An Approach and Case Study of Cloud Instance Type Selection for Multi-Tier Web Applications

Christian Davatz, Christian Inzinger, Joel Scheuner, and Philipp Leitner
Department of Informatics, University of Zurich
{firstname.lastname}@uzh.ch

***Abstract*—A challenging problem for users of Infrastructure-as-a-Service (IaaS) clouds is selecting cloud providers, regions, and instance types cost-optimally for a given desired service level. Issues such as hardware heterogeneity, contention, and virtual machine (VM) placement can result in considerably differing performance across supposedly equivalent cloud resources. Existing research on cloud benchmarking helps, but often the focus is on providing low-level microbenchmarks (e.g., CPU or network speed), which are hard to map to concrete business metrics of enterprise cloud applications, such as request throughput of a multi-tier Web application. In this paper, we propose Okta, a general approach for fairly and comprehensively benchmarking the performance and cost of a multi-tier Web application hosted in an IaaS cloud. We exemplify our approach for a case study based on the two-tier AcmeAir application, which we evaluate for 11 real-life deployment configurations on Amazon EC2 and Google Compute Engine. Our results show that for this application, choosing compute-optimized instance types in the Web layer and small bursting instances for the database tier leads to the overall most cost-effective deployments. This result held true for both cloud providers. The least cost-effective configuration in our study provides only about 67% of throughput per US dollar spent. Our case study can serve as a blueprint for future industrial or academic application benchmarking projects.**

I. Introduction

The commoditization of cloud computing infrastructure gave rise to a variety of Infrastructure as a Service (IaaS) providers. While compute resources from different services (e.g., Amazon EC2[1] or Google Compute Engine[2]) appear comparable on paper, their actual performance can vary significantly across providers [1], [2] and resource types [3]. Issues, such as hardware heterogeneity, contention, and virtual machine placement can result in considerably differing performance across supposedly equivalent resources [4].

Consequently, it is difficult to know a priori which set of resources from which provider(s) will offer the best cost/performance ratio for any given application. Previous work in cloud performance engineering either tries to predict application performance using simulation (e.g., [5]) or relies on gathering data using deployment and execution of benchmarks on cloud resources. The accuracy of simulation approaches can be low, as it is difficult to account for the intricacies of cloud infrastructure (e.g., resource sharing) in prediction models [6]. In contrast, data-driven approaches attempt to make cloud resources comparable by executing micro-benchmarks on a large number of instance types from multiple providers that assess specific aspects of the offered resources [7]. While this allows for objective comparison of distinct resource properties across providers, it is hard for practitioners to map their applications' characteristics to such micro-benchmark results directly. To address this problem, reference application benchmark approaches (e.g., [8], [9]) assess cloud infrastructure using realistic application deployments to give a complete picture of infrastructure performance across providers and resource types for a given workload. Naturally, such benchmark results are only directly applicable to applications and workloads that are very similar to the reference applications used to gather the initial performance measurements.

Hence, to accurately and confidently select the (combination of) IaaS instance types that offers the optimal cost and performance for an application, cloud customers are often required to conduct their own measurements. This includes using their own application (or a simplified version of it) and workloads that are realistic for the application. Unfortunately, even though multiple frameworks for defining and executing cloud benchmarks have been proposed in the past (e.g., Expertus [10], Cloud-Bench [11] CloudCrawler [12], or Cloud Workbench [13]), none of them sufficiently enables the execution of complex benchmarks on realistic multi-tier applications.

This paper contributes two-fold to the state of research. Firstly, we propose Okta[3], an approach and conceptual framework for fair and repeatable application benchmarking of IaaS instance types. In this paper, we focus specifically on transaction-oriented multi-tier Web applications, although we expect that our approach also generalizes to other application models such as Big Data. The framework is based on previously published best practices for cloud and Web benchmarking [14], [15]. Core tenets include cloud portability, scripting and automation, and a clear separation of benchmarking framework, test driver, and system-under-test (SUT). Secondly, to validate this approach, we use this framework to evaluate the deployment costs and performance of AcmeAir, a two-tier sample OLTP application. We compare the costs and performance of AcmeAir in six real-life deployment configurations on Amazon EC2 and five deployments on Google Compute Engine (GCE). We show that for this application, using compute-optimized instance types for the Web tier and

[1] https://aws.amazon.com/ec2/

[2] https://cloud.google.com/compute/

[3] An okta is a unit of cloud coverage in meteorology.

small bursting instance types for the database tier leads to the best performance per US dollar spent. From the 11 configurations in our study, the least cost-effective configuration only provides about 67% of throughput for the same monetary costs as the best-performing configuration. We have not observed a systematic difference between cloud providers, i.e., none of the providers is clearly cheaper or more expensive. Our case study makes use of various industry-strength tools for benchmarking, including Cloud Workbench [13] as benchmarking framework, Chef [16] for provisioning and automation, and Apache JMeter [17] as load generator. Hence, we argue that this case study can serve as a blueprint for future academic or industrial application benchmarking projects. Further, given that AcmeAir represents a typical Web application, we envision that various technical assets from our current study (e.g., Chef recipes), which we provide as open source software, can be reused or adapted for future projects.

II. Related Work

The research community has for a long time relied on open-source benchmarks for establishing the performance of Web applications, such as TPC-W [18], [19] and RUBiS [20], [21]. However, with the advent of Web 2.0, interactive content, mobile clients, and cloud computing, these benchmarks became outdated. Binning et al. [22] started a discussion why traditional benchmarks are insufficient for analyzing cloud services. They identify shortcomings of the common TPC-W benchmark and outline ideas on how to design such a new cloud benchmark.

Building upon this work, Folkerts et al. [14] provide guidelines on the subject of benchmark design and implementation and list general requirements and challenges for modern cloud benchmarks. Iosup et al. [23] describe a generic approach to IaaS and PaaS cloud benchmarking, namely a general benchmark architecture. While the architecture shows several participating entities in an abstracted way, it leaves the concrete implementation of the benchmark to the user.

Since TPC-W [18] and RUBiS [20], a number of new benchmarks have been proposed, such as TPC-E [24] and SPECWeb2009 [25]. However, as there were no open source or free implementations available to the general public, these benchmarks have only been used by commercial vendors [26]. Hence, and in response to Web 2.0 requirements, Sobel et al. [27] developed CloudStone, a toolkit consisting of a social Web 2.0 application, scripts for workload generation, and guidelines for computing the key metric dollars-per-user-per-month. Furthermore, results from Amazon EC2 show the maximum number of users for a particular setup consisting of a VM type and a software configuration [27]. But without capturing or emulating client-side JavaScript or AJAX interactions, an important aspect of current Web 2.0 applications falls short.

To address complex Web 2.0 interaction patterns, Cecchet et al. [28] propose BenchLab, an open-source benchmark suite based on multiple modern Web 2.0 applications. BenchLab provides as SUT several backends, which represent different domains [26] and are already known from existing benchmarks (RUBiS [21], TPC-W [18] and CloudStone [27]). Moreover, BenchLab [26], [28] provides an alternative, novel approach to the emulation of complex interactions. In contrast to other benchmarks targeting at Web applications, BenchLab makes use of real Web browsers in combination with Selenium to capture and emulate client-side JavaScript or AJAX interactions. To manage and orchestrate all components of the benchmark, and to store the results, BenchLab provides a Web-based user interface. In their studies, Cecchet et al. show the need to use real Web applications as benchmarks and present a tool that authentically reproduces user interactions [26], [28].

Smart CloudBench [29], [30] is a framework, which supports the whole benchmarking process from cloud provider selection to decommissioning of resources acquired during the benchmark setup. The Smart CloudBench application can be used to deploy a Java-based implementation of the TPC-W [18] benchmark to all cloud providers supported by Apache jClouds. From the data gathered with the benchmark, the performance of the tested instance types (i.e. instance sizes) is inferred for different scenarios.

Dejun et al. [3] study performance stability and performance homogeneity of VMs provided on Amazon EC2. While performance stability behaves as expected, performance homogeneity emerges as an issue. The study shows that the performance from VMs of the same type exhibits very heterogeneous performance profiles, up to a ratio 4 in response time from each other. While this is an issue concerning performance predictability, they believe that exploiting performance variability could result in an improvement of the overall resource usage. To simulate CPU-intensive (processing) and I/O intensive workload patterns, they develop three custom micro-service applications but withhold details about their concrete implementation [3].

To empirically evaluate our proposed multi-VM benchmark at scale, we require means to easily define and execute benchmarks over different cloud providers and in an automated manner. Previous work has proposed multiple approaches to achieve this, including Expertus [10], Cloud-Bench [11] CloudCrawler [12], Smart Cloud-Bench [29], [30], Cloud-Gauge [31], and BenchLab [26], [28]. Although either of these systems could have been used, we decided to use the Cloud Workbench [13] framework that we proposed in previous work.

III. A Framework for Application Benchmarking

In this section, we introduce Okta, our approach and conceptual framework for OLTP benchmarking. The primary goal of Okta is to provide a mechanism for fair and repeatable benchmarking of custom applications across IaaS instance types. A conceptual overview of the framework is shown in Figure 1 and consists of the following components: (i) System under Test (SuT), (ii) Benchmark Driver and Load Generator, (iii) Cloud Provider under Test, (iv) Benchmark Manager, and (v) Provisioner. As Okta is designed to be a black box

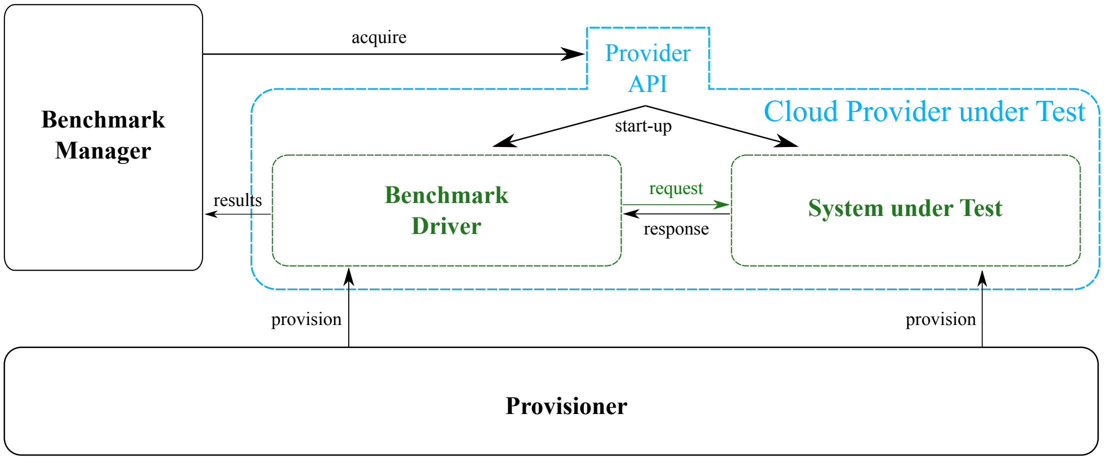

Fig. 1. An overview of the conceptual Okta benchmark approach and framework.

benchmarking approach, we do not require access to low-level server-side execution metrics, but measure performance solely using QoS metrics available at the client load generator. In general, the framework design does not mandate specific implementations for any of the components but can be instantiated using a combination of suitable components, as discussed in Section IV. In the following, we discuss the components of Okta in greater detail.

System under Test. Our framework is designed to allow for benchmarking arbitrary applications. Hence, the SuT is primarily treated as a black box. The main requirement for the SuT is that it can be automatically deployed, configured, and started with a configuration management tool that can act as the provisioner as described below.

Benchmark Driver. During benchmark execution, the benchmark driver executes a configurable workload to simulate realistic application load. Conceptually, the benchmark driver is a load generator and is treated as a component that is automatically deployed alongside the SuT. Hence, the driver must support automated deployment and configuration so that the framework can inject benchmark workloads and application endpoints. The benchmark workload should be designed to represent typical application usage and exercise the application in a production-like manner. Configurable end-to-end integration tests can be composed to represent typical application users that can be independently replicated to represent arbitrary load on the application.

Cloud Provider under Test. Okta is designed to support benchmark deployments on cloud infrastructure of providers. The central requirement is the ability to programmatically launch resources (i.e., VM instances) by the Benchmark Manager. One important consideration is the preselection of candidate providers. It would be infeasible to benchmark an application on a large number of cloud providers and all available instance types due to the resulting significant investment of time and money.

Benchmark Manager. The benchmark manager coordinates deployment, execution, and gathering of results. This component provides the primary user interface for our framework and serves as its central point of interaction. The user supplies the deployment manifest for the SuT along with workload definitions to the benchmark manager, and can further specify benchmark execution parameters. These include, but are not limited to, number and type of different instances and cloud providers to evaluate. Further, schedules to gather application benchmark data over longer periods of time to identify any performance differences caused by varying load on the provider infrastructure need to be provided.

Provisioner. The provisioner is responsible for deploying and configuring the components of the SUT and load generator on infrastructure instantiated by the cloud provider under test. We rely on state of the art configuration management tools to perform this task, e.g., Ansible [32], Chef [16], or Puppet [33]. Okta requires a configuration management tool that supports multi-host deployments to successfully provision and configure all necessary components.

IV. Case Study

We now substantiate the general approach outlined in Section III using a concrete case study. We base our case study on AcmeAir, a sample two-tier airline ticketing application that is designed to support public cloud deployments, offer scalable APIs, and allow for interaction using multiple user interfaces. AcmeAir thus represents a compelling industry example Web application[4]. Further, we use Cloud Workbench [13] (CWB) as Benchmark Manager, Chef as Provisioner, and Apache JMeter as Benchmark Driver. We evaluate various realistic deployment options in two public IaaS services, namely Amazon EC2 and Google Compute Engine. An overview of this instantiation of Okta is given in Figure 2. Note that our approach is general in the sense that a different Benchmark Manager (e.g., Expertus [10]), Provisioner (e.g., Puppet [33]), or load generator (e.g., Faban [34]) could have also been

[4] http://ispyker.blogspot.com/2013/05/announcing-acme-air-performance.html

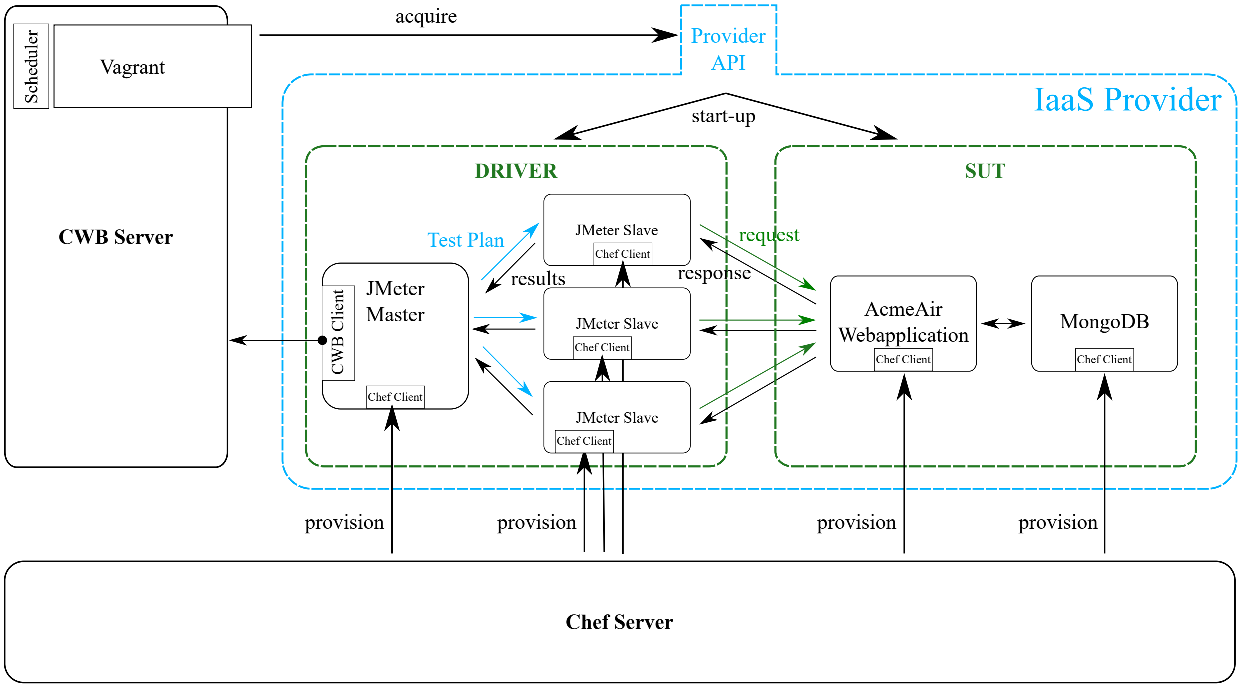

Fig. 2. An instantiation of our application benchmarking methodology for the AcmeAir application, using CWB, JMeter, and Chef.

used instead. We maintain an online appendix[5] for this paper, containing all Chef cookbooks for the entire case study, as well as all resulting raw data.

A. *System under Test*

Our example SuT AcmeAir is composed of two largely independent tiers, a Web application, and a database.

Web application. The Web tier is comprised of three distinct components: the user interface, a REST API, and a data service integrating the database. A key requirement of the Web tier is supporting multiple channels for user interaction, hence AcmeAir provides a GUI dedicated to traditional desktop browsers as well as a mobile app. The desktop browser GUI is based on current Web standards (HTML5, CSS3, JavaScript) and makes use of the Dojo JavaScript framework [35]. The mobile app is designed as a hybrid app to provide a consistent design across Android and iOS devices and also makes use of technologies such as Apache Cordova [36] to get access to mobile-only features (such as location and camera), allowing the mobile application to fully take advantage of device specifics. Further, the application provides a REST API which is based on IBM's Worklight server technology. The application uses IBM WebSphere Liberty [37] as application server for this tier.

Database. The backing database stores bookings, customers, sessions, flights, flight segments, and airports. During deployment, test data is seeded containing 394 flight segments connecting 31 airports. During the test run, bookings and sessions are dynamically generated as part of the application workload. Technically, two implementation alternatives are made available, a MongoDB 2.6.4 database or an IBM WebSphere Extreme Scale 8.6.0.8 backend. We have selected the MongoDB backend for our case study.

B. *Benchmark Manager.*

The benchmarking lifecycle is managed by CWB [13], [38]. Its Web interface allows us to periodically schedule benchmark executions over the course of multiple weeks using Cron expressions. CWB uses Vagrant [39] to acquire cloud resources (e.g., VM instances, disk volumes) based on the established Vagrant DSL. The subsequent Chef-based VM provisioning is highly configurable (e.g., dynamic IP address resolution) and thus suitable to support applications across multiple VMs. A lightweight CWB Client is automatically provisioned on one coordinating benchmark VM (e.g., JMeter Master in Fig. 2) to facilitate communication with the CWB Server (e.g., submitting results). CWB encourages to define benchmarks entirely in code, without relying on static assets such as prepared VM images. Thus, such benchmarks are easily portable across cloud providers.

C. *Provisioner*

As Provisioner, we have chosen to use Chef, primarily because CWB is itself based heavily on this tool and integrates well with it. Hence, we have built Chef cookbooks for all relevant components in Figure 2 (i.e., Benchmark Driver, the AcmeAir Web application, and database). All Chef cookbooks and other important artifacts are part of our online appendix.

D. *Benchmark Driver*

To generate representative workloads on the AcmeAir SuT for our measurements, we use a distributed setup of Apache

[5] https://sealuzh.github.io/benchmarking_online_appendix/

JMeter. We use a hierarchical topology with a single JMeter master node, which also serves as the interface of the Benchmark Driver to CWB, and a varying number of JMeter slave nodes. The master node aggregates benchmark outcomes from the slaves and sends the resulting data back to CWB. Similar to the SuT itself, this Benchmark Driver setup is provisioned via Chef, which allows us to regenerate the Benchmark Driver freshly for each benchmark run. To minimize the impact of latency on our results, we have chosen to provision the Benchmark Driver in the same IaaS cloud as the SuT (i.e., the EC2 benchmarks are generated with a Benchmark Driver in EC2, and the GCE benchmarks with a driver in GCE). Consequently, the resulting response times will be substantially faster than what is experienced by real users accessing the SuT from an outside network. However, given that the main driver behind this difference in perceived response time is the speed of the client's network link (and, consequently, out of our control), we have decided to exclude this factor from our study.

Workload Configuration. The request mix we used for the case study is derived from the workload originally employed in the AcmeAir example and consists of 9 different request types. The request mix covers all major features of AcmeAir in a realistic distribution to simulate real user sessions. During initial experiments, we have not found our experimental results to be particularly sensitive towards smaller changes in the used request mix.

- 1 × Login $[POST, 12\%]$
- ¼× Update Customer:
  View Profile Information $[GET, 3\%]$
  Update Customer $[POST, 3\%]$
- 5 × Query Flight $[GET, 50\%]$
- 1 × Book flight (if last query result valid) $[POST, 7\%]$
- 1 × List all Bookings $[GET, 8\%]$
- ¼× Cancel all but 2 Bookings $[POST, 10\%]$
- 1 × Logout $[GET, 7\%]$

Total GET $\simeq 68\%$
Total POST $\simeq 32\%$
¼= triggered every 4th iteration

Using this request mix, we define a "growing" workload [40], where the number of requests from the mix sent to the SuT per time unit steadily increases until saturation is reached. We define saturation as the point when on average, the SuT is unable to process requests faster than new requests arrive). This maximum number of requests the SuT can sustain long term using a given configuration is the metric we are primarily interested in our study.

E. Cloud-Provider-under-Test

Due to the substantial number of cloud providers that are currently in the market, the scope of our case study did not allow for a comprehensive comparison of all, or even of all large, public IaaS providers. Hence, we have chosen to study Amazon EC2 as a representative "market leader", and Google Compute Engine (GCE) as a "visionary" service which is still backed by a major company (see also Serrano et al. [41]). We have chosen to evaluate both cloud providers in their European regions, i.e., `eu-central-1` for EC2 and `europe-west1` for GCE. We evaluate 11 different hosting combinations, as outlined in Table I.

Configuration	**Webapp**	**DB**	**Costs**	**# of Runs**
$c \in C$			m_c	$\lvert c_r \rvert$
EC2				
A_gp2_1	m4.large	t2.small	\$0.173	37
A_gp2_2	m4.large	m3.medium	\$0.222	27
A_gp4	m4.xlarge	t2.small	\$0.315	23
A_co2_1	c4.large	t2.small	\$0.164	35
A_co2_2	c4.large	m3.medium	\$0.213	26
A_co4	c4.xlarge	t2.small	\$0.297	19
GCE				
G_gp1	n1-standard-1	n1-standard-1	\$0.110	26
G_gp2	n1-standard-2	n1-standard-1	\$0.165	26
G_gp4	n1-standard-4	n1-standard-1	\$0.270	24
G_co2	n1-highcpu-2	n1-highcpu-2	\$0.168	18
G_co4	n1-highcpu-4	n1-standard-1	\$0.223	23

TABLE I
INSTANCE TYPE CONFIGURATIONS

The column *Costs* indicates the total hourly costs in US dollars for both instances used in the configuration, but excluding any additional cost factors, such as disk IO, costs for elastic IP addresses, or dedicated hosting. Further, the costs here should be understood as the price of Linux on-demand instances at the time of writing in November 2016. The last column, *# of Runs*, shows how many data points we have collected for each configuration. The differences in the number of collected data points stems from (1) transitive errors during benchmarking in some runs (e.g., network link temporarily down), (2) pricing differences between configurations (we avoided collecting statistically unnecessary data for more expensive configurations), and (3) including valid data from early experimental benchmark runs.

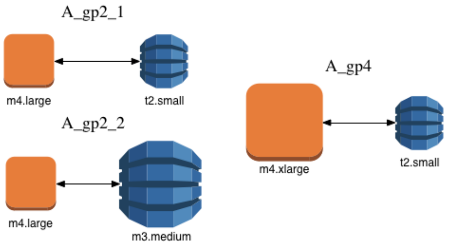

Fig. 3. Three example configurations. We combine different instance sizes for both components to identify which tier is most important to end-to-end performance.

The concrete configurations we have selected are various instance type combinations that are promising for hosting a two-tier application such as AcmeAir for small-scale workloads, as appropriate e.g., for a Web-based startup company. As a starting point for defining configurations, we have built on the knowledge we generated in previous studies [2], [8]. As shown in Figure 3, for EC2, we have explicitly evaluated various combinations of instance types for both, the Web and database tier. All configurations are in a similar range of total hourly costs (between \$0.110 per hour, or approximately \$80 per month, and \$0.315 per hour, or around \$230 per month). We have not evaluated more sophisticated setups, such as scale-out or high-availability setups with multiple instances in either, or both, tiers, although our experiment could be extended to include such configurations as well.

V. Benchmark Results and Discussion

After introducing the setup of our benchmarking case study, we now discuss concrete results for the instance type combinations listed in Table I.

A. Benchmarking Research Questions

Fundamentally, our case study answers three research questions.

***RQ 1**: What sustained performance, measured in throughput of successful requests per second, can we achieve with each configuration?*

Successful requests are defined to return the HTTP status code 200[6]. Further, note that we are specifically interested in what number of successful requests per second can be sustained indefinitely, and less in whether a configuration can handle short-term bursts. The main advantage of focussing on sustainable throughput is that this metric is largely independent of server-side timeout and queuing configurations. Further, this metric is (unlike other commonly-used Web application metrics, such as the response time) neither network sensitive nor sensitive on the request arrival rate, as long as more requests are arriving than can be served.

***RQ 2**: Can we observe statistically significantly different performance for each configuration?*

Given that all configurations have different hourly costs, it will be interesting to observe whether there are even statistically significant differences in the sustainable throughput between all configurations, especially among the ones where the same instance type is used for the Web application layer with a differing database instance type (e.g., `A_gp2_1` versus `A_gp2_2`).

***RQ 3**: Which configuration is the most cost-effective way to host AcmeAir for the defined workload?*

Following the reasoning of previous studies [2], [27], we need to not only observe the performance provided by each configuration, but also put it in contrast to the hourly costs of the configuration. Doing so will allow us to reason about which configuration can provide the best "bang for the buck", i.e., the largest sustainable request throughput per US dollar spent.

B. Used Metrics

To answer these research questions, we monitor and analyze the following two metrics:

Sustainable throughput. For each configuration $c \in C$, we use $\overline{SRPS}_c$ ("successful requests per second") as a metric of maximum sustainable throughput of the configuration. To monitor this metric, for each configuration $c \in C$ we execute $r \in r_c$ benchmark runs using an identical technical setup (referred to as *# of Runs* or $|r_c|$ in Table I). In each run, we gradually increase the pressure on the SuT by linearly increasing the number of requests per second over a time span t. We monitor how many successful requests per second (i.e., requests with a status code of 200) we receive back. This number flattens out at some point when the system starts to overload. We keep the run going with constant load for an additional defined time period t_s to ensure that a steady state has been reached, and then calculate $\overline{SRPS}_{c,r}$ for the run as the average of received successful requests per second (Equation 1, where $SRPS_{i,c,r}$ represents the successful requests per second for a configuration $c \in C$ at experiment second i in run r).

$$\overline{SRPS}_{c,r} = \frac{1}{t_s} \sum_{i=t}^{t+t_s} SRPS_{i,c,r} \tag{1}$$

This measurement procedure is illustrated for example runs of the configurations `A_gp2_1`, `A_gp4`, `G_gp2`, and `G_gp4` in Figure 4. In this example, the total duration of the experiment run is 5 minutes. After 140 seconds, all configurations have reached a steady state, so we calculate $\overline{SRPS}_{c,r}$ as the average successful requests per second between second 140 and 500. $\overline{SRPS}_{c,r}$ for each configuration is indicated with a black bar in the figure.

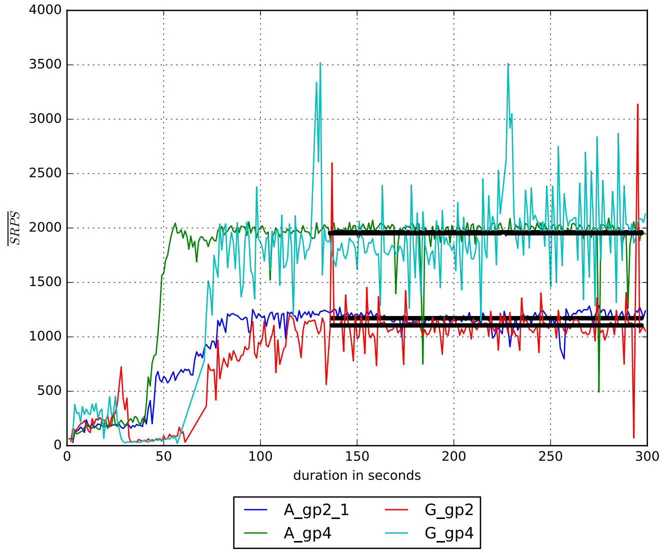

Fig. 4. Example experiment run for four configurations, including an illustrations of the calculation of $\overline{SRPS}_{c,r}$.

[6] https://tools.ietf.org/html/rfc7231#section-6.3.1

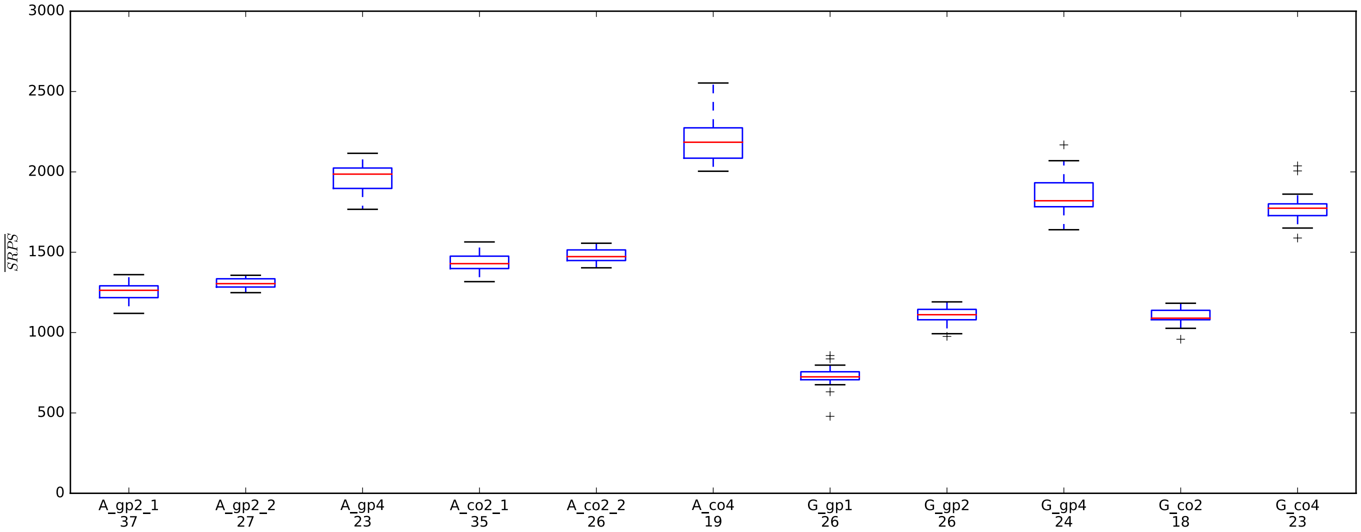

Fig. 5. Boxplots of $\overline{SRPS}$ for each configuration over all runs. The number on the x-axis is the sample size for each configuration.

Performance-cost index. We use the sustainable request throughput achieved per US dollar spent as a measure of cost-effectiveness. For simplicity and ease of comparison, we only focus on costs of computation and use regular on-demand hourly instance prices for Linux instances in the region we used. In our AcmeAir case study, the hourly compute costs m_c of a configuration $c \in C$ consist of the sum of the hourly costs of both used instances. We report on this total hourly price in Table I. It should be noted that using reserved or sustained usage instances, or instances acquired via the spot market, could drastically change the outcome, which we have not formally investigated in this paper.

$SRPS_c$ is the arithmetic mean of $\overline{SRPS}_{c,r}$ for all runs, as defined in Equation 2.

$$SRPS_c = \frac{1}{|r_c|} \sum_{r \in r_c} \overline{SRPS}_{c,r} \qquad (2)$$

We now define the performance-cost index of a configuration pci_c as in Equation 3. This performance metric is multiplied by 3600 to account for the fact that instances are priced by the hour, while $\overline{SRPS}_{c,r}$ represents throughput per second. This metric can be understood as successful requests that can be served per US dollar spent. As this metric evidently leads to large numbers, we report on the final performance-cost index as millions of requests per US dollar (i.e., we divide the final outcome by 10^6).

$$pci_c = \frac{3600 * SRPS_c}{10^6 * m_c} \qquad (3)$$

C. Results

We now present the main results regarding all three RQs.

RQ 1 – Comparison of Performance. Figure 5 depicts $\overline{SRPS}_{c,r}$ for all configurations and runs using boxplot notation. On the x-axis, we have also given the sample size for each configuration (i.e., $|r_c|$, the number of runs). Given the different sizes of instances and their associated costs, we observe strongly varying results for sustainable throughput without surprise. The lowest-performing configuration in our test was `G_gp1` with an $SRPS_c$ of 722.21. The best-performing configuration was `A_co4` with an $SRPS_c$ of 2192.07. The remaining configurations vary between those two extreme values, with most configurations being able to serve between 1000 and 2000 successful requests on average per second.

Note that we have observed very low variability of outcomes between runs for most configurations. Concretely, we have observed relative standard deviations of less than 2% (e.g., `A_co2_2` and 5% (e.g., `G_co4`) between runs. This is substantially more stable than previous studies have reported for micro-benchmarks [2]. Interestingly, the three overall best-performing (and expensive) configurations (`A_gp4`, `A_co4`, and `G_gp4`) are also the ones with the highest relative standard deviations in the study.

RQ 2 – Statistical Significance. To establish whether the performance differences we have observed for different configurations are also statistically significant, we have conducted crosswise Mann-Whitney-U tests for all pairs. Mann-Whitney-U has been selected as this statistical test is non-parametric and does not assume normal distribution of samples. We consider a pair of configurations to provide statistically significantly different performance if Mann-Whitney-U leads to a p-value of 0.05 or less. Further, for statistically significantly different pairs, we also establish the effect size r, with the standard labels of $r \leq 0.1$ representing a *small* effect, $0.1 < r \leq 0.5$ representing a *medium* effect, and $r > 0.5$ a *large* effect. These effect sizes have been color-coded in the table, with green shading representing a large effect, and yellow a medium effect. The red shading represents that no statistical significance was established.

The results of these crosswise tests are summarized in

		A_gp2_1	A_gp2_2	A_gp4	A_co2_1	A_co2_2	A_co4	G_gp1	G_gp2	G_gp4	G_co2
A_gp2_2	p =	0.000378117									
	r =	0.45									
A_gp4	p =	1.04E-10	1.60E-09								
	r =	0.84	0.85								
A_co2_1	p =	4.24E-12	2.70E-09	1.66E-10							
	r =	0.82	0.76	0.84							
A_co2_2	p =	1.97E-11	4.49E-10	2.23E-09	0.001301028						
	r =	0.85	0.86	0.86	0.41						
A_co4	p =	1.25E-09	1.12E-08	2.94E-06	1.81E-09	1.47E-08					
	r =	0.81	0.84	0.72	0.82	0.85					
G_gp1	p =	1.97E-11	4.49E-10	2.23E-09	3.39E-11	6.55E-10	1.47E-08				
	r =	0.85	0.86	0.86	0.85	0.86	0.85				
G_gp2	p =	6.50E-10	4.48E-10	2.23E-09	3.39E-11	6.54E-10	1.47E-08	6.54E-10			
	r =	0.78	0.86	0.86	0.85	0.86	0.85	0.86			
G_gp4	p =	5.84E-11	1.03E-09	0.036057377	9.60E-11	1.46E-09	6.46E-07	1.46E-09	1.46E-09		
	r =	0.84	0.86	0.31	0.84	0.86	0.76	0.86	0.86		
G_co2	p =	3.14E-08	1.93E-08	5.80E-08	3.49E-09	2.49E-08	2.21E-07	2.49E-08	0.496293569	4.31E-08	
	r =	0.75	0.84	0.85	0.81	0.84	0.85	0.84		0.85	
G_co4	p =	1.04E-10	1.60E-09	8.41E-05	1.84E-10	2.52E-09	7.34E-08	2.23E-09	2.23E-09	0.005132678	5.80E-08
	r =	0.84	0.85	0.58	0.84	0.85	0.83	0.86	0.86	0.41	0.85

TABLE II
P-VALUES PROVIDED BY MANN-WHITNEY-U TEST RESULTS AND EFFECT SIZES.

Table II. All pairs of configurations lead to statistically significantly different performance, with the exception of G_gp2 versus G_co2. These two configurations are indeed technically quite similar, and are also similarly costed. Further, most pairs are significantly different with large effect sizes, with the exceptions of A_gp2_1 versus A_gp2_2, A_co2_1 versus A_co2_2, A_gp4 versus G_gp4, and G_co2 versus G_co4, which only differ with a medium effect size. From these, the pairs A_gp2_1 versus A_gp2_2 and A_co2_1 versus A_co2_2 are particularly interesting, as these are configurations with identical instances for the Web tier, but different database instances. From these results, we can conclude that the decision which cloud provider and instance types to use has a substantial impact on the possible request throughput. Further, we conclude that the instance choice for the database tier has a noticeable, albeit lower, impact on the sustained throughput.

RQ 3 – Performance per US dollar spent. In isolation, the performance values observed in Figure 5 mean little, as they ignore the costs of each configuration. Hence, we now compare the configurations based on their performance-cost index pci_c. The resulting ranking is displayed in Table III, sorted from best to worst pci_c. These results show that there are indeed substantial differences in cost-effectiveness when comparing configurations, which further supports the need for benchmarking to foster effective instance type selection. The most cost-effective configuration for AcmeAir is A_co2_1, i.e., combining a c4.large instance for the Web tier with a t2.small database instance in the EC2 cloud. The configuration with the worst performance-cost index in our study was A_gp2_2, i.e., a m4.large for the Web tier and m3.medium for the database, again in the EC2 cloud. This worst-performing configuration provides only about 67% of the performance per US dollar spent in comparison to the best configuration.

Notably, our results indicate that configurations that use CPU-optimized instance types for the Web layer outperform general-purpose configurations. Further, we see that opting for a larger database instance is not cost-effective. The G_co2 configuration, which uses GCE's entry-level CPU-optimized instance types for both tiers is substantially less cost-effective than similar configurations that use a general-purpose instance type for the database. Finally, the best-performing configuration is A_co2_1, which is the second-cheapest configuration overall. However, the second-best configuration G_co4 is among the more expensive configurations in the setup. Hence, we feel that neither price nor performance is in isolation a good indicator of cost-effectiveness.

Configuration	**Avg. Throughput**	**Costs**	**Mio. Requests per $**	**Rank**
$c \in C$	$SRPS_c$	m_c	pci_c	
A_co2_1	1417.09	$0.164	31.107	1
G_co4	1791.98	$0.223	28.929	2
A_co4	2192.07	$0.297	26.571	3
A_gp2_1	1247.37	$0.173	25.957	4
G_gp4	1888.37	$0.270	25.178	5
A_co2_2	1472.01	$0.213	24.879	6
G_gp2	1102.49	$0.165	24.054	7
G_gp1	722.21	$0.110	23.636	8
G_co2	1095.28	$0.168	23.470	9
A_gp4	1939.74	$0.315	22.168	10
A_gp2_2	1302.83	$0.222	21.127	11

TABLE III
COMPARISON OF COST PERFORMANCE INDICES

D. Discussion

We now discuss the main lessons learned from our study.

The importance of benchmarking. Our results show that there are indeed substantial differences, both in the performance and in the cost-effectiveness of different configurations. This further supports the need for benchmarking to foster effective instance type selection. In our study, the least cost-effective configuration for AcmeAir provides only about 67% of the performance per US dollar spent in comparison to the best configuration. However, when comparing not the most extreme configurations, but ones that are more in the middle of the ranking, the difference becomes much more negligible. For instance, the difference between A_co2_2 on rank 6 and G_gp2 on rank 7 is only about 3.5%. Hence, even though substantial savings can be achieved in some cases, there are still some configurations that are functionally identical regarding performance per US dollar spent. In these cases, secondary selection criteria, such as a preference for one provider over another, can be used without significant costs.

No clear cheaper cloud provider. Another interesting observation is that there is no obvious ordering of cloud providers in our results. That is, neither EC2 nor GCE are per se cheaper than the other provider. Contrarily, comparable offerings from both services (e.g., A_co4 and G_co4) typically provide similar performance per US dollar spent. This is in stark contrast to the often-repeated common knowledge in the cloud market that GCE is using a particularly aggressive pricing strategy to attack EC2's dominant market position[7].

No easy rules of thumb for instance type selection. In our study, configurations that use CPU-optimized instance types for the Web layer outperform general-purpose configurations. Further, we have seen that in our case study choosing the cheapest, specifically bursting, database instance type is cost-effective. This result is in line with a previous study of bursting cloud instances [42], where we have already argued that these instance types are particularly well-suited for database applications. However, both of these results are due to specifics of our case study and used workload rather than general rules that can be followed blindly for all application models. Users should conduct their own experiments specific for their applications and workloads, for instance using the Okta approach and a benchmarking tooling such as CWB.

E. Threats to Validity

As with every empirical research, there are some limitations and threats to the validity of our case study, which we discuss in the following.

Construct validity. Designing a cloud benchmarking study requires deciding on a large number of parameters, starting from which providers and instance type combinations to evaluate, which request mixes and workload patterns to use, all the way down to deciding how to technically configure the SuT (e.g., setting maximum queue lengths of the Web server). To mitigate this threat, we have carefully constructed our experiments based on existing guidelines [7], [43], [44] as well as based on our own previous experience [2], [8], [42]. Another challenge inherent to cloud benchmarking is that the performance of cloud providers changes over time, typically without external notice. Hence a reader should utilize our case study as an illustration of a general method of cloud benchmarking, and not as a numerical ground truth that can be trusted to accurately represent future cloud provider performance or costs for all applications.

Internal validity. The major threat to the internal validity of our results is that we have executed all benchmarks in a relatively short time period of approximately one month in spring 2016. This means that our results could have been influenced by intermittent quality-of-service problems of the providers during this time. However, we are not aware of any publicized issues of either selected provider in this time span, so we judge this threat to be low.

External validity. We have selected AcmeAir as a representative two-tier Web application. It is an open question to what extent our results generalize to other application models (e.g., single-VM applications, scientific computing, or latency-sensitive applications). Further, a reader should take care to not generalize our results to cloud providers, regions, or instance types that have not been evaluated in the study.

Reproducibility. Accurately reproducing cloud benchmarking studies tends to be hard, as it is hard for study authors to report on all of the many parameters and experiment design choices that may influence their results in the scope of a scientific paper. We mitigate this problem by open sourcing the tool we use to execute benchmarks, and by providing all benchmark code and configurations in an online appendix.

VI. Conclusion

In this paper, we have introduced Okta, an approach and conceptual framework for conducting benchmarking experiments of multi-instance IaaS applications. The goal of Okta is to foster application benchmarking that is fair, easy to repeat, and easy to port to different cloud providers or instance type configurations. Consequently, we have focused on a clear conceptual separation between SuT, Cloud Provider under Test, provisioning framework, benchmark driver, and benchmark manager. We have illustrated the Okta approach in a concrete case study, where we have evaluated AcmeAir, a two-tier sample application, in 11 instance type combinations in Amazon EC2 and Google's GCE. We used a combination of CWB, Apache JMeter, and Chef to instantiate Okta.

Our case study results showed that cloud benchmarking is indeed crucial to identify cost-effective combinations of IaaS instance types, as the least cost-effective configuration in our study provides only about 67% of throughput per US dollar spent of the best-performing configuration. Further, we have seen that using compute-optimized instance types for the Web tier and small bursting instances for the database tier is the most cost-effective way to run our case study for the envisioned workload. Finally, we have not observed a clear ordering of cloud providers. Instead, we have seen comparable costs for comparable performance for both providers.

Our work enables relevant follow-up studies. Most importantly, the results presented in this paper only reflect costs of computation, and exclude factors such as disk or network IO, or the costs of additional services (e.g., static IP addresses). Similarly, more research should be conducted on using Okta for application models other than multi-tier Web applications. For instance, we consider it an interesting research question whether our approach can also be applied to

[7] http://www.fool.com/investing/general/2014/10/08/google-incs-price-war-is-creating-a-headache-for-a.aspx

low-latency applications (e.g., game servers), or data-intensive applications [45].

Acknowledgments

The research leading to these results has received funding from the Swiss National Science Foundation (SNF) under project MINCA (Models to Increase the Cost Awareness of Cloud Developers).

References

[1] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, "More for your Money: Exploiting Performance Heterogeneity in Public Clouds," in *Proc. Symp. on Cloud Computing*. ACM, 2012, pp. 20:1–20:14.

[2] P. Leitner and J. Cito, "Patterns in the Chaos – a Study of Performance Variation and Predictability in Public IaaS Clouds," *ACM Trans. Internet Technol.*, vol. 16, no. 3, pp. 15:1–15:23, 2016.

[3] J. Dejun, G. Pierre, and C.-H. Chi, "EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications," in *Proc. ICSOC/ServiceWave 2009 Workshops*. Springer, 2010, pp. 197–207.

[4] L. Gillam, B. Li, J. O'Loughlin, and A. P. S. Tomar, "Fair Benchmarking for Cloud Computing Systems," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, no. 1, p. 1, 2013.

[5] A. Li, X. Zong, S. Kandula, X. Yang, and M. Zhang, "Cloudprophet: Towards Application Performance Prediction in Cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 426–427, 2011.

[6] M. Gonçalves, M. Cunha, N. C. Mendonça, and A. Sampaio, "Performance Inference: a Novel Approach for Planning the Capacity of IaaS Cloud Applications," in *Proc. 8th Int. Conf. on Cloud Computing*. IEEE, 2015, pp. 813–820.

[7] Z. Li, H. Zhang, L. O'Brien, R. Cai, and S. Flint, "On Evaluating Commercial Cloud Services: a Systematic Review," *Journal of Systems and Software*, vol. 86, no. 9, pp. 2371–2393, 2013.

[8] A. H. Borhani, P. Leitner, B.-S. Lee, X. Li, and T. Hung, "WPress: Benchmarking Infrastructure-as-a-Service Cloud Computing Systems for On-line Transaction Processing Applications," in *Proc. Int. Enterprise Distributed Object Computing Conference*, 2014, pp. 101–109.

[9] M. Cunha, N. Mendonça, and A. Sampaio, "Cloud Crawler: a Declarative Performance Evaluation Environment for Infrastructure-as-a-Service Clouds," *Concurrency and Computation: Practice and Experience*, 2016.

[10] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Q. Wang, J. Park, and C. Pu, "Expertus: a Generator Approach to Automate Performance Testing in IaaS Clouds," in *Proc. Int. Conf. on Cloud Computing*. IEEE, 2012, pp. 115–122.

[11] I. Silva-Lepe, R. Subramanian, I. Rouvellou, T. Mikalsen, J. Diament, and A. Iyengar, "SOALive Service Catalog: a Simplified Approach to Describing, Discovering and Composing Situational Enterprise Services," in *Proc. Int. Conf. on Service-Oriented Computing*. Springer, 2008, pp. 422–437.

[12] M. Cunha, N. C. Mendonça, and A. Sampaio, "A declarative environment for automatic performance evaluation in iaas clouds." *IEEE CLOUD*, vol. 2013, pp. 285–292, 2013.

[13] J. Scheuner, P. Leitner, J. Cito, and H. Gall, "Cloud WorkBench - Infrastructure-as-Code Based Cloud Benchmarking," in *Proc. Int. Conf. on Cloud Computing Tech. and Science*. IEEE, 2014, pp. 246–253.

[14] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun, "Benchmarking in the Cloud: What it Should, Can, and Cannot Be," in *Proc. TPC Technology Conference*. Springer, 2012, pp. 173–188.

[15] E. Li, L. O'Brien, R. Cai, and H. J. Zhang, "Towards a Taxonomy of Performance Evaluation of Commercial Cloud Services," in *IEEE 5th International Conference on Cloud Computing*. Honolulu, USA: IEEE Computer Society, jun 2012, pp. 344–351.

[16] Chef Software Inc., "Chef," https://www.chef.io/chef/, 2008 – 2016.

[17] Apache Software Foundation, "Apache JMeter," http://jmeter.apache.org, 1999 – 2016.

[18] W. D. Smith, "TPC-W: Benchmarking an E-Commerce Solution," http://www.tpc.org/tpcw/tpcw_ex.asp, 2000.

[19] ObjectWeb Consortium, "ObjectWeb Implementation of the TPC-W Benchmark," http://jmob.objectWeb.org/tpcw.html, 2005.

[20] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite, "Specification and Implementation of Dynamic Web Site Benchmarks," in *Proc. Int. Workshop on Workload Characterization*. IEEE, 2002, pp. 3–13.

[21] OW2 Consortium, "Rubis: Rice university bidding system," http://rubis.ow2.org/, 2009.

[22] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the Weather Tomorrow?: Towards a Benchmark for the Cloud," in *Proc. Int. Workshop on Testing Database Systems*. ACM, 2009, p. 9.

[23] A. Iosup, M. Capotă, T. Hegeman, Y. Guo, W. L. Ngai, A. L. Varbanescu, and M. Verstraaten, "Towards Benchmarking IaaS and PaaS Clouds for Graph Analytics," in *Proc. Workshop on Big Data Benchmarks*. Springer, 2014, pp. 109–131.

[24] TPC, "TPC-E a new On-Line Transaction Processing (OLTP) Workload Developed by the TPC," http://www.tpc.org/tpce/, 2015.

[25] Standard Performance Evaluation Corporation, "SPEC specweb2009," https://www.spec.org/web2009/, 2009.

[26] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy, "BenchLab: an Open Testbed for Realistic Benchmarking of Web Applications," in *Proc. Int. Conf. on Web application development*. USENIX, 2011.

[27] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0," in *Proc. of CCA*, vol. 8, 2008.

[28] E. Cecchet, V. Udayabhanu, T. Wood, P. Shenoy, F. Mottet, V. Quema, and G. Pierre, "Benchlab: Benchmarking with real web applications and web browsers." Eurosys, 2011.

[29] M. B. Chhetri, S. Chichin, Q. B. Vo, and R. Kowalczyk, "Smart CloudBench - Automated Performance Benchmarking of the Cloud," in *Proc. Int. Conf. Cloud Computing*. IEEE, 2013, pp. 414–421.

[30] Q. B. Vo and R. Kowalczyk, "Smart Cloudbench - Test Drive the Cloud Before you Buy," *Service Research and Innovation*, p. 59, 2014.

[31] M. A. El-Refaey and M. A. Rizkaa, "CloudGauge: A Dynamic Cloud and Virtualization Benchmarking Suite," in *Int. Workshop on Enabling Tech.: Infrastructures for Collaborative Enterprises*. IEEE, 2010, pp. 66–75.

[32] Red Hat Inc., "Ansible," https://www.ansible.com, 2016.

[33] Puppet, "Puppet devops solutions," https://puppet.com, 2016.

[34] "Faban," http://faban.org, 2016.

[35] The Dojo Foundation, "Dojo toolkit," https://dojotoolkit.org, 2016.

[36] Apache Software Foundation, "Apache cordova," https://cordova.apache.org, 2012 – 2016.

[37] IBM DeveloperWorks, "IBM WebSphere Liberty," https://developer.ibm.com/wasdev/websphere-liberty/, 2016.

[38] J. Scheuner, J. Cito, P. Leitner, and H. Gall, "Cloud workbench: Benchmarking iaas providers based on infrastructure-as-code," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15 Companion. New York, NY, USA: ACM, 2015, pp. 239–242. [Online]. Available: http://doi.acm.org/10.1145/2740908.2742833

[39] HashiCorp, "Vagrant," https://www.vagrantup.com, 2016.

[40] M. Mao and M. Humphrey, "Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows," in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 49.

[41] N. Serrano, G. Gallardo, and J. Hernantes, "Infrastructure as a Service and Cloud Technologies," *IEEE Software*, vol. 32, no. 2, pp. 30–36, Mar 2015.

[42] P. Leitner and J. Scheuner, "Bursting with Possibilities–An Empirical Study of Credit-Based Bursting Cloud Instance Types," in *Proc. Int. Conf. on Utility and Cloud Computing*. IEEE, 2015, pp. 227–236.

[43] Z. Li, L. O'Brien, H. Zhang, and R. Cai, "On a Catalogue of Metrics for Evaluating Commercial Cloud Services," in *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*. IEEE Computer Society, 2012, pp. 164–173.

[44] P. J. Fleming and J. J. Wallace, "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results," *Commun. ACM*, vol. 29, no. 3, pp. 218–221, Mar. 1986. [Online]. Available: http://doi.acm.org/10.1145/5666.5673

[45] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. D. Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. F. Pérez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, and D. Vladušič, "DICE: Quality-driven Development of Data-intensive Cloud Applications," in *Proceedings of the Seventh International Workshop on Modeling in Software Engineering*, ser. MiSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 78–83. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820489.2820507