Performance Modeling for Cloud Microservice Applications

Anshul Jindal
Technical University of Munich
Garching (near Munich), Bavaria
Germany
anshul.jindal@tum.de

Vladimir Podolskiy
Technical University of Munich
Garching (near Munich), Bavaria
Germany
v.podolskiy@tum.de

Michael Gerndt
Technical University of Munich
Garching (near Munich), Bavaria
Germany
gerndt@in.tum.de

ABSTRACT

Microservices enable a fine-grained control over the cloud applications that they constitute and thus became widely-used in the industry. Each microservice implements its own functionality and communicates with other microservices through language- and platform-agnostic API. The resources usage of microservices varies depending on the implemented functionality and the workload. Continuously increasing load or a sudden load spike may yield a violation of a service level objective (SLO). To characterize the behavior of a microservice application which is appropriate for the user, we define a *MicroService Capacity (MSC)* as a maximal rate of requests that can be served without violating SLO.

The paper addresses the *challenge of identifying MSC individually for each microservice.* Finding individual capacities of microservices ensures the flexibility of the capacity planning for an application. This challenge is addressed by *sandboxing* a microservice and building its performance model. This approach was implemented in a tool **Terminus**. The tool estimates the capacity of a microservice on different deployment configurations by conducting a limited set of load tests followed by fitting an appropriate regression model to the acquired performance data. The evaluation of the microservice performance models on microservices of four different applications shown relatively accurate predictions with mean absolute percentage error (MAPE) less than 10%.

The results of the proposed performance modeling for individual microservices are deemed as a major input for the microservice application performance modeling.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;

KEYWORDS

Performance modeling, Microservice capacity, Kubernetes

ACM Reference Format:
Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2019. Performance Modeling for Cloud Microservice Applications. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297663.3310309

ICPE '19, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6239-9/19/04…$15.00
https://doi.org/10.1145/3297663.3310309

1 INTRODUCTION

Cloud computing is widely adopted in the industry as a technology enabling cheap and easy access to data processing and storage resources. The cloud is backed by physical servers which host virtual machines (VMs) that are provided to the users. This relieves the cloud user from the obligation to own and maintain the hardware; only the actual use is being paid for.

Cloud computing, however, exhibits a significant drawback - the actual performance could significantly differ from the expected depending on the type of VM and on the type of application running [22]. The difference in resources utilization and performance patterns for cloud applications introduces the inability to create a universal performance model for any cloud application.

The only general performance characteristic that could be captured in the load test for an interactive application is the critical workload value which, when achieved, results in a significant drop of the quality of service. This drop could be manifested differently, e.g. response time for user requests becomes higher than the value specified in the SLO. Identifying this critical value is equivalent to finding the approximate capacity of the application as characterized by the workload. This value is usually identified by stress-testing the application, i.e. by putting it under the increasing workload until the SLOs are violated [3]. While giving the information on the overall capacity of the application, this value fails to help answer the question: *how to increase the capacity of the application by utilizing the cloud elasticity in a cost-efficient way?*

The ongoing adoption of microservices applications enabled higher degree of flexibility in managing the application at run-time. Instead of changing the capacity of the whole application, one can change the capacity of the particular part by changing the amount of microservice replicas. Depending upon the type of algorithms implemented in different microservices, they can have different requests rate that they can cope with, i.e. different capacity. In case of multilayered virtualization - microservices running on VMs - the capacity also depends on the characteristics of the underlying VM.

Microservice capacity (MSC) is identified in the paper as the maximal rate of requests that a microservice can cope with without violating SLOs. MSC can be used in multiple cases, e.g. to detect the cloud application bottleneck, to plan the capacity of the application in terms of workload.

Application bottleneck detection serves to identify the microservice(s) responsible for the performance degradation. The bottleneck of an interactive microservice application is such microservice, replication of which results in the increase of the quality of service (e.g. less SLO violations). Existing analytic methods to detect the bottlenecks are limited to multi-tier applications with a small number of tiers [26].

Application capacity planning allows to cope with a certain workload. Capacity planning takes the predicted workload and the performance model of the application as inputs. With these inputs, one can derive the number of replicas of an individual microservice to cope with a forecasted workload [2, 19]. Automating the forecasting procedure and the performance modeling of microservices allows to automate capacity planning. In particular, predictive autoscaling incorporates capacity planning.

The outlined use-cases of cloud application bottleneck detection and capacity planning highlight the potential applications of performance modeling of microservice applications.

The main contribution of the paper is a novel performance modeling approach determining the service capacity of individual microservices. This challenge is addressed by modifying the conventional stress-testing with a proxy service-based sandboxed testing. The exhaustive testing for all possible configurations from some limited configuration space is avoided by deriving the regression of the workload capacity on the deployment characteristics. The results of performance modeling with the developed tool Terminus for 4 distinct microservices are reported.

Section 2 provides background on multilayered cloud virtualization and microservice applications. Section 3 describes the general approach to sandboxing-based performance modeling of microservices. Section 4 discusses the implementation of Terminus. Section 5 is devoted to the description of the experimental settings. The results of conducted tests are provided in Section 6. Section 7 summarizes the discussion of the test results. Section 8 studies related works. Section 9 concludes the paper and outlines future research.

2 BACKGROUND

2.1 Multiple layers of cloud virtualization

As of now, two virtualization layers are widely used in the industry - virtual infrastructure layer consisting of VMs and container virtualization layer.

A container is a lightweight, standalone, executable package of software that includes everything needed to run it. Containers can run both on bare metal and on the virtual infrastructure. They use fractions of virtual CPUs (vCPU) or real CPUs. Container-based type of virtualization increases the utilization potential of a server, e.g. the single-threaded microservice can be instantiated in multiple containers thus utilizing multiple cores. Using container-based virtualization, a single operating system on a host can run multiple isolated cloud services [10].

A container orchestrator such as Kubernetes can run containers at scale. It schedules and orchestrates containers on the shared set of physical or virtual resources [16]. In Kubernetes, containers are grouped into pods sharing storage and network resources.

Container resource constraints might be set in the pod configuration file. With this, the user can control the amount of CPU and memory resources at container level. Each resource type has a base unit, e.g. CPU is specified in millicores, i.e. 1 millicore equals 0.001 of the virtual CPU. Constraints can be set for each resource through requests and limits.

A CPU request specifies the fraction of CPU time that the system has to guarantee to a container. A container can use more in case no other container uses CPU and the limits are not reached [1, 15]. Kubernetes uses this value to decide on which node to place the pod. Kubernetes will schedule as many pods on a single node as possible until the requested CPU share can be guaranteed for each pod on that node. Setting request less than limits allows over-subscription of resources.

2.2 Microservice Applications

Microservice applications gained popularity due to fine-granular design and loosely coupled services implementing limited functionality which results in higher scaling flexibility.

Microservice applications allow deployment of individual services to physically separated VMs. Each microservice behaves as an independent, autonomous process and communicates with other microservices through APIs. Commonly, each such process is put into individual container to achieve higher flexibility and manageability of the application, though running multiple services in a container is also allowed[1]. Microservice 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.

Containers and pods are used to deploy microservices to simplify the management of cloud application components [13, 17].

2.3 Microservices deployment strategy

Microservices can be implemented using variety of languages and frameworks. Each microservice could be considered as a mini-application with its specific deployment, resource, scaling, and monitoring requirements [21]. The following paragraphs briefly describe various microservices deployment strategies.

2.3.1 *Multiple microservices instances per host.*

This strategy aims to run multiple microservices on one or more physical or virtual hosts. This is achieved:

(1) **without containers**: services are directly deployed on host;
(2) **with containers**: services are packaged individually in different containers and then deployed.

This strategy is used to take the advantage of the full potential of the host by running multiple services. If some microservices do not use their share of resources fully, then other microservices can use these resources. However, the quality of service is not guaranteed.

2.3.2 *Single microservice instance per host.*

This strategy aims to run each microservice instance on its own host. This is achieved:

(1) **without containers**: each microservice instance is packaged in a VM image, which is used to start the host;
(2) **with containers**: microservice instances are packaged into containers and then deployed on different hosts.

This deployment type allows the guaranteed quality of service for any microservice at a cost of idle resources. Further, if the container scalability of each microservice is not used, the user has only a single point where he specifies the VM scaling settings.

3 PERFORMANCE MODELING

3.1 Terminology

3.1.1 *Service Level Objective.*

Service level objective (SLO) specifies guarantees for the level of performance, reliability, and availability.

[1] docs.docker.com/config/containers/multi-service_container

Following common SLOs for interactive web applications are considered in the paper - (1) requests success rate being higher than the threshold and (2) the response time for 90% of requests being lower than the threshold.

Requests success rate (RSR) is a ratio of the user requests successfully completed within a given time to the overall number of the requests submitted by the users within the same time.

90%tile response time (P90RT) is such a value of the response time for a time interval T that 90% of the requests that arrived during interval T received the response in the given time. P90RT is used throughout the paper because the behavior of the tail latency can significantly differ from the mean response time which can result in sub-optimal resource allocation and degraded performance.

3.1.2 Deployment Configuration. Deployment configurations means the amount of the resources allocated to the microservice pod and type of VM used as slave nodes in a Kubernetes cluster. Further, resource requests, limits and replicas count are used to allocate the desired amount of resources. As part of this research, resource requests and limits are kept equal so that a consistent performance is obtained. Resource requests and limits value of 0.100 of vCPUs and 100 MB of virtual memory allocated to a single pod replica when deployed on a t2.micro type of VM.

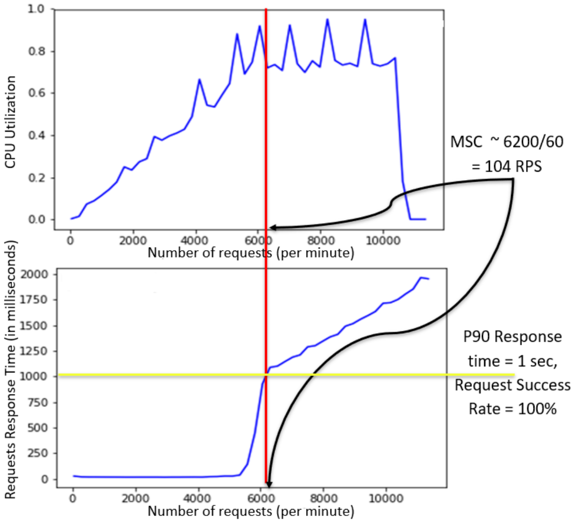


Figure 1: Microservice capacity identification.

3.1.3 Microservice Capacity (MSC). Application microservices use various VM resources. The resource utilization changes with the variation in the user workload. Given an increasing workload, a resource might become nearly fully utilized. At that point, some user requests might require longer processing times as specified by the SLO. The maximal number of successfully processed user requests per second for the given service such that no SLO is violated is called Microservice Capacity (MSC).

Let us identify MSC for a microservice on a single-core VM with the number of requests per minute completed versus 90%tile response time and CPU utilization characteristic as shown in Fig. 1. The user has specified SLO with RSR to be 100% and P90RT as 1 second. From Fig. 1, we can observe that at around 6200 requests per minute the CPU is almost fully utilized and the 90%tile response time had crossed 1 second. Ideally, if the count of requests per second would have remained below $104 = 6200/60$ then all the requests would have been completed without violating SLOs. Hence, MSC is approximately 104 requests per second.

3.2 Approach

3.2.1 Sandboxing of microservices. Sandboxing is a software management strategy that isolates applications from critical system resources and other programs [20]. Automatic sandboxing of microservices requires to replace each dependent microservice by a dummy service with no or minimal changes to the original application code. Dummy services receive requests from the sandboxed microservice at the API endpoint used by original services and respond instantly. The goal is to evaluate the performance of an individual microservice independently without any performance impact from the dependent services.

First stage of the sandboxing aims to record the combination of the request with the correct response for each microservice. For this purpose, a proxy microservice **hoxy_app** is attached to all the microservices in the given docker-compose file. Attaching this microservice to all the microservices and passing its URL as an environment variable to all the microservices ensures that each request and response is intercepted. After the interception phase is completed, the dependent microservices are replaced by their dummy versions. With this new deployment ready, the following load test will uncover the capacity of an individual microservice.

Following the modification of the docker-compose files, the microservice with its dummy connections is deployed in a Kubernetes cluster and is put under the linearly increasing workload. During this process, resource utilization data and the workload parameters are collected. When the violation of the SLO is detected, the collected data is dumped to the database and MSC is determined. This procedure is repeated for a sample of different deployment configurations. The sample configurations are selected to ensure the desired accuracy of the performance model through the continuous addition of the sample configurations until the accuracy of the derived model ceases to significantly improve.

3.2.2 Regression-based Performance Modeling. The data resulting from the sandboxed load testing of an individual microservice is used to fit the regression model that relates MSC or replica counts to the deployment configuration and resources utilization.

Deployment configuration-based model to estimate MSC captures the connection between MSC and the deployment configurations (i.e. number of pod replicas, allocated virtual CPUs, and the memory). Theil-Sen estimator is used to derive the regression model. This estimator is used because of its simplicity in computation, robustness to outliers, a priori limited information regarding measurement errors and the capability to fit both linear and non-linear models [7]. The input parameters are the deployment configuration and the total CPU utilization for N pods which is calculated by the Equation 1.

$$Util_{total}^{(CPU)} = \sum_{i=1}^{N} Util_{i}^{(CPU)} \tag{1}$$

Incorporation of this model in the tool aims to prove the concept that it is not necessary to conduct an exhaustive load testing of the microservices to determine MSC for each deployment.

Other regression models like Support Vector Regression (SVR) and simplistic linear and polynomial regression were also tested. However, the Theil-Sen estimator had shown higher accuracy and lower fitting duration, therefore we omit SVR and other estimators for the focused discussion.

Model to predict the replica count is derived using the same estimator. It allows to predict the number of microservice pod replicas required to handle a particular amount of requests on a specific deployment configuration i.e. the input data is represented by the number of requests and the total CPU utilization and the output data by the number of replicas.

Fitted models can be used to predict the capacity of the individual microservices even for the VMs with larger amount of resources. This statement will be proven in the results section.

4 MICROSERVICE PERFORMANCE MODELING TOOL TERMINUS

4.1 Architecture and functionality

Terminus is a performance modeling tool for microservice applications written in Golang and Python. The tool automates the setup of a Kubernetes cluster and deploys the monitoring services and a load generator. Terminus comprises both the API and the user interface allowing the user to easily interact with the tool and conduct comprehensive performance modeling for any microservice.

Terminus consists of multiple components. It has a microservices architecture, thus the components can be scaled. The architecture of the tool and the communication paths between its components in a typical use-case are shown in Fig. 2.

The performance modeling 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 tested individually using the sandboxing. Afterwards, performance modeling is applied to each sandboxed microservice to analyze and build the corresponding model. Finally, the analyzed data and performance model are presented to the user in the form of graphs or JSON data.

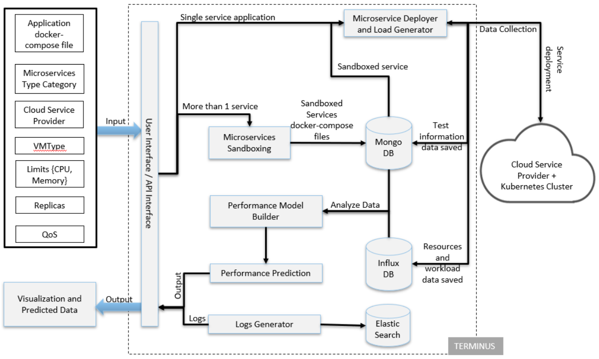

Application docker-compose file
Microservices Type Category
Cloud Service Provider
VMType
Limits {CPU, Memory}
Replicas
QoS
Input
User Interface / API Interface
Single service application
Microservice Deployer and Load Generator
Data Collection
Service deployment
Sandboxed service
More than 1 service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

Figure 2: Architecture of Terminus.

4.2 Inputs

Application's docker-compose file is used to build the microservice application and to sandbox each microservice. Either this sandboxed microservice file or the converted Kubernetes configuration file is used to deploy the application and load-test it.

Microservice Type Category: The research focuses on the interactive applications with single function per microservice. Supported microservice categories include: compute-intensive, database access, simple web app, and the mediator interacting with other microservices. Specification of the category allows to form the datasets to conduct comparisons and generalizations for performance models of particular microservice category.

Cloud Service Provider (CSP): IaaS CSP with the deployed Kubernetes cluster. Currently, only AWS is supported.

VM Type is a type of VM instance to be used as slave nodes in the Kubernetes cluster. Currently, only t2 family of AWS instances is supported as it is best suited for microservices [23].

Limits and Replicas include CPU limits and memory limits to be allocated to each microservice replica as well as the number of microservice replicas that equals the number of pods.

4.3 Microservice Deployer / Load Generator

This component is responsible for the deployment of the microservice with the specified resource limits on the Kubernetes cluster as well as for the generation of the load and collection of the monitoring data to build the performance model. To handle these tasks, Terminus creates an agent to test the specific deployment configuration with multiple components described below.

Kubernetes Cluster Deployment is responsible for the deployment of the Kubernetes cluster using KOPS - a tool to set up and manage production-grade Kubernetes clusters [14]. Heapster is deployed in the cluster to monitor the consumed resources [8]. The data collected by Heapster is stored in InfluxDB [9].

Load generator generates a linearly increasing workload to the exposed endpoint of the microservice using K6 [12]. The characteristics of the generated workload are stored in InfluxDB and can be viewed in real time using Grafana.

Elasticsearch and Kibana are used to store and view in real time the logs produced by the components [6].

The process of collecting the data allows to start multiple tests in parallel and involves the following steps:

(1) **Agent deployment**: A request is sent to AWS to start a t2.large VM and to deploy the agent on VM boot.
(2) **Start of the Kubernetes cluster and microservice deployment**: Terminus sends a request to the agent to create a Kubernetes cluster with the specified configuration and to deploy the sandboxed microservice in it. The number of slave nodes is computed based on the number of replicas, type of VM and limits specified in the inputs.
(3) **Load generation and data collection**: When the deployment is completed, the agent starts to generate the linearly increasing workload and monitors all the resources and stores the performance parameters and the resources utilization data in local database along with the workload data. When the violation of the SLO is detected, all the data are dumped back to the InfluxDB of the Terminus. Afterwards, the cluster and the agent are terminated.

4.4 Microservice Sandboxing Component

This component allows to build performance models for individual microservices through simulation of their neighbors. It isolates each microservice and substitutes its direct neighbors with dummy microservices accepting the requests and sending the responses of the same format, but without any additional processing. The sandboxing component takes a docker-compose file, the main microservice name and an API endpoint as an input. Following, a proxy application microservice is added as a dependency to every microservice. The application is deployed using the modified docker-compose file so that each microservice request and response passes through the proxy. Copies of these requests and responses are stored in the Terminus MongoDB database.

Following, each microservice receives its own sandboxed deployment where the direct neighbors are replaced by dummy microservices. These dummy microservices will respond at the same port and at the same endpoint with the response stored in MongoDB. As a result, the time taken by the dependent microservices to respond becomes negligible. By generating the increasing workload for the sandboxed versions of microservices, a relatively pure performance model and capacity for each microservice is determined.

4.5 Performance Model Builder and Predictor

This component derives the performance model of a microservice based on the data collected during the load tests. It is written in Python and performs the regression modeling to capture the connection between the deployment characteristics, resources consumption and the performance in terms of the workload. The component is used to derive two models described in 3.2.2.

Fitted regression models are used by Performance Predictor to estimate the workload capacity of the deployment configurations.

4.6 UI, API and the Logs Generator

User Interface (UI) and API components of Terminus provide the opportunity to select or configure parameters of the tool. The user has the option to start the tests, view the results (tables or graphs), fit the performance model, view the ongoing tests and logs. Terminus supports REST API to interact with other programs. Logs Generator generates logs during the use of Terminus to simplify the debugging.

4.7 Tool outputs

The outputs of Terminus were designed to support predictive autoscaling. These outputs are briefly presented below.

Microservice Capacity, MSC is a test-based evaluation of MSC for the particular deployment configuration and SLO.

Estimated MSC is a regression model output for a specific deployment configuration.

Predicted number of pod replicas is a regression model output for a given number of requests and resource limits.

Sandboxed microservice configuration includes a docker-compose file for a sandboxed version of all the microservices.

5 EXPERIMENTAL SETTINGS

5.1 Test application

Experimental application consists ofthe following 4 microservices and a MongoDB connected to the movieapp:

- **primeapp** (compute-intensive) computes the sum of prime numbers starting from 1 and up to 1000000 when called;
- **movieapp** (database access) queries MongoDB for a fixed amount of movies when called and returns the results found;
- **webacapp** (web access) instantly responds to each request with the body containing only the *hello world* string;
- **serveapp** receives a request from the user and dispatches it to the dependent (primeapp, movieapp and webacapp) microservices. After receiving responses from all the microservices it combines them and returns the result.

5.2 Deployment Configurations

Experiments were conducted for the t2 family of AWS instances comprising t2.nano, t2.micro, t2.small, t2.medium, t2.large and t2.xlarge types. Resource and replicas count limits were set for pods running on each instance type. These limits are provided in Table 1. The numerical resource limit value, e.g. 100, means the corresponding fraction of vCPU and the memory will be utilized, e.g. 0.100 of virtual CPU cores and 100 MB of virtual memory. Resource limit that is referred by the type of instance, e.g. t2.nano means that pods limits are equivalent to the parameters of t2.nano type. The limits on the replicas count specified in the table are caused by the resource capacity of the host VM type.

Table 1: Experimental Deployment Configurations

VM type[a]	Resource Limits	Replicas
t2.nano	100; 200	1 - 3
t2.micro	100; 200; 500	1 - 3
t2.small	100; 200; 500	1 - 3
t2.medium	100; 200; 500; t2.nano; t2.micro; t2.small	1 - 3
t2.large	100; 200; 500; t2.nano; t2.micro; t2.small	1 - 3
t2.xlarge	100; 200; 500; t2.nano; t2.micro; t2.small	1 - 3

[a] https://aws.amazon.com/ec2/instance-types/t2

5.3 Load generation settings

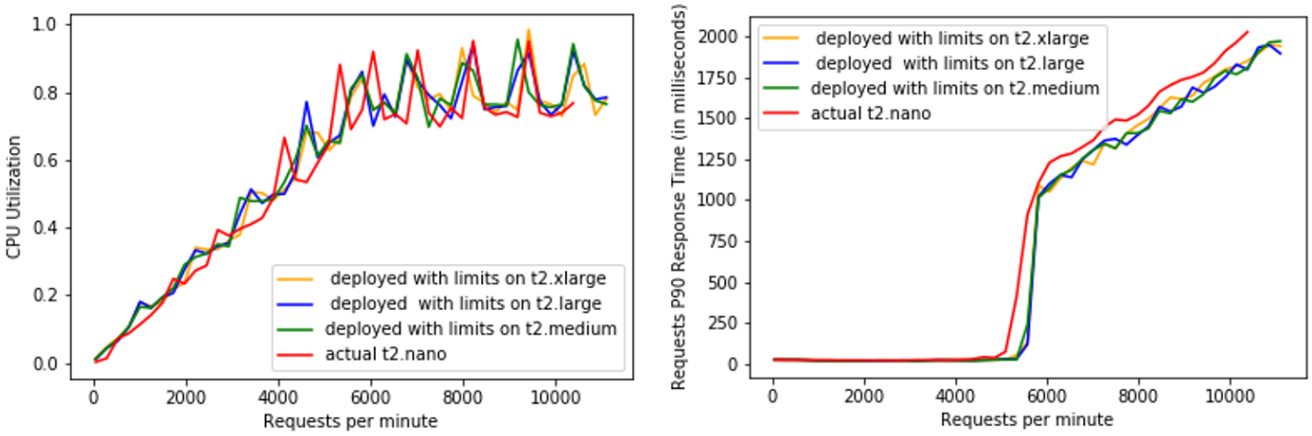
The workload request rate was linearly increasing during the experiment to determine MSC. The rate increased every minute according to the type of microservice: compute-intensive - by 4; database access - by 6; web access - by 50; other types - by 20.

Different load generation rates are determined experimentally based on the change in the resource utilization and time required for the requests processing by each types of the microservice. An appropriate number which causes the resource utilization change approximately by 0.05% was selected. The load testing in all the cases starts with single request. The SLO with RSR equal to 98% and P90RT of 3 second was used to identify experimental MSC.

6 EXPERIMENTAL RESULTS

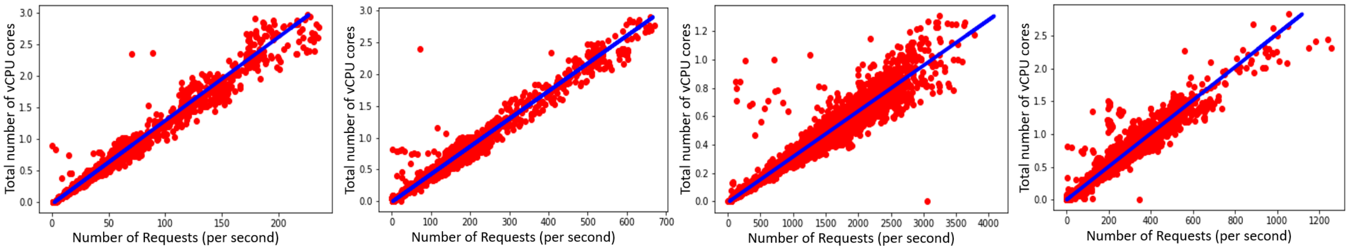
The time taken to determine experimental MSC varies with the deployment configuration. The minimum time taken for the configuration with VM type as t2.nano with resource limits of 100 and 1 replica and the maximum time taken for the configuration with VM type as t2.xlarge with resource limits of t2.small and 3 replicas for different microservices is shown in Table 2. These large durations of tests do not hinder the applicability of the approach for real systems since such tests should be conducted only once.

Deploying pods with the same resource limits on different instance types results in almost the same performance. Comparison



(a) Number of Requests (per minute) vs CPU Utilization **(b) Number of Requests (per minute) vs Requests P90 Response Time**

Figure 3: Same resources limit pod on different types of VM shows similar performance.



(a) primeapp **(b) movieapp** **(c) webacapp** **(d) serveapp**

Figure 4: Total number of vCPU cores required to handle a certain number of requests per second vary linearly for different types of microservices.

Table 2: Minimum and maximum time taken for determining MSC for different Microservices

Micrserivce Name	Minimum Time (t2.nano, 100, 1 replica)	Maximum Time (t2.xlarge, t2.small, 3 replicas)
primeapp	23 minutes	3 hours 7 minutes
movieapp	8 minutes	3 hours 5 minutes
serveapp	33 minutes	2 hours 33 minutes
webacapp	9 minutes	1 hours 40 minutes

of resource-limited pods to the equivalent instances also reveals similarity in the performance. Fig. 3a shows the number of requests per minute versus the CPU utilization of pods with *t2.nano* instance limits when deployed on t2.xlarge, t2.large, and t2.medium. The requests percentile 90 response time captured for each requests per minute is shown in Fig. 3b. The figure points out that the exhibited performance is similar in all the conducted tests.

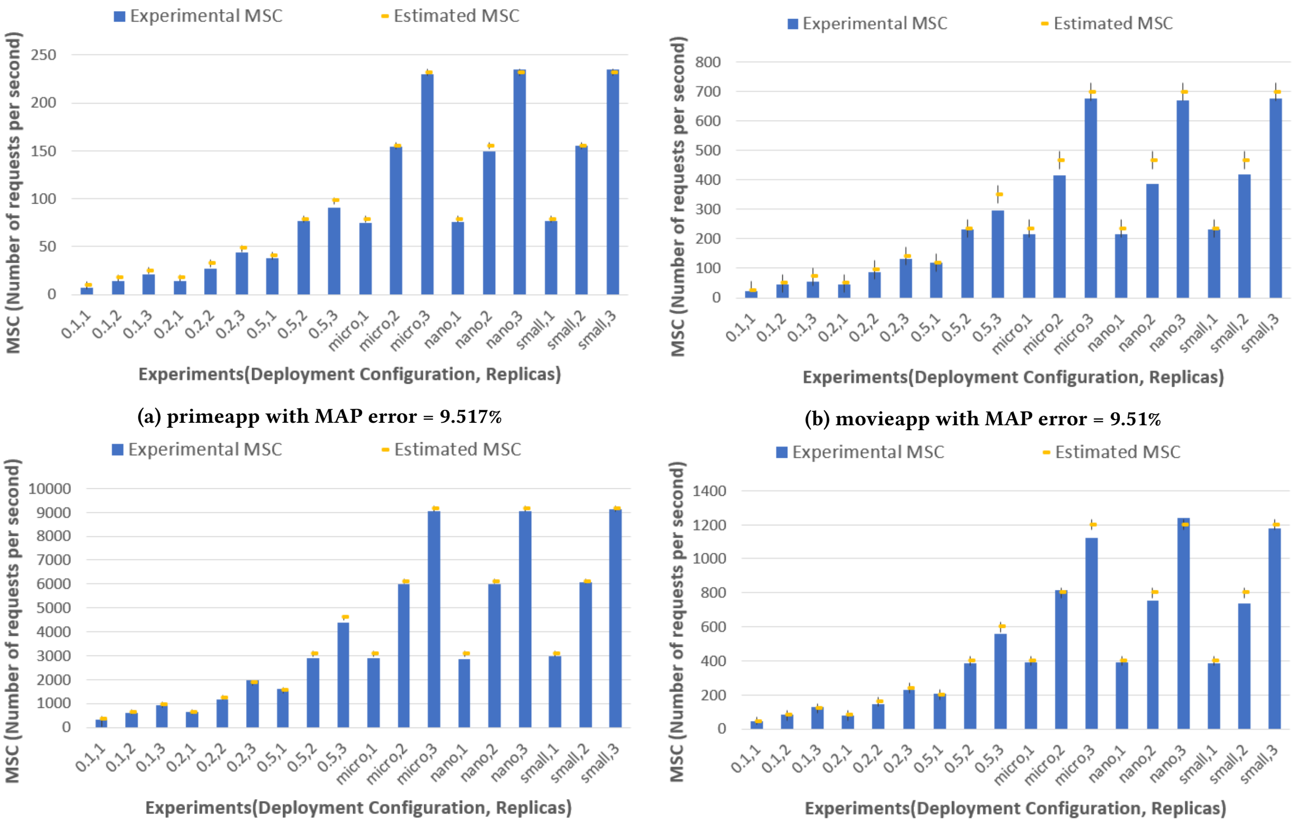
It was found that the number of virtual CPU cores required to process the requests grows linearly as shown in Fig. 4. The change in the number of requests did not affect the memory utilization significantly; it becomes constant for all the microservices after a certain threshold is reached.

Fitted regression models estimated MSCs and replicas quite accurately. Fig. 5 compares estimated MSC versus the experimental value for all the microservices along with the mean absolute percentage error (MAPE). Similarly, the fitted model for predicting the number of replicas estimated the replicas with root mean squared error (RMSE) of 0.1, 0.11, 0.12 and 0.11 for primeapp, movieapp, webacapp and serveapp respectively. RMSE value less than 1 (replica) shows high accuracy of the model.

Table 3: Actual capacity for the combined application and estimated capacity for its microservices

Config.	Replicas	Combined	primeapp	movieapp	webacapp	serveapp
0.10	1	5.3	9.52	25.51	334.5	39.8
0.10	3	7.8	24.8	71.83	942.5	119.6
0.20	1	13.7	17.2	48.67	638.5	79.8
0.20	2	25.9	32.4	95.0	1246.8	159.7
0.50	1	32.8	40.1	118.1	1550.6	199.7
0.50	3	105.5	97.7	349.7	4590.4	599.4
t2.micro	1	64.5	78.4	233.9	3070.5	399.5
t2.nano	2	126.7	155.1	465.5	6110.4	799.2
t2.nano	3	192.4	231.6	697.2	9150.4	1198.8

Fig. 5 highlights that the highest rate of change of the CPU utilization is achieved for **primeapp**. This points at the **primeapp** as the potential computational bottleneck microservice. Table 3 shows the actual capacity for the combined application as well as the predicted MSC for the sandboxed individual microservices. These data indicate that the overall capacity of the application is limited by the bottleneck microservice. Also, individual MSCs acquired for different deployment configurations allow to determine the amount of



(a) primeapp with MAP error = 9.517%

(b) movieapp with MAP error = 9.51%

(c) webacapp with MAP error = 2.61%

(d) serveapp with MAP error = 4.96%

Figure 5: MSC Actual vs Estimated for Microservices along with error distribution

microservices replicas needed to increase the combined application performance. For example, if microservices are deployed with the configuration of CPU and memory equal to 0.1, then approximately 37 replicas for **primeapp** (334.5/9.52), 14 replicas for **movieapp** (334.5/25.51), 9 replicas for **serveapp** (334.5/39.8) and 1 replica for **webacapp** will be enough to raise the combined application capacity to the capacity of the webacapp frontend of 334. Depending upon the resources limitation, one can estimate how many replicas are required for microservices for the desired overall as well as the individual microservices capacity and performance.

7 DISCUSSION

The results point out several research hypotheses for future extensive proof on different microservices and on their combinations.

As was shown, the performance of a pod with the limited instance resources nearly equals that of a VM with the same virtual resources. Therefore, it is not necessary to conduct the exhaustive load tests on every VM type. VM type with a large capacity can be selected to house the pods with various limits. The capacity of these pods will still be limited by the capacity of the VM, therefore more powerful VMs are preferred.

Sandboxing microservices allows to understand the performance of the individual microservices constituting the application. MSC determined for each sandboxed microservice could be used to identify the bottleneck microservice. The identified microservice could be horizontally scaled or modified to match the performance of other microservices. Microservices with similar capacity are candidates for scaling in groups. The individual MSC values known for each microservice are the necessary prerequisite for the accurate fine-grained microservice application capacity planning.

Capacity prediction is a promising technique implemented in Terminus; it helps to determine the individual MSC of each application microservice without testing all possible deployments. Along with the forecast of the user workload, its results can be used in predictive autoscaling [5].

The same technique allows to determine the number of replicas required to handle the given number of user requests in reactive autoscaling systems. Reactive IaaS autoscaling dynamically adjusts number of VMs based on the cluster's load. When the metric stays higher than a threshold, the reactive autoscaling engine adds VMs to the cluster. The number of VMs adjusted in the scaling action is called *scaling adjustment number*. Scaling adjustment can either be some constant or could specified as a percent of the current VMs count [24]. The predicted number of replicas can be used to compute the scaling adjustment number automatically.

Knowing the capacities of the microservices enables the distributed autonomous applications management. However, the relatively high accuracy in determining the capacities exhibited by the sandboxing approach comes at a price - the time necessary for the load testing and performance modeling. This renders the proposed approach limited in use for the applications with the frequent structural changes. In addition, accurate prediction models demand a lot of preliminary tests. The prospective direction is to classify the microservices and to create the performance models libraries.

8 RELATED WORK

Joydeep et al. claim that the performance delivered by AWS is unpredictable when running web applications [18]. This statement points at the necessity to capture the real performance. However, neither native cloud monitoring nor the advanced performance monitoring solutions like CloudMonix [4] support evaluation of the performance for different types of resources and deployments.

Detection of a bottleneck component is another essential task for increasing the performance of the cloud application. Bhuvan et al. presented an analytical model for the bottlenecks detection and the performance prediction of multi-tier applications [25]. Method presented in [17] uses the capability profiling to identify the potential resource bottlenecks and make recommendations regarding achieving adequate performance. These wide-spread analytic approaches to bottlenecks detection emphasize the need for human involvement to interpret the results of the analysis.

Some systems detect the performance anomalies in the deployed applications. For example, Root is a system to automatically identify the root cause of performance anomalies in web applications deployed in Platform-as-a-Service (PaaS) clouds [11].

9 CONCLUSION & FUTURE WORK

Performance modeling of microservice applications allows to determine the capacity distribution among the microservices. This enables fine-granular capacity planning for applications and the detection of the bottleneck microservices.

The approach and the Terminus tool proposed in the paper aim to solve the problem of finding the best-suited resources for the microservice to be deployed on so that the whole application achieves the best performance at the same time minimizing the resources consumption. After analyzing the results acquired by the Terminus tool on the example application consisting of 4 microservices, it was identified that the microservices follow a common pattern in the performance versus the workload - the performance degrades slowly with the increase in the workload up until a certain point when all the virtual resources are exhausted. Commonly for the tested microservices, the CPU utilization increased linearly with the increase in the number of requests.

The main focus of the further work is to incorporate Terminus into the predictive autoscaling framework enabling one of the two main inputs - capacity estimate for the scalable components of the cloud application, i.e. microservices. The other input to the predictive autoscaling is the forecasted workload.

ACKNOWLEDGMENTS

This work was supported by the AWS Cloud Credits for the research program. The authors thank anonymous reviewers.

REFERENCES

[1] Peter Arijs. 2018. How to use resource requests and limits to manage resource usage of your Kubernetes cluster. Retrieved 2-October-2018 from https://jaxenter.com/manage-container-resource-kubernetes-141977.html

[2] André Bauer, Nikolas Herbst, and Samuel Kounev. 2017. Design and Evaluation of a Proactive, Application-Aware Auto-Scaler: Tutorial Paper. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 425–428. https://doi.org/10.1145/3030207.3053678

[3] Betsy Beyer, Niall Richard Murphy, David K. Rensin, Kent Kawahara, and Stephen Thorne. 2018. *The Site Reliability Workbook: Practical Ways to Implement SRE* (1st ed.). O'Reilly Media, Inc., Farnham, UK.

[4] Cloudmonix. 2018. CloudMonix. Retrieved 9-May-2018 from http://www.cloudmonix.com/

[5] Danny Yuan Daniel Jacobson and Neeraj Joshi. 2013. Scryer: Netflix's Predictive Auto Scaling Engine. Retrieved 29-September-2018 from https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270

[6] Elastic. 2018. Elasticsearch. Retrieved 25-September-2018 from https://www.elastic.co/products/elasticsearch

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

[8] Heapster. 2018. Heapster. Retrieved 25-September-2018 from https://github.com/kubernetes/heapster

[9] Influxdata. 2018. InfluxDB. Retrieved 25-September-2018 from https://www.influxdata.com/time-series-platform/influxdb/

[10] D. Jaramillo, D. V. Nguyen, and R. Smart. 2016. Leveraging microservices architecture by using Docker technology. In *SoutheastCon 2016*. 1–5. https://doi.org/10.1109/SECON.2016.7506647

[11] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 469–478. https://doi.org/10.1145/3038912.3052649

[12] K6. 2018. K6. Retrieved 25-September-2018 from https://docs.k6.io/docs

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

[14] Kubernetes. 2018. Kubernetes Operations. Retrieved 25-September-2018 from https://github.com/kubernetes/kops

[15] Kubernetes. 2018. Managing Compute Resources for Containers. Retrieved 2-October-2018 from https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

[16] kubernetes.io. 2018. What is Kubernetes. Retrieved 3-May-2018 from https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[17] Frank Leymann, Uwe Breitenbücher, Sebastian Wagner, and Johannes Wettinger. 2017. Native Cloud Applications: Why Monolithic Virtualization Is Not Their Foundation. In *Cloud Computing and Services Science*, Markus Helfert, Donald Ferguson, Victor Méndez Muñoz, and Jorge Cardoso (Eds.). Springer International Publishing, Cham, 16–40.

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

[19] V. Podolskiy, A. Jindal, M. Gerndt, and Y. Oleynik. 2018. Forecasting Models for Self-Adaptive Cloud Applications: A Comparative Study. In *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*.

[20] Vassilis Prevelakis and Diomidis Spinellis. 2001. Sandboxing Applications. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 119–126. http://dl.acm.org/citation.cfm?id=647054.715767

[21] Chris Richardson. 2015. Introduction to Microservices. Retrieved 25-May-2018 from https://www.nginx.com/blog/introduction-to-microservices/

[22] F. Samreen, Y. Elkhatib, M. Rowe, and G. S. Blair. 2016. Daleel: Simplifying cloud instance selection using machine learning. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. 557–563. https://doi.org/10.1109/NOMS.2016.7502858

[23] Amazon Web Services. 2018. Amazon EC2 Instance Types. Retrieved 25-September-2018 from https://aws.amazon.com/ec2/instance-types/

[24] Amazon Web Services. 2018. Simple and Step Scaling Policies for Amazon EC2 Auto Scaling. Retrieved 29-September-2018 from https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.html

[25] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. 2005. Dynamic Provisioning of Multi-tier Internet Applications. In *Second International Conference on Autonomic Computing (ICAC'05)*. 217–228. https://doi.org/10.1109/ICAC.2005.27

[26] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. 2013. Detecting Transient Bottlenecks in n-Tier Applications through Fine-Grained Analysis. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. 31–40. https://doi.org/10.1109/ICDCS.2013.17