A Cost-Efficient Multi-cloud Orchestrator for Benchmarking Containerized Web-Applications

Devki Nandan Jha(✉), Zhenyu Wen, Yinhao Li, Michael Nee, Maciej Koutny, and Rajiv Ranjan

Newcastle University, Newcastle upon Tyne, UK
{d.n.jha2,zhenyu.wen,y.li119,maciej.koutny,raj.ranjan}@ncl.ac.uk,
info@michael-nee.co.uk

Abstract. Benchmarking the containerized web-applications across multiple cloud gives web-application owners more chance to deploy their applications on cheaper host while meeting their performance requirements. However, benchmarking a large number of cloud hosts (about 267 cloud providers in the world) to find a flexible deployment option becomes a grand challenge. Users need to evaluate as many hosts as possible to find an option which offers expected performance at the lowest price. It is also necessary to benchmark the hosts for longer duration so that it can capture the uncertainty of cloud environment.

In this paper, we present **S**mart **D**ocker **B**enchmarking **O**rchestrator (**SDBO**), a general orchestrator that automatically benchmarks containerized web-applications in multi-cloud environment. At the same time, SDBO is able to maximize the numbers of evaluated cloud providers and type of hosts without exceeding users' budgets. Moreover, we propose a *flexible execution* module which enhances SDBO's ability to capture the performance variation of benchmark web-application for longer period of time in the defined users' budgets.

Keywords: Cloud computing · Benchmarking · Orchestrator · Web-application

1 Introduction

Evolution of microservice architecture that modularizes the application into smaller independent components gives the flexibility for developers to implement each component as a standalone service. Every microservice component in the web-application chain can communicate either via synchronous (HTTP/HTTPS) or asynchronous (AMQP) network communication protocols depending on the level of desired *component autonomy*. Note that many cloud providers such as Amazon and Microsoft offer containers virtualized at the operating system level which facilitates the deployment of microservices i.e. each component of the web-application can be encapsulated into a container. Since containers have

R. Cheng et al. (Eds.): WISE 2019, LNCS 11881, pp. 407–423, 2019.
https://doi.org/10.1007/978-3-030-34223-4_26

many advantages including light-weight, fast start up/shut down, packaged; as a result, users can move their web-applications fast and deploy them efficiently.

However, the multi-cloud environment provide diverse options for users to deploy their web-applications, which means users have more chance to find a cheaper host which still meets their deployment requirements such as cost, throughput, latency. To this end, the users need to test the performance in these hosts before actually deploying and publishing their web-applications. The common practice is to use the standard benchmarking applications to test the hosts instead of using users' own application. This is because these benchmarking applications have the standard procedures to evaluate the performance of the host, thereby obtaining more comprehensive results. Moreover, benchmarking all the hosts from different cloud providers is very challenging as each provider has their own architecture and programming interface [9]. Existing research [10,14] focuses mainly on evaluating the benchmark web-application on different host configurations alone. However, [3,6,13] discuss some frameworks that provide the automatic systems to perform the benchmark across multiple clouds.

Web-application is a long running system and its performance must be guaranteed all the time. On the other hand, the underlying cloud environment is very dynamic and resource preemption happens frequently in the virtualized environment [8]. The performance is also affected by the interference caused by other applications deployed on the same server [5]. Observing the performance variation for a longer duration is an important task for benchmarking web-application. Unfortunately, running the benchmark applications in various hosts over different clouds for a longer duration (say at least 24 h) is very costly. To the best of our knowledge, we could not find any study that considers *cost efficiency* for benchmarking i.e. maximize the number of evaluated hosts and benchmarking time within a defined budget.

In this paper, we aim to build a smart orchestrator for benchmarking containerized web-applications in multi-cloud. SDBO is designed to solve the complexity of deploying benchmark applications in multi-cloud environment that have different programming APIs and numerous ways to interact. To achieve the *cost efficiency*, first, we develop an algorithm that maximizes the number of evaluation hosts based on users' budgets and pre-defined benchmarking time. Then, the *flexible execution* module is designed to capture the performance variation of cloud environment by partitioning the pre-defined benchmarking time into a set of slots. In summary, this paper makes the following contributions:

- We developed a novel orchestrator, SDBO that automates the definition and execution of benchmarks for containerized web-applications. In particular, the orchestrator allows the user to choose the benchmark applications and hosts across different cloud providers.
- SDBO has a native feature of optimization that maximizes the utility of user's budget by maximizing the number of cloud providers and the hosts for benchmarking.

– Based on the optimized execution plans, we interact the plans with the *flexible execution* module to run the benchmarks in a set of time interval thereby capturing the performance variation for longer duration.

2 Related Work

The web-application benchmarks need to be deployed on various host configurations in the multi-cloud environment. Orchestrating the systematic deployment consists of following steps [16]: (i) defining the benchmark with their attributes and relationships, (ii) defining the host machine configuration (e.g. CPU cores, location), (iii) instantiating the cloud host complying the application requirements, (iv) monitoring the resources to ensure the QoS and SLA parameters, and (v) controlling the overall processes. Performing all these steps manually is tedious, error-prone and requires a lot of time and diverse knowledge of architecture and accessing mechanism of all these environments. There are different frameworks available that automate/semi-automate the orchestration steps. [7,12] evaluated the performance of containers for scientific applications where a few of them [17,18] evaluate for big data applications. However, most of these works are intended for single cloud environment, without considering the complexity of interacting with various APIs/SDKs provided by different cloud providers.

There are few existing frameworks that handle the orchestration of benchmarks in multi-cloud environment. CloudBench [13] and Smart CloudBench [3] automates the benchmark execution in multi-cloud environment. However, it is not easy to define the benchmarks using these frameworks. Also they are not specific for containerized environment. Additionally, Varghese et al. proposed a framework called DocLite [15] to evaluate the performance of VMs using containerized microbenchmarks. Microbenchmarks are executed on different VMs and the ranking is evaluated by using the set of weights provided by the user for different system parameters. This framework is specific for scientific application and may not be applicable for web-application. Our proposed SDBO orchestrates the benchmark for web-application while allowing users to define and deploy the benchmark in a very interactive and user-friendly way.

Additionally, there are some commercial tools, e.g. CloudHarmony[1] available that perform the benchmark for users but are not specific for particular application. Also, they do not provide all the required metrics specific to that particular application for making proper decisions before final resource selection and provisioning. The limitations of the existing work are briefly summarized as follows.

Limitations. The cloud providers offers shared computing resources to their customers, which makes the cloud environments dynamic and the SLA very hard to guarantee [11]. Moreover, the web-application is very sensitive to the

[1] https://cloudharmony.com/.

dynamically changing environment that directly affect user's satisfaction. Capturing or monitoring the changing behavior of the cloud environments requires the users to run their benchmark applications over a considerable time, which is very costly. Existing benchmark frameworks are not able to solve the trade-off between the limited budgets and the long-time benchmarking experiments.

Additionally, the variety of cloud providers offer a massive configuration choices of host. For instance, Amazon EC2 provide 43 types of host for their customers excluding self customized hosts. It is not possible to run the benchmark applications over all available resources. The state-of-the-art systems do not consider this case that provides an optimized recommendation to help users in selecting the hosts from the massive number of available hosts spanned across multiple cloud providers.

3 System Overview

This section discusses the architecture and system design details of SDBO.

3.1 SDBO Architecture

Figure 1 illustrates the architecture of SDBO and the dependencies of each component. SDBO is implemented as a web-application that provides a *User Interface* for users to interact, explore and manage their benchmarking experiments. The *User Interface* allows the user to choose an existing benchmark application or customize a new application. Moreover, users can easily select the available hosts from different cloud providers, define the benchmarking time for each selected host, and specify the total budget for running the experiments. Next, this configuration information is stored in a relational Database.

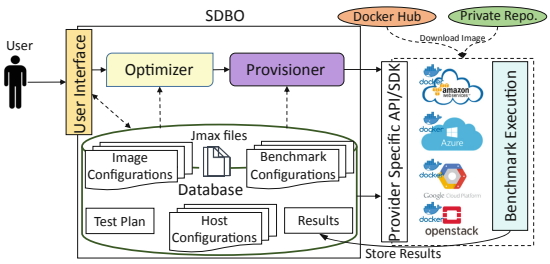


Fig. 1. System architecture of SDBO

The *Optimizer* is designed to create an optimized host list based on the information provided by the user. It retrieves the necessary information (host configurations, benchmark duration and budget) from the Database and applies a heuristic algorithm to generate an optimized host list for running the benchmarking experiments. More details about the *Optimizer* are given in Sect. 3.2.

The generated host list is automatically stored in the Database. Next, users can choose the *flexible execution* option for benchmark execution. If the user chooses to execute the benchmark experiments, the *Provisioner* will be triggered to provision the resources, deploy the benchmark applications and execute the applications based on the user entered information and optimized host list. The benchmark is executed for the specified interval of time and the completion is notified to the *Provisioner*. The results are stored in the Database in real-time for further evaluation and analysis. Finally, the user is notified after completion of the benchmarking experiment and following that cloud resources are released. The main steps of the execution workflow of SDBO is shown in Fig. 2.

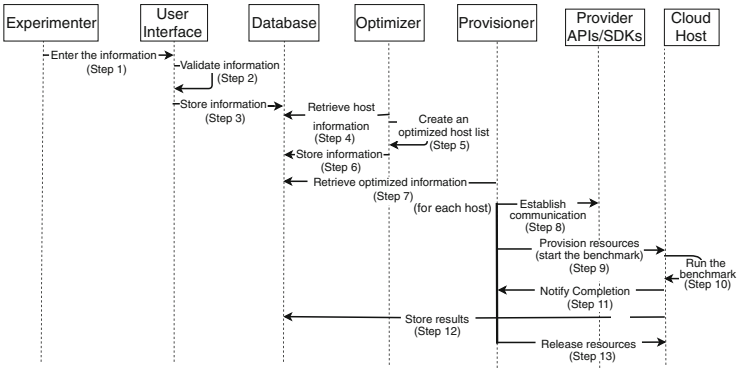


Fig. 2. SDBO execution workflow

3.2 SDBO Design

Optimizer and Its Formal Model. SDBO benchmarks containerized web-application in a multi-cloud environment. Let N represent the number of cloud providers $C_i|i \in \{1, N\}$ where each provider C_i has T type of hosts $v_{i,t}|t \in \{1, T\}$. In our model, we assume a one-to-one mapping between host and container. Consider $\mathcal{C}(v_{i,t})$ to be the unit cost of using $v_{i,t}$, $\tau_{i,t}$ is the time units for which $v_{i,t}$ is chosen to run and $\mathcal{B}$ is the user budget for the benchmark, finding an optimal set of hosts for the benchmark is modelled as a Binary Integer Linear Programming problem (BILP). The defined objective function is given in Eq. 1 subject to constraints as given in Eq. 1a–1c.

$$\textbf{maximize:} \sum_{\mathbf{i=1}}^{\mathbf{N}} \sum_{\mathbf{t=1}}^{\mathbf{T}} \mathbf{x_{i,t}} + \lambda \sum_{\mathbf{i=1}}^{\mathbf{N}} (\sum_{\mathbf{t=1}}^{\mathbf{T}} \mathbf{x_{i,t}} - \mathbf{T}) \tag{1}$$

$$\sum_{i=1}^{N}\sum_{t=1}^{T}(\mathcal{C}(v_{i,t}) \times \tau_{i,t}) \leq \mathcal{B} \tag{1a}$$

$$\forall i \; \forall t \; \tau_{i,t} \geq 0 \tag{1b}$$

$$\forall i \sum_{t=1}^{T} x_{i,t} \geq 1, \; \forall t \sum_{i=1}^{N} x_{i,t} \geq 1 \tag{1c}$$

Where, $x_{i,t}|x_{i,t} \in \{0,1\}$ is a binary variable which represents whether $v_{i,t}$ is selected or not. The first factor of the optimization problem is to comprehend maximum selection of hosts and the second considers a penalizing factor to boost the spanning of maximum number of cloud providers. λ is a tunable parameter which is incorporated to maintain a balance.

Constraint 1a states that the total cost of benchmarking different containers running inside the host must be less than the defined budget. Also, the cost is calculated only if $x_{i,t}$ is 1 with a positive execution time for host $v_{i,t}$ (constraint 1b). Finally constraint 1c enforces the selection of at least one cloud provider and at least one host configuration.

We developed and implemented a heuristic algorithm for the *Optimizer* to solve the problem formalized above. The algorithm generates an optimized list of hosts while satisfying all the defined constraints. The details about how to create an optimized list of hosts is discussed in Algorithm 1. It first calculates the total cost, $\mathcal{C}^T(v_{i1,t1})$ for each selected host, $v_{i1,t1}$ (line 4). It then performs a local sorting (using merge sort) for each selected cloud provider, $i1$ according to the increasing host cost and stores it in a temporary list, $List_{i1}$ (line 6). Following that it selects a host with minimum cost globally and adds to the final host list, $V2$ (line 14, line 24) until the final_cost is less than budget, $\mathcal{B}$ (line 8). To maintain the fairness and diversity among different cloud providers, there is a provision to add a penalty if the cloud has been selected (line 17). A host is selected only if the penalty imposed to that cloud is less than a defined value (100 for our case) or if there is no other providers left for selection (line 10).

Provisioner. Once the *Optimizer* generates a benchmark plan, the users can decide whether they want to submit the plan for execution via the user friendly web interface. If the user agrees to perform the experiment, the functions implemented in *Provisioner* will be triggered. First, the *Provisioner* will check the connection and the requirement of the resources on different clouds. Next, it uses a background process application, Hangfire to create and launch the hosts on the selected cloud providers.

Flexible Execution. SDBO offers two types of execution strategy (a) *solitary execution* and (b) *manifold execution*. *Solitary execution* is the basic strategy where users can set a particular time interval for evaluating the benchmark on the desired host configuration. The performance evaluation in this case is limited as it executes only for the particular time interval. We know that the host's QoS performance is highly dependent on the system parameters, e.g. current

Algorithm 1: *optimizer*

Input: $V1$ - list of hosts $v_{i1,t1}$ selected by the user, $\tau_{i1,t1}$ - time for executing the benchmark on host $v_{i1,t1}$, $\mathcal{C}(v_{i1,t1})$ - unit cost of using host $v_{i1,t1}$, $\mathcal{B}$ - budget

Output: $V2$ - optimized list of hosts

$\forall i1\ fine_{i1} = 0$, $V2 = []$, $final\_cost = 0$
for *each selected provider* $i1$ **do**
  for *each selected host type* $t1$ **do**
    $\mathcal{C}^T(v_{i1,t1}) = \mathcal{C}(v_{i1,t1}) \times \tau_{i1,t1}$
  end
  Sort the host $v_{i1,t1}$ in ascending order of total cost $\mathcal{C}^T(v_{i1,t1})$ using Merge sort and store in a list, $List_{i1}$
end
while $(final\_cost \leq \mathcal{B})$ **do**
  Search the first element of all list and find the host $v_{i1',t1'}$ with smallest cost
  if $(fine_{i1'} > 100\ \&\ \forall i1\ (!empty(List_{i1})))$ **then**
    Skip $List_{i1}$ from current calculation
    continue
  else if $(fine_{i1'} \leq 100\ \&\ \forall i1\ (!empty(List_{i1})))$ **then**
    Add $v_{i1',t1'}$ to $V2$
    Delete $v_{i1',t1'}$ from the list $List_{i1'}$
    $final\_cost = final\_cost + \mathcal{C}^T(v_{i1',t1'})$
    $fine_{i1'} = fine_{i1'} \times 10$
    for $(\forall\ i1\ <>\ i1')$ **do**
      if $(fine_{i1} \geq 10)$ **then**
        $fine_{i1} = fine_{i1}/10$
      end
    end
  else
    Add $v_{i1',t1'}$ to $V2$
    Delete $v_{i1',t1'}$ from the list $List_{i1'}$
    $final\_cost = final\_cost + \mathcal{C}^T(v_{i1',t1'})$
  end
end

workload, network state, etc. which may vary with time [4]. This variation is especially significant for the web-applications due to the continuous execution and the resource preemption in the virtualized environment.

To capture this variation, we propose *manifold execution* strategy that executes the benchmark application in the same host but in multiple time intervals. The user is asked to define the number of iterations along with other parameters for the optimizer. The optimizer then generates an optimized list of hosts which is associated with the execution timestamps. As a result, the *Provisioner* can schedule the deployment and execution based on the host configurations and its associated execution timestamps.

4 Metrics Profiling

SDBO can support benchmarking for different type of web-applications including e-commerce, social media and banking system. It does not only capture the basic web-application features, e.g. response time, throughput illustrated in Sect. 4.1, but also supports more complex and advanced metrics (see Sect. 4.2).

4.1 Basic Metrics

Response Time (ΔT). Response time is the total time taken by the web-application to process a request and generate its response. It is a basic metrics to evaluate the performance of any web-application. Normally, response time depends on many factors varying from the host infrastructure, scheduling policy and the current load on the system to the host capability and network capacity to handle a user's request. Average response time $\mu(T)$ and standard deviation of the response time $\sigma(T)$ are used frequently to measure the performance of the web-applications. Lower response time represents better performance.

Throughput (TP). Throughput represents the host performance in terms of number of requests that can be handled per unit time. Consider that there are total N number of sample requests which are successfully executed in Δt time interval where $\Delta t = (Start\ time - Finish\ time)$, throughput is calculated as $TP = N/\Delta t$.

CPU Usage (CPU). It gives the percentage of CPU used by the container while executing the process. We obtain this information from docker stats APIs [1] embedded with our orchestrator.

Memory Usage ($Memory$). Docker stats APIs also allow us to obtain percentage of memory used by the monitored container.

Network Throughput (Net). This metric indicates how much data can be transferred from a client to the target container in a unit time interval and is represented in Mega bits per seconds (Mbps).

Block I/O (I/O). Block input/output refers to the amount of data written to or read from the block storage devices in a unit time interval and is also represented in Mbps. We collect Net and I/O also from Docker stats APIs.

4.2 Advanced Metrics

Based on the collected basic metrics which are stored in our database, users can perform more complex queries to profile the complex systems.

Apdex Score. Apdex (Application Performance Index)[2] is considered as an open standard developed to standardize the methods for benchmarking, tracking

[2] http://www.apdex.org/index.html.

and reporting the application performance. It utilizes the Response Time (ΔT) to check the user satisfaction level for an application's performance. Based on a defined threshold for the response time **T**, Apdex defines three acceptable zones namely Satisfied, Tolerated or Frustated.

An Apdex score is calculated using the number of requests satisfied and tolerated out of the total requests received. The contribution of satisfied and tolerated requests for the user satisfaction level is 100% and 50% respectively. Let NR, SR and TR be the total number, satisfied number and tolerated number of requests respectively, an Apdex score is calculated as given in Eq. 2. The value of an Apdex score lies between 0 and 1 with higher values representing better satisfaction levels.

$$Apdex\ Score = (SR + TR/2)/NR \tag{2}$$

Host Stability. Stability of host machine is the metric to measure the consistency of the system performance. It is defined as the inverse of variability experienced by different basic metrics. Given the average μ_i and standard deviation σ_i for ith basic system metric ($i \in M$) executed for time T, variability is calculated as given in Eq. 3.

$$Variability = 1/T \sum_{t=0}^{T} \sum_{i=0}^{M} (\sigma_{i,t}/\mu_{i,t}) \tag{3}$$

Thereby, host stability is calculated as $Host\ Stability = 1/Variability$. Hosts with smaller stability values show that the performance is inconsistent and is not suggested for execution.

Host Suitability. Host suitability metric represents the worthiness of a host in terms of performance and cost. It is computed using Eq. 4.

$$Host\ Suitability = TP/Cost \tag{4}$$

where, TP is the throughput, $Cost$ is the per unit execution cost for that particular host. The higher the value of host suitability the better is the host.

5 Evaluation

To illustrate the effectiveness of SDBO, we performed a case study using a simple web-application benchmark. The details are presented in the section below.

5.1 Experiment Setup

SDBO is tested both in simulation and on a real testbed. The simulation is to test the scalability of our proposed optimization algorithm, and the real testbed is to evaluate the system performance. The experiment setup is detailed as follows.

Scalability Evaluation. Our algorithm is tested on a Lenovo PC with Intel(R) Core(TM) i5-6200U CPU @2.3 GHz - 2.4 GHz with 16 GB memory and 512 GB

Table 1. Experiment host configuration

CSP	Host Type	CPU cores	Memory	Disk	Price/hr($)
AWS	t2.nano	1	0.5	EBS	0.0066
	t2.micro	1	1	EBS	0.0132
	t2.small	1	2	EBS	0.026
	t2.medium	2	4	EBS	0.052
	t2.large	2	8	EBS	0.1056
	m4.large	2	8	EBS	0.116
	t2.xlarge	4	16	EBS	0.2112
	c4.xlarge	4	7.5	EBS	0.237
	m4.2xlarge	8	32	EBS	0.464
	c4.2xlarge	8	15	EBS	0.476
Azure	Standard_B1s	1	1	2	0.0118
	Standard_B1ms	1	2	2	0.0236
	Standard_B2s	2	4	4	0.0472
	Standard_F2	2	4	8	0.119
	Standard_B2ms	2	8	4	0.0944
	Standard_D2_v3	2	8	4	0.116
	Standard_B4ms	4	16	8	0.189
	Standard_A4_v2	4	8	8	0.222
	Standard_B8ms	8	32	16	0.378
	Standard_D8_v3	8	32	16	0.464

SSD. We collected 20 host configurations from AWS and Azure as the input dataset as shown in Table 1.

Benchmark Application and Its Deployment. SDBO is published on Google Cloud App Engine (*B2 instance class*) London (europe-west2). Therefore, the users can access to the system from any place and run their benchmarking applications via the user interface. PostgreSQL Database is associated with the SDBO, and stores different configuration of hosts and benchmark images for running the experiments. The database is also deployed on a Google cloud *n1-standard-2* instance with 2 vCPUs, 7.50 GB memory, 128 GB disk. All these components are running independent following the microservice architecture.

We utilized a popular benchmark application, TPC-W[3] with SimplCommerce that emulate the activities of a sample e-commerce web-application. This application is containerized and used to benchmark various type of hosts on AWS and Azure as shown in Table 1. The load on the web-application is created by Apache JMeter[4] according to the test plans defined by the user. To emulate real traffic, JMeter is not configured on the same cloud where the benchmark applications are running. The containerized load generator is deployed on Digital Ocean cloud and the host is a *Standard droplet* machine with 6 vCPUs, 16 GB memory and 320 GB SSD disk.

[3] https://cs.nyu.edu/~totok/professional/software/tpcw/tpcw.html.
[4] https://jmeter.apache.org/.

5.2 Cost Optimization

In this section, we evaluate the performance of our optimizer which aims to maximize the number of hosts within the constraint of users' budgets and predefined benchmarking time.

To highlight the advantages of the optimizer, we considered 20 host configurations from AWS and Azure (see Table 1). Moreover, we assume that the user would like to run their benchmarking experiment for 3.5 h, with four different budgets $ 0.5, $ 1.0, $ 1.5 and $ 2.0. We compared the performance of our optimized selection method (*Opt*) with the random selection method (*Rand*).

Figure 3 demonstrates that the optimized option selects the higher number of host in all the cases, compared to the random selection method. The reason is because the optimizer always selects the host with lower price first and then it moves to higher cost host. This is based on the logic that a user wants to deploy their web-application on the cheapest hosts that can meet their QoS requirements. Our algorithm design fits to this logic very much. However, the random selection method selects any host which may not be cost optimized. In addition to this, our method can provide a more stable numbers of hosts as shown in Fig. 3, where the *Opt* has much smaller variance than *Rand*.

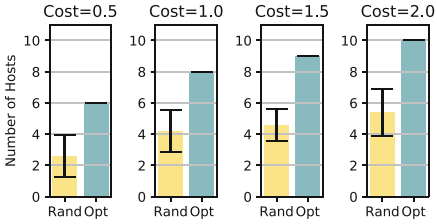


Fig. 3. Comparing the optimized result with random selected result

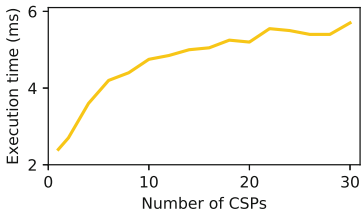


Fig. 4. Schematic diagram showing the execution time complexity of the Optimizer

We also evaluate the scalability of our algorithm by simulating a scenario with varying number of cloud providers with each provider having 50 different host configurations available. Figure 4 shows the execution time of different cases with increasing number of cloud providers varying from 1 to 30. The result shows that the execution time only increases linearly as the number of cloud providers increases. Moreover, the maximal execution time is 5.7 ms for 30 cloud providers, which is comparatively very small as compared to the deployment time.

5.3 Basic Metrics Profiling

In this subsection, we present the benchmark results of an optimized test case. We select a subset of the hosts from Table 1 as the input to our optimizer that

then generates 6 hosts (highlighted with gray color in Table 1) for benchmarking experiments.

To obtain the throughput of each deployed benchmark application, we emulated the *bursty request*, i.e. we send the maximal number of requests to the web-applications simultaneously without causing any response error. In other words, the web-applications are fully saturated. Table 2 shows the maximal number of requests for each selected host. Figure 5 illustrates the value of basic metrics (as specified in Sect. 4.1) of the selected hosts, collected from the experiments.

Table 2. Number of requests to saturate the host

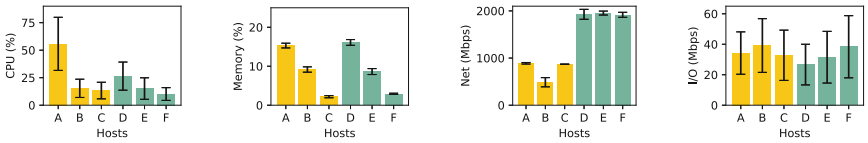
CSP	Seq	Host	No. of req (to saturate)	CSP	Seq	Host	No. of reg (to saturate)
AWS	A	t2.small	300	Azure	D	Standard_B1ms	300
	B	t2.medium	600		E	Standard_B2s	600
	C	t2.xlarge	1500		F	Standard_B4ms	1500

CPU Usage. Figure 5(a) shows the CPU usage of each selected host. The result clearly shows that CPU usage decreases as we increase the size of host. Also, the more powerful hosts have less variation of CPU usage. For example, the variance of the CPU usage for the big size hosts C and F is only 7% and 5% and that for small host A and D reaches 24% and 12% respectively. Except for small sized hosts, the performance of AWS to Azure is almost comparable. For the small size host, there is a huge performance difference (52.5% degradation) as Azure has less CPU usage compared to AWS for processing the same number of request.

Memory Usage. The memory usage (Fig. 5(b)) also shows the similar trend except the variation which is much less (highest is 0.76 for host E) as compared to CPU usage. Highest memory usage is noticed for host D followed by host A with 16.1% and 15.3% respectively. Note that the memory usage is varying only in the initial phase, after that the usage is almost constant. It is caused by the property of the Docker container, the memory once allocated is not released back until the container is terminated or restarted.

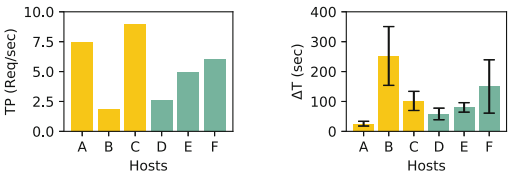
Network Throughput. The result in Fig. 5(c) shows that the hosts from Azure have about twice the network throughput, compared to the hosts from AWS. The hosts from the same cloud provider have the same network throughput except for host B from AWS where the throughput is much less (487.2 Mbps) than others A (889.2 Mbps) and C (869.2 Mbps).

I/O Throughput. As compared to the above three basic metrics, block I/O shows different trends (see Fig. 5(d)). The I/O throughput of AWS hosts is very random which is because of the selected hosts that offers EBS support. Thus, the throughput is also affected by the time elapsed between the I/O requests and EBS server [2]. Moreover, our benchmark application is block I/O intensive, so



(a) CPU Usage (b) Memory Usage (c) Network (d) Block I/O

Fig. 5. Basic container system metrics while specifying the workload to 300 requests per second with ramp up period as 0 s. CPU and memory usage are given in percentage while network and block I/O throughput are in Megabits per second (Mbps). Black bar on top represents the standard deviation.



(a) Throughput (b) Response Time

Fig. 6. System throughput and response time.

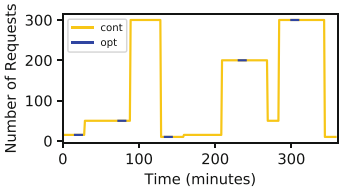


Fig. 7. Workload pattern for continuous and optimized execution

the collected statistics are not the maximal I/O throughput of each host. This is demonstrated very well in Azure hosts (see D, E, F in Fig. 5(d)) that the I/O throughput increases with the increase in the number of requests.

System Throughput. Figure 6(a) illustrates the throughput of different hosts. It is clearly depicted from the figure that the throughput increases linearly with the capacity of the host and AWS hosts have comparatively better throughput than Azure for similar sized machine except for host *B*. The reason of lower throughput for host *B* is the degraded network throughput as depicted in Fig. 5(c). Therefore, the network becomes the bottleneck in this case. The highest throughput achieved by AWS host is 8.93 requests per second (host *C*).

Response Time. Figure 6(b) shows the results of the response time. The results show that the response time is significantly affected by the network throughput and the number of requests. Figure 5(c) show that host *B* has the worst network throughput which causes the significantly higher response time as shown in Fig. 6(b), i.e. 252.2 s. If the network throughput is constant, the response time increases with the increase of number of requests. Note that we try to saturate the web-application until it reaches the maximal number of requests that it can handle without causing errors. Thus, a large number of requests are queuing and waiting for being processed, and this is the main reason that causes the high response time for many requests.

5.4 Advanced Metrics Profiling

In this subsection, we compute different advanced metrics based on the collected basic metrics that can help in selecting the cloud provider and the hosts for the actual deployment.

Apdex Score. We calculate the Apdex score for same case as discussed in Sect. 5.3. Since we are considering the case of a saturated system where the response time is high, we set the threshold for the response time to 50 s. The higher the Apdex score, the better users' satisfaction. Table 3 shows the Apdex score for all selected hosts. The result clearly shows that smaller hosts have better Apdex scores as compared to larger hosts. We have explained why the smaller hosts have lower response time (see Sect. 5.3 **Response Time**). The lowest score of 0.15 is noticed for host B due to its bad network throughput (see Fig. 5(c)).

Table 3. Advanced metrics profile

CSP	Host seq	Apdex score	Host stability	Host suitability	CSP	Host seq	Apdex score	Host stability	Host suitability
AWS	A	1	1.115	286.399	Azure	D	0.8	0.921	109.979
	B	0.15	0.790	36.313		E	0.5	0.772	104.470
	C	0.5	0.834	42.311		F	0.4	0.846	32.042

Host Stability. A host with a higher stability value is considered best as it signifies less performance variation with the elapsed time. The result in Table 3 shows that the stability of small and large host instances are higher. The highest value is for host A with stability index of 1.115 followed by host D with the index of 0.921. The worst stability index is for host E with the value of only 0.772.

Host Suitability. Host suitability is computed as discussed in Eq. 4. A host with higher suitability value is considered to be better as it provides better throughput to cost ratio. The suitability index for the selected hosts show a downward trend with increasing size. For both AWS and Azure, smaller machines have better suitability values as shown in Table 3.

5.5 Flexible Execution

Continuous execution of a benchmark for longer duration is the best way to capture the performance variation. However, the benchmarking cost in this case is very high. To capture the performance variation of changing environment in a defined budget, SDBO offers the *flexible execution* module.

We do not have access to the cloud hypervisor, therefore, we are not able to emulate the resource changing or preemption of the hosts. As an alternative, we emulate the performance variation of our web-application by changing the number of requests in a period of time. If our tool can observe the performance

variation with the changing number of requests, it can also capture the variations that may be caused by other reasons.

To this end, we define 9 test plans, each plan is defined with a timestamp and the number of requests need to be sent as shown in Fig. 7 depicted by *Cont*. For example, the first plan is to send 15 requests starting at 00:00 min timestamp. Following that, the second plan sends 50 requests starting at 30:00 min timestamp. We keep the web-application (benchmark application) running for 360 min to cover all timestamps from the test plans. For the *Opt* case, we randomly selected 5 test plans and sort them based on the timestamp as shown in Fig. 7 and Table 4. The web-application (benchmark application) is executed for 10 min, if and only if the timestamp is reached. The above described two experiments were executed simultaneously with the same host configuration (AWS $t2.medium$).

Table 4 shows response time and throughput collected from both scenarios. The result clearly shows that SDBO can capture the same performance with a maximal variation of 15% in Case I for response time and 5.9% in Case III for throughput. The cost for the optimized method is much less than continuous way of deployment as the total time of deployment for *Opt* is only 50 min as compared to 360 for *Cont*.

Table 4. Comparison of Optimized; *Opt* and Continuous; *Cont* method for Response time and Throughput. Values in [] represent standard deviation.

Case	No. of req	Response time (sec)		Throughput (req/sec)	
		Cont.	Opt.	Cont.	Opt.
Case	10	0.684 [±0.29]	0.789 [±0.36]	6.71	6.54
Case II	15	2.114 [±0.10]	2.122 [±0.15]	6.23	6.17
Case III	50	4.105 [±0.13]	4.441 [±0.18]	10.49	9.87
Case IV	200	21.381 [±1.34]	22.482 [±2.04]	6.90	6.59
Case V	300	23.463 [±6.13]	24.649 [±4.18]	7.56	7.25

6 Conclusion

To facilitate web-application benchmarking in multiple cloud with cost efficiency and flexibility, we proposed SDBO which is the first cost-efficient web-application benchmarking orchestrator. SDBO provides the smart user interface that expedites the handling of benchmark even for a non-expert user. Also, the cost optimization offered by the orchestrator helps the user to select a variety of hosts while *flexible execution* captures the long time performance variation in a limited budget.

Future work. The *flexible execution* module allow users to execute their benchmark applications at any pre-defined timestamp. We can leverage this feature and develop an advanced sampling method to collect the system metrics that

can feed to some machine learning methods to have a better observation of the uncertainty in cloud environments. Moreover, we will extend our orchestrator to benchmark other applications such as stream processing and big data applications.

References

1. Docker stats. https://docs.docker.com/engine/reference/commandline/stats/. Accessed 17 Apr 2019
2. I/o characteristics and monitoring. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-io-characteristics.html/. Accessed 17 Apr 2019
3. Chhetri, M.B., et al.: Smart cloudbench—a framework for evaluating cloud infrastructure performance. Inf. Syst. Front. **18**(3), 413–428 (2016)
4. Duan, Q.: Cloud service performance evaluation: status, challenges, and opportunities - a survey from the system modeling perspective. Digit. Commun. Networks **3**(2), 101–111 (2017)
5. Jha, D.N., Garg, S., Jayaraman, P.P., Buyya, R., Li, Z., Morgan, G., Ranjan, R.: A study on the evaluation of HPC microservices in containerized environment. Concurrency Comput. Pract. Experience e5323 (2019)
6. Jha, D.N., Nee, M., Wen, Z., Zomaya, A., Ranjan, R.: SmartDBO: smart docker benchmarking orchestrator for web-application. In: The World Wide Web Conference, pp. 3555–3559. ACM (2019)
7. Kozhirbayev, Z., Sinnott, R.O.: A performance comparison of container-based technologies for the cloud. Future Gener. Comput. Syst. **68**, 175–182 (2017)
8. Leitner, P., Cito, J.: Patterns in the chaos - a study of performance variation and predictability in public iaas clouds. ACM Trans. Internet Technol. **16**(3), 15:1–15:23 (2016)
9. Ranjan, R., Benatallah, B., Dustdar, S., Papazoglou, M.P.: Cloud resource orchestration programming: overview, issues, and directions. IEEE Internet Comput. **19**(5), 46–56 (2015)
10. Scheuner, J., Cito, J., Leitner, P., Gall, H.: Cloud workbench: benchmarking IaaS providers based on infrastructure-as-code. In: Proceedings of the 24th International Conference on World Wide Web, pp. 239–242. ACM (2015)
11. Serrano, D., et al.: SLA guarantees for cloud services. Future Gener. Comput. Syst. **54**, 233–246 (2016)
12. Sharma, P., Chaufournier, L., Shenoy, P., Tay, Y.C.: Containers and virtual machines at scale: a comparative study. In: Proceedings of the 17th International Middleware Conference, Middleware 2016, pp. 1:1–1:13 (2016)
13. Silva, M., Hines, M.R., Gallo, D., Liu, Q., Ryu, K.D., Da Silva, D.: CloudBench: experiment automation for cloud environments. In: 2013 IEEE International Conference on Cloud Engineering (IC2E), pp. 302–311. IEEE (2013)
14. Sobel, W., et al.: Cloudstone: multi-platform, multi-language benchmark and measurement tools for web 2.0. In: Proceedings of CCA (2008)
15. Varghese, B., Subba, L.T., Thai, L., Barker, A.: Doclite: a docker-based lightweight cloud benchmarking tool. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 213–222. IEEE (2016)
16. Weerasiri, D., Barukh, M.C., Benatallah, B., Sheng, Q.Z., Ranjan, R.: A taxonomy and survey of cloud resource orchestration techniques. ACM Comput. Surv. (CSUR) **50**(2), 26 (2017)

17. Xavier, M.G., Neves, M.V., De Rose, C.A.F.: A performance comparison of container-based virtualization systems for mapreduce clusters. In: 2014 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 299–306. IEEE (2014)
18. Zhang, Q., Liu, L., Pu, C., Dou, Q., Wu, L., Zhou, W.: A comparative study of containers and virtual machines in big data environment. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 178–185. IEEE (2018)