DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# Performance Model Derivation for Cloud-based Microservices Applications

Anshul Jindal

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# Performance Model Derivation for Cloud-based Microservices Applications

# Leistungsmodellableitung für Cloud-basierte Microservices-Anwendungen

| | |
|---|---|
| Author: | Anshul Jindal |
| Supervisor: | Prof. Dr. Michael Gerndt |
| Advisor: | Vladimir Podolskiy |
| Submission Date: | 11th October 2018 |

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 11th October 2018 Anshul Jindal

# Acknowledgments

First of all, I would like to express my deepest sense of gratitude to my advisor Vladimir Podolskiy, who offered his continuous advice and encouragement throughout the course of this thesis. I also thank him for the systematic guidance and great effort he put into training me in the scientific field.

I would also like to express my very sincere gratitude to Prof. Dr. Michael Gerndt for his support and guidance to make this thesis possible.

This work was carried on AWS cloud for which credits were provided by them for conducting the research.

# Abstract

Microservices application being a distributed system allows deployment of individual services to physically separated different or same cloud virtual machine (VM) instances. Each microservice is responsible for completing their part and communicate with others through language and platform-agnostic application programming interfaces (APIs). These microservices when deployed on a VM uses many system resources (e.g., CPU, memory, disk I/O, and network I/O) to process user requests. The resource usage varies according to the type of task (CPU intensive, memory intensive or Create Read Update Delete (CRUD)) microservice's is doing. The resources usage change with the variation in the number of user requests or the user workload. Depending on the type of work, particular resource usage will have a more significant effect than others. As a result, at a certain stage, the response time of requests would go beyond the desired time. The maximum number of requests that could be served within this time is called as the Maximum Service Capacity (MSC) of the microservice. Performance of a microservice is directly related to the MSC. Automatically detecting MSC for each microservice can be challenging in practice as microservices applications are deployed using multiple abstraction layers. These numerous abstraction layers result in additional overhead for the indirect usage of hardware resources and also obscuring the run-time details. This challenge grows more with the availability of the different deployment configurations (e.g., Virtual machine type, Cloud service provider, deployment strategy, etc.). MSC varies with varying configurations of deployment.
This thesis aims to address this problem of identifying the MSC for each microservice of the application by building the performance model of it in all the possible deployment configurations. A novel approach to microservice's performance modeling and sand-boxing microservices from an application is introduced in this research. This approach is implemented in a tool called as Terminus. The proposed tool predicts the MSC of the microservice on different deployment configurations by training a model from the conducted tests. The derived results from the tool could also be used to find out the bottleneck microservice along with the number of microservice replicas needed for achieving adequate performance.

# Contents

# 1 Introduction

Cloud computing is a type of Internet-based computing that provides shared computer processing resources and data on demand. In recent times, it has rapidly become a popular method for deploying applications [30]. This success is based on the virtualization technology that considers hardware as an unlimited pool of resources for the users [33]. The requested amount of CPU cores and memory are provided on demand from this pool of resources, hence helping in to get on-demand resources and cost efficiency for the applications [16]. Previously, the applications were used to be packaged and deployed as a monolith. These applications architecture is called monolithic architecture. Although, in the early stages of the business these applications may work but once the business starts to grow the applications also grow, as a result, the problems like scalability, redeployment, reliability etc. start to arise. With the introduction of microservices architecture applications, where an application is split into a set of smaller and interconnected services instead of building a single monolithic application these problems are solved. Also, the introduction of new virtualization layers such as containers and pods has turned the deployment and management of cloud applications into a simple routine. These microservices architecture applications are deployed using such virtualization layers to keep the structure scalable and manageable. Microservices, when deployed using different types of virtualization layers, perform differently. They use many system resources (e.g., CPU, memory, disk I/O, and network I/O) to process user requests. The resources usage varies according to the type of task (CPU intensive, memory intensive or Create Read Update Delete (CRUD)) a microservice is performing. The resources usage change with the variation in the number of user requests or the user workload. As a result, at a certain stage, the response time of requests would go beyond the desired time. The maximum number of requests that could be served within this time is called as the Maximum Service Capacity (MSC) of the microservice. Apart from having many advantages of such applications, they also impose several important challenges:

- Microservices application being a distributed system adds more complexity to the project as an inter-process communication mechanism based on either messaging or Remote Procedure Call(RPC) needs to be handled.
- Testing of such an application.

- Deployment of such an application is more complex as it typically consists of a large number of services and each service can have multiple instances which can be individually configured, deployed, and scaled.
- Performance of individual microservices and identifying the bottleneck microservice.

The required first step to build a good microservice application is to determine and understand the performance of each microservices when deployed on a particular type of virtual machine. MSC is directly related to the performance of the microservice and determining MSC means determining the performance of the microservice. The measured performance would lead to understanding the major architectural drawbacks both for individual microservices scaling and their combinations. Once the individual microservice performance is determined then the overall application performance can be improved by scaling the services that need scaling instead of scaling the whole application. However, automatically detecting MSC for each microservice can be challenging in practice as microservices applications are deployed using multiple virtualization layers. These numerous virtualization layers result in additional overhead for the indirect usage of hardware resources and also obscuring the run-time details. This challenge grows more with the availability of the different deployment configurations (e.g., Virtual machine type, Cloud service provider, deployment strategy, etc.). MSC varies with varying configurations of the deployment. Also, microservices need to be sandboxed from the application to determine the actual MSC for them.
The key contribution of this thesis is the approach and a tool to solve the problem of deriving the performance model for a microservice. Also, an approach to sandbox microservices is presented and implemented in the tool. Understanding the performance of microservices inside an application paves a path for understanding the structural and scalability aspect of the application. Leaving aside pure research challenges, the developed tool, namely Terminus, could also be used to predict the MSC of the microservice on different deployment configurations and also predicts the number of replicas required to handle a given number of requests when deployed using a certain deployment configuration. The derived results from the tool could also be used to find out the bottleneck microservice along with the number of microservice replicas in an application needed for achieving adequate performance.

# 2 Background

## 2.1 Multiple layers of cloud virtualization

In general, there are two primary virtualized abstraction layers being used in the cloud. The first type of layer allows servers to be broken up into virtual machines (VMs) that can run same or different operating systems and a minimum amount of 1 CPU can be provided to the end-user, therefore increasing the utilization potential of a physical server for multiple users. On the other hand, a container which is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, run-time, system tools, system libraries, settings - offers an alternative way of virtualization. Using containers CPU as little as 0.1 CPU can be provided to the end-user, therefore increasing further the utilization potential of a physical server. CPU shares are used, instead of reserving specific CPUs for such fractional CPU reservation. This means that, if 0.1 CPU is reserved, then the process will be allowed to use a total of 0.1 seconds of CPU time each second and if 2 CPUs are reserved, then the process will be allowed to use a total of 2 seconds of CPU time each second by utilizing more than one CPU per core. A Container that requests 0.1 CPU is guaranteed one-tenth as much CPU as a Container that requests 1 CPU. Using container-based virtualization, a single operating system on a host can run multiple cloud services [17]. Docker is a widely adopted container-based virtualization [26]. Docker creates a set of namespaces for a container to provide isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace only [10]. Docker also uses linux control groups (cgroups) to enforce limits and constraints on an application to a specific set of resources. To run containers in production at scale, a container orchestrator such as Kubernetes is needed to efficiently schedule and orchestrate containers on the underlying shared set of physical resources [24]. In Kubernetes, containers are grouped into pods. A pod is a group of one or more containers that share storage and network resources as well as the specification on how to run the containers.
However, there is no control over how much resources each pod can use. Some images might be more resource heavy or have certain "minimum resource" requirements. Also, if different teams are running different projects on the same cluster, there is no control on how much resources each team can use. Within Kubernetes, by default, a pod will run with no limits on CPU and memory in a default namespace causing several

problems related to contention for resources.
This is where container resource limits and resource quotas come in which can control the amount of CPU and memory resources per pod. The setting of resource requests and limits can be done in the pod configuration file. Within the pod configuration file, CPU and memory are each a resource type for which constraints can be set at the container level. A resource type has a base unit. CPU is specified in units of cores, and memory is specified in units of bytes. Two types of constraints can be set for each resource type: requests and limits as shown in Fig. 2.1. CPU requests constraint

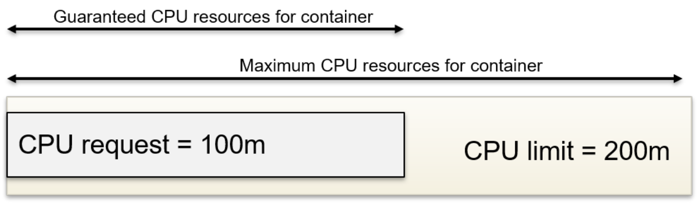

Guaranteed CPU resources for container
Maximum CPU resources for container
CPU request = 100m
CPU limit = 200m

Figure 2.1: Resource requests and limits in Kubernetes.

provides minimum guaranteed CPU resource for a container and if the container running process is able to utilize more than its share, and no other task would use an otherwise idle CPU, the first task can potentially use more than its share till the specified limits. The upper limit is usually achieved by throttling the process when it fully consumes its allocated CPU time. As a result, the CPUs reserved provide a guaranteed minimum of CPU time available to the container and if additional capacity is available, it will be allowed to use more. Kubernetes uses request constraint to decide on which node to place a pod [23, 2]. Setting request less than limits allows some over-subscription of resources i.e temporarily use unused/idle resources as long as there is spare capacity on the node. This allows the Kubernetes scheduler to make better decisions on handling contention for resources on a node.

## 2.2 Cloud based microservice applications

Scalability has become the major selling point of modern cloud infrastructure and services [34, 27]. This scalability allows enterprises to scale their applications on demand. Microservice design for applications is a new basis for cloud application development and has gained popularity due to its fine granular design and loosely coupled services unlike monolithic design with a single code base. These microservice applications result in better scaling and flexibility. Microservices application being a distributed system allows deployment of individual services to physically separated different or same cloud virtual machine (VM) instances. Each microservice is responsible for completing their part and communicate with others through language and platform-agnostic Application Programming Interfaces (APIs). These APIs are typically exposed as Representational State Transfer (ReST) endpoints. Each microservice behaves as an independent, autonomous process with no dependency on other microservices. Microservices applications have an advantage that instead of launching multiple instances of the whole application, it is possible to scale-in or scale-out a specific microservice on-demand. This way it is a cost-efficient solution. For such type of applications, virtual machines of Cloud Service Providers (CSPs) could become inconvenient either because of coarse granularity of VMs or because of the inconvenience of microservices management on the levels of VMs and their groups. Therefore, the new cloud abstraction layers, such as containers, pods, and clusters are used for deploying them. Such an abstraction not only helps businesses and individuals to scale their applications in the cloud fast but also makes management of cloud application components easy [21, 25].

## 2.3 Microservices deployment strategy

A microservices application consists of many services. Services are written in a variety of languages and frameworks. Each one is a mini-application with its specific deployment, resource, scaling, and monitoring requirements [32]. Following subsections briefly describe some different microservices deployment strategies.

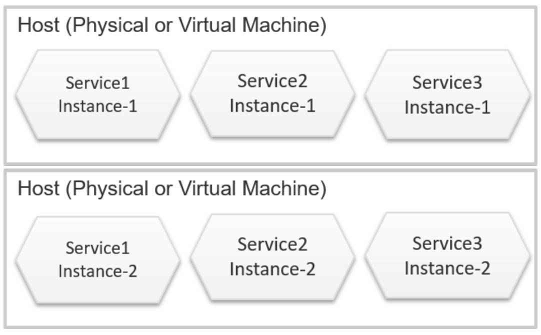


Figure 2.2: Multiple services instances per host (without containers)

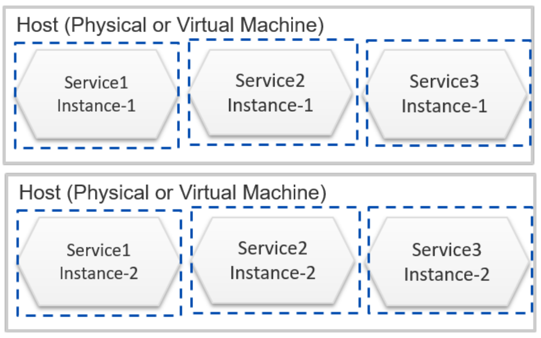


Figure 2.3: Multiple services instances per host (with containers)

### 2.3.1 Multiple services instances per host pattern

Here one or more physical or virtual hosts are provisioned, and multiple services instances are run on each one of them. This can be done in two ways:

1. Without Containers: Services are directly deployed on the host as shown in Fig. 2.2.
2. With Containers: Services are packaged into containers and then deployed on the host as shown in Fig. 2.3.

This type of pattern is used to take advantage of the full utilization potential of the host server by running multiple services. If any service is not using its share of resources to a full extent then the others can use these idle resources. However, the Quality of Service (QoS) is not guaranteed as there are no dedicated resources to the services. If used without the container, scaling a microservice means the replication of the whole VM and unnecessary scaling of the other services, in turn, wasting resources and money. The packaging inside the containers allows the easy manageability and scalability of the services.

### 2.3.2 Service instance per host pattern

Here each service instance is run in isolation on its own host. This also can be achieved in two ways:

1. Without Containers: Each Service instance is packaged as the virtual machine image, and then that image is used to start the host. Therefore each host represents a service as shown in Fig. 2.4.
2. With Containers: Service instances are packaged into containers and then deployed on different hosts as shown in Fig. 2.5.

This type of deployment allows the guaranteed QoS to the services at a cost of some idle resources if the service's is not using the complete dedicated resources. Further, if deployed without using container scalability of each microservice is easier as the instance needs to be replicated.

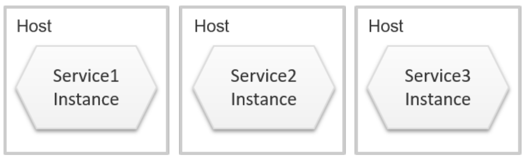


Figure 2.4: Service instance per host (without containers)

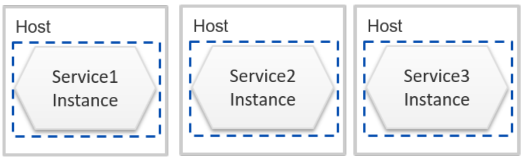


Figure 2.5: Service instance per host (with containers)

### 2.3.3 Serverless deployment

In this deployment strategy, a microservice is packaged into a ZIP file or a container image and uploaded to some framework like AWS lambda [6]. Along with it, some metadata is also specified like the name of the function that is to be invoked to handle a request. The framework automatically runs enough instances of the microservice to process requests without user worrying about any aspect of servers, virtual machines, or containers. The billing is done for each request based on the time taken and the memory consumed by it.

## 2.4 Service Virtualization

Nowadays, to solve the complexities of businesses many interdependent microservices based cloud applications are used. For developing such type of applications, developers may face some common problems, e.g., all the services of the application may not be developed at the same time or some of the services might be down, or one might want to test each microservice independently. Such problems if not encountered could impact the delivery time of the application. One solution could be service mocking using some popular mocking frameworks. However, it has a drawback that mocks are scenario specific and require a lot of effort to create a mocked response for those services. Other solution could be to use stubbed services, where fake services are developed with fixed responses – again here something needs to be developed to make it work.

Since both mocking and stubbed service has few problems, so to solve the above problems, a technique called service virtualization is used. Service virtualization is a method to emulate the behavior of specific components by virtualizing the services. This virtualization of services is done using capturing and simulation of actual services. Hoverfly is one such tool developed in Golang [14].

# 3 Literature Review

These days most of the applications are being hosted in the cloud; therefore performance evaluation and monitoring of the applications play a crucial role. Rocco et al. have introduced an agent-based monitoring system [3] for monitoring the cloud applications. Also, the Cloud Service Providers (CSPs) provide services like Amazon CloudWatch [1] from Amazon AWS, Azure monitor [28] from Microsoft for monitoring cloud applications and cloud resources (e.g., virtual machines). However, these monitoring solutions only monitor the resources and then let the user figure out if there is any performance issue or not. For example, Joydeep et al. [29] have claimed that the performance delivered by the AWS compute platform is quite unpredictable when running web applications. The user deploys the application without considering this variable performance; as a result, it might cost him some money. Therefore the user must know before deploying approximately what performance will be achieved if he or she deploys the application on a VM. There are some performance management solutions like CloudMonix [8] which continuously monitors the utilization of the various resources and send a notification once they are saturated, but these solutions do not derive a general conclusion about what performance will be achieved when the application uses different types of resources and deployment configurations. The user has to test the application on different deployment configurations to find the performance of the application on them. Our approach focuses on this problem and derives the performance model of the application when deployed using different types of resources and deployment configurations.
According to Google SRE book [7], there are four golden signals by which a problem can be identified: latency, traffic, errors, and saturation. Latency is the time taken by the deployed system to service a request. It's important to distinguish between the latency of successful requests and the latency of failed requests as failed requests may have less latency due to errors. Also, a slow error is worse than a fast error. Traffic is a measure of how much demand is being placed on the system, for a web service, this measurement is usually HTTP requests per second. Understanding the errors and error rate can help in identifying the real cause for the errors. For example, errors could arise due to either some policy (any request taking time over one second is an error) or explicitly (e.g., HTTP 500s) or implicitly (a success response, but coupled with the wrong content). Saturation tells how full the service is. Identifying utilization target

is essential as systems degrade in performance before they achieve 100% utilization. Latency increases are often a leading indicator of saturation. Identifying the traffic and saturation point can be helpful with predictions of impending saturation i.e when the service will get full. Our research focuses on latency to identify the saturation point of the service and building the performance model to predict the saturation point for different deployment configurations.

Along with the performance evaluation, detection of bottleneck component for the deployed application is essential. Bhuvan et al. presented an analytical model [35] for bottleneck detection and performance prediction of multi-tier applications. It predicts system performance based on burst workloads and then, determines how much resources need to be allocated for each tier of the application for the target system response time. On the other hand, method like [25] have used the capability profiling to identify the potential resource bottlenecks and make recommendations regarding the resources required for achieving adequate performance. Some systems detect the performance anomalies in the deployed system like Jayathilaka et al. presented Root [18], a system for automatically identifying the root cause of performance anomalies in web applications deployed in Platform-as-a-Service (PaaS) clouds. However, all the above methods do not give the user a beforehand insight of what approximately the performance can be achieved if specific deployment configuration is used. Also, none of them has considered the microservice applications and derived the performance model for them in different deployment configurations. Our research focuses on microservice applications and derives the performance model of it under different deployment configurations. Also, the developed tool is not only limited to performance derivation but can also detect the bottleneck microservice in the whole application.

# 4 Performance Modeling of Microservices

## 4.1 Performance of microservices

### 4.1.1 Quality-of-Service (QoS)

Quality-of-Service (QoS) management for any application deployed on the Cloud plays an important role. QoS denotes the levels of performance, reliability, and availability offered by an application and by the platform or infrastructure that hosts it. QoS is fundamental for the cloud users as well as for the CSPs. As part of this research there are two QoS parameters considered :

1. Request Success Rate (RSR): It is the degree to which the number of user requests sent is successfully completed within a time period. It is calculated based on the equation (4.1)

$$\text{Request Success Rate (RSR)} = \frac{\text{Number of Requests Successfully Completed}}{\text{Total Number of Requests Sent}} \tag{4.1}$$

2. Mean Response Time (MRT): It is the mean for all the requests response time within a time period. Some of the applications are highly interactive and clients typically have strict expectations about response time. Therefore for such applications, MRT plays an important role in determining the performance of an application.

### 4.1.2 Maximum Service Capacity (MSC)

Microservices of an application when deployed on VMs use their resources (e.g., CPU, memory, disk I/O, and network I/O) to process user requests. The resources usage change with the variation in the number of user requests or the user workload. A resource gets fully used by a service when the number of user requests or the user workload increases beyond a threshold number. At that point, some user requests might take time beyond MRT specified by the user to get processed and are therefore returned with an error. The maximal number of user requests per second that can be successfully processed by the service such that the requests response time stays below the MRT is called as the Maximum Service Capacity (MSC). It should be noted that it's not always true that CPU utilization will be 100% when the RRT goes beyond the specified MRT and vice versa. For example, consider a service with the number of

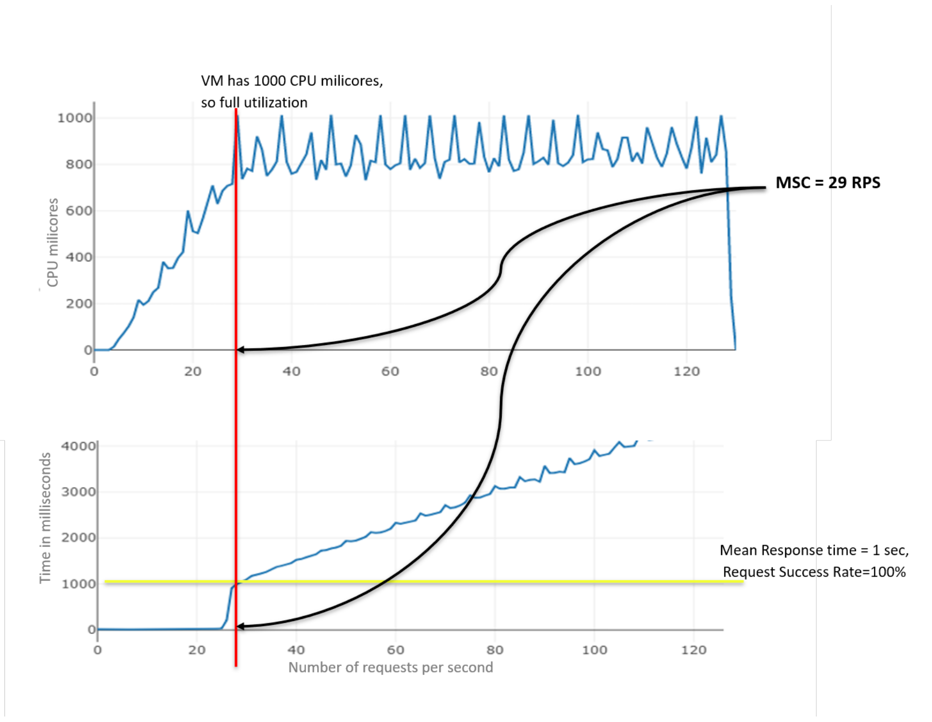


Figure 4.1: Maximum Service Capacity

requests per second completed vs Mean Response Time and number of CPU millicores used per request, graphs in Fig. 4.1. Suppose the user has specified the QoS parameters RSR as 100% and MRT to be 1 second. Also, the machine on which the service is deployed has 1 core or 1000 millicores. From Fig. 4.1, we can see that at around 29 RPS, the total CPU millicores used are around 1000 and the response time of requests is going beyond 1 second. Therefore, in an ideal scenario if Requests Per Second(RPS) remain below 29 then all the requests are successfully completed, hence achieving 100% RSR with MRT less than 1 second. Therefore, MSC for this service is 29 RPS.

## 4.2 Approach

### 4.2.1 Sandboxing of microservices

Sandboxing is a software management strategy that isolates applications from critical system resources and other programs[31]. In this research, we want to separate each microservices and add in place of each dependent service a virtualized service which responds instantly. This is done in order to test individual microservice performance independently without the performance impact of the other dependent services. In order to determine the request and response of each microservice, a proxy application is attached to all the services. Attaching this application to all the services makes sure that each request and response to and from the services is intercepted by it. After the interception of the requests and responses, all the dependent services are replaced by their virtualized services. The task of these virtualized services is to receive the request from the main microservice at the same API endpoint on which dependent services were receiving and respond back with the response intercepted by the proxy application without any computation. This whole process will make sure that the main service gets an instant reply to its requests from dependent virtualized services.

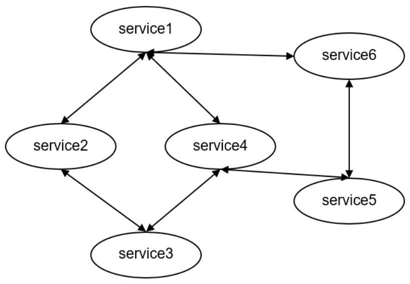


Figure 4.2: Example original application structure

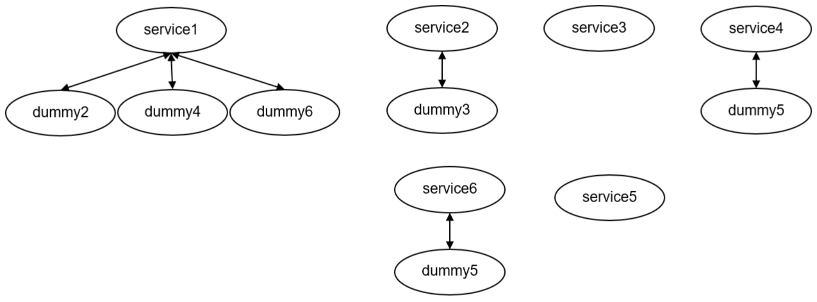


Figure 4.3: Example sandboxed microservices

Fig. 4.2 shows an example application structure which is broken down into **6** sandboxed microservices as shown in Fig. 4.3. Now, the performance model of the microservices can be derived.

### 4.2.2 Modeling of microservices

The basic idea behind the approach for deriving the performance model of the microservice is to test the deployed microservice with the linear increasing load pattern until a point is reached where either the deployed system is not able to process any request further or the QoS has gone down below the desired value. For building the model, a method is proposed with the following steps :

1. Firstly, each microservice of the given application is sandboxed using the sandboxing approach.
2. Now the microservice for which the model is to be built and along with the other virtualized microservices are deployed in a Kubernetes cluster on a Cloud Service Provider using some deployment configuration on the main microservice pod (resource requests, limits and replicas).
3. A linearly increasing load pattern, increasing at a certain rate T is generated on the exposed API endpoint of the deployed microservice with the starting number of requests per second equal to 1 to some requests per second corresponding to N, where N is quite a large number.
4. During the load generation, the main microservice pod resources utilization (e.g., CPU, memory, disk I/O, and network I/O) data and the generated workload data is collected. Table 4.1 shows one such example of the monitoring pod data based upon the RPS obtained from the cluster.

Table 4.1: Monitoring data of the resources for the Pod

| vCPU | Memory | RPS | Initial CPU | CPU Util. | Initial Memory | Memory Util. |
|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 0.08 | 0.1 | 0.02 | 0.0201 |
| 1 | 2 | 2 | 0.08 | 0.123 | 0.02 | 0.0202 |
| 1 | 2 | 3 | 0.08 | 0.145 | 0.02 | 0.020104 |
| 1 | 2 | 4 | 0.08 | 0.16 | 0.02 | 0.02024 |
| 1 | 2 | 5 | 0.08 | 0.181 | 0.02 | 0.020109 |

5. After the load generation is finished, the collected data for each resource of the cluster along with the number of requests served and the QoS parameters specified by the user, MSC is determined.

6. The calculated MSC for the microservice along with all the other collected data is saved.

7. The steps 2 - 6 are repeated for a sample of different deployment configurations (resource requests, limits and replicas).

8. All the collected data: resources utilization, number of requests served and deployment configuration is fitted to some mathematical function using regression. As a result, a model is derived of the microservice. Currently, in this research three models are derived :

   a) **Model trained for predicting MSC**: This derived model is used to predict the MSC at different deployment configurations by changing the parameters like the number of CPU cores or memory. Such a model is built by training with deployment configurations, resources utilization and QoS parameters as input values, and the number of requests served as the output value using regression.

   b) **Smart Model trained for determining actual MSC**: This model is used to find the actual MSC for a particular deployment configuration without conducting the load generation test till the endpoint. The training data here contains only some fixed sample values like 40 or 50. The actual MSC could be predicted from this trained model by finding the value at the 100% utilization of resources. Such a model is also built by training with deployment configurations, resources utilization and QoS parameters as input values, and the number of requests served as the output value using regression.

   c) **Model trained for predicting replicas**: This derived model is used to find the replicas required to handle the given number of user requests with a particular deployment configuration. Such a model is built by training with deployment configurations, resources utilization, QoS parameters and number of requests served as input values, and the number of replicas as the output value using regression.

   Fig. 4.4 shows the summarized diagram of the performance modeling. Also, the total resource utilization of the deployed microservice can maximally reach nearly 100%. Therefore, we can find the theoretical maximum number of requests the

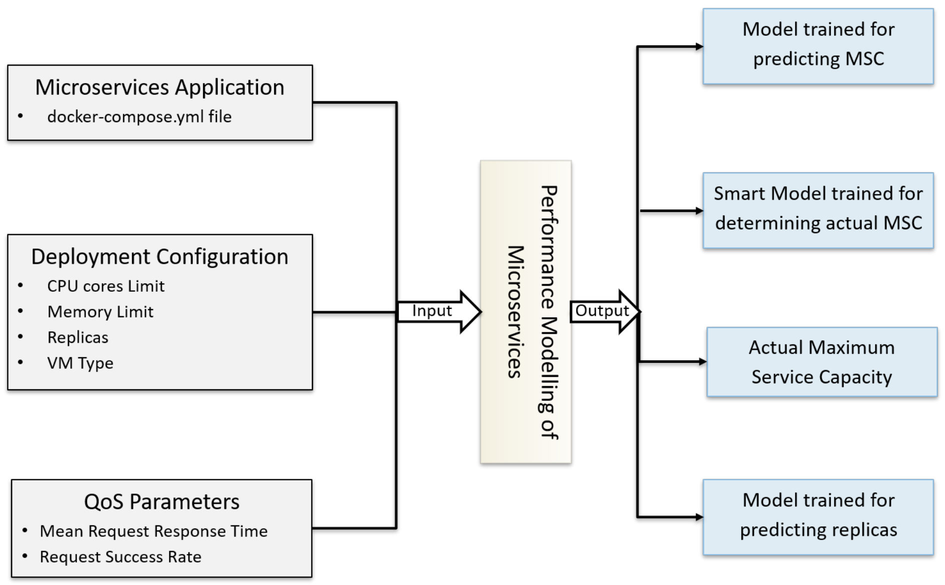

Microservices Application
docker-compose.yml file
Deployment Configuration
CPU cores Limit
Memory Limit
Replicas
VM Type
QoS Parameters
Mean Request Response Time
Request Success Rate
Input
Performance Modelling of Microservices
Output
Model trained for predicting MSC
Smart Model trained for determining actual MSC
Actual Maximum Service Capacity
Model trained for predicting replicas

Figure 4.4: Performance modeling of microservices

microservice can serve using the above-trained model for predicting MSC and compare it with the actual found MSC.

Afterward above all steps are repeated for all the other microservices in the given application and performance model is derived for each one of them.

# 5 Use Case Discussion

Performance prediction derived from the performance modeling of microservices could have many use cases, below sections describe a few of them.

## 5.1 Bottleneck microservice detection use case

Application microservices can be deployed directly on the VMs or inside the containers as described in the Microservices deployment strategy subsection in the background chapter 2.3. The deployed microservice uses many resources (e.g., CPU, memory, disk I/O, and network I/O) of the VM to process user requests. The resources usage change with the variation in the number of user requests or the user workload. The microservice that is able to serve the least number of requests in the given interval of time becomes the bottleneck microservice of the whole system. This could be either because one of the resources allocated to that microservice has reached around 100% utilization or because of the limited bandwidth of memory or network. Possible solution could be the upgrading of the VM to a better VM with the higher resources or increasing the number of replicas of the microservice.

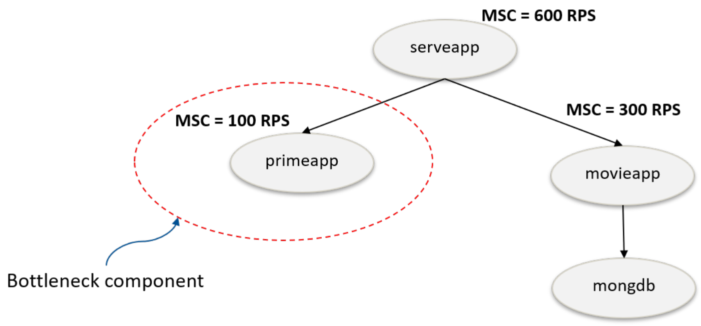


Figure 5.1: Bottleneck microservice detection use case

The microservice with the least MSC becomes the bottleneck microservice of that application. Individual microservice's scaling could be done to equalize the MSC of the whole system. For example, in Fig. 5.1, 3 microservices and a mongo database are shown along with their MSC. The microservice primeapp has the least MSC, therefore becomes the bottleneck microservice of the whole application. Now based upon MSC, one can decide how many replicas are required for each microservice. Here 6 replicas of primeapp, 2 of movieapp, 1 of serveapp makes the whole system's MSC same. After scaling microservices, the whole system can be scaled together as one unit.

## 5.2 Predictive autoscaling use case

Predictive autoscaling is an approach of autoscaling VMs where the autoscaling engine predicts the user traffic, calculate the resources(VMs) required prior to the time of need and provisions them based on those predictions [9]. Performance prediction of microservices by Terminus helps to determine the individual MSC of each microservice belonging to an application. Based upon the forecasting of user traffic and determination of MSC for each microservice, one can find out the number of resources required to serve those forecasted user requests and finally provision those many resources in advance to handle the traffic. The whole process is briefly shown in Fig. 5.2

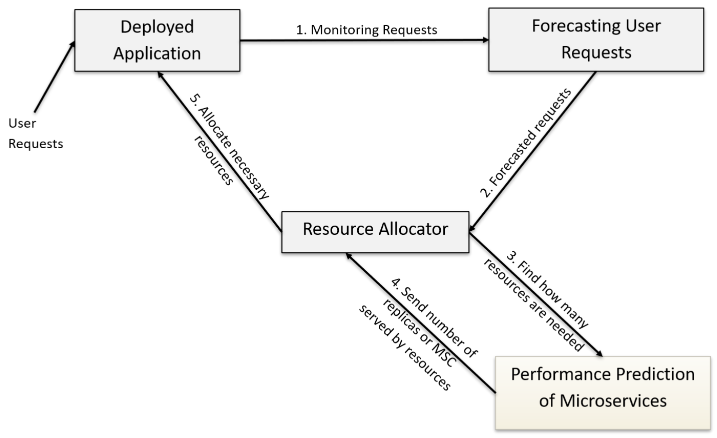

Deployed Application
1. Monitoring Requests
Forecasting User Requests
User Requests
5. Allocate necessary resources
2. Forecasted requests
Resource Allocator
3. Find how many resources are needed
4. Send number of replicas or MSC served by resources
Performance Prediction of Microservices

Figure 5.2: Predictive autoscaling use case

## 5.3 Reactive autoscaling scaling adjustment

Reactive autoscaling dynamically adjusts resources(VMs) counts based on the cluster's current workload (most often the metric used is average CPU utilization)[9]. When the chosen metric goes beyond a certain threshold, the reactive autoscaling engine will either add or remove the resources from the cluster. The number by which scaling is adjusted is referred to as the scaling adjustment number. Scaling adjustment can either be a constant number by which those many numbers of instances are added or removed or can be an exact number of instances or increment or decrement the current capacity of the group by the specified percentage [5]. However, determination of this scaling adjustment number can be difficult. Performance prediction of microservices

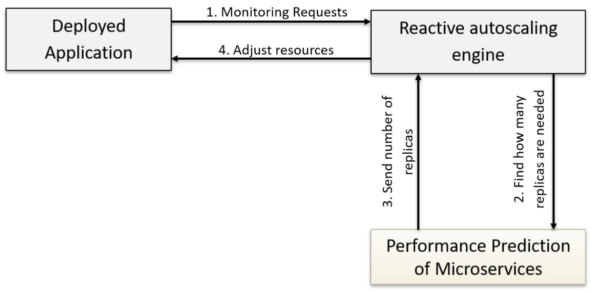

Deployed Application
1. Monitoring Requests
4. Adjust resources
Reactive autoscaling engine
3. Send number of replicas
2. Find how many replicas are needed
Performance Prediction of Microservices

Figure 5.3: Reactive autoscaling scaling adjustment use case

by Terminus helps to determine the number of replicas required to handle the given number of user requests of each microservice belonging to an application. Based upon the current number of user requests, one can determine the exact number of replicas required to serve those requests using Terminus. Therefore, that predicted number of replicas can be used as the scaling adjustment number. The whole process is briefly shown in Fig. 5.3

y

# 6 Implementation

The first step towards building the performance model of a microservice is the collection of the data of the microservice when deployed using different configurations on the Cloud Service Provider. This deployment process requires the setting up of Kubernetes Cluster, necessary monitoring services, and a load generator. Setting up all these manually again and again for conducting different tests can be cumbersome. Therefore, Terminus: a performance modeling tool for microservices is developed using the Golang and Python to automate the whole process. It comprises of both the API and a user interface with which a user can easily interact and perform various functions. In order to provide a comprehensive overview of the tool, the section below presents an overall architecture of the tool and then in next subsections, each individual components are explained briefly.

## 6.1 Overall architecture

Terminus is a performance modeling tool for the microservices. Terminus is derived from the word "terminal" which means end point of something. Terminus consists of components implementing microservices architecture; these components can be scaled up and down on demand. An overall architecture of the Terminus and communications between components thereof in a typical use case is shown in Fig. 6.1. The process starts with the user providing an application and its parameters either through the user interface or API. If the application consists of multiple microservices, then each microservice is sandboxed using the sandboxing approach. Afterward, performance modeling approach is applied to each of the sandboxed microservice or a single microservice application to analyze and build the performance model of it. Finally, the analyzed data and performance model is presented back to the user in the form of graphs or JSON data.
In further subsections, different components of Terminus along with the input to the Terminus and output from the Terminus are presented.

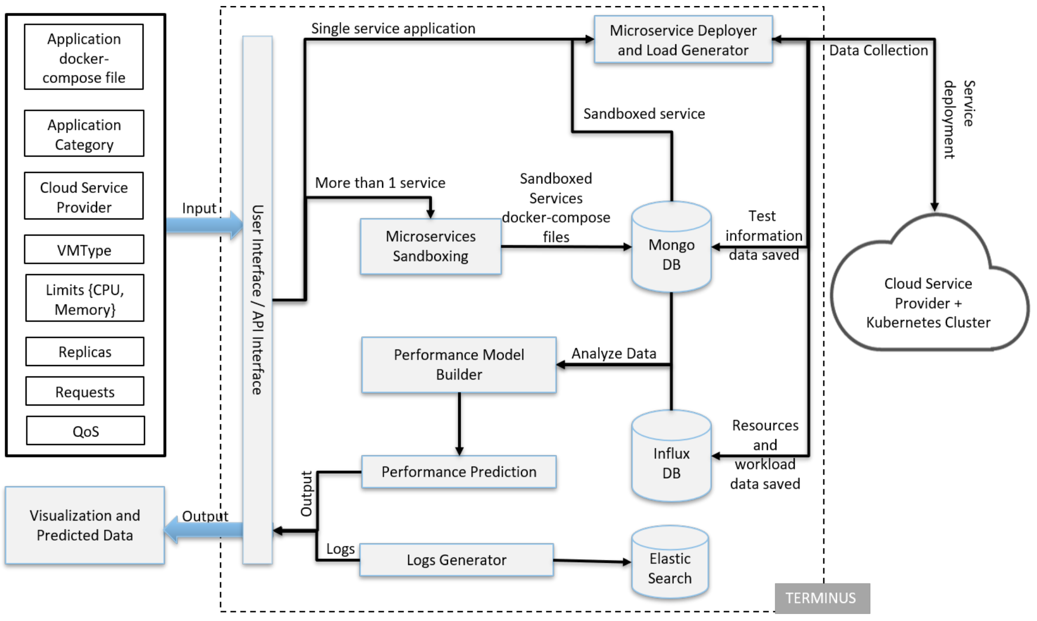

Application docker-compose file
Application Category
Cloud Service Provider
VMType
Limits {CPU, Memory}
Replicas
Requests
QoS
Input
User Interface / API Interface
Single service application
More than 1 service
Microservice Deployer and Load Generator
Sandboxed service
Microservices Sandboxing
Sandboxed Services docker-compose files
Mongo DB
Test information data saved
Data Collection
Service deployment
Cloud Service Provider + Kubernetes Cluster
Performance Model Builder
Analyze Data
Performance Prediction
Influx DB
Resources and workload data saved
Output
Logs
Logs Generator
Elastic Search
Visualization and Predicted Data
Output
TERMINUS

Figure 6.1: Overall architecture of the system.

### 6.1.1 Input to Terminus

- **Application docker-compose file**: Here the application for which performance model is to be built is provided in the form of a docker-compose file. This docker-compose file is used to build the structure of the application showcasing interactions between microservices and is also used to sandbox each microservice. For the deployment and building of the performance model, either this sandboxed microservice YAML file can be used or the file can be converted into Kubernetes configuration YAML file and then uploaded on the GitHub[1] repository and providing the name of the file as the input.
- **Application category**: It specifies the category to which the application belongs. Currently, 4 types of application categories are supported: compute category for compute-intensive applications, dbaccess category for the applications which accesses a database to perform some operations, web category for simple web applications and mix category for the applications consisting of the mixture of different categories.
- **Cloud Service Provider (CSP)**: The Cloud Service Provider on which the Kubernetes cluster is deployed. Currently, only Amazon Web Services is supported.

[1]https://github.com/ansjin/apps

- **Virtual Machine Type**: This specifies the type of virtual machine instance to be used as slave nodes in the Kubernetes cluster. AWS has different types of instances for the different family of instances. According to AWS, T2 family of instances is best suited for microservices, therefore currently only T2 family of instances is supported [4].
- **Limits and Replicas**: Here the resources configuration of the pod on which the microservice is deployed is done. CPU Limits are specified in the form of the number of cores and memory limits in GBs to be allocated to each replica of the microservice. Along with the limits, the number of replicas of the microservice is also specified here. The number of replicas corresponds directly to the number of pods.

### 6.1.2 Terminus components

#### Microservice Deployer and Load Generator

This component is responsible for the deployment of the microservice with the specified resource limits on the Kubernetes cluster, generation of the load and collection of the data for building performance model. For handling all these tasks an agent is created, whose components are shown in the Fig. 6.2. The functions of agent's components are as follows:
**Kubernetes cluster deployment**. This sub-component of the agent is responsible for

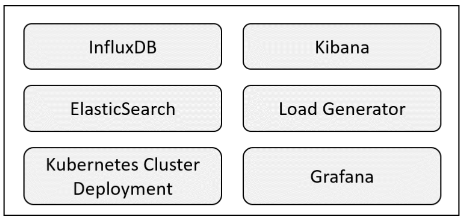
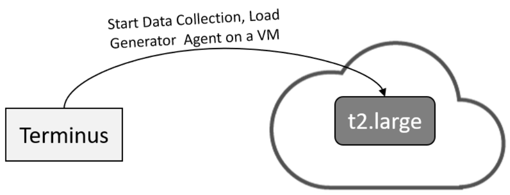


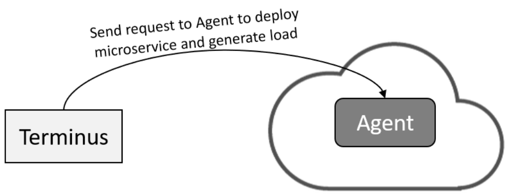
Figure 6.2: Agent components.

the deployment of the Kubernetes cluster using KOPS - Kubernetes Operations [22]. KOPS can set up highly available and production-grade Kubernetes cluster. Along with the deployment of the microservice, heapster is also deployed as part of the cluster to monitor all the resources inside the Kubernetes cluster[13]. The data collected by heapster is stored inside the InfluxDB [15].
**Load generator**. This sub-component generates a linearly increasing load to the exposed external endpoint of the deployed microservice. K6 is used for generating the load [19]. All the workload data is stored inside the InfluxDB. This data can be viewed

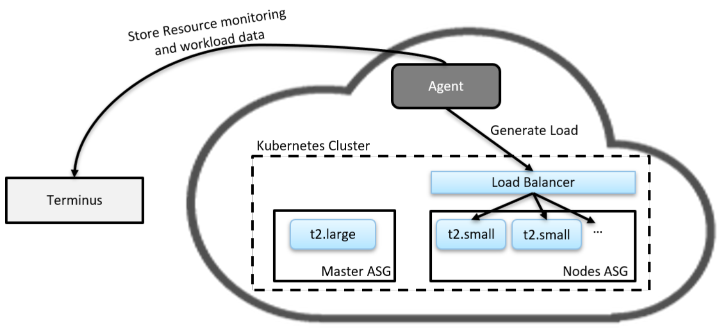
in real time using the Grafana[2] attached with this agent.
**Elasticsearch and Kibana**. The agent also generate logs for each step therefore, Elasticsearch is used to store them and with Kibana those can be viewed in real time[11, 20].



(a) Agent Deployment


Send request to Agent to deploy microservice and generate load
Terminus
Agent

(b) Start Kubernetes Cluster



(c) Load generation and data collection

Figure 6.3: Microservice Deployer and Load Generator Steps

The process of collecting the data involves three steps as described below:

1. **Agent deployment**: Firstly, the agent described above is deployed on a VM instance. Terminus sends a command with necessary instructions to the CSP to start a VM and deploy this agent on a VM as shown in Fig. 6.3a.
2. **Start Kubernetes cluster and generate load**: After the deployment of the agent, Terminus sends another request to the agent to create a Kubernetes cluster and deploy the microservice inside it. After the deployment, it starts to generate the load as shown in Fig. 6.3b.
3. **Data collection**: During the load generation phase, agent continuously monitors all the resources and stores all the data inside the InfluxDB along with the workload data. Once the test is finished all the stored data is restored back to InfluxDB inside the Terminus and the cluster, as well as the agent, are terminated. This step is shown in Fig. 6.3c.

[2]https://grafana.com/docs

### Microservice sandboxing

This is one of the important components of the Terminus which comes into action when an application consists of multiple microservices. This component sandboxes each microservice and adds in place of each dependent microservice a pseudo virtualized microservice. Such a process is carried out so that individual microservice can be independently tested and a performance model is built for it. Following are the steps performed:

1. docker-compose yaml file along with the main microservice name and api end point is provided as input. For example in code 6.1, there are two services: primeapp and serveapp, and serveapp is dependent on primeapp.

```yaml
services:
 primeapp:
  image: "[..]/primeapp"
  ports:
  - "9001:9001"
 serveapp:
  image: "[..]/serveapp"
  links:
  - primeapp # links primeapp
  ports:
  - "8080:8080"
version: "2"
```

Code Snippet 6.1: Initial docker compose file

```yaml
services:
 primeapp:
  image: "[..]/primeapp"
  ports:
  - "9001:9001"
  links:
  - hoxy_app # new dependency
  environment: # newly added
   - HTTP_PROXY:
     —"http://hoxy_app:8085"
 serveapp:
  image: "[..]/serveapp"
  links:
  - primeapp # links primeapp
  - hoxy_app # new dependency
  ports:
  - "8080:8080"
  [...]
```

Code Snippet 6.2: Intermediate docker compose file

2. A dependency of hoxy_app service to all the services present inside the docker-compose file is added. This application's docker image can be found here[3]. It ensures that every user request and response to and from any of the microservice

[3]https://hub.docker.com/r/terminusimages/hoxy_app/

passes through it first. As a result, this application intercepts the requests sent to the service and the responses from it. At the end of this step, a new intermediate docker-compose file is formed as shown in code 6.2.

3. This new docker-compose file is deployed on a virtual machine as shown in Fig. 6.4a. Once the deployment is completed, a request is sent to the externally exposed endpoint. This request will flow through all the microservices and the hoxy_app will intercept all the requests and responses to and from each microservices respectively. After intercepting of requests and responses, it stores back all these data inside the mongo database attached to the Terminus. This is shown in Fig. 6.4b

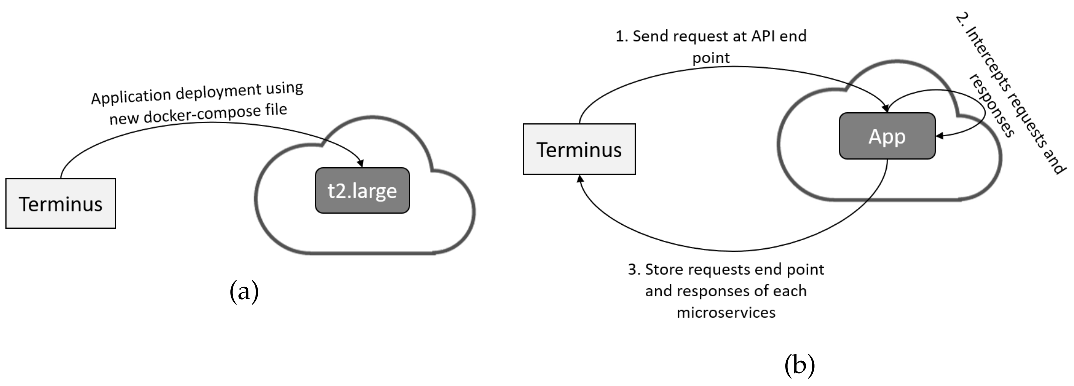

Application deployment using new docker-compose file
Terminus
t2.large
(a)
1. Send request at API end point
2. Intercepts requests and responses
Terminus
App
3. Store requests end point and responses of each microservices
(b)

Figure 6.4: Requests and response intercepting step of sandboxing microservices

4. For each microservice, dummy virtualized microservices replaces their dependent microservices as shown in code 6.3. These dummy microservices will respond at the same port and at the same endpoint with the response collected in the above step. As a result, now the time taken by the dependent services to respond is next to negligible.

```
services:
 primeapp:
  #dummy service
  container_name: primeapp
  environment:
  - TEST_API=/api/test
  - DUMMY_RESPONSE={"result":249995787104.9515}
  - PORT=9001
  image: [...]dummy_response_app:latest
```

```
    ports:
    - "9001:9001"
  serveapp:
    depends_on:
    - primeapp
    image: [..]/serveapp
    links:
    - primeapp
    ports:
    - "8080:8080"
version: "2"
```

Code Snippet 6.3: Final docker compose file for sandboxed microservice

Now, each sandboxed microservice can be used to build the performance model for them like other single microservice applications.

### Performance Model Builder

This is another major component of Terminus. It is responsible for building the performance model of a microservice from the collected data. This component is written in python and uses a regression approach to train the model. Currently, there are three models being trained as described below:

1. The first model that is trained is based upon the collection of different deployment configurations combined together with the collected data to predict the MSC for all the configurations. Theil–Sen estimator is used for training the model. Theil–Sen method has been used to estimate trends in software aging[36]. It is used due to its simplicity in computation, robustness to outliers and high precision in the presence of skewed data"[12]. Here the training input is the deployment configuration and total CPU utilization which is calculated by equation 6.1

$$\text{CPU\_Util}_{total} = \sum_{n=1}^{N} \text{CPU\_Util}_{n} \tag{6.1}$$

"N" being the number of pods

and the number of requests is the output value.
2. The second model trained is per deployment configuration which is also combined together with the collected data to predict the MSC for that specific configuration.

The training data here contains only some fixed sample values like 40 or 50. This is done to prove the concept that, we don't need to conduct the full test for finding out MSC. It could be predicted from this regression trained model by finding the value at the 100% utilization of resources. Here also, Theil–Sen estimator is used for training the model. Here also the training input is the deployment configuration and total CPU utilization and the number of requests is the output value.

3. Third model trained is used to predict the number of replicas required to handle a particular amount of requests on a specific deployment configuration. Here the data from a sample of different deployment configurations along with the collected data is taken and trained for prediction of replicas. Here the training input is the number of requests and total CPU utilization and the number of pods is the output value.

### Performance Prediction

Trained models by the performance model builder are used by this component to predict values.

### Logs Generator

This component generates logs while Terminus tool is being used. This allows easy debugging of the issues and detailed understanding of the flow path. All the logs are saved in a file and are also shown on the user interface.

### User Interface and API interface

User Interface (UI) and API interface components of Terminus interacts with the user and provides him or her an opportunity to select or configure different tool parameters. The user has the option to start various tests, view results in the form of table or graphs, train the model and can also view ongoing tests and logs from the UI. The APIs supported by Terminus are in standard REST API format as shown in the code 6.4. Also, a swagger documentation YAML file is generated to understand the APIs input and output.

```
/getPredictedRegressionReplicas/{apptype}/{appname}/{mainservicename}/{msc}/{
    numcoresutil}/{numcoreslimit}/{nummemlimit}
```

Code Snippet 6.4: Example REST API

### 6.1.3 Output from Terminus

Apart from the graphs and data consisting of the generalized performance model of different microservices for different deployment configurations presented in the UI, Terminus output :

- **MSC**: It provides the actual experimental MSC calculated based upon the deployment configuration and the QoS parameters specified.
- **Predicted MSC**: It outputs predicted MSC for a deployment configuration specified in the terms of CPU Utilization, CPU Limit, Memory Limit, Mean Request Response Time and the number of replicas.
- **Predicted Number Of Replicas**: Predicts the number of replicas required to handle the certain number of requests given the deployment configuration specified in the terms of CPU Utilization, CPU Limit, Memory Limit, Mean Request Response Time and the number of requests.
- **Sandbox Microservices from the Application**: It provides the individual docker-compose file for the sandboxed microservices.

## 6.2 Experimental settings

### 6.2.1 Test application

In our experiments, we took an application consisting of 4 microservices and a mongo database connected to the movieapp as shown in Fig. 6.5.

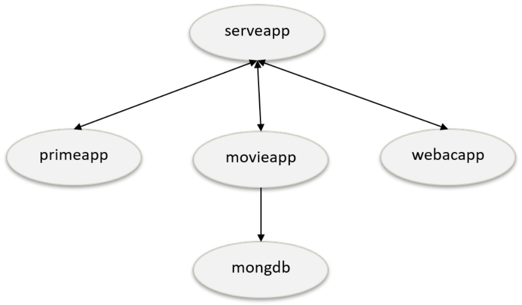


Figure 6.5: Test Application Structure.

- **primeapp**: It computes the sum of prime numbers starting from **1** and up to **1000000** when called using a particular API call. This is categorized as the compute intensive microservice.

- **movieapp**: It queries mongo database for a fixed amount of movies information[4] when called using a particular API call and returns the results as the response. This is categorized as the database access microservice.
- **webacapp**: It is a simple web application which responds back with the "hello world" when called using a particular API call. This is categorized as the web microservice.
- **serveapp**: Its job is to receive the request from the user and send requests to all the three dependent microservices. After receiving responses from all the microservices it combines them into one and responds back.

### 6.2.2 Deployment Configurations

In our experiments, we use T2 family of instances from AWS which comprised of t2.nano, t2.micro, t2.small, t2.medium, t2.large and t2.xlarge instances. Additional deployment configuration for each test is described in Table 6.1. It is to be noted that, when the resource limits are given a value for example of "100", then it means that CPU cores of 0.100 and memory of 100MB are allocated. Also, if the resource limit is referred by an instance name for example "t2.nano", then it means that limits equivalent to t2.nano instance are allocated. Depending on the number of replicas, slave nodes inside the Kubernetes cluster are adjusted.

Table 6.1: Experimental deployment configurations

| **Instance type** | **Resource Limits** | **Replicas** |
|---|---|---|
| t2.nano | 100 and 200. | 1, 2 and 3 |
| t2.micro | 100,200 and 500. | 1, 2 and 3 |
| t2.small | 100,200 and 500 | 1, 2 and 3 |
| t2.medium | 100,200, 500, t2.nano, t2.micro and t2.small | 1, 2 and 3 |
| t2.large | 100,200, 500, t2.nano, t2.micro and t2.small | 1, 2 and 3 |
| t2.xlarge | 100,200, 500, t2.nano, t2.micro, t2.small, t2.medium and t2.large | 1, 2 and 3 |

[4]the database consists of dummy movies information

### 6.2.3 Load generation setting

We have used the linearly increasing load pattern. The rate at which the requests are increased differs from type to type of the application and is shown in Table 6.2.

Table 6.2: Load generation requests rate

| Application type | Request Rate |
|---|---|
| compute | Starts with 1, increases by 4 every minute. |
| dbaccess | Starts with 1, increases by 6 every minute. |
| web | Starts with 1, increases by 50 every minute. |
| other | Starts with 1, increases by 20 every minute. |

### 6.2.4 Performance modeling

For training the models, data collected from the deployment configurations listed in Table 6.1 were used except t2.xlarge one. Once the model is trained then they were used to predict the values for the experiments listed in Table 6.3. These resource limits were set on the t2.xlarge instance and also the number of nodes were adjusted based upon the resource requirement and replicas. For predicting replicas, actual found experimental MSC value was used instead of the replicas column. For calculating the actual MSC, QoS parameters MRT equals to 1 second and 99% RSR are used.

Table 6.3: Deployment configurations for experiments

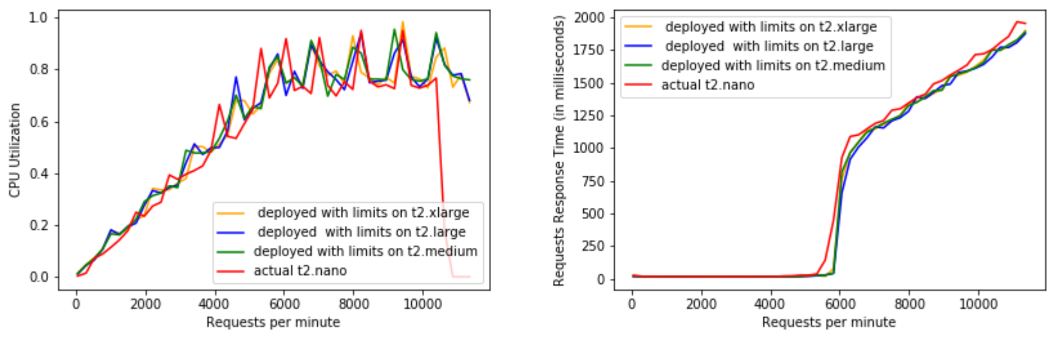
| Experiment Number | CPU Limit | Memory Limit | Replicas |
|---|---|---|---|
| 1 | 0.1 | 0.1 | 1 |
| 2 | 0.1 | 0.1 | 2 |
| 3 | 0.1 | 0.1 | 3 |
| 4 | 0.2 | 0.2 | 1 |
| 5 | 0.2 | 0.2 | 2 |
| 6 | 0.2 | 0.2 | 3 |
| 7 | 0.5 | 0.5 | 1 |
| 8 | 0.5 | 0.5 | 2 |
| 9 | 0.5 | 0.5 | 3 |
| 10 | 1 | 1 | 1 |
| 11 | 1 | 1 | 2 |
| 12 | 1 | 1 | 3 |

Table 6.3: Deployment configurations for experiments

| **Experiment Number** | **CPU Limit** | **Memory Limit** | **Replicas** |
|---|---|---|---|
| 13 | 1 | 0.5 | 1 |
| 14 | 1 | 0.5 | 2 |
| 15 | 1 | 0.5 | 3 |
| 16 | 1 | 2 | 1 |
| 17 | 1 | 2 | 2 |
| 18 | 1 | 2 | 3 |
| 19 | 2 | 4 | 1 |
| 20 | 2 | 8 | 1 |

# 7 Results

Deploying same resource limits on different types of instances provides almost the same performance with the condition that the host instance has enough resources for the Kubernetes cluster to provide dedicated resources to a pod. Also, comparing this limited resources pod with its equivalent instance also results in almost the same performance. There will be a slight drop of performance towards the peak for the VM instance due to the fact that VM is using some resources underneath for the OS and not all resources are available as compared to the pod which has all the limited resources. Fig. 7.1a shows the number of requests per minute vs the CPU utilization of pods with "t2.nano" instance as limits when deployed on t2.xlarge, t2.large, and t2.medium. Also, the performance of the actual instance is shown. Correspondingly, the mean request response time compared with the number of requests is shown in Fig. 7.1b. We can see that almost everyone has similar performance.

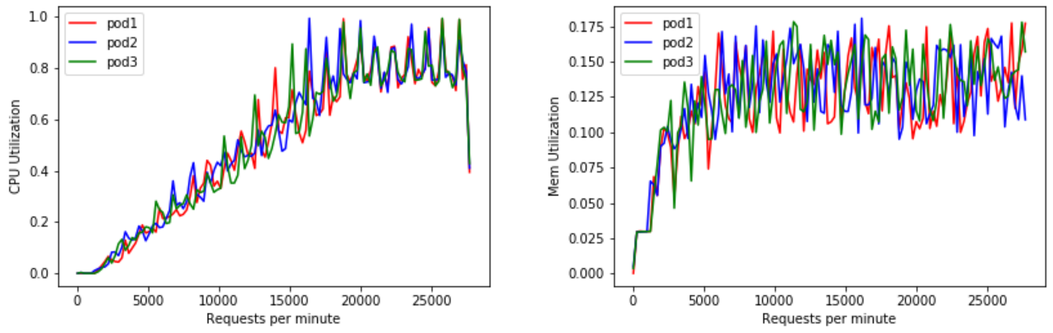


(a) Requests vs CPU Utilization

(b) Requests vs MRT

Figure 7.1: Same resources limit pod on different types of VM.

Another important result derived is that the resources utilization by different pods or replicas belonging to the same microservice on the same type of VM is almost the same. This could be because for load balancing among replicas of a microservice the Kubernetes scheduler uses the round-robin algorithm. As a result, the load gets equally distributed among the replicas. Fig. 7.2a shows the CPU utilization and Fig. 7.2b shows the memory utilization of three pods with the increasing number of requests per



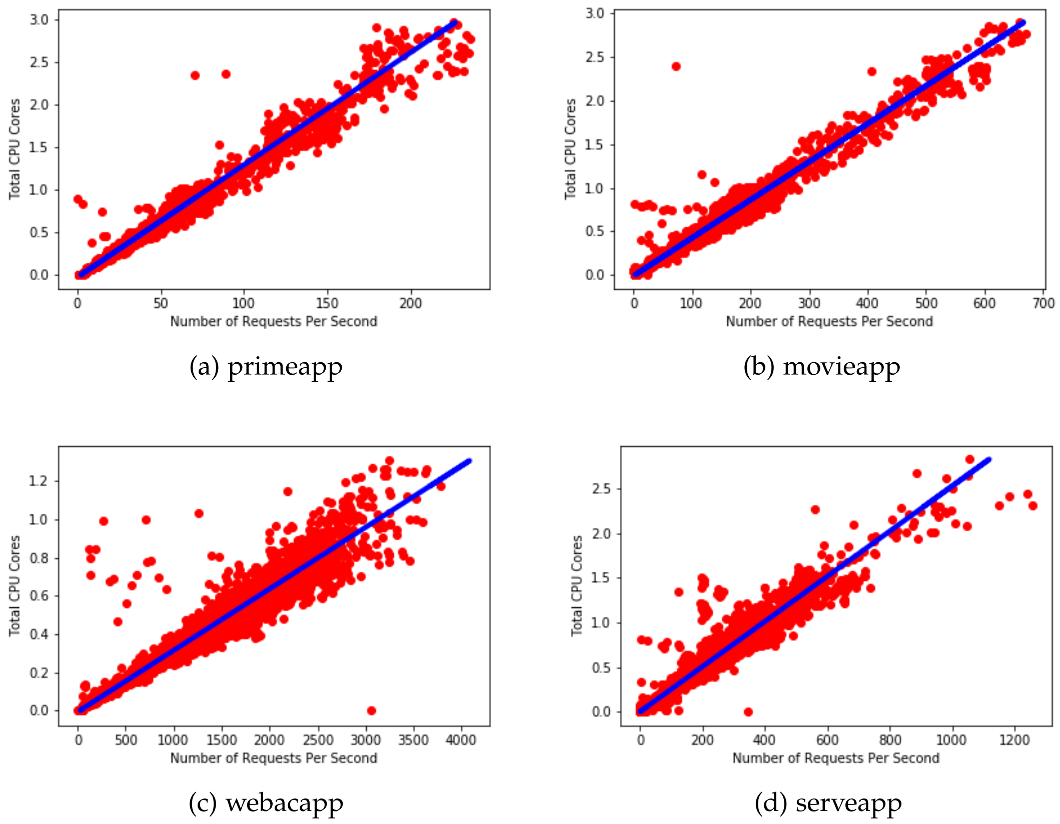
(a) Pods CPU utilization (b) Pods memory utilization

Figure 7.2: Different replicas of microservice resources utilization.

minute. We can see from both the figure that pods resources utilization is almost the same.

The application used for testing consists of 4 microservices and for each sandboxed microservice the number of CPU cores required to process user requests was found to be linearly increasing with the number of requests as shown in Fig. 7.3. Memory utilization did not have much effect on the change in the number of requests, it becomes constant for all the microservices after a certain number of requests. For determining the experimental actual MSC for the primeapp, movieapp and serveapp the load was generated till the threshold point was reached and calculated afterwards from the data. However, for finding the MSC of the webacapp the load could not be generated till its threshold point because the machine which generated the load could not handle the simulation of very high number of requests. Therefore either we need to use very high end resource machine or use the smart model developed to determine the MSC for a particular deployment configuration based upon the limited amount of load generation data. This was first tested on the primeapp and found to be very close to the actual MSC as shown in Fig. 7.4. Later on, this model **smart approach** was used to find the actual MSC for the webacapp.

The predicted MSC gives an upper bound on how many requests different types of VMs can handle based upon the fixed number of tests conducted. Trained model was able to model the microservices quite well and predicted MSC as well as replicas quite close to the actual found one. Fig. 7.5 shows the comparison of predicted vs the actual MSC for all the microservices and similarly Fig. 7.6 shows the comparison of predicted vs the actual replicas for all the microservices along with their RMS errors as well. **webacapp** has highest RMS error for predicted MSC model because of the



(a) primeapp (b) movieapp

(c) webacapp (d) serveapp

Figure 7.3: Number of requests vs CPU cores vary linearly for various types of microservices

range of number of requests per second it can handle. We can notice towards the end of the MSC predicted values a big difference in Fig. 7.5 for all the microservices except webacapp where the experiments has the number of cores is equal to 2. This is the result of the microservice not being able to use more than 1 core even though it was deployed on the pod which had the CPU core limit of 2. This could be possible because the application is made in Node.js and it is single threaded therefore it could not use more than one core. But the model takes into account the linearity of the variation and predicted based upon 2 cores. For webacapp, the actual was also predicted using the smart testing therefore it is more close to predicted one.

Also, from the Fig. 7.5 one can notice that rate at which the CPU utilization changes with the number of requests is highest for **primeapp** and least for **webacapp**. It clearly depicts that the primeapp is the bottleneck microservice of the whole application. Table

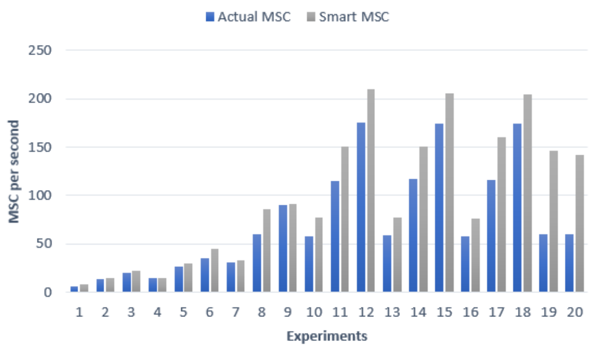


Figure 7.4: Actual found MSC vs the Smart MSC calculated using Smart model for the primeapp.

7.1 shows the predicted MSC for the combined application as well as for the sandboxed individual microservices for some deployment configurations. One can now easily decide by looking at the MSC for each microservice that how many replicas for each is needed to make the performance of the whole application same. For example, if deployed with the configuration of CPU and memory equal to 0.1 and replicas equal to 1 then we need approximately 37,14, 9 and 1 replicas for primeapp,movieapp, serveapp and webacapp respectively to make the performance of the whole application one.

## 7.1 Performance modeling hypothesis

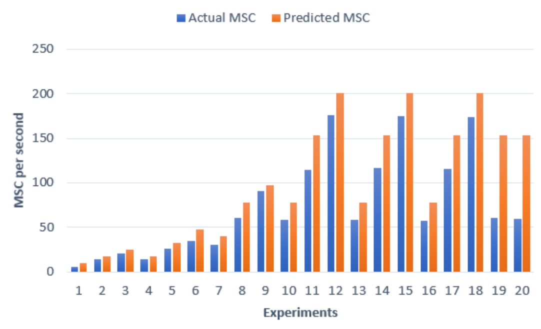
Based on the experimental results, the following major research hypotheses might be formulated for future extensive proof on different microservices and on their combinations thereof with the use of Terminus:

- Performance of pod with limited instance resources is equivalent to the VM instance with similar resources. Therefore, one doesn't need to conduct tests for performance analysis on the smaller machines. A big VM instance can be selected and tested with different types of limits.
- MSCs found for each sandboxed microservices could be used to determine the potential candidates for the dependency of microservices on each other. As a result, it could be used to pack and scale the dependent microservices together.
- A better load balancer scheduling algorithm can be added while distributing the load to the pod replicas inside Kubernetes cluster.

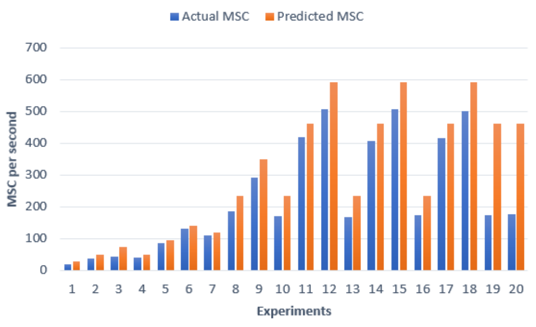
Table 7.1: MSC Predicted for the combined application and its microservices

| **No.** | **CPU Cores** | **Mem-ory** | **Replicas** | **Comb-ined** | **prime-app** | **movie-app** | **webac-app** | **serve-app** |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.10 | 0.10 | 1.00 | 5.25 | 9.59 | 25.90 | 364.53 | 40.11 |
| 2 | 0.10 | 0.10 | 2.00 | 6.55 | 17.18 | 48.90 | 674.29 | 79.84 |
| 3 | 0.10 | 0.10 | 3.00 | 7.77 | 24.77 | 71.90 | 984.05 | 119.58 |
| 4 | 0.20 | 0.20 | 1.00 | 13.67 | 17.18 | 48.90 | 674.29 | 79.84 |
| 5 | 0.20 | 0.20 | 2.00 | 25.87 | 32.36 | 94.91 | 1293.80 | 159.31 |
| 6 | 0.20 | 0.20 | 3.00 | 38.32 | 47.53 | 140.91 | 1913.32 | 226.45 |
| 7 | 0.50 | 0.50 | 1.00 | 32.80 | 39.95 | 117.91 | 1603.56 | 199.05 |
| 8 | 0.50 | 0.50 | 2.00 | 62.32 | 77.88 | 232.92 | 3152.36 | 397.72 |
| 9 | 0.50 | 0.50 | 3.00 | 105.48 | 97.65 | 347.94 | 4701.16 | 596.39 |
| 10 | 1.00 | 1.00 | 1.00 | 67.20 | 77.88 | 232.92 | 3152.36 | 397.72 |
| 11 | 1.00 | 1.00 | 2.00 | 126.55 | 153.76 | 462.95 | 6249.96 | 795.07 |
| 12 | 1.00 | 1.00 | 3.00 | 208.32 | 200.64 | 592.98 | 9347.56 | 1192.41 |
| 13 | 1.00 | 0.50 | 1.00 | 65.15 | 77.88 | 232.92 | 3152.36 | 397.72 |
| 14 | 1.00 | 0.50 | 2.00 | 126.73 | 153.76 | 462.95 | 6249.96 | 795.07 |
| 15 | 1.00 | 0.50 | 3.00 | 211.23 | 200.64 | 592.98 | 9347.56 | 1192.41 |
| 16 | 1.00 | 2.00 | 1.00 | 63.25 | 77.88 | 232.92 | 3152.36 | 397.72 |

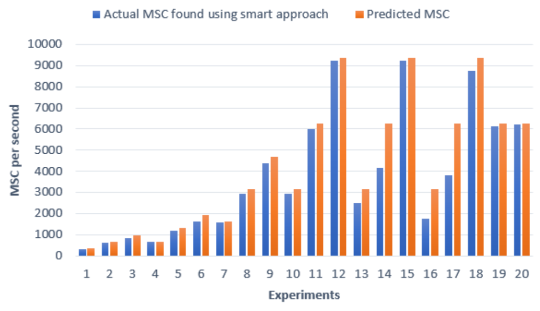
- Most microservices application consume CPU resource more intensively than other resources, that is why we can see most Autoscaling solutions provided by Cloud Service providers use CPU utilization parameter for scaling.
- A brute-force testing for determination of actual threshold point could be avoided by using the smart approach. This method is cost-efficient and save resources too.
- Sandboxing microservices allows to understand the real performance of them and further modeled microservice can be used to improve the performance of the whole application as well as efficient scaling and resources allocation.
- It is better to use microservice with multiple replicas of resource limits below or equivalent to 1 CPU cores and 2GB of memory until unless the microservice has higher minimum requirements.



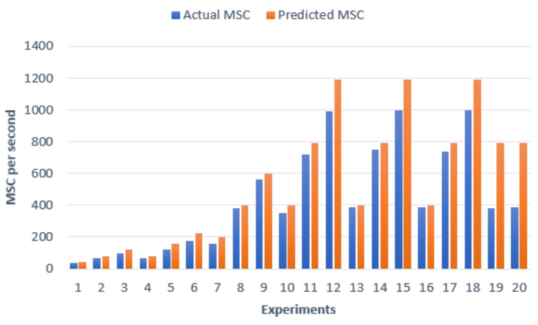
(a) primeapp with RMS error = 8.1



(b) movieapp with RMS error = 20.57

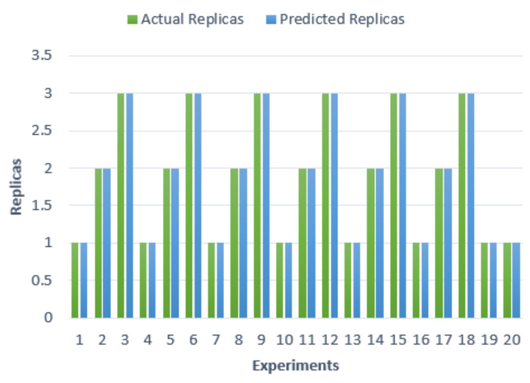


(c) webacapp actual MSC found using smart model with RMS error = 100.66

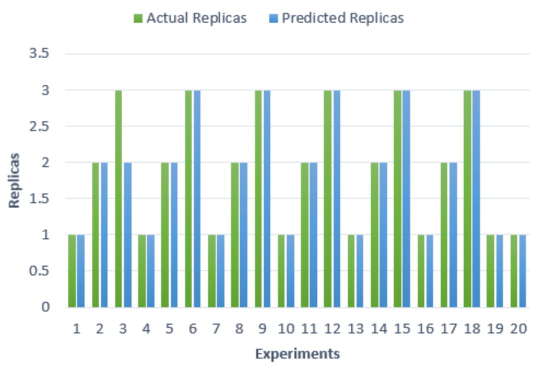


(d) serveapp with RMS error = 40.57

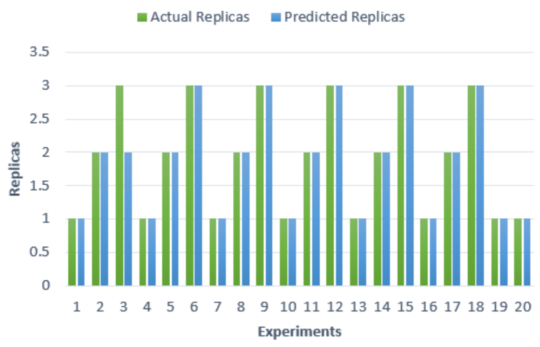
Figure 7.5: MSC Actual vs Predicted for Microservices



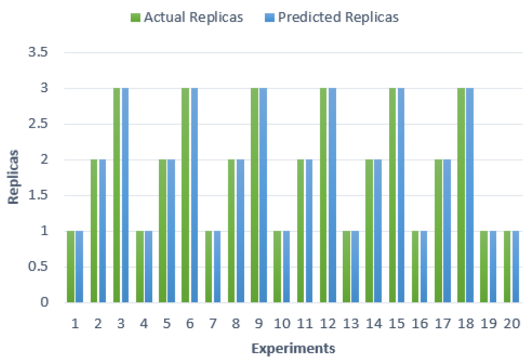
(a) primeapp with RMS error = 0.10



(b) movieapp with RMS error = 0.11



(c) webacapp with RMS error = 0.12



(d) serveapp with RMS error = 0.11

Figure 7.6: Replicas Actual vs Predicted for Microservices

# 8 Conclusion & Future Scope

## 8.1 Conclusion

Performance modeling of the cloud-based microservices application allows to find out the resources that need to be allocated to it while deployment. Each microservice performs differently when deployed using different deployment configurations, however, they follow a common pattern in their performance. All the tested microservices in this research has a higher CPU resource usage as compared to other resources usage and the usage changed linearly with the change in the number of requests. Other different types of applications like memory intensive applications will behave differently. In cases where an application consists of multiple microservices, the overall performance is limited by the least performing microservice, so scaling all the microservices of an application without understanding their resource requirements is not a good idea. In order to capture the performance and functionality issues and track down architectural reasons for it, sandboxing of microservices from the application is required. However, challenges in sandboxing and then deriving efficient model issues occur during the building of performance model of the microservice. The approach and the tool proposed in the research serve to solve the problem of finding the best-suited resources for the microservice to be deployed on so that it gives the best performance based on the need for the whole application.
The presented approach and the tool could be widely applied in the cloud-powered industry for various use cases. A key example would be to use Terminus to allocate resources efficiently to the deployed microservices. This could be done by building the performance model of the microservices and then based upon the possible incoming user workload finding out the resources required by them. Such an extension would enable many real-world use cases which will help companies to decrease the cost of cloud services and increase the manageability and scalability of the cloud-based microservices applications. A correct identification of bottleneck microservices and scaling replicas contributes to the possible solution of yet another research and practical problem - structural analysis problem of the microservices application.

## 8.2 Future Scope

In the scope of the further research, a comprehensive performance modeling of existing microservices applications using the developed approach and Terminus can be conducted. As a result, a dataset of performance models of microservices can be created and then the classification of microservices based upon their derived performance model can be done. This dataset then could further be used to compare with a new microservice and based upon the class to which it belongs to, the performance of the microservice could be predicted. Furthermore, this tool could be extended for many applications. Following are a few to point :

- Extension of Terminus to different cloud providers(Microsoft Azure, IBM Bluemix, LRZ etc).
- Support for different types of applications and building performance models.
- Currently, regression is used for building the performance model, in future support for different approaches like using Neural networks can be added.
- Sandboxing approach developed currently only supports homogenous GET requests, it can be extended to include heterogeneous GET and POST requests.

# List of Figures

# List of Tables

# List of Code Snippets

# Bibliography

[1] *Amazon cloud watch*. [Online; Accessed: 7-May-2018]. URL: `https://aws.amazon.com/cloudwatch`.

[2] Peter Arijs. *How to use resource requests and limits to manage resource usage of your Kubernetes cluster*. [Online; Accessed: 2-October-2018]. 2018. URL: `https://jaxenter.com/manage-container-resource-kubernetes-141977.html`.

[3] R. Aversa, N. Panza, and L. Tasquier. "An Agent-Based Platform for Cloud Applications Performance Monitoring." In: *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*. 2015, pp. 535–540. DOI: `10.1109/CISIS.2015.79`.

[4] AWS. *Amazon EC2 Instance Types*. [Online; Accessed: 25-September-2018]. Sept. 2018. URL: `https://aws.amazon.com/ec2/instance-types/`.

[5] AWS. *Simple and Step Scaling Policies for Amazon EC2 Auto Scaling*. [Online; Accessed: 29-September-2018]. URL: `https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.html`.

[6] *AWS Lambda*. [Online; Accessed: 25-May-2018]. URL: `https://aws.amazon.com/lambda/`.

[7] Jennifer Petoff Betsy Beyer Chris Jones and Niall Richard Murphy, eds. *Site Reliability Engineering*. [Online; Accessed: 9-October-2018]. 2016. URL: `https://landing.google.com/sre/book/index.html`.

[8] *CloudMonix*. [Online; Accessed: 9-May-2018]. URL: `http://www.cloudmonix.com/`.

[9] Danny Yuan Daniel Jacobson and Neeraj Joshi. *Scryer: Netflix's Predictive Auto Scaling Engine*. [Online; Accessed: 29-September-2018]. Nov. 2013. URL: `https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270`.

[10] *Docker overview*. [Online; Accessed: 9-October-2018]. URL: `https://docs.docker.com/engine/docker-overview/`.

[11] *Elasticsearch*. [Online; Accessed: 25-September-2018]. URL: `https://www.elastic.co/products/elasticsearch`.

[12] R. Fernandes and S. G. Leblanc. “Parametric (modified least squares) and non-parametric (Theil-Sen) linear regressions for predicting biophysical parameters in the presence of measurement errors.” In: *Remote Sensing of Environment* 95 (Apr. 2005), pp. 303–316. DOI: 10.1016/j.rse.2005.01.005.

[13] *Heapster*. [Online; Accessed: 25-September-2018]. URL: https://github.com/kubernetes/heapster.

[14] *Howerfly*. [Online; Accessed: 10-June-2018]. URL: https://hoverfly.readthedocs.io/en/latest/.

[15] *InfluxDB*. [Online; Accessed: 25-September-2018]. URL: https://www.influxdata.com/time-series-platform/influxdb/.

[16] N. Jain and S. Choudhary. “Overview of virtualization in cloud computing.” In: *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. 2016, pp. 1–4. DOI: 10.1109/CDAN.2016.7570950.

[17] D. Jaramillo, D. V. Nguyen, and R. Smart. “Leveraging microservices architecture by using Docker technology.” In: *SoutheastCon 2016*. 2016, pp. 1–5. DOI: 10.1109/SECON.2016.7506647.

[18] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. “Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications.” In: *Proceedings of the 26th International Conference on World Wide Web*. WWW ’17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 469–478. ISBN: 978-1-4503-4913-0. DOI: 10.1145/3038912.3052649. URL: https://doi.org/10.1145/3038912.3052649.

[19] *K6*. [Online; Accessed: 25-September-2018]. URL: https://docs.k6.io/docs.

[20] *Kibana*. [Online; Accessed: 25-September-2018]. URL: https://www.elastic.co/products/kibana.

[21] Nane Kratzke. “About Microservices, Containers and their Underestimated Impact on Network Performance.” In: *CoRR* abs/1710.04049 (2017). arXiv: 1710.04049. URL: http://arxiv.org/abs/1710.04049.

[22] *Kubernetes Operations*. [Online; Accessed: 25-September-2018]. URL: https://github.com/kubernetes/kops.

[23] kubernetes.io. *Managing Compute Resources for Containers*. [Online; Accessed: 2-October-2018]. URL: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/.

[24] kubernetes.io. *What is Kubernetes*. [Online; Accessed: 3-May-2018]. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[25] Frank Leymann et al. "Native Cloud Applications: Why Monolithic Virtualization Is Not Their Foundation." In: *Cloud Computing and Services Science*. Ed. by Markus Helfert et al. Cham: Springer International Publishing, 2017, pp. 16–40. ISBN: 978-3-319-62594-2.

[26] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583. URL: `http://dl.acm.org/citation.cfm?id=2600239.2600241`.

[27] D. Moldovan, H. L. Truong, and S. Dustdar. "Cost-Aware Scalability of Applications in Public Clouds." In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 2016, pp. 79–88. DOI: `10.1109/IC2E.2016.23`.

[28] *Monitoring Azure applications and resources*. [Online; Accessed: 7-May-2018]. May 2018. URL: `https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/monitoring-overview`.

[29] J. Mukherjee, M. Wang, and D. Krishnamurthy. "Performance Testing Web Applications on the Cloud." In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. 2014, pp. 363–369. DOI: `10.1109/ICSTW.2014.57`.

[30] Pedro Pinheiro, Manuela Aparicio, and Carlos Costa. "Adoption of Cloud Computing Systems." In: *Proceedings of the International Conference on Information Systems and Design of Communication*. ISDOC '14. Lisbon, Portugal: ACM, 2014, pp. 127–131. ISBN: 978-1-4503-2713-8. DOI: `10.1145/2618168.2618188`. URL: `http://doi.acm.org/10.1145/2618168.2618188`.

[31] Vassilis Prevelakis and Diomidis Spinellis. "Sandboxing Applications." In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 119–126. ISBN: 1-880446-10-3. URL: `http://dl.acm.org/citation.cfm?id=647054.715767`.

[32] Chris Richardson. *Introduction to Microservices*. [Online; Accessed: 25-May-2018]. 2015. URL: `https://www.nginx.com/blog/introduction-to-microservices/`.

[33] John Rittinghouse and James Ransome. *Cloud Computing: Implementation, Management, and Security*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2009. ISBN: 1439806802, 9781439806807.

[34] N. Serrano, G. Gallardo, and J. Hernantes. "Infrastructure as a Service and Cloud Technologies." In: *IEEE Software* 32.2 (2015), pp. 30–36. ISSN: 0740-7459. DOI: `10.1109/MS.2015.43`.

[35] B. Urgaonkar et al. “Dynamic Provisioning of Multi-tier Internet Applications.” In: *Second International Conference on Autonomic Computing (ICAC’05)*. 2005, pp. 217–228. DOI: `10.1109/ICAC.2005.27`.

[36] K. Vaidyanathan and K. S. Trivedi. “A comprehensive model for software rejuvenation.” In: *IEEE Transactions on Dependable and Secure Computing* 2.2 (2005), pp. 124–137. ISSN: 1545-5971. DOI: `10.1109/TDSC.2005.15`.