod is based on the system load, we are not interested in request rates. The intent of increasing the request rate is to increase the load on the system.

Any response time that goes beyond a certain threshold is considered to be a delay, in which case our model will initiate scaling out of the pod.

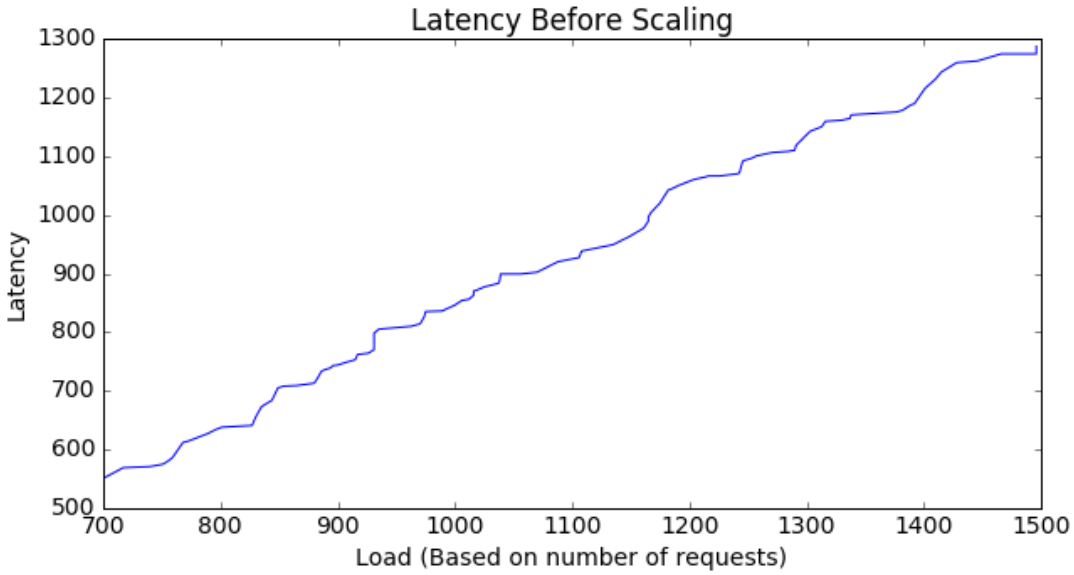

FIGURE 3.1. Latencies for different loads before Scaling.

For scaling in the pods, we continuously decrease the request rates so that the system load decreases and the latency of requests also decreases. We started with a request rate of 1400 requests/sec, which is load balanced between two pods. When the request rates are reduced drastically, the load on the system decreases, and the latency decreases. Using the calculation below, we can calculate the total latency of the pods and kill one pod.

$$\text{Latency Pod 1} + \text{Latency Pod 2} < 450ms : \text{ kill one pod.}$$

The threshold for scaling in or scaling out pods are user configurable, and it can be set to any value. For our testing purpose, we have configured the scaling out threshold to 900ms and the scaling in threshold at 450ms. The model calls the Kubernetes replication controller API to accomplish the scaling in and scaling out of pods.

3.4. Testing Scaling Out Pods

Once a new pod is created, the service in the master load balances the traffic that is coming in between the two pods. Our model will start monitoring the second pod created in another node.

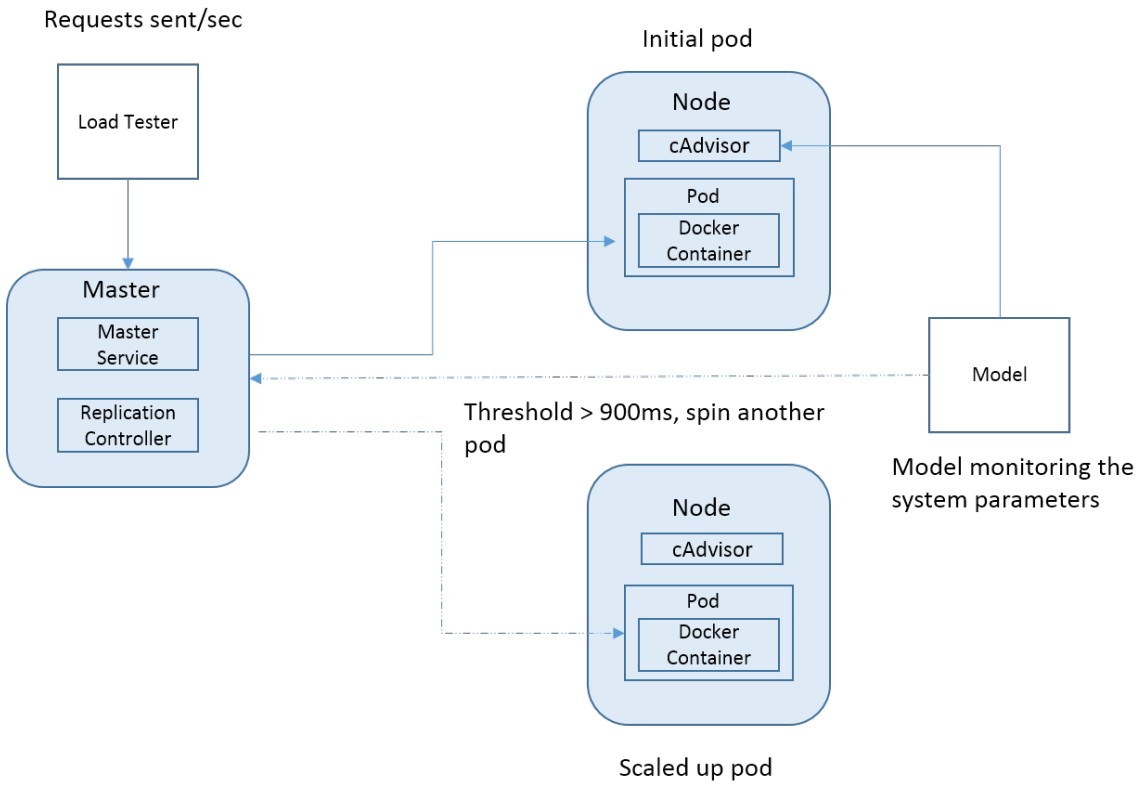

Figure 3.2. Scaling out Pods

Figure 3.2 shows how the pods are scaled out when the threshold is greater than 900ms. We have forced the pod creation to another node for this experiment so that Node 1 is not overloaded.

3.5. Testing Scaling In Pods

Once the latency is below 450ms, all of the traffic from Node 2 will be redirected to Node 1 by the master service. Our model will stop monitoring the second pod once the traffic

is redirected. Figure 3.3 shows how the pods are scaled in when the threshold goes below 450ms.

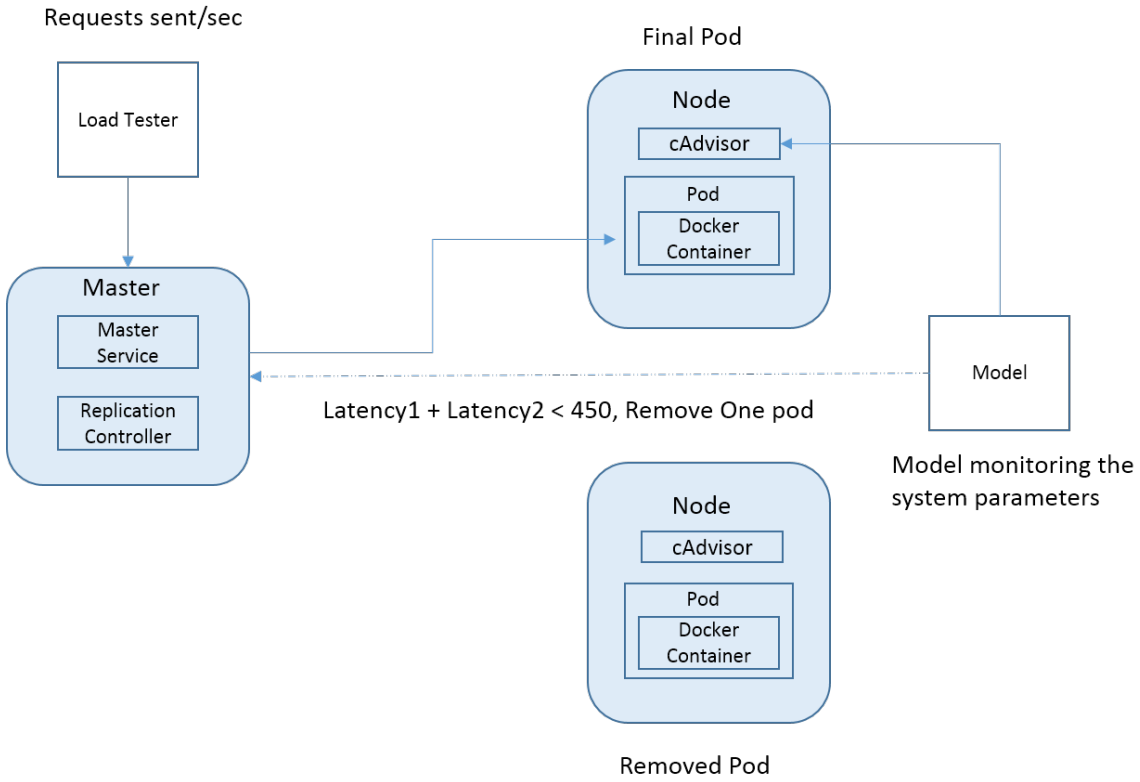

FIGURE 3.3. Scaling in Pods

In summary, this chapter contains the method used to collect the data, models applied to the data, processes for testing the system with constant load, and thresholds used to make decisions to scale out or scale in.

CHAPTER 4

Evaluation

In this chapter we discuss the models applied to the dataset, compare their performances, and select the best one. After running the test for 30 hours we collected 10,800 data points for different sets of loads. The model is currently installed in a separate machine which resides in the same environment.

4.1. Mean Absolute Error and Mean Square Error

Mean Absolute Error(MAE) is a quantity that measures how close predicted values are to actual values. Mean Square Error(MSE) measures the average of the squares of the errors, which is the difference between the estimator and what is estimated. The formula for MAE and MSE are calculated with the following equations:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where,

$$Actual = y_i$$

$$Predicted = \hat{y}_i$$

Table 4.1 shows the MSE and MAE values obtained using the current model for 70-30 training-testing.

Table 4.1. MAE and MSE.

Models	MAE	MSE
Random Forest	145.29	48997.60
Linear Regression	155.02	49494.79
Gradient Boosting	156.16	51310.388

From table 4.1, we can see that Mean Absolute Error and Mean Squared Error are low for Random Forest.

4.2. Regression Error Characteristic Curves

The Regression Error Characteristic Curve (REC) plots the error tolerance on the x-axis and the percentage of points predicted within the tolerance on the y-axis. Users can quickly assess the relative merits of many regression models by visually checking the curve. Figure 4.1 shows a plot of the REC curve for 99^{th} percentile latency.

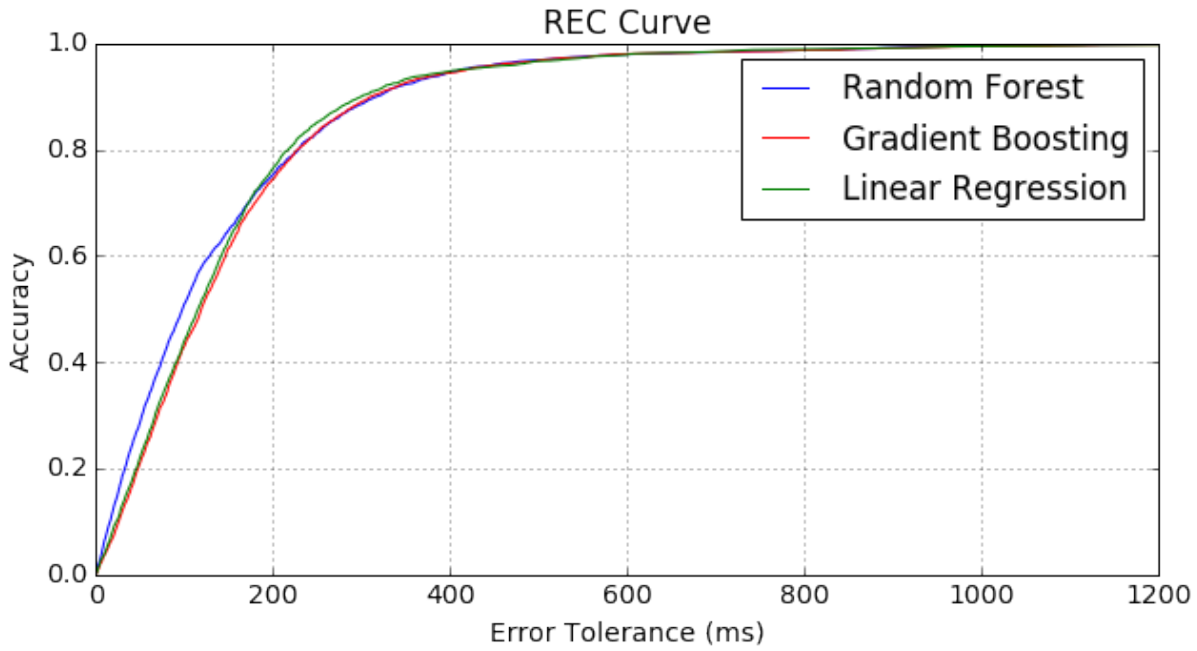

Figure 4.1. Regression Error Characteristic Curve

Accuracy appears to be higher for the Random Forest model compared to the other two regression models. Area Over the Curve (AOC) is the measure of the expected error for a regression model. If the AOC is smaller, then the model performs better. As the error tolerance increases, accuracy increases and finally touches one. Also we can see that 80% of the requests are serviced within 200ms. From Figure 4.1, we can see that Random Forest performs better when compared to other models.

4.3. Scaling Out Pod

Since Random Forest results in lower MAE when compared to the two other models, we used Random Forest to scale the pod based on the specified thresholds. Figure 4.2 shows the latency of the system before and after scaling out. The scaling out test is run by constantly increasing the load so that latency is continuously increased.

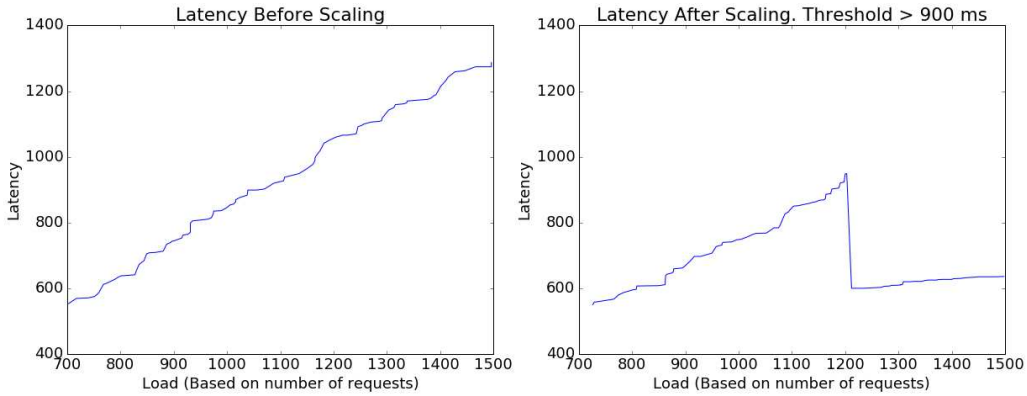

Figure 4.2. Latencies before and after Scaling Out Pods.

From Figure 4.2, we can see that the scaling out threshold is set at 900ms. In this case, the error was around 54ms, and the model started scaling down when the threshold reached around 954ms. Once Kubernetes created another pod, the requests were sent to both pods by the load balancer service, and the latency dropped to less than 600ms, gradually increasing after that.

4.4. Scaling In Pod

Figure 4.3 shows the latency of the system before and after scaling in. The scaling in test is run by constantly decreasing the load so that the latency is continuously decreasing.

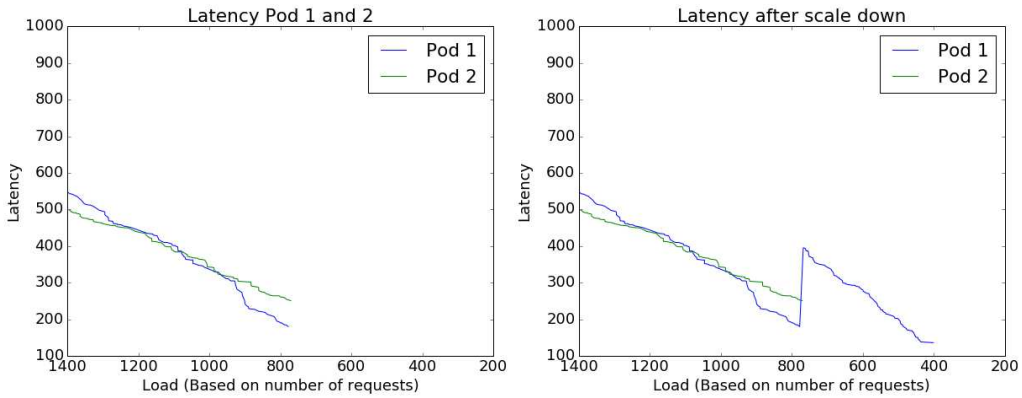

Figure 4.3. Latencies before and after Scaling In Pods when threshold was 395ms.

From Figure 4.3, we see that as load in the system decreases, the latency also decreases. When the cumulative latencies of both pods are less than 450ms, pod 2 is killed, and pod 1 starts processing the requests. The actual latency at which the pod got killed was 395ms.

Figure 4.4 shows the same test run with the pod scaling in at a threshold of 485ms.

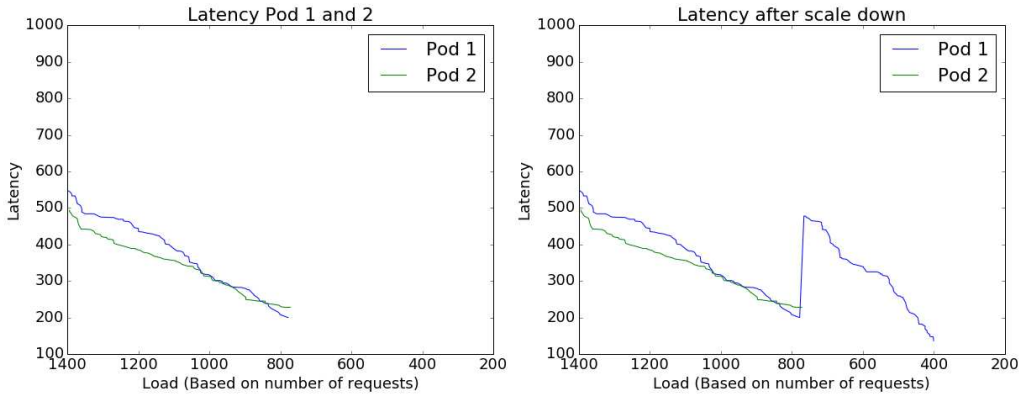

Figure 4.4. Latencies before and after Scaling In Pods when threshold was 485ms.

Scaling in pods is done by comparing latencies from two pods with the threshold set at 450ms, which is half of the scaling out latency. This is to ensure no delays when the traffic is redirected to one pod.

4.5. Feature Importance

Figure 4.5 shows the results for feature importance obtained from the Random Forest Regression algorithm. The cumulative total of feature importance values is 1.

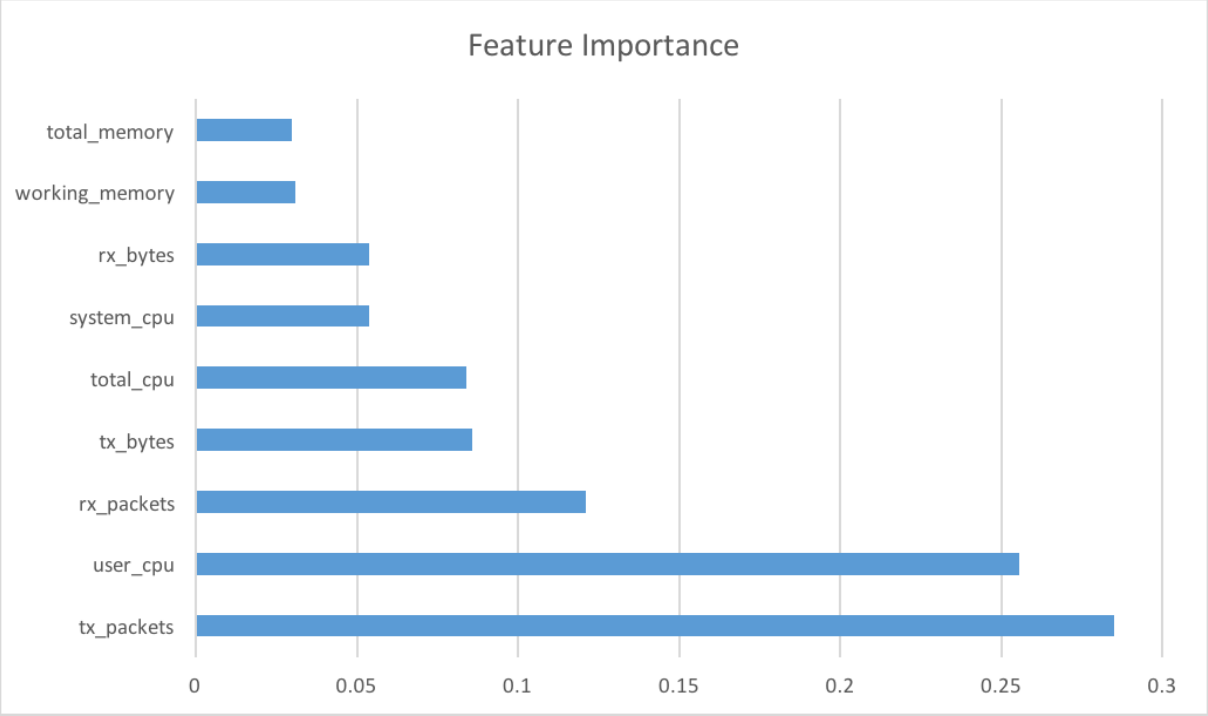

Figure 4.5. Feature Importance from Random Forest.

Packets transmitted and user CPU were the two strongest predictors of the latency compared to other features. Features like total memory and working memory didn't have much influence on the overall system. Also, total CPU and system CPU didn't have much influence on predicting latency.

CHAPTER 5

Related Work

Several methods have been proposed for the placement of VMs based on various strategies. Energy consumption has always been a problem in cloud computing, and this work and, many other algorithms for VM placement, have proven to reduce energy consumption, thereby reducing the operating cost. Rahimi, et al. introduced a new priority routing VM placement algorithm, and they compared it with an existing algorithm that uses a best fit algorithm PABFD (power-aware best fit decreasing) [5].

Meng, et al., proposed a VM placement strategy based on the network latency and congestion situation to maintain the application performance [6].The main idea here is to reduce the data access time by placing the VM at appropriate physical machines. Meng, et al., proposed a VM consolidation as a novel random variable packing problem that models the bandwidth demands of VMs as a probabilistic distribution [7]. The authors talk about the VM consolidation problem when network devices impose bandwidth constraints.

Docker Swam container scheduling is based on three strategies: spread, binpack and random, and it computes the rank of available nodes for placement of containers [8] based on the chosen strategy.The first two strategies consider CPU and RAM for computing the rank, but the third strategy is completely random. Kubernetes scheduling of pods is done by a three step process [9].First the scheduler applies the predicates to a filter, secondly it sets the priority functions that rank the nodes, and finally it schedules the pod on an appropriate node.

Gupta et al., proposed an enhanced VM placement mechanism on top of openstack for High Performance Computing Environments (HPC), which characterizes applications along

two dimensions, Cache-intensiveness and Parallel Synchronization, and Network Sensitivity [10]. The experimental results for the sample application indicated a 45% improvement in HPC performance and a 32% increase in throughput.

Lloyd et al., proposed an approach to maximize performance in a multi-tier application deployment in Cloud [11] [12] [13]. These researchers address maximizing the service compositions of multi-tier applications deployed to IaaS cloud. This empirical study investigates the implications for application performance and resource requirements that results from applications deployed in IaaS clouds. The authors have developed a model based on resource utilization and performance to predict application deployment performance.

Our proposed approach is more flexible because it considers the load on the system for scaling up and scaling down of the pods. The tests that were run showed that scaling up is more efficient, and it reduces the latency of the requests dramatically.

CHAPTER 6

Conclusion and Future Work

6.1. Conclusion

Creating a model for scaling up containers in a low latency environment provides optimal resource utilization, thereby increasing quality of service. The model created predicts the 99^{th} Percentile latency rather than mean latency for the request, which is considered as an acceptable parameter in a low latency environment.

RQ1: User CPU utilization and Packets transmitted were the strongest predictors for scaling decisions; whereas, memory and its related statistics were much less useful. Also, other CPU parameters like overall CPU utilization and kernel CPU utilization did not have much impact on the system.

RQ2: The model created by Random Forest predicts the latency of the system based on the system load, which includes all of the system features. Based on the predicted latency, the decision can be made to scale out or scale in the containers.

RQ3: Pods can be scaled out when the threshold is crossed, which will decrease the latency for the requests. If the aggregate of the latencies is less than half the threshold (scaling out threshold), pods can be scaled in. The threshold is user configurable, and it can be changed according to user requirements.

6.2. Future Work

As future work, we propose a different approach for scaling in and scaling out of pods in the Kubernetes environment. The first approach involves aggregating the system resources across all of the containers by considering both node wise system utilization and consolidated system utilization to generate a model and predict the scaling of pods. This involves making

changes to the current monitoring solution to aggregate the results for each container. The current monitoring solution, cAdvisor, provides only a few parameters and has a place holder for numerous other system parameters. Once more features are added to the monitoring solution, our model can be trained to consider those parameters. Another approach involves having the model built into the Kubernetes environment itself for inter containers communication inside the cluster. When the request rates are higher, we can scale out containers' based on request rates instead of latency.

References

[1] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*. 2015.

[2] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, (Bordeaux, France), 2015.

[3] "Executive summary — the definitive guide to docker containers. the definitive guide to docker containersexecutive summary — the definitive guide to docker containers (n.d.): n. pag.3."

[4] "Treat, author tyler. "everything you know about latency is wrong." brave new geek. n.p., 12 dec. 2015. web. 12 oct. 2016.."

[5] A. Rahimi, L. M. Khanli, and S. Pashazadeh, "Energy efficient virtual machine placement algorithm with balanced resource utilization based on priority of resources," *Computer Engineering and Applications Journal*, vol. 4, no. 2, pp. 107–118, 2015.

[6] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM, 2010 Proceedings IEEE*, pp. 1–9, IEEE, 2010.

[7] M. Wang, X. Meng, and L. Zhang, "Consolidating virtual machines with dynamic bandwidth demand in data centers," in *INFOCOM, 2011 Proceedings IEEE*, pp. 71–75, IEEE, 2011.

[8] "Docker swarm. docker, 11 oct. 2016, https://docs.docker.com/swarm/.."

[9] "kubernetes kubernetes/kubernetes. github, https://github.com/kubernetes/release-1.3/docs/devel/scheduler.md.."

[10] A. Gupta, L. V. Kale, D. Milojicic, P. Faraboschi, and S. M. Balle, "Hpc-aware vm placement in infrastructure clouds," in *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pp. 11–20, IEEE, 2013.

[11] W. Lloyd, S. Pallickara, O. David, M. Arabi, T. Wible, and J. Ditty, "Demystifying the clouds: Harnessing resource utilization models for cost effective infrastructure alternatives,"

[12] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, "Performance implications of multi-tier application deployments on infrastructure-as-a-service clouds: Towards performance modeling," *Future Generation Computer Systems*, vol. 29, no. 5, pp. 1254–1264, 2013.

[13] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, "Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds," in *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pp. 73–80, IEEE, 2012.