Towards Faster Response Time Models for Vertical Elasticity

Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-Rodriguez and Erik Elmroth

Department of Computing Science
Umeå University, Sweden
{*ewnetu, cristian.klein, francisco, elmroth*}*@cs.umu.se*

***Abstract*—Cloud computing is mostly allocating resources at course grain, e.g., entire CPU cores are allocated for as long as an hour. For improving resource efficiency of clouds, the Resource-as-a-Service cloud is envisioned, which allocated resources at fraction of a core and second granularity. Despite technology enabling such infrastructure, e.g., through lightweight virtualization such as LXC or vertical elasticity in the Xen hypervisor, performance models to decide how much capacity to allocate to each application are lagging behind.**

In this paper, we evaluate two performance models for mean response time, a previously proposed one and a novel one. The models are tested with 3 applications in both open and closed system models. Results show that the inverse model reacts fast and remains stable for targets as low as 0.5 seconds.

I. Introduction

One of the defining features of Infrastructure as a Service (IaaS) cloud computing, leading to its widespread adoption, is elasticity. It consists in the ability to provision and release computing resources, usually packed as Virtual Machines (VMs), on-demand, rapidly and automated. Elasticity can be of two types. Horizontal elasticity consists in adding or removing VMs to or from an application, e.g., based on the number of end-users. It requires more support from the application, e.g., to clone and synchronise state among VMs, but requires no extra support from the hypervisor, hence its widespread adoption in public clouds.

Vertical elasticity consists in adding or removing resources, such as CPU cores and memory, from a VM. It requires little support from the application, which in essence only needs to be multi-threaded, while the bulk support of elasticity is handled by the hypervisor and guest operating system's kernel. In general, horizontal elasticity is course-grained, i.e., a whole core is exclusively allocated to a VM for a relative long duration such as an hour, while vertical elasticity is fine-grained, i.e., only a fraction of a core can be allocated to a VM for as short as a few seconds. Indeed, vertical elasticity might be a key enabler for the resource-as-a-service cloud [6], an infrastructure that leases resources at CPU-cycle and second granularity, benefiting both cloud users, as they truly pay only for the resources they use, and the cloud providers, as they can make better use of their resources and accept more users. In fact, the technological cornerstone is already laid by lightweight virtualization frameworks such as LXC [7] and commercial offerings such as dotCloud.

However, to fully enable vertical elasticity, a key challenge is to decide the amount of capacity, e.g., cores and fraction of cores, to allocate to an application. For example, it is well-known that users are sensitive to response time [15] and that response times as high as 4 seconds may determine them to abandon the application. While a lot of research has been done for horizontal elasticity, these solutions are not directly applicable to the vertical case, due to their course-grain allocation. Other works deal with the vertical case, but suffer from a number of deficiencies, amongst others, they may require a lot of time for training, e.g., up to 20 minutes [14], [16] (see Section II). We argue that, given the accelerating pace at which new versions of a cloud application are deployed [18], truly enabling the envisioned resource-as-a-service cloud would require faster performance models: They should require minimal knowledge and training about the hosted applications, while at the same time react as fast as possible to environmental changes, such as a sudden increase in the number of users.

In this paper, we present an empirical study of two performance models for mean response time (Section III): *Queue length based*, which has been previously proposed in literature, works by measuring queue length and applying Little's law. Our novel *inverted response time* works by inverting the response time of M/M/1 queue. The highlight of our contribution is to validate the performance models both for open- and closed-system models for targets as low as 0.5 seconds (Section IV). We show using several widely-used prototype cloud applications that our inverse response time model outperforms the queue length based: It maintains response times around the target when no environmental changes occur and reacts within 40 seconds (or 8 control intervals) when changes do occur. We present our initial findings here, hoping to promote early feedback from the research community in the issue of faster performance models.

II. Related Work

Guitart et al. published an extensive survey on performance management for Internet application [12]. According to their taxonomy, our work best fits in dynamic resource management on virtualized platforms, that base their decisions on a combined approach. We gather run-time observations of response time and queue length to fit a queuing model, and use a control theoretic approach to filter potential noise and modeling errors. The cited authors themselves highlighted a deficiency of works that use combined approaches, which showed to be the most promising. Hence, our works comes at a timely moment to fill this gap.

We have noted two more shortcomings of related work presented in the above survey. First, validation is done only using simulations or using a single application. Second, some works base their decisions on CPU usage, which is not a reliable measure of spare capacity in vertical elasticity, due to hypervisor preemption of virtual machines, also called steal time [10]. In contrast, our work bases its decision on response time and queue length, that can be reliably measured.

Among some newer works, we observed that they require up to 20 minutes of training [14] or off-line profiling [16]. As businesses embrace the lean movement, several application versions are deployed on a daily basis [18], which makes cited approaches cumbersome. Indeed, no training and fast reaction was one of our work's main objectives.

III. Performance Models for Response Time

In this section we first describe our constraints and assumptions, then we describe two models to predict capacity requirements of a cloud application, given past behavior and a target response time.

A. Assumptions

Performance models need to fulfill several constraints. First, due to the heterogeneity of hosted applications, the performance models need to be as generic as possible and should require no knowledge of application internals. Second, they should predict average behavior and ignore sporadic noise observed in the past. Such noise may be caused, for example, by variation in retrieving data, some being cached in memory, other needing to be fetched from disk. Third, these performance models should quickly adjust to variation in workload and capacity requirements. For example, an increase in the number of users, or a change in request distribution may need refitting the model parameters.

Ideally, given a Key Performance Indicator (KPI) value as input, a performance model should return the exact capacity that needs to be allocated to an application to reach that KPI value. However, as this is difficult to achieve, a performance model should at least *drive* capacity allocations to the correct value, i.e., by periodically refitting the model parameters when allocating estimated capacity, the application should eventually reach the desired performance.

The load of the cloud application can be of three types: open, closed or partly-open [19]. In an open system model, typically modeled as Poisson process, requests are issued with an exponentially-random inter-arrival time, characterized by a rate parameter, without waiting for requests to actually complete. In contrast, in a closed system model, a number of users access the application, each executing the following loop: issue a request, wait for the request to complete, "think" for a random time interval, repeat. The resulting average request inter-arrival time is the sum of the average think-time and the average response time of the application, hence dependent on the performance of the evaluated application. A partly-open system model is a mixture between the two: Users arrive according to a Poisson process and leave after some time, but behave closed while in the system. As with the closed model, the inter-arrival time depends on the performance of the evaluated system.

For the performance models, we start from an open system assumption. The response time in such a scenario increases faster as the system is approaching saturation [19], hence capacity allocation through vertical elasticity has to be performed more carefully. Nevertheless, in Section IV we also evaluate the proposed performance models if the load is closed, to test their suitability in case our open assumption does not hold.

B. Response Time Models

End-users of interactive applications are sensitive to response time. Indeed, several studies show that increased response time reduces revenue. In particular, end-users abandon the service if response time is above 4 seconds [15] and are likely to join the competition, thus incurring long-term revenue loss. Therefore, it is desirable to maintain target response time for an application. However, modeling response time is challenging due to its non-linear relationship with capacity. We present two different response time models: the **queue length** model, which was tested using only simulations in [8], and our novel **inverse** model.

1) Queue Length Model: Starting from Little's Law, the relation between average response time R and capacity c for an application can be represented as:

$$q = \lambda \cdot R, \tag{1}$$

where q is the average queue length, i.e., the number of requests that entered the application but have yet to exit, and λ is the arrival rate. Next, we use the formula for average response time given by the M/M/1 queuing model:

$$R = \frac{1}{\mu - \lambda}, \tag{2}$$

where μ is the average service rate of the application. To model the relationship between capacity and response time, one can model $\mu = c/\alpha$, where α is a model parameter. By replacing λ from Eq. (1) in Eq. (2) and resolving R, one obtains the formula for the average response time as:

$$R = \alpha(q+1)/c, \tag{3}$$

where α is a model parameter and q is the number of requests waiting to be serviced. The parameter α can be estimated online from past measurements of average response time, average queue length and capacity, thus compensating dynamically for small non-linearities in the real system. However, to reduce the impact of measurement noise, we decided to use a Recursive Least Square (RLS) filter [13]. In essence, such a filter takes past estimation of α and the current ratio $(Rc)/(q+1)$ to output a new value that minimizes the least-squares error. A *forgetting factor* allows to trade the influence of old values for up-to-date measurements. In our experiments, we use a forgetting factor of 0.2. The queue length model was only validated using simulation [8] and has yet to be tested in a real environment.

2) Inverse model: We model the inverse relationship between average response time R of an application and the capacity allocated to it as:

$$R = \beta/c, \tag{4}$$

where β is a model parameter. As with queue length, the parameter model β can be estimated using past measurements of capacity and average response time using Eq. (4). As before, to reduce the influence of measurement noise, we use an RLS filter with forgetting factor 0.2.

Note that the inverse model needs less information from the application as compared to the queue length model.

In our experiments, we recompute capacity to the application periodically, with a **control interval** of 5 seconds, which is short enough to make the system reactive and long enough to observe the effects of the new capacity allocation on the performance of the application [17].

IV. Evaluation

In this section, we evaluate our contribution. First, we describe the experimental setup. Next, we perform time series and cumulative analyses of the performance models under different configurations.

A. Setup

Experiments were conducted on a single Physical Machine (PM) equipped with a total of 32 cores[1] and 56 GB of memory. To emulate a typical cloud environment and easily perform vertical elasticity, we used the Xen hypervisor [5]. To our knowledge, Xen is the only hypervisor that support hot-unplugging virtual cores from VM, which was necessary for our tests. Each tested application was deployed with all its components – e.g., web server, database server – inside its own VM, as is commonly done in practice [21], e.g., using a LAMP stack [1]. Since we are primarily interested in evaluating CPU allocation strategies, we configured each VM with 6 GB of memory, enough to avoid disk activity.

To test the applicability of our contribution to a wide range of applications, we performed experiments using three applications: RUBiS [3], RUBBoS [4] and Olio [2]. These applications are widely-used cloud benchmarks (see [11], [20], [25], [23], [24], [22], [9]) and represent an eBay-like e-commerce application, a Slashdot-like bulleting board and an Amazon-like book store, respectively.

To emulate the users accessing the applications, we used our custom `httpmon` workload generator[2], which supports clients behaving both as open and closed system models. For open clients, we changed the arrival rate during experiments, as required to stress-test the system. For closed clients, the think-time of each client is constantly 1 second, whereas the number of users is varied. These parameters allow for a meaningful comparison of the behavior of the system among the two clients models. Indeed, as the application's response time decreases, the throughput of closed clients approaches the same value as for open clients.

Metrics: The **response time** of a request is defined as the time elapsed from sending the first byte of the request to receiving the last byte of the reply. We are mostly interested in the mean response time over 20 seconds intervals (4 control intervals), which is a long enough interval to filter measurement noise, but short enough to highlight the transient behavior of an application.

B. Time series analysis

To evaluate the performance models, we injected a variable load, so as to test how each model reacts during sudden workload spikes under both open and closed system models. Furthermore, we configured the system with relatively high to small target values in order to see how the models behave.

The plots in this section are structured as follows. Each figure shows the results of a single experiment. The bottom x-axis represents the time elapsed since the start of the experiment. The time is divided in 5 equal intervals, each featuring a different arrival rate (for open system model) or number of users (for closed system model), as presented on the top x-axis. The top graph of each figure plots the measured mean response time value. The bottom graph plots the required capacity as computed by the performance models and allocated to the application over the next 5 second interval.

Figs. 1a to 1d show the two models with different target response time configurations under open and closed system models for RUBiS application. In general, both performance models are stable for higher target values under both system models. Moreover, both models converge to the target values within short period (see Section IV-D) after detecting a sudden increase or decrease in workload which is manifested as quick increase or decrease in response time.

[1]Two AMD Opteron$^{\text{TM}}$ 6272 processors, 2100 MHz, 16 cores each, no hyper-threading.

[2]https://github.com/cloud-control/httpmon

The other important point to note is that the two models properly detect and adapt to the capacity requirements for both open and closed model systems. Indeed, for higher target response times, the open system model requires more capacity compared to the closed system model for the same value of arrival rate and users, respectively. For lower target response time, the capacity requirements needed for both system models becomes almost the same. These situations are properly dealt with by the two models as depicted in Fig. 1a and Fig. 1c for higher targets, and Fig. 1b and Fig. 1d for lower targets.

We also did experiments with Olio and RUBBoS. However, due to lack of space, we only present time series plots for target response time of 0.5. As can be observed from Figs. 2 and 3, the queue length model does not behave well for lower target values while the inverse model remains stable.

In general, the inverse model remains relatively stable irrespective of the target values under both system models. On the other hand, the queue length model is less stable for lower target values under both system models.

The time series results show that the models behave as intended. In the following sections, we analyse the cumulative behaviors such as the aggregate errors over the span of the experiment (Section IV-C) and the total time it takes each model to arrive stable state after a change in load occurs (Section IV-D).

C. Aggregate Analysis

To see the aggregate behavior of the models over the course of the experiment, we use two control theoretic metrics which measure the total error observed during the life span of the system. These metrics are Integral of Squared Error (ISE) and Integral of the Absolute Error (IAE) which are computed as shown in below:

$$ISE = \sum (e(t))^2, \tag{5}$$

$$IAE = \sum |e(t)|, \tag{6}$$

Tables I and II show the aggregate errors of the two models for the three applications under different targets and system models. For higher target values, both the ISE and IAE are relatively smaller for the queue length model compared to the inverse model under the open system model. On the contrary, the reverse holds true for smaller target values. The implication is that, under open system model the queue length model is slightly preferable for higher target values while the inverse model is more preferable for lower target values. Under the closed system model, the error values for ISE and IAE are smaller for inverse model than the queue length model irrespective of the targets. This indicates that the inverse model is more preferable for closed system model.

Table I: Errors of the two models for RUBiS.

Target [seconds]	System Model	Performance Model	ISE	IAE
1.5	open	inverse	80.20	34.80
		queue length	73.61	32.62
	closed	inverse	148.07	70.47
		queue length	131.14	64.71
1.0	open	inverse	55.04	29.82
		queue length	43.36	26.87
	closed	inverse	119.40	67.05
		queue length	103.30	72.98
0.5	open	inverse	6.01	75.23
		queue length	88.38	85.75
	closed	inverse	43.68	57.05
		queue length	110.76	100.69
0.1	open	inverse	2.60	8.91
		queue length	7.11	14.33
	closed	Inverse	2.05	7.05
		queue length	2.21	8.74

Table II: Errors of the two models for Olio and RUBBoS with 0.5s target.

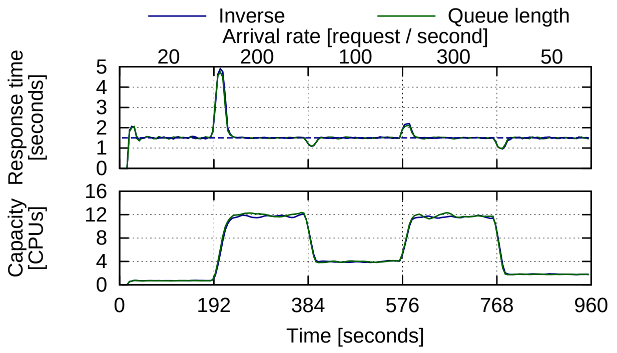
Application	System Model	Performance Model	ISE	IAE
Olio	open	inverse	19.27	35.42
		queue length	48.57	41.73
	closed	inverse	17.01	33.70
		queue length	188.74	93.54
RUBBoS	open	inverse	10.19	14.65
		queue length	9.95	16.61
	closed	inverse	50.78	53.76
		queue length	319.86	160.25

D. Adaptation Period

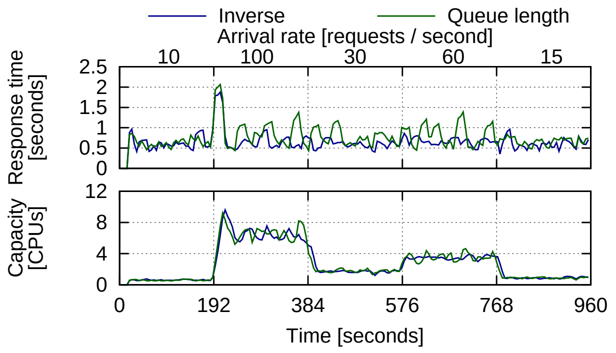
The adaptation period measures the duration (in the worst case) each model takes to converge to the target value after a change in the system (i.e., number of users) was introduced. We say that the system converged if the deviation of measured response time is within 10% of the target. Table III shows the values of the adaptation period of each performance model for different applications. With higher target values, the system always converges within 10 to 30 seconds. With lower targets, the queue length model does not converge according to the above definition, as it does not maintain response time close enough to the target. The inverse model failed to keep response time close to target with an open system model, however, it managed to converge with a closed system model within 40 seconds.

Table III: Adaptation period for the three applications.

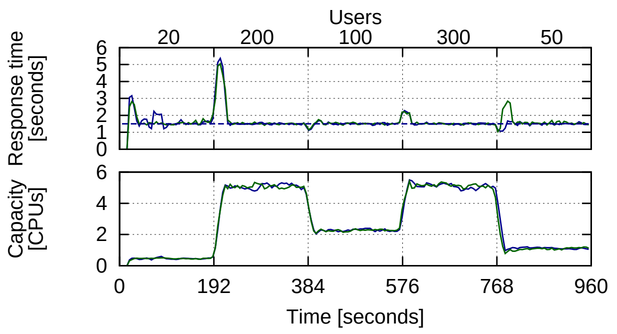
Application	Target [seconds]	Adaptation period [seconds]			
		Open		Closed	
		Inverse	Queue length	Inverse	Queue length
Olio	1.5	20	20	20	20
	0.5	–	–	25	–
RUBBoS	1.5	10	20	10	10
	0.5	20	35	40	–
RUBiS	1.5	25	25	20	25
	1.0	25	20	30	30
	0.5	–	–	10	–



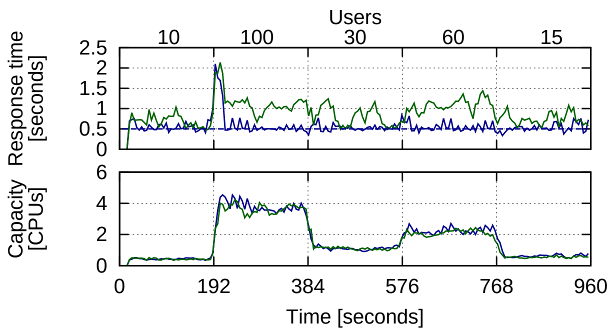
(a) open system model, 1.5s target



(b) open system model, 0.5s target

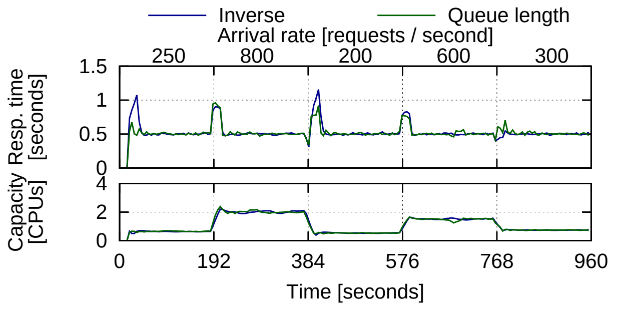


(c) closed system model, 1.5s target

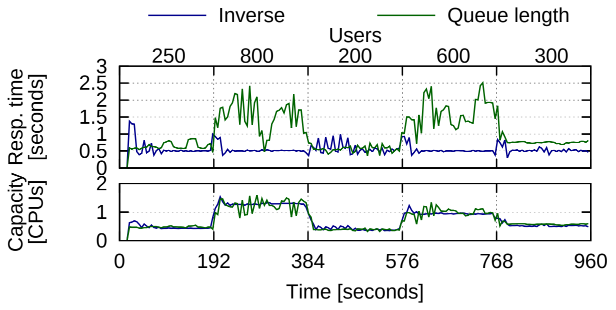


(d) closed system model, 0.5s target

Figure 1: RUBiS–under open and closed system models with 0.5s and 1.5s target response time.

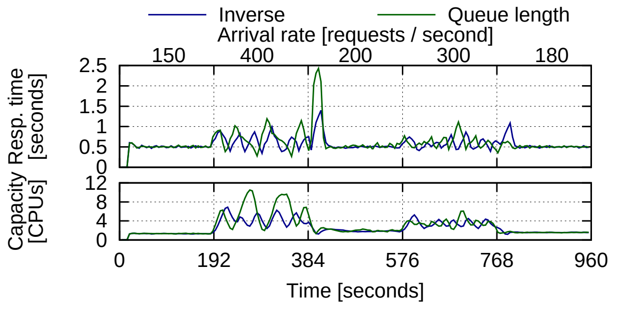


(a) open system model, 0.5s target

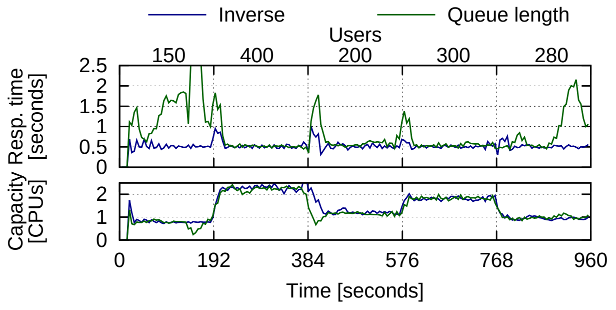


(b) closed system model, 0.5s target

Figure 2: RUBBoS–under open and closed system models with 0.5s target response time.



(a) open system model, 0.5s target



(b) closed system model, 0.5s target

Figure 3: Olio–under open and closed system models with 0.5s target response time.

E. Discussion

Experiments highlighted that both models behave well for relatively higher response time targets under both closed and open system models. However, for lower response time targets the inverse model performs better. Specifically, the results show the following **key findings**:

1) The models properly detect and allocate the capacity required for both open and closed system models.
2) Both models show more stability for relatively higher targets. However, the inverse model is more stable for lower targets than queue length model.
3) The inverse model is more stable under closed system model than under open system model for lower targets.
4) Both models reach stability very fast (i.e., less than 40 seconds or 8 control intervals) after detecting change in the system.

V. Conclusion

We presented two generic performance models for mean response time that map performance to capacity in order to provide performance guarantees for interactive applications deployed in the cloud. We carried out an extensive set of experiments using different applications by varying the workload mix over time under both closed and open system models. We also varied the target response time of each application to see how the models behave. The results demonstrate that the two models are stable for higher response time targets. However, our inverse model, shows more stability than the queue length model for lower targets. Furthermore, our inverse model converges within 40 seconds, which is relatively low compared to the 20 minutes required by the state-of-the-art. Thus, our contribution paves the way to capacity allocation for vertical elasticity, enabling the future Resource as a Service (RaaS) cloud.

Future work includes improving the stability and reaction time of our inverse model and extending it to handle the tail of response time.

Acknowledgement

This work was partially supported by the Swedish Research Council (VR) project *Cloud Control* and the Swedish Government's strategic effort *eSSENCE*.

References

[1] Tutorial: Installing a LAMP web server, 2013. available online: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-LAMP.html.

[2] Olio, 2014. Available online: http://incubator.apache.org/projects/olio.html.

[3] Rice university bidding system, 2014. Available online: http://rubis.ow2.org.

[4] Rubbos, 2014. Available online: http://jmob.ow2.org/rubbos.html.

[5] Paul Barham et al. Xen and the art of virtualization. In *SOSP*. ACM, 2003.

[6] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The rise of raas: The resource-as-a-service cloud. *Commun. ACM*, 57(7), 2014.

[7] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42(5):104–113, 2008.

[8] Abhishek Chandra et al. Dynamic resource allocation for shared data centers using online measurements. In *IWQoS*, pages 381–398. Springer, 2003.

[9] Yuan Chen et al. SLA decomposition: Translating service level objectives to system level thresholds. In *ICAC*. IEEE, 2007.

[10] Christian Ehrhardt. Cpu time accounting. Last accessed: Aug. 2013.

[11] Zhenhuan Gong et al. PRESS: Predictive elastic resource scaling for cloud systems. In *CNSM*. IEEE, 2010.

[12] Jordi Guitart, Jordi Torres, and Eduard Ayguadé. A survey on performance management for internet applications. *Concurr. Comput. : Pract. Exper.*, 22(1):68–106, 2010.

[13] Weifeng Liu, Jose C. Principe, and Simon Haykin. *Kernel Adaptive Filtering: A Comprehensive Introduction*. Wiley Publishing, 1st edition, 2010.

[14] Lei Lu, Xiaoyun Zhu, Rean Griffith, Pradeep Padala, Aashish Parikh, Parth Shah, and Evgenia Smirni. Application-driven dynamic vertical scaling of virtual machines in resource pools. In *NOMS*, 2014.

[15] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour and Information Technology*, 23(3), 2004.

[16] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, 2013.

[17] Pradeep Padala et al. Automated control of multiple virtualized resources. In *European Conference on Computer Systems (EuroSys)*. ACM, 2009.

[18] Eric Ries. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, 2011.

[19] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Networked Systems Design and Implementation (NSDI)*, 2006.

[20] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *SoCC*. ACM, 2011.

[21] Kunwadee Sripanidkulchai et al. Are clouds ready for large distributed applications? *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.

[22] Christopher Stewart et al. Exploiting nonstationarity for performance prediction. In *EuroSys*. ACM, 2007.

[23] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *NSDI*, pages 71–84. USENIX, 2005.

[24] Nedeljko Vasić et al. DejaVu: accelerating resource allocation in virtualized environments. In *ASPLOS*. ACM, 2012.

[25] Wei Zheng et al. JustRunIt: experiment-based management of virtualized data centers. In *ATC*, pages 18–28. USENIX, 2009.