This is the pre-peer reviewed version of the following article: “Black-box Load Testing to Support Auto Scaling Web Applications in the Cloud”
by M. Catillo, L. Ocone, M. Rak, U. Villano, which has been published in final form in *International Journal of Grid and Utility Computing*,
and is available at http://www.inderscience.com/offer.php?id=114823,
DOI: 10.1504/IJGUC.2021.114823

Black-box Load Testing to Support Auto Scaling Web Applications in the Cloud

Abstract: One of the most interesting features of cloud environments is the possibility to deploy scalable applications, which can automatically modulate the amount of leased resources so as to adapt to load variations and to guarantee the desired level of quality of service. As auto scaling has severe implications on execution costs, making optimal choices is of paramount importance. This paper presents a method based on off-line black-box load testing that allows to obtain performance indexes of a web application in multiple configurations under realistic load. These indexes, along with available resource cost information, can be exploited by auto-scaler tools to implement the desired scaling policy, making a trade-off between cost and user-perceived performance.

1 Introduction

The widespread and systematic use of cloud computing resources is currently a solid reality. In cloud environments there is the possibility to rapidly and automatically expand or reduce the amount of the computing resources obtained from a provider in order to adapt to load variations, keeping leasing costs to a minimum. According to the NIST definition [20], this is an essential characteristic of cloud environments, called *rapid elasticity*: "Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time." In practice, elasticity makes it possible to exploit at any time, without human intervention, the right amount of cloud resources, so as to adapt to actual load without performance degradation, avoiding at the same time to incur unnecessary extra costs.

Elasticity is undoubtedly one of the reasons that have led web application developers to migrate to the cloud. However, the implementation of an *auto-scaler* able to exploit optimally (or, at least, reasonably well) cloud elasticity has proven to be far from simple. This is due to a number of reasons, including:

- the difficulty to scale the performance of web applications. More resources should lead to higher performance, but finding the best way to scale performance is not a trivial task, often complicated by the presence of non-replicable application components;
- the characterization of the workload. Its modeling is difficult or even impossible, making it hard to foresee future workload changes and to apply proactive resource scaling;
- the garden variety of cloud resources offerings and pricing models, which add further complexity to the choice of optimal resource acquisition.

Informally speaking, a successful auto-scaler must know *when to scale*, so as to adapt promptly or even proactively to workload changes, and *how to scale*, provisioning or deprovisioning computer resources leased from the cloud, obtaining the desired performance level without dangerous oscillations. Currently several auto-scaling tools are available, albeit with limited capabilities. Sometimes they are readily offered from cloud service providers (e.g., AWS [8], Azure [1], Google [2]). Moreover, even if a wide literature has been produced on the topic, a final solution is not available at the state of the art. Furthermore, the constant growth of cloud offerings and pricing models and the concrete possibility to resort to multiple clouds rise the complexity of the problem. A short review of the auto-scaling literature will be presented in the related work section.

In this paper we propose a methodology based on off-line black-box load testing that allows to obtain performance indexes of a web application in multiple configurations under realistic load. These indexes, along with available resource cost information, can be exploited by auto-scaler tools to implement the desired scaling policy with negligible monitoring overhead, making a trade-off between cost and user-perceived performance. Our approach consists in benchmarking in an almost automated way the web application in different deployment configurations to discover its processing capacities, making it easier to choose scaling policies in the presence of a too high/low number of requests. As a reference for our work we will use a pretty simple, but real and highly diffused, web application (a WordPress stack). We will present the obtained performance figures, showing how and when its performance limits are reached for a number of alternative configurations, obtaining information that, in conjunction with resource leasing costs, can support scaling decisions.

This paper will go on as follows. Section 2 presents the basics of the auto scaling problem. Section 3 deals with related work. The paper will go on by illustrating our methodology (Section 4) and by a case study

Copyright © 201X Inderscience Enterprises Ltd.

Copyright © 201X Inderscience Enterprises Ltd.

(Section 5) discussing in detail its application. Finally, the conclusions are drawn and our future work outlined.

2 The Auto Scaling Problem

The problem of the auto scaling of web applications involves the automatic provisioning or deprovisioning of computing resources leased from a single or multiple cloud providers, so that the application SLA is satisfied under dynamically variable load and the total resource cost is minimized. It is commonly approached as an autonomous control problem, involving the use of a MAPE (Monitoring, Analysis, Planning, Execution) loop [19]. The Monitoring phase requires the cyclic measurement of purposely-selected performance indicators. These measurements may suggest the necessity of a scaling action. Actual scaling decisions are taken in the Analysis phase, taking care to avoid oscillations due to opposite actions in a short time interval. In the Planning phase, the actual amount of resources to be provisioned/deprovisioned is estimated. This involves the choice between *horizontal* and *vertical scaling*. In a cloud context, horizontal scaling involves the acquisition of additional VMs (scaling out) or the release of useless VMs (scaling in). Vertical scaling, instead, involves adding (scaling up) or subtracting (scaling down) resources (e.g., compute cores, RAM) to/from existing VMs. In fact, horizontal and vertical scaling are not mutually exclusive. Sometimes it is worth using them jointly, for example when an application contains both replicable services and stateful ones (e.g., databases) that cannot be easily replicated. Once the scaling decisions are taken, they are implemented in Execution phase. Even this phase involves complex decisions, especially when resorting to multiple cloud providers, possibly with multiple data centers in different geographical regions.

All things considered, the decision space for an auto-scaler is decidedly huge, with an expansion trend due to the multiplicity of cloud providers and to the growing variety of cost plans to be considered.

3 Related Work

As mentioned in the introduction, the auto-scaling problem has been studied extensively, from multiple perspectives. The solutions proposed in the literature at the state of the art are based on one (or more) of the following scaling methods: *load predictive*, *resource-aware*, *SLA-aware*, *cost-aware*, depending on whether the focus is on the prediction of load peaks, on resource provisioning policies, on meeting the SLAs signed by application providers, or on cost reduction. Since the work presented in this paper is essentially *cost-aware*, for brevity's sake the focus of this section will be on works based on a cost-aware approach. The interested reader is referred to [18], [23], [12], [11] for general and wide surveys of auto-scaling literature.

Currently only a few contributions in the literature follow a cost-aware approach. In [7] the authors underline the importance of cost analysis for complex applications. In particular, they state that a cost-oriented study is crucial from the design phase to the deployment and runtime phases. Hawng *et al.* [16] highlight the performance/cost ratio as a key factor in cloud productivity. The paper [27] describes a cost-aware solution based on workload predictions to provide an elastic service cloud. Qu *et al.* [22] instead propose a cost-efficient auto-scaling system for web applications based on the lease of Amazon EC2 spot instances. The MELODIC framework [15] can support cost-aware auto-scaling. It finds a good initial deployment in the cloud and continuously optimizes it according to the variable execution context, possibly taking into account the cost in its utility function. It is worth pointing out that, unlike the contributions mentioned above, our work does not entail the development of an auto-scaling system. Our aim in this paper is instead to propose a benchmarking and analysis method able to provide data to support the scaling policies used by commonly-used scalability frameworks and tools for clouds.

Other solutions in the literature analyze cost-awareness from different perspectives. An interesting approach that defines an innovative, cost-aware mechanism, is described in [17]. In particular, the Authors introduce a mediator between application and auto-scaler. They use a 'forward-looking' approach by delaying or omitting the release of resources in order to avoid additional charging costs if the resource will be required in the future. This solution can interface with any auto-scaler. A method that instead analyzes the cost-aware scalability of applications in public clouds is presented in [21], where the Authors introduce a model for capturing the pricing schemes of cloud services. They analyze the application cost depending on its used cloud services and their billing cycles, specifying a function for evaluating the cost efficiency of cloud applications. Our proposal, though aimed at the reduction of cost, similarly to the two works mentioned above, exploits different performance indexes and cost metrics.

The methodology proposed in this paper will be demonstrated by application to a widespread *three-tier* web application, a WordPress stack. The performance modeling of three-tier applications has been thoroughly dealt with in [14] and previously in [26] and [28]. All the above contributions analyze the application performance by detailed models taking into account the hardware resources exploited and the software components of the application. Our proposal, instead, relies on a *black-box* approach. All performance indexes are obtained solely by interacting with the application at the client interface, i.e., submitting *http* requests from purposely-crafted clients and recording responses and their timings.

4 Methodology

As outlined in Section 2, the problem we address here is to support cost-based scalability policies by information obtained by off-line profiling. The obvious objective is to grant that the costs of the resources leased from the cloud are adequate to the performance results obtained. Our methodology is based on use of custom benchmarks for the target web application. These benchmarks can be executed off-line once and for all, and will provide information used to identify the conditions when the application should scale.

In the following subsections we will show how the whole web application is modeled by a black-box approach, load testing it and collecting information on the messages sent/received at the server-client interface.

4.1 Application modeling

Our model relies on the basic assumption that in steady-state conditions a web application is a state system. In each state, it is characterized by performance indexes which are substantially stable and predictable. Hence our objective is to measure the performance indexes by load testing, stressing the system and settling it in a given operating state. The procedure has to be repeated for all the states that correspond to meaningful working conditions and for all system configurations, as obtained by scaling up/down, out/in the basic deployment.

Our working hypothesis is that the application state is determined to a large extent by the number of active concurrent users of the application. This will be proven to be true for the application chosen for our case study; however, we have found that it is also true for a large class of *http*-based systems [24] where all users generate *similar* load on the system.

Measuring the number of concurrent users in real operating conditions is possible, but unpractical for an external scaling agent. As a matter of fact, there is a direct relation between the number of concurrent users and the request rate, which can be easily measured externally even in the absence of any information on the application internal structure.

Putting all together, the actual (measured) request rate in operating conditions makes it possible to deduce the number of concurrent users that characterizes the state of the system. The pre-determined performance parameters for that load state and for each possible application configuration, along with the leasing cost of these configurations allow to make scaling decisions, by a reasoned choice of the system configuration to be used next.

As far as the application quality of service delivered (QoS) is concerned, the analysis of the performance results characterizing the behavior in this state must allow to find out if scaling decisions are possible or immediately required. By "immediately required" we mean that the system QoS is unacceptably low, and a scaling up or out choice must be taken soon.

In practice, the event that has the highest impact on the level of *user patience* (the perception end users have from a service QoS) is the failure of the *http* requests used for the interaction client-server. The failure of an *http* request is commonly known as KO; it corresponds to any status code different from 2xx or 304. On the other hand, for completed *http* requests (i.e., not KO), the fundamental index is the response time (RT), which clearly should be as low as possible.

The MAR index (Maximum Allowed Rate), defined in [10], is an index linked to the maximum number of failed *http* requests. In particular, it is the request rate for which the application reaches $\%KO_{max}$, a given maximum allowed $\%KO$ value. The MAR is a monotonically non-decreasing function of $\%KO_{max}$: if a higher failure ratio is allowed, the application can be feed with higher request rates.

4.2 Load testing and analysis

Given the assumptions at the previous Subsection, the issue discussed here is how to perform load tests, forcing the application to settle in a given operating state.

The method we propose consists of three main phases, sketched in figure 1: (i) Application Behavior Profiling, (ii) Profile Benchmarking, (iii) Cost Analysis.

Application Behavior Profiling consists in building a simplified model of the application workload, used to feed off-line the application through stressing tools. In this first phase we run and analyze the application to build a set of *application user profiles*, scripts that can issue suitably-timed sequences of application invocations that model the behavior of a typical user in a single session. Such profiles can be easily built through web automation tools as Selenium [3], or the Gatling recorder [4] (the one actually used in our experiments). For example, for the small commerce site based on the WordPress CMS software that will be used in the following as a running example, it is possible to identify four "typical" navigation profiles: an Editor profile (browsing and adding comments), an Author profile (browsing and adding posts and comments), a ShopManager profile (browsing and adding products, posts and comments), a Reader profile (just browsing). Once the scripts for each profile have been built, a workload for benchmarking purposes can be obtained by multiple concurrent executions of one of the load profiles, and load variability can be obtained by modulating the number of load profiles launched per second.

In the **Profile benchmarking** phase, an off-line application profiling is obtained by running the application in the cloud under the synthetic workload generated by the scripts produced in the previous phase.

As our aim is to reach a steady-state condition for each state, i.e., for progressively higher numbers of concurrent users, we exploit the load tests with closed workload (capped number of users) provided by Gatling [13]. In each working condition, users are injected to keep the number of concurrent users stable. Throughout the

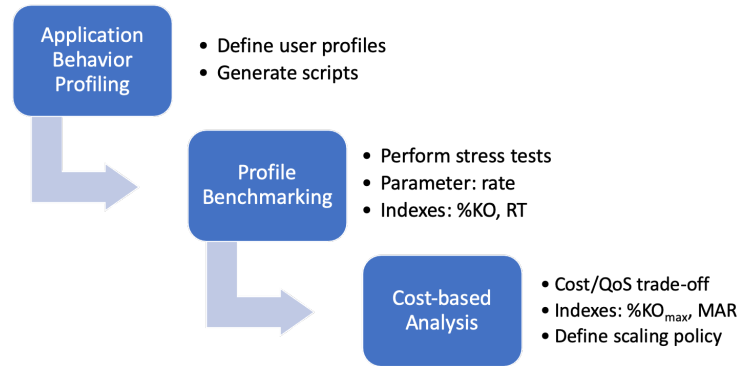


Figure 1: Scalability Benchmarking Methodology

test we record the measurements of $\%KO$, percentage ratio of failed requests, and of Response Time (RT).

In the last phase, **Cost-based Analysis**, we produce a set of graphs that outline the trade-off among cost-related indexes and quality of service delivered, enabling the application administrator to define a scalability policy with an optimal balance between costs and QoS.

5 A Case Study

In order to demonstrate the effectiveness of the proposed approach, we applied the suggested methodology to a simple, but very typical web application: a WordPress-based Content Management System (CMS). We performed the full flow described in the previous section on this application, in order to clarify all details of the cost analysis process.

WordPress [5] is open-source software running on top of PHP using any web server, as Apache or NGNIX. It exploits a MySQL data base to store the web application data. WordPress is one of the most widely used CMS, commonly adopted for blogs or simple e-commerce sites. For this reason, it has been used many times in the literature as a benchmark (see for example [9], [25]).

5.1 Application Behavior Profiling

We configured WordPress in order to host a simple e-commerce site, where it is possible to navigate through the products and make comments and publish articles or new products. Accordingly, we defined four simple *application user profiles*:

- **Editor**: a user that registers new operations, navigates through the website and posts comments;
- **Author**: a user that behaves as an Editor, but in addition posts new articles (not only comments);
- **ShopManager**: a user that add products, articles and comments on the website;
- **Reader**: a user that navigates through the website without posting or commenting.

Thanks to the already mentioned Gatling recorder, we generated a set of SCALA scripts that automate the navigation and interaction with the website. The automatically-generated scripts were successively modified, in order to add parameters in any phase that requires a user data input. Raw scripts need sample data when adding a new comment, a new post, a new product and/or the registration of a new user. We stored a set of possible data inputs in a collection of JSON files, to be selected to feed the automation tool that performs the tests. The following code snippet shows a fragment of one of the generated SCALA scripts used to setup an execution.

```
val reader = scenario("User that reads only")
           .exec(Browse.browse)
val editor = scenario("Editor registers and adds comments")
           .exec(Registration.addUser,
                 NewComment.addComment,
                 Browse.browse)
val author = scenario("Author registers, adds new posts
                       and adds new comments")
           .exec( Registration.addUser,
                 NewPost.addPost,
                 Browse.browse,
                 NewComment.addComment)
val shopManager = scenario("ShopManager registers,
                            adds new products,
                            adds new posts and
                            adds new comments")
           .exec(Registration.addUser,
                 Browse.browse,
                 NewProduct.addProduct,
                 NewPost.addPost,
                 NewComment.addComment)
```

Depending on the type cof CMS operation to be performed and the type of data needed, sample input data can be injected into the automation tool in three ways: (i) direct, (ii) indirect and (iii) two-steps.

Direct data injection is performed directly in the SCALA script: the Gatling automation tool has a feeder that, correctly configured, randomly selects a JSON file to upload. In indirect injection, the feeder accesses a JSON file containing a Javascript code that produces the

text to be injected as the body of an HTTP request. The last injection type requires the use of both the techniques illustrated above.

5.2 Profile benchmarking

The application benchmarks were executed on a private cloud , based on the OpenStack environment and hosted in a datacenter at the University of Sannio. Table 1 shows the VM flavor used for our tests. These are roughly equivalent to the VM leased by AWS in Table 2, which also reports their cost in dollars per hour (we used the AWS prices available at `https://aws.amazon.com/it/ec2/pricing/on-demand/`)

TYPE	VCPU	RAM (GB)	STORAGE (GB)
m1.medium	2	4	40
m1.large	4	8	60
m1.xlarge	8	16	80

Table 1 VMs used in our experiments

TYPE	AWS equivalent	AWS cost
m1.medium	a1.large	0.0576 $/h
m1.large	a1.xlarge	0.1152$/h
m1.xlarge	a1.2xlarge	0.2304 $/h

Table 2 Equivalent AWS VMs and costs ($ per hour)

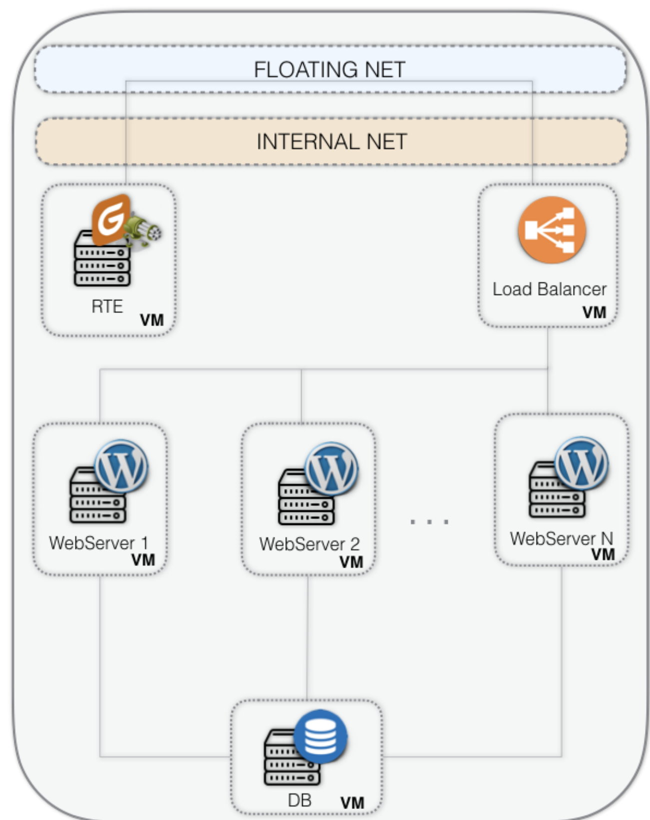


Figure 2: WordPress deployment configuration

The simplest deployment configuration (Fig. 2) consists of four VMs: the first VM hosts our *Remote Terminal Emulator* (RTE), which emulates users that perform continuous requests to the server. The other three VMs are devoted to host the WordPress application, the MySQL DB and the load balancer (based on HAProxy). This choice corresponds to a typical WordPress configuration [6] and makes it possible to verify easily the scalability issues. The WordPress module can be scaled horizontally adding (or removing) VMs that host additional WP front-ends. The DB cannot be easily replicated, and so the simplest scaling solution is to perform only vertical scaling (i.e., to execute it on a more/less powerful VM flavor). Some of our tests, not shown here for brevity's sake, show that the use of *large* or *xlarge* VMs for the WP front-end does not entail performance improvements. At the same time, the use of *xlarge* VMs for the DB is not beneficial for performance. Therefore, in the discussion that follows, we will present and discuss the results obtained with six different configurations, made up of one to three *medium* flavor WP front-ends, along with one *medium* or *large* DB. The strings used to describe these configurations are self explanatory (e.g., `2WPmedium_1DBmedium` is the configuration made up of two WP front-ends on *medium* VMs and one DB on *medium* VM). In all our tests, the load balancer is hosted on a dedicated *medium* flavor VM.

As previously described, we stressed the server starting each application user profile and collecting the main performance indexes, as shown graphically in Fig. 3. The number of concurrent users (C_{users}) is progressively made higher and higher, paying attention to maintain any given C_{users} value for enough time to stabilize the server behavior (applying the approach described in [10] and [24]).

Figure 4 shows the results obtained in our tests. Each row shows, for one of the four profiles and for the six deployment configurations, the plots as function of C_{users} of:

- $\%KO$, the percentage of failed replies;
- *rate required*, the mean request rate generated by that number of concurrent users;
- *RT-measured*, the mean response time in ms;
- *COV of RT*, the coefficient of variation of response times.

A first analysis of these diagrams shows that the results obtained for the four user profiles are very similar. In other words, the actual operations performed, whether they involve reading or writing in the database, have negligible influence on the performance results. It is also possible to observe that the lowest response times are obtained by the configurations with three WP front-ends. The flavor of the VM used for the DB deployment, instead, is almost uninfluential. It is worth noting that the WordPress application shows almost constant $\%KO$ until the rate is fairly low. For progressively higher rates,

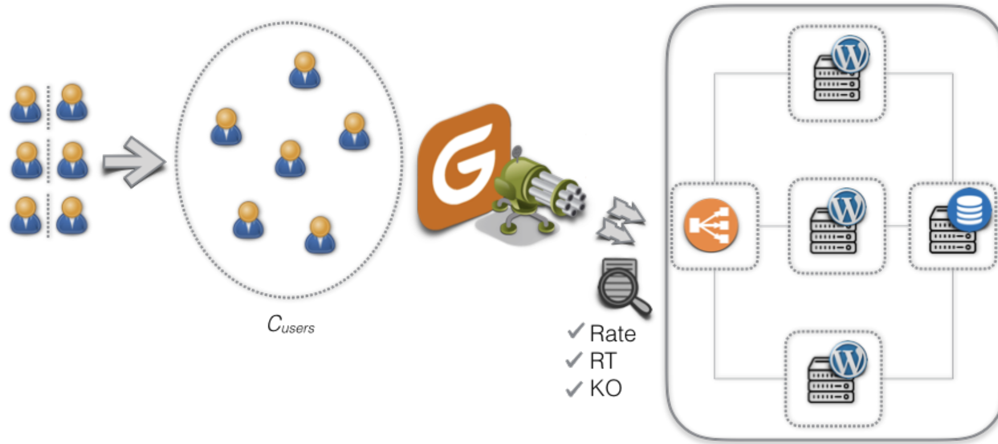


Figure 3: Benchmarking schema

from a saturation point onward, $\%KO$ grows almost linearly.

Even if the actual *http* requests generated by each user profile are largely heterogeneous, and so physiologically involve highly variable response times, the diagrams of the COV of the response time show that the actual response times are scarcely dispersed around the mean. For high numbers of C_{users}, the majority of configurations show very low COV (around 1.5). This is the proof that the state and performance behavior of the system is in large part determined by the number of concurrent users of the application, as assumed in Section 4. Stated another way, under heavy load the application behaves as a LDS system [24].

5.3 Cost-based Analysis

As mentioned above, we aim at introducing a cost-based analysis method based on high-level scaling indicators and on information obtained beforehand by load testing, that should enable to scale optimally the application. This solution is based on the use of the MAR (Maximum Allowed Rate) performance indicator. As discussed in Section 4, a fundamental requisite to avoid QoS loss is to keep the number of KO under a very low limit value, as KOs have great impact on user patience. Hence all the application scaling activity should guarantee that the limit value is not reached. The MAR is the rate for which the application reaches $\%KO_{max}$, a given maximum allowed $\%KO$ value.

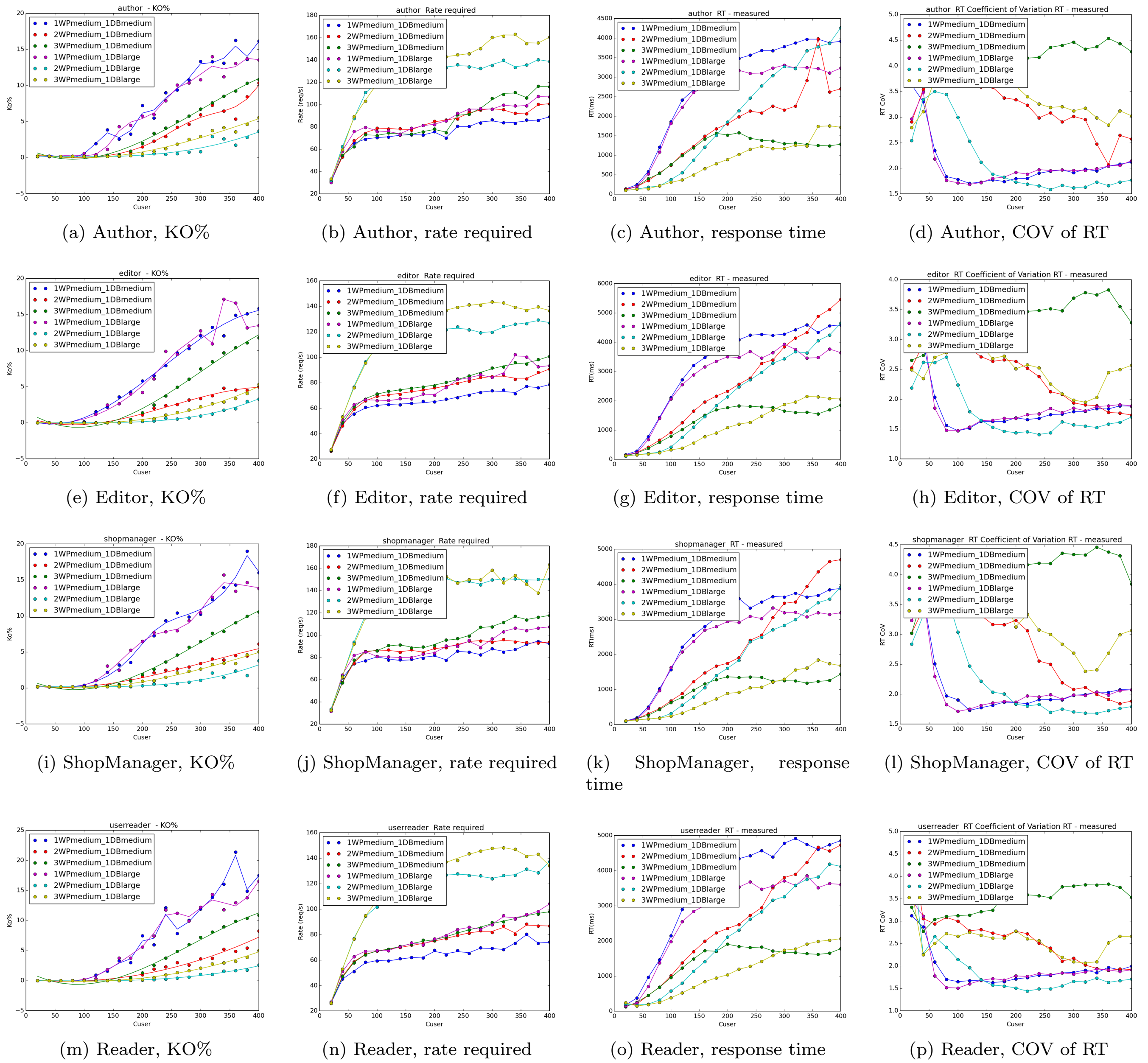
The performance results plotted in Fig. 4 provide KOs and rate required as functions of C_{users}. From these figures, choosing a $\%KO_{max}$ value of 5%, it is possible to obtain the histograms in the left column of Fig. 5 ((a), (c), (e), (g)). On each column of the histogram is also shown the configuration cost on AWS, as resulting from the values in Table 2. As was to be expected, more expensive the configuration, higher the rate of requests that can be sustained without trespassing the desired $\%KO_{max}$ value.

It is worth pointing out explicitly that these diagrams show the maximum rate and not the maximum number of concurrent users, because the latter is not measurable easily by an external scaling/monitoring agent. On the other hand, the rate of requests can be obtained with negligible monitoring overhead by measuring it at the load balancer node, without requiring the use of probes inside the application, which can thus still considered as a black box.

The plots in the right column of Fig. 5 show instead the cost for one thousand requests for each rate sustainable by each deployment configuration. In these plots it is interesting to note that the configurations `2WPmedium_1DBmedium` and `1WPmedium_1DBlarge` have similar cost behavior. However, they are not completely equivalent, as they are characterized by different MAR and response times. The same observation is true for the configurations `3WPmedium_1DBmedium` and `2WPmedium_1DBlarge`.

The diagrams in the left column of Fig. 5 make it possible to find for each profile the optimal configuration at any given request rate. *If the only optimization objective is cost reduction*, the optimal configuration at a given rate is the one with minimum cost “covered” by one of the columns in the diagram at that rate. For example, for the Author profile, it is possible to observe that:

- **Rate 0 - 73.06**: the optimal configuration is `1WPmedium_1DB medium`;
- **Rate 73.06 - 81.34**: the optimal configurations are `1WPmedium_1DB large` and `2WPmedium_1DB medium`. These are roughly equivalent as cost is concerned, but, as noted above, are characterized by different performance indexes;
- **Rate 81.34 - 95.08**: the optimal configuration is `2WPmedium_1DB medium`;
- **Rate 95.08 - 140.24**: the optimal configuration is `2WPmedium_1DB large`.



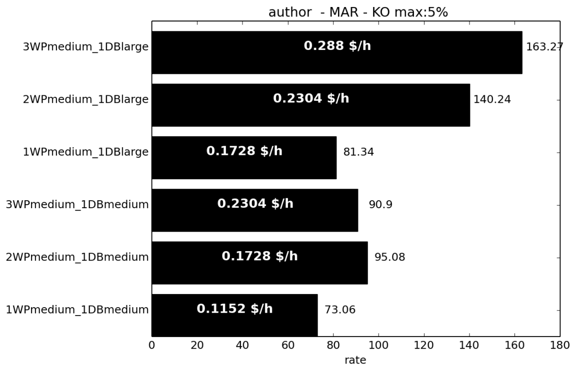
(a) Author, KO% (b) Author, rate required (c) Author, response time (d) Author, COV of RT

(e) Editor, KO% (f) Editor, rate required (g) Editor, response time (h) Editor, COV of RT

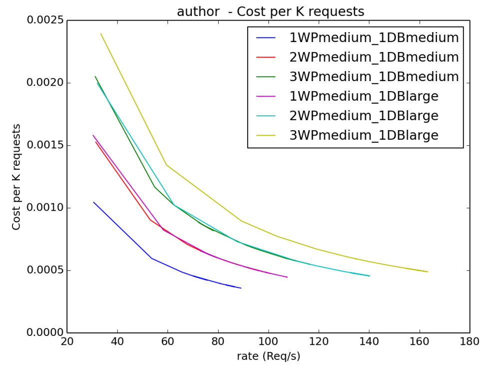
(i) ShopManager, KO% (j) ShopManager, rate required (k) ShopManager, response time (l) ShopManager, COV of RT

(m) Reader, KO% (n) Reader, rate required (o) Reader, response time (p) Reader, COV of RT

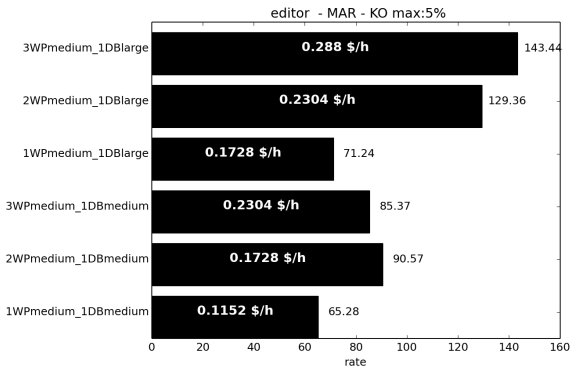
Figure 4: Benchmarking results for the four user profiles



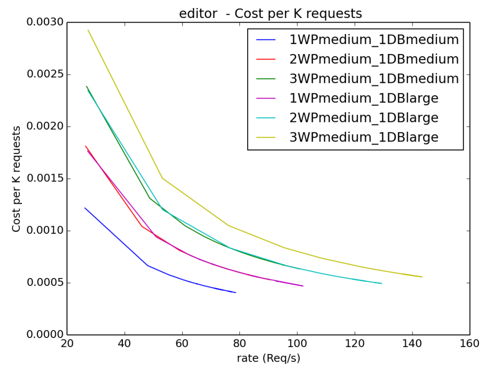
(a) Author, max rate and cost



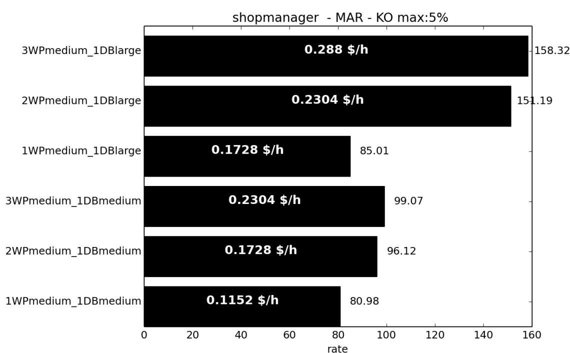
(b) Author, cost per k requests



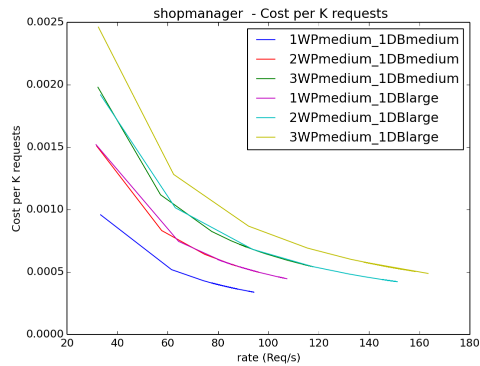
(c) Editor, max rate and cost



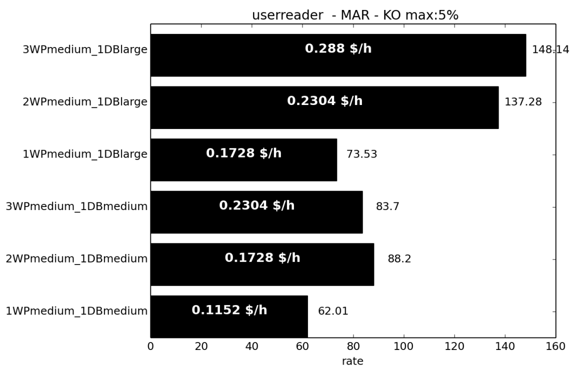
(d) Editor, cost per k requests



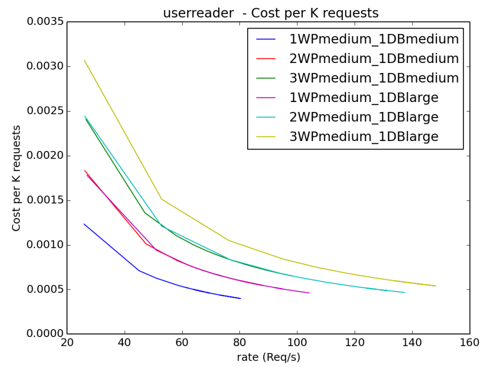
(e) ShopManager, max rate and cost



(f) ShopManager, cost per k requests



(g) Reader, max rate and cost



(h) Reader, cost per k requests

Figure 5: Maximum sustained rate and cost per k requests for $\%KO_{max}$=5%

Even in cases when the cost is not the sole optimization factor, the figures found by load testing the application before its actual deployment in the cloud can be employed by auto-scalers to support effectively any provisioning policy.

6 Conclusions

In this paper we have presented a methodology that makes it possible to support scalability policies for web applications running in the cloud with leased resources, taking into account both performance and cost. Our proposal relies on the use of off-line profiling, which allows to obtain system performance information through automated benchmarking based on simplified models of the possible application workloads. The information measured off-line makes it possible to find beforehand acceptable user request rates, taking into account the leasing cost of any chosen configuration. Dynamic run-time measurements of user request rates make it possible to scale the application making a trade-off between cost and user-perceived performance.

Our method relies on the use of an innovative performance index, the MAR (Maximum Allowed Rate). Such index is measured by black-box load testing the application with synthetic workloads that take into account common usage patterns. As shown for a case study application, the analysis enables to identify easily performance indexes that can be exploited by auto-scalers to found trade-offs between costs and quality of service of the application, when deployed in cloud in different configurations.

As a future development, we plan to extend our methodology in order to consider more complex type of workloads, composing different application user profiles. Another important research activity regards the possible use of prediction techniques to measure our benchmark indexes without performing expensive measurements, as the ones that led to the results presented here. At the state of the art, we simply neglect the scaling time from one configuration to another. We plan to extend our analyses in order to take into account these parameters. Last but not least, we plan to apply our methodology to commonly-used scalability frameworks and tools for clouds, in order to supply scalability policies to be enforced taking into account both user satisfaction and application costs.

References

[1] `https://azure.microsoft.com/en-in/features/autoscale/`.

[2] `https://cloud.google.com/compute/docs/load-balancing-and-autoscaling`.

[3] `https://selenium.dev`.

[4] `https://gatling.io/docs/current/quickstart\#using-the-recorder`.

[5] `https://wordpress.org`.

[6] `https://wordpress.org/support/article/installing-multiple-blogs`.

[7] V. Andrikopoulos, S. Gómez Sáez, F. Leymann, and J. Wettinger. Optimal distribution of applications in the cloud. In M. Jarke, J. Mylopoulos, C. Quix, C. Rolland, Y. Manolopoulos, H. Mouratidis, and J. Horkoff, editors, *Advanced Information Systems Engineering*, pages 75–90, Cham, 2014. Springer International Publishing.

[8] J. Barr. New AWS auto scaling - unified scaling for your cloud applications, 2018.

[9] A. H. Borhani, P. Leitner, B. Lee, X. Li, and T. Hung. Wpress: An application-driven performance benchmark for cloud-based virtual machines. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference*, pages 101–109, Sep. 2014.

[10] V. Casola, A. De Benedictis, M. Rak, and U. Villano. An automatic tool for benchmark testing of cloud applications. In *CLOSER 2017 - Proc. of the 7th Int. Conf. on Cloud Computing and Services Science*, pages 701–708, 2017.

[11] M. Catillo, M. Rak, and U. Villano. Auto-scaling in the cloud: Current status and perspectives. In Barolli L., Hellinckx P., and Natwichai J., editors, *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, volume 96 of *Lecture Notes in Networks and Systems*, pages 616–625, 2019.

[12] T. Chen, R. Bahsoon, and X. Yao. A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems. *ACM Comput. Surv.*, 51(3):1–40, June 2018.

[13] Gatling Corp. Gatling - Performance testing for web applications - web site. https://gatling.io/, 2018.

[14] N. Grozev and R. Buyya. Performance Modelling and Simulation of Three-Tier Applications in Cloud and Multi-Cloud Environments. *The Computer Journal*, 58(1):1–22, January 2015.

[15] G. Horn and P. Skrzypek. MELODIC: Utility Based Cross Cloud Deployment Optimisation. In *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 360–367, Krakow, May 2018. IEEE.

[16] Kai Hwang, Xiaoying Bai, Yue Shi, Muyang Li, Wenguang Chen, and Yongwei Wu. Cloud performance modeling and benchmark evaluation of elastic scaling strategies. *IEEE Transactions on Parallel and Distributed Systems*, 27:1–1, 01 2015.

[17] V. Lesch, A. Bauer, N. Herbst, and S. Kounev. FOX: Cost-awareness for autonomic resource management in public clouds. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 4–15, New York, NY, USA, 2018. ACM.

[18] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, 12(4):559–592, December 2014.

[19] M. Maurer, I. Breskovic, V. C. Emeakaroha, and I. Brandic. Revealing the MAPE loop for the autonomic management of Cloud infrastructures. In *2011 IEEE Symposium on Computers and Communications (ISCC)*, pages 147–152, Corfu, Greece, June 2011. IEEE.

[20] P. Mell and T. Grance. The NIST definition of cloud computing. *NIST Special Publication*, 800:145, 2011.

[21] D. Moldovan, H. Truong, and S. Dustdar. Cost-aware scalability of applications in public clouds. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 79–88, April 2016.

[22] C. Qu, R. N. Calheiros, and R. Buyya. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *CoRR*, abs/1509.05197, 2015.

[23] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. *ACM Computing Surveys*, 51(4):1–33, July 2018.

[24] M. Rak, R. Aversa, B. Di Martino, and A. Sgueglia. Web services resilience evaluation using LDS load dependent server models. *Journal of Communications*, 5(1):39–49, 2010.

[25] A. Sampaio, T. Rolim, N. C. Mendonça, and M. Cunha. An approach for evaluating cloud application topologies based on TOSCA. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 407–414, June 2016.

[26] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An Analytical Model for Multi-tier Internet Services and Its Applications. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 291–302, June 2005.

[27] J. Yang, C. Liu, Y. Shang, B. Cheng, Z. Mao, L. Chunhong, L. Niu, and C. Junliang. A cost-aware auto-scaling approach using the workload prediction in service clouds. *Information Systems Frontiers*, 16:7–18, 03 2014.

[28] Q. Zhang, L. Cherkasova, and E. Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Fourth International Conference on Autonomic Computing (ICAC'07)*, pages 27–27, Jacksonville, FL, USA, June 2007. IEEE.