2013 IEEE International Conference on Cloud Engineering

Techniques for Optimizing Cloud Footprint

Arun Kejariwal
Netflix Inc.

***Abstract*—Infrastructure as a Service (IaaS) has emerged as a popular service model in the context of cloud computing. Examples of IaaS vendors include, but not limited to, Amazon's EC2, Rackspace, GoGrid and the Google Compute Engine. Use of IaaS obviates the need for set up and maintenance of infrastructure and thereby boosts product development agility – a key in increasingly competitive landscape. However, the use of IaaS is much more expensive compared to use of an in-house datacenter. This calls for development of techniques to minimize the cost overhead associated with the use of an IaaS without sacrificing its various benefits, e.g., elasticity of the cloud. In this paper we present novel techniques to optimize operational efficiency in the cloud. Specifically, we present three techniques targeted to different production scenarios. The techniques have been deployed in production and resulted in up to 50% reduction in operational costs for the target Netflix applications.**

I. Introduction

Cloud computing has emerged as a dominant paradigm in both academic and industry circles. Mell and Grance from the National Institute of Standards and Technology (NIST) defined cloud computing as follows [1]:

> Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (such as networks, servers, storage, applications, and services) that can be quickly provisioned and released with minimal management effort or service provider interaction.

In [2], Feldman reports that the adoption of cloud computing in the industry has been rising on a Year-over-Year (YoY) basis. In a recent report International Data Corporation (IDC) predicted cloud services to generate $100 billion in revenue in 2016 (from $21.5 billion in 2010) [3].

Infrastructure as a Service (IaaS) [4] is one of the service models of cloud computing. In [5], Garrison et al. describe IaaS in the following way: *"With IaaS, the cloud vendor provides the servers (such as processing capability), storage (such as replication, backup, and archiving), and connectivity domains (such as firewalls and load balancing), with the client organization charged based on their use."* Several vendors provide IaaS such as, but not limited to, Amazon's EC2 [6], Rackspace [7], GoGrid [8] and the Google Compute Engine [9].

Netflix [10] migrated its infrastructure from a traditional data center to Amazon's EC2 to leverage, amongst other benefits, elasticity and support for multiple availability zones. Migration to the cloud eliminated the cycles spent on hardware procurement, datacenter maintenance and resulted in higher development agility [11], [12], [13]. Having said that, capitalizing on the elasticity of cloud – a cluster can be dynamically scaled up or down – efficiently is non-trivial. Specifically, one has to be vary of the following:

- Aggressive scale down can potentially adversely impact latency and throughput (in the worst case, the service may become unavailable). Higher latency would degrade the experience of the end users (Netflix subscribers in the current context). Further, from a corporate standpoint, lower throughput would adversely impact the bottomline (this holds in general for any end-user facing service).
- Aggressive scale up can result in over provisioning thereby ballooning the footprint on the cloud. As above, higher operational costs would adversely affect the bottomline.

Additionally, efficient exploitation of elasticity of the cloud can contain the overall footprint, i.e., across multiple applications. In the long run, from Table I we note that *on-demand* usage is much more expensive than the use of *reserved* instances [14].[1]

Instance Type	Price ($ per Hour)	
	On-Demand	Reserved
m1.xlarge	0.64	0.192
m2.xlarge	0.45	0.133
m2.2xlarge	0.90	0.266
m2.4xlarge	1.80	0.532
cc2.2xlarge	2.40	0.54

Table I
COST PER HOUR FOR *on-demand* AND *reserved* INSTANCES ON EC2

Consequently, it is critical to develop novel techniques to exploit elasticity of the cloud systematically.

Amazon's EC2 supports dynamic scaling – referred to as *Auto Scaling* [15] – of a cluster/ASG (Auto Scaling Group) [16]. There are multiple parameters associated with each scaling policy type (an overview of the parameters is presented in Section II). This in conjunction with time varying nature of incoming traffic (as illustrated in Figure 1) makes optimizing the footprint on the cloud difficult.

To this end, we present novel techniques to exploit the elasticity of the cloud, thereby reducing the operational

[1]The costs for reserved instances corresponds to the use of Linux as the operating system and assuming medium utilization.

978-0-7695-4945-3/13 $26.00 © 2013 IEEE
DOI 10.1109/IC2E.2013.14

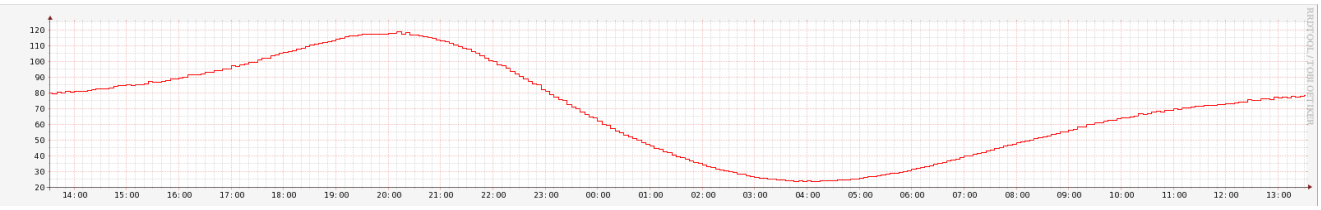


Figure 1. Illustration of variation in load in production

footprint on the cloud. Specifically, the main contributions of the paper are as follows:

- First, we present novel techniques to optimize the operational footprint on the cloud. In particular,
  - We present a technique to guide the scaling – both up and down – of a cluster by a fixed amount.
  - We present a technique to guide the scaling – both up and down – of a cluster by a percentage of the current capacity.
  - Many applications such as the Netflix recommendation service incur long start up times owing to, for example, loading of large amount of data. In such cases, it is critical to account for the start up time during the design of the scaling policy. We present a technique to guide the scaling – both up and down – application with long startup times.

  Note that the aforementioned techniques were deployed in production and resulted in up to 50% reduction in operational costs for the target Netflix applications.
- Second, we illustrate the efficacy of the proposed techniques using Netflix applications as case studies. Although the evaluation is done on Amazon's EC2, the proposed techniques are applicable in general to any IaaS service.

The rest of the paper is organized as follows: Section II presents a brief overview of autoscaling on Amazon's EC2 and the associated terminology. Section III details the techniques mentioned above and their use cases at Netflix. Previous work is discussed in Section IV. Finally, in Section V, we conclude with directions for future work.

II. Background

In this section, we present a brief overview of autoscaling on Amazon's EC2 and the associated terminology.

Amazon's Auto Scaling service [15] lets one launch or terminate EC2 instances (up to a defined minimum and maximum respectively) automatically based on user-defined policies, schedules, and health checks. We used Amazon's CloudWatch [17] for real-time monitoring of EC2 instances. Metrics such as CPU utilization, latency, and request counts are provided automatically by CloudWatch. Further, we used CloudWatch to access up-to-the-minute statistics, view graphs, and set alarms (defined below).

Definition 1: An Amazon CloudWatch alarm is an object that watches over a single metric. An alarm can change state depending on the value of the metric. An action is invoked when an alarm changes state and remains in that state for a number of time periods.

One can configure a CloudWatch alarm to send a message to autoscaling whenever a specific metric has reached a threshold value. When the alarm sends the message, autoscaling executes the associated policy [18] on an Auto Scaling Group (ASG) to scale the group up or down. Note that an Auto Scaling action is invoked when the specified metric remains above the threshold value for a number of time periods. This is to ensure that a scaling action is not triggered due to a sudden spike in the metric.

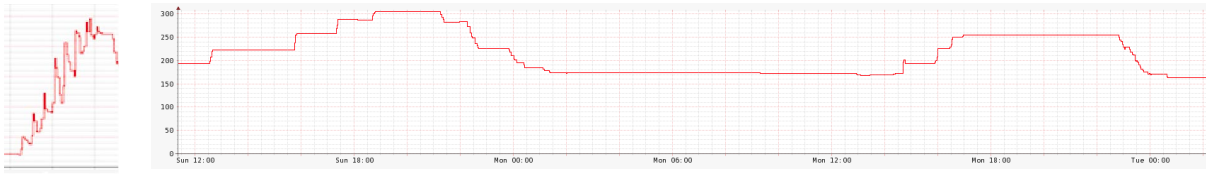
Definition 2: A policy is a set of instructions for Auto Scaling that tells the service how to respond to CloudWatch alarm messages.

Separate policies are instituted for autoscaling up and autoscaling down. The two key parameters associated with an Auto Scaling Policy are the following:

- `ScalingAdjustment`: The number of instances by which to scale. `AdjustmentType` determines the interpretation of this number (e.g., as an absolute number or as a percentage of the existing ASG size). A positive increment adds to the current capacity and a negative value removes from the current capacity.
- `AdjustmentType`: It specifies whether the ScalingAdjustment is an absolute number or a percentage of the current capacity. Valid values are `ChangeInCapacity` or `PercentChangeInCapacity` (described in subsections III-D and III-F respectively).

An autoscaling action, say scale up, usually takes a while to take effect. In light of this, one can specify a `cooldown` period (defined below) to ensure that a new autoscaling event is triggered after the completion of the previous autoscaling event.

Definition 3: Cooldown is the period of time after autoscaling initiates a scaling activity during which no other scaling activity can take place. A cooldown period allows the effect of a scaling activity to become visible in the metrics that originally triggered the activity. This period



(a) (b)

Figure 2. (a) Illustrating Ping-Pong effect (b) Desired autoscaling profile (Y-axis corresponds to the number of nodes in the ASG and X-axis corresponds to time)

is configurable, and gives the system time to perform and adjust to any new scaling activities (such as scale-in and scale-out) that affect capacity.

In this paper we employ *scaling by policy* [19] wherein a given cluster was scaled up/down based on the incoming RPS (request per second) of a given application. We used incoming RPS as the metric to drive autoscaling as it is independent of the application and directly relates to throughput.

III. Techniques

In this section, we detail the techniques to optimize the operational footprint on the cloud.

A. Design Guidelines

In this subsection we brief the considerations behind the design of the proposed techniques.

1) Avoiding Ping-Pong Effect: During a scale up event, new nodes are added to a given ASG. As a consequence, the RPS per node drops. However, if the RPS per node drops below the threshold specified for scaling down an ASG, it would trigger a scale down event. This would result in alternating scale up and scale down events, as illustrated in Figure 2(a) and referred to as *ping-pong* effect. At Netflix, we observed that ping-ponging can potentially result in higher latency and, in the worst case, may cause violation of the SLA (service level agreement) of the service.

Thus, when defining the autoscaling policies, it is imperative to make sure that the policy is not susceptible to ping-pong effect. The desired autoscaling profile is exemplified in Figure 2(b).

2) Being Proactive, Not Reactive: As mentioned earlier, applications such as the recommendation engine at Netflix take a long time to start up. This may be ascribed to a variety of reasons, e.g., loading of metadata of Netflix subscribers, precomputation of certain features. For such applications it is critical to trigger the scale up event in a proactive fashion, not reactively.

Let us consider the scenario shown in Figure 3. The solid arrow in the figure corresponds to the need to scale up a given ASG as mandated by the SLA and increasing traffic. However, owing to a long application start up time, the autoscaling up is triggered, signified by the dashed arrow in the figure, proactively. The proactive approach ensures that the ASG is sufficiently provisioned by the time the latency approaches the SLA and that the SLA is never violated!

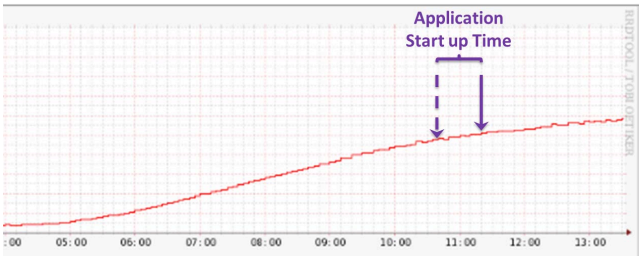


Figure 3. Illustration of scaling in a proactive fashion. Solid arrow signifies the need to scale up (as governed by SLA of the application at hand) and the dashed arrow signifies the corresponding autoscaling event (governed by the start up of the application).

3) Aggressive Upwards, Conservative Downwards: Delivering the best user experience is critical for business. Thus, we employ an aggressive scale up policy so as to be able to handle more than expected increase in traffic. Also, an aggressive scale up approach provides a buffer for increase in traffic during the cooldown period.

In contrast, we employ a conservative scale down policy so as to be able to handle slower (than the historical trend) ramp down of traffic. Aggressive scale down may accidentally result in under provisioning thereby adversely impacting latency and throughout.

B. Scalability Analysis

Determining the threshold for scale up is an integral step is defining an autoscaling policy. A low threshold will result in underutilization of the instances in the ASG; on the other hand, a high threshold may result in higher latency thereby

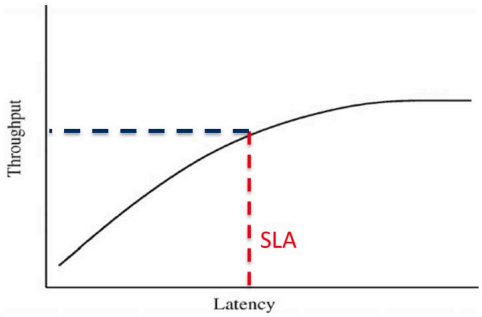


Figure 4. Trade-off between Latency and Throughput (load)

Scale Up						Scale Down					
# Nodes (Current)	Nodes Added	RPS_{ASG}	RPS_n	Total Nodes	New RPS_n	# Nodes (Current)	Nodes Added	RPS_{ASG}	RPS_n	Total Nodes	New RPS_n
6	0	500	83.33	6		18		5000	277.78	18	
		1740						3240			
	3			9	193.33		2			16	202.50
		2610						2880			
	3			12	217.50		2			14	205.71
		3480						2520			
	3			15	232.00		2			12	210.00
		4350						2160			
	3			18	241.67		2			10	216.00
		5220						1800			
	3			21	248.57		2			8	225.00

Table II
ILLUSTRATION OF ALGORITHM 1 ($\mathcal{D} = 2, \mathcal{U} = 3, \mathcal{T}_D = 180, \mathcal{T}_U = 230$)

degrading the user experience. To this end, we determine the throughput corresponding to the SLA of application, refer to Figure 4.

Given an application, we carried scalability analysis to determine the profile akin to Figure 4. Specifically, we captured Apache access logs from production nodes, massaged the logs such that resulting request log could be fed to JMeter [20]. Load was pushed, using JMeter, against canaries in production to determine the scalability profile.

C. Properties

The algorithms presented in the rest of the section must satisfy the following properties. Each scale up event should satisfy the following:

Property 1: RPS per node after scale up should be more than the scale down threshold ($\mathcal{T}_D$).

Property 1 ensures that scale up does not induce a ping-pong effect (refer to subsubsection III-A1). Likewise, each scale down event should satisfy the following:

Property 2: RPS per node after scale down should be less than the scale up threshold ($\mathcal{T}_U$).

Akin to Property 1, Property 2 also ensures that scale up does not induce a ping-pong effect.

D. Algorithm 1

In this subsection we present a technique for scaling an ASG up/down by a fixed number of instances and as per the guidelines laid out earlier in this section. `AdjustmentType` (refer to Section II) for the scaling policy is set to `ChangeInCapacity` (defined below).

- `ChangeInCapacity`: This `AdjustmentType` is used to increase or decrease the capacity, by a fixed amount, on top of the existing capacity. For instance, let us assume that the capacity of a given ASG is three and that `ChangeInCapacity` is set to five. When the policy is executed, autoscaling will add five more instances to ASG.

Algorithm 1 details the parameters and the steps to determine the scaling thresholds (for both scaling up and scaling

Algorithm 1 Autoscaling up/down by a fixed amount

Input: An application with a specified SLA.

Parameters: $\mathcal{D}$ Scale down value
$\mathcal{U}$ Scale up value
$\mathcal{T}_D$ Scale down threshold (RPS per node)
$\mathcal{T}_U$ Scale up threshold (RPS per node)
$N_{\min}$ Minimum number of nodes in the ASG

Let $\mathcal{T}(\text{SLA})$ return the maximum RPS per node for the specified SLA.

$\mathcal{T}_U \leftarrow 0.90 \times \mathcal{T}(\text{SLA})$

$\mathcal{T}_D \leftarrow 0.50 \times \mathcal{T}_U$

Let $\text{RPS}_{\text{Peak}}, \text{RPS}_{\min}$ denote the peak and minimum RPS observed for the ASG over the last, say, two weeks

Let N_c, RPS_n denote the current number of nodes and RPS per node respectively

L1: /* Scale Up (if $\text{RPS}_n > \mathcal{T}_U$) */
repeat
  $\text{RPS}_{\text{ASG}} \leftarrow N_c \times \text{RPS}_n$
  $N_c \leftarrow N_c + \mathcal{U}$
  $\text{RPS}_n \leftarrow \text{RPS}_{\text{ASG}}/N_c$
until $\text{RPS}_n \times N_c \leq \text{RPS}_{\text{Peak}}$

L2: /* Scale Down (if $\text{RPS}_n < \mathcal{T}_D$) */
repeat
  $\text{RPS}_{\text{ASG}} \leftarrow N_c \times \text{RPS}_n$
  $N_c \leftarrow max(N_{\min}, N_c - \mathcal{D})$
  $\text{RPS}_n \leftarrow \text{RPS}_{\text{ASG}}/N_c$
until $\text{RPS}_n \times N_c \geq \text{RPS}_{\min}$ **or** $N_c = N_{\min}$

if Properties 1 and/or 2 are not satisfied for each scale up and scale down respectively **then**
  Adjust $\mathcal{D}, \mathcal{U}, \mathcal{T}_D, \mathcal{T}_U$ incrementally
  Revisit L1 and L2
end if

down). The scale down value $\mathcal{D}$ and the scale up value $\mathcal{U}$ are inputs to the algorithm. The constants – 0.90 and 0.50 - used in defining $\mathcal{T}_U, \mathcal{T}_D$ were determined empirically so as to minimize impact on user experience and contain ASG under utilization. Loop `L1` in Algorithm 1 corresponds to scaling up an ASG as the incoming traffic increases. Loop `L2` in Algorithm 1 scales down an ASG as the incoming traffic decreases. If Properties 1 and/or 2 are not satisfied then the algorithm adjusts the parameters $\mathcal{D}, \mathcal{U}, \mathcal{T}_D, \mathcal{T}_U$ in an incremental fashion and iterates through the loops `L1` and `L2`.

For better understanding of Algorithm 1, we walk through a case study (refer to Table II) of a Netflix application. The parameters of the algorithm are mentioned in the caption of Table II. RPS_{Peak} for the application was 5300. Initially, $\text{RPS}_{\text{ASG}} = 500$ and $N_c = 6$. As RPS_{ASG} increases to 1540, RPS_n approaches $\mathcal{T}_U$. An autoscaling up event gets triggered thereby adding 3 ($= \mathcal{U}$) nodes to the ASG. Subsequently, the ASG scales up until $\text{RPS}_n \times N_c \leq \text{RPS}_{\text{Peak}}$. Note that all the entries in the column six satisfy Property 1.

During scale down, the initial $\text{RPS}_{\text{ASG}} = 5000$ and $N_c = 18$. As RPS_{ASG} decreases to 3240, RPS_n approaches $\mathcal{T}_D$. An autoscaling down event gets triggered thereby deleting 2 ($= \mathcal{D}$) nodes from the ASG. Subsequently, the ASG scales down until $\text{RPS}_n \times N_c \geq \text{RPS}_{\text{min}}$ or $N_c = N_{\text{min}}$. Note that all the entries in the column twelve satisfy Property 2.

E. Algorithm 2

In this subsection we present a technique for scaling an ASG up/down by a percentage of current capacity and as per the guidelines laid out earlier in this section. `AdjustmentType` (refer to Section II) for the scaling policy is set to `PercentChangeInCapacity` (defined below).

- `PercentChangeInCapacity`: This `AdjustmentType` is used to increase or decrease the capacity by a percentage of the desired capacity. For instance, let an ASG have 15 instances and a scaling up policy of the type `PercentChangeInCapacity` and adjustment set to fifteen. When the policy is executed, autoscaling will increase the ASG size by two.

  Note that if the `PercentChangeInCapacity` returns a value between 0 and 1, autoscaling will round it off to 1. If the `PercentChangeInCapacity` returns a value greater than 1, autoscaling will round it off to the lower value.

Algorithm 2 details the parameters and the steps to determine the scaling thresholds (for both scaling up and scaling down). The scale down value $\mathcal{D}$ and the scale up value $\mathcal{U}$ (note that both are percentages) are inputs to the algorithm. The constants – 0.90 and 0.50 - used in defining $\mathcal{T}_U, \mathcal{T}_D$ were determined empirically so as to minimize impact on user experience and contain ASG under utilization. Loop

Algorithm 2 Autoscaling up/down by a percentage of current capacity

Input: An application with a specified SLA.

Parameters: $\mathcal{D}$ Scale down percetage value
$\mathcal{U}$ Scale up percetage value
$\mathcal{U}_{\text{min}}$ Minimum scale up value
$\mathcal{T}_D$ Scale down threshold (RPS per node)
$\mathcal{T}_U$ Scale up threshold (RPS per node)
N_{min} Minimum number of nodes in the ASG

Let $\mathcal{T}(\text{SLA})$ return the maximum RPS per node for the specified SLA.

$\mathcal{T}_U \leftarrow 0.90 \times \mathcal{T}(\text{SLA})$

$\mathcal{T}_D \leftarrow 0.50 \times \mathcal{T}_U$

Let $\text{RPS}_{\text{Peak}}, \text{RPS}_{\text{min}}$ denote the peak and minimum RPS observed for the ASG over the last, say, two weeks

Let N_c, RPS_n denote the current number of nodes and RPS per node respectively

L1: /* Scale Up (if $\text{RPS}_n > \mathcal{T}_U$) */
repeat
  $\text{RPS}_{\text{ASG}} \leftarrow N_c \times \text{RPS}_n$
  $N_c \leftarrow N_c + \max(1, N_c \times \mathcal{U}/100)$
  $\text{RPS}_n \leftarrow \text{RPS}_{\text{ASG}}/N_c$
until $\text{RPS}_n \times N_c \leq \text{RPS}_{\text{Peak}}$

L2: /* Scale Down (if $\text{RPS}_n < \mathcal{T}_D$) */
repeat
  $\text{RPS}_{\text{ASG}} \leftarrow N_c \times \text{RPS}_n$
  $N_c \leftarrow \max(N_{\text{min}}, N_c - \max(1, N_c \times \mathcal{D}/100))$
  $\text{RPS}_n \leftarrow \text{RPS}_{\text{ASG}}/N_c$
until $\text{RPS}_n \times N_c \geq \text{RPS}_{\text{min}}$ **or** $N_c = N_{\text{min}}$

if Properties 1 and/or 2 are not satisfied for each scale up and scale down respectively **then**
  Adjust $\mathcal{D}, \mathcal{U}, \mathcal{T}_D, \mathcal{T}_U$ incrementally
  Revisit L1 and L2
end if

`L1` in Algorithm 2 corresponds to scaling up an ASG as the incoming traffic increases. Loop `L2` in Algorithm 2 scales down an ASG as the incoming traffic decreases. If Properties 1 and/or 2 are not satisfied then the algorithm adjusts the parameters $\mathcal{D}, \mathcal{U}, \mathcal{T}_D, \mathcal{T}_U$ in an incremental fashion and iterates through the loops `L1` and `L2`.

For better understanding of Algorithm 2, we walk through a case study (refer to Table III) of the same Netflix application as in the previous subsection. The parameters of the algorithm are mentioned in the caption of Table II. N_{min} is set 1. Recall that RPS_{Peak} for the application was 5300 and initially, $\text{RPS}_{\text{ASG}} = 500$ and $N_c = 6$. As RPS_{ASG} increases to 1540, RPS_n approaches $\mathcal{T}_U$, an autoscaling up

Scale Up						Scale Down					
# Nodes (Current)	Nodes Added	RPS_{ASG}	RPS_n	Total Nodes	New RPS_n	# Nodes (Current)	Nodes Added	RPS_{ASG}	RPS_n	Total Nodes	New RPS_n
6	0	500	83.33	6		18		5000	277.78	18	
		1740						4140			
	1			7	248.57		1			17	243.53
		2030						3910			
	1			8	253.75		1			16	244.38
		2320						3680			
	1			9	257.78		1			15	245.33
		2610						3450			
	1			10	261.00		1			14	246.43
		2900						3220			
	1			11	263.64		1			13	247.69
		3190						2990			
	1			12	265.83		1			12	249.17
		3480						2760			
	1			13	267.69		1			11	250.91
		3770						2530			
	1			14	269.29		1			10	253.00
		4060						2300			
	1			15	270.67		1			9	255.56
		4350						2070			
	1			16	271.88		1			8	258.75
		4640						1840			
	1			17	272.94		1			7	262.86
		4930						1610			
	1			18	273.89		1			6	268.33
		5220									
	1			19	274.74						

Table III
ILLUSTRATION OF ALGORITHM 2 ($\mathcal{D} = 8$, $\mathcal{U} = 10$, $N_{\min} = 1$, $\mathcal{T}_D = 230$, $\mathcal{T}_U = 290$)

event gets triggered thereby adding 1 ($= \max(1, 6 \times 10/100)$) node to the ASG. Subsequently, the ASG scales up until $\text{RPS}_{\text{ASG}} \times N_c \leq \text{RPS}_{\text{Peak}}$. Note that all the entries in the column six satisfy Property 1.

During scale down, the initial $\text{RPS}_{\text{ASG}} = 5000$ and $N_c = 18$. As RPS_{ASG} decreases to 4140, RPS_n approaches $\mathcal{T}_D$. An autoscaling down event gets triggered thereby deleting 1 ($= \max(1, \lfloor 18 * 8/100 \rfloor)$) nodes[2] from the ASG. Subsequently, the ASG scales down until $\text{RPS}_n \times N_c \geq \text{RPS}_{\min}$ or $N_c = N_{\min}$. Note that all the entries in the column twelve satisfy Property 2.

On comparing Tables II and III we note that the threshold values $\mathcal{U}$ and $\mathcal{D}$ are higher in case of the latter. This boosts hardware utilization and reduces the footprint on the cloud.

F. Algorithm 3

In this section we extend Algorithm 2 to guide autoscaling for applications with long start up times. Long application start up times may be ascribed to a variety of reasons, e.g., loading of metadata of Netflix subscribers. As discussed earlier in subsubsection III-A2, in the presence of long start up times, autoscaling up need to be done proactively. For this, we employ the following steps:

- For a historical time series of RPS in production, determine the change in RPS over every $\mathcal{A}_{\text{start}}$ minutes, where $\mathcal{A}_{\text{start}}$ denotes the application start up time. This would yield a time series with the the following data points:

$$\begin{aligned} &\text{RPS}_{\mathcal{A}_{\text{start}}} - \text{RPS}_0 \\ &\text{RPS}_{\mathcal{A}_{\text{start}}+1} - \text{RPS}_1 \\ &\text{RPS}_{\mathcal{A}_{\text{start}}+2} - \text{RPS}_2 \\ &\text{RPS}_{\mathcal{A}_{\text{start}}+3} - \text{RPS}_3 \\ &\ldots \end{aligned}$$

  where RPS_t denotes the RPS at time t. The derived time series, referred to as *rolling RPS change*, captures the change in RPS in any window of width $\mathcal{A}_{\text{start}}$ minutes.
- Compute the 99$^{\text{th}}$ percentile of the rolling time series, denoted by $\mathcal{R}_{\text{RPS}}$.
- Compute $\Upsilon = \mathcal{T}_U - \mathcal{R}_{\text{RPS}}$. The parameter Υ is the effective threshold for scale up. The use of 99$^{\text{th}}$ percentile of the *rolling RPS change* time series is consistent with the *Aggressive Upwards* guideline outlined earlier in subsection III-A.

An example RPS time series (with 1 minute granularity) of a Netflix application is shown in Figure 5. The start up time of the application was 30 minutes. The corresponding rolling RPS time series is shown in Figure 6. The 99$^{\text{th}}$ percentile

[2]Recall that if `PercentChangeInCapacity` returns a value greater than 1, autoscaling rounds it off to the lower value.

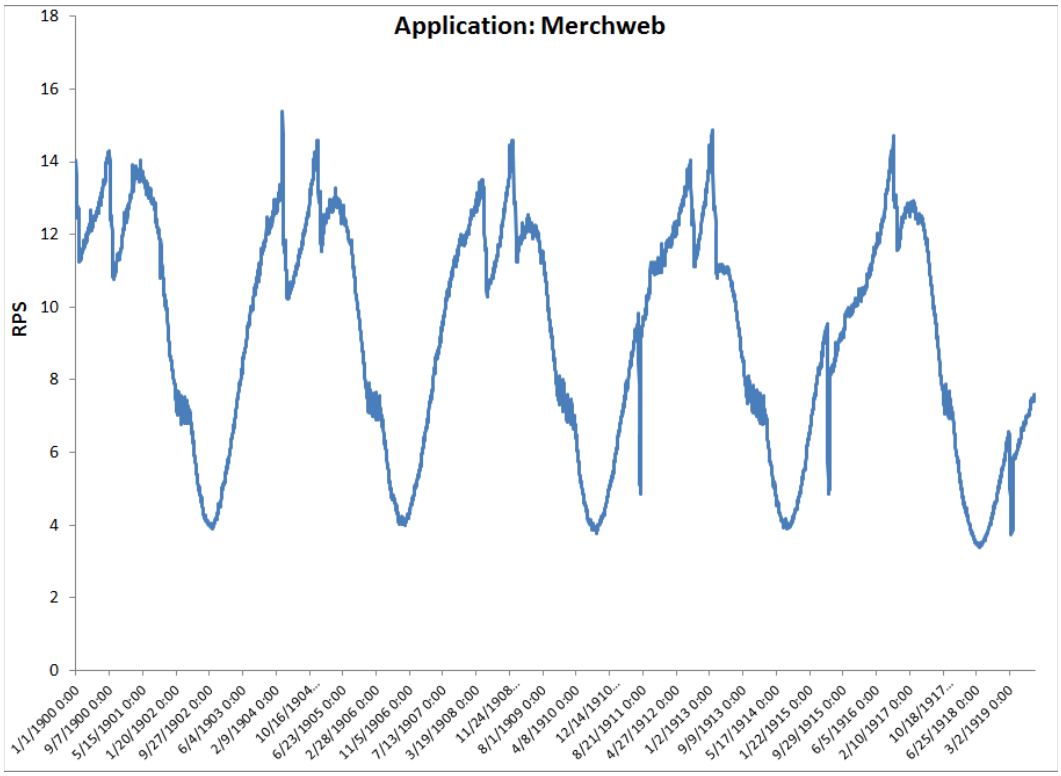


Figure 5. Time series of RPS_n for a Netflix application

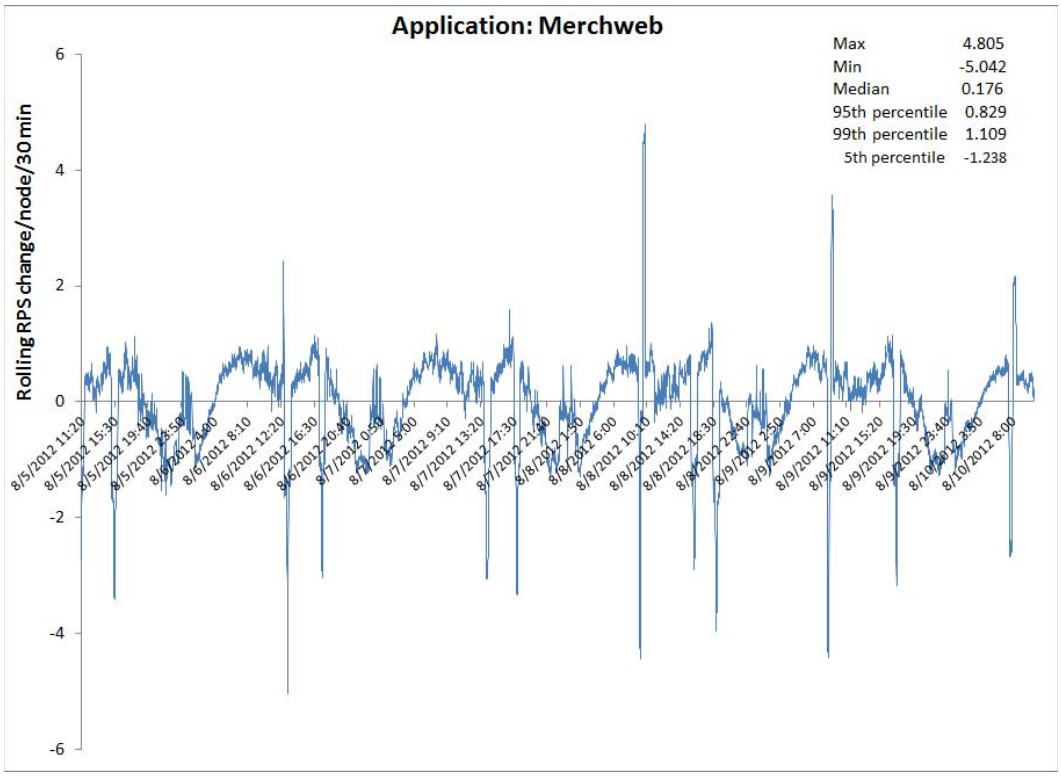


Figure 6. Illustration of rolling change in RPS_n for a Netflix application start up time of 30 minutes

Algorithm 3 Application start up aware autoscaling up/down by a percentage of current capacity

Input: An application with a specified SLA.

Parameters: $\mathcal{D}$ Scale down percetage value
$\mathcal{U}$ Scale up percetage value
$\mathcal{U}_{\min}$ Minimum scale up value
$\mathcal{A}_{\text{start}}$ Application start up time (mins)
$\mathcal{T}_D$ Scale down threshold (RPS per node)
$\mathcal{T}_U$ Scale up threshold (RPS per node)
$N_{\min}$ Minimum number of nodes in the ASG

Let $\mathcal{T}(\text{SLA})$ return the maximum RPS per node for the specified SLA.

$\mathcal{T}_U \leftarrow 0.90 \times \mathcal{T}(\text{SLA})$

$\mathcal{T}_D \leftarrow 0.50 \times \mathcal{T}_U$

Let $\text{RPS}_{\text{Peak}}, \text{RPS}_{\min}$ denote the peak and minimum RPS observed for the ASG over the last, say, two weeks

Let N_c, RPS_n denote the current number of nodes and RPS per node respectively

Transform RPS time series to a rolling $\mathcal{A}_{\text{start}}$ (min) time series

Let $\mathcal{R}_{\text{RPS}}$ denote the 99^{th} percentile of the rolling time series

Let $\Upsilon = \mathcal{T}_U - \mathcal{R}_{\text{RPS}}$

L1: /* Scale Up (if $\text{RPS}_n > \Upsilon$) */
repeat
$\text{RPS}_{\text{ASG}} \leftarrow N_c \times \text{RPS}_n$
$N_c \leftarrow N_c + \max(1, N_c \times \mathcal{U}/100)$
$\text{RPS}_n \leftarrow \text{RPS}_{\text{ASG}}/N_c$
until $\text{RPS}_n \times N_c \leq \text{RPS}_{\text{Peak}}$

L2: /* Scale Down (if $\text{RPS}_n < \mathcal{T}_D$) */
repeat
$\text{RPS}_{\text{ASG}} \leftarrow N_c \times \text{RPS}_n$
$N_c \leftarrow \max(N_{\min}, N_c - \max(1, N_c \times \mathcal{D}/100))$
$\text{RPS}_n \leftarrow \text{RPS}_{\text{ASG}}/N_c$
until $\text{RPS}_n \times N_c \geq \text{RPS}_{\min}$ **or** $N_c = N_{\min}$

if Properties 1 and/or 2 are not satisfied for each scale up and scale down respectively **then**
Adjust $\mathcal{D}, \mathcal{U}, \mathcal{T}_D, \mathcal{T}_U$ incrementally
Revisit L1 and L2
end if

of the rolling time series is 1.109.

Algorithm 3 details the parameters and the steps to determine the scaling thresholds (for both scaling up and scaling down). The scale down value $\mathcal{D}$ and the scale up value $\mathcal{U}$ are inputs to the algorithm. The constants – 0.90 and 0.50 - used in defining $\mathcal{T}_U, \mathcal{T}_D$ were determined empirically so as to minimize impact on user experience and contain ASG under utilization. Loop `L1` in Algorithm 3 corresponds to scaling up an ASG as the incoming traffic increases. Loop `L2` in Algorithm 3 scales down an ASG as the incoming traffic decreases. Unlike scale up, the threshold for scale down $\mathcal{D}$ need not be adjusted as applications do not induce long delay during termination of instances on Amazon's EC2. If Properties 1 and/or 2 are not satisfied then the algorithm adjusts the parameters $\mathcal{D}, \mathcal{U}, \mathcal{T}_D, \mathcal{T}_U$ in an incremental fashion and iterates through the loops `L1` and `L2`.

For better understanding of Algorithm 3, we walk through a case study (refer to Table IV) of the Netflix application. The RPS and the rolling RPS change time series for the application are shown in Figures 5 and 6 respectively. The parameters of the algorithm are mentioned in the caption of Table IV. $N_{\min}$ is set 1. RPS_{Peak} for the application was 4500 and initially, $\text{RPS}_{\text{ASG}} = 800$ and $N_c = 170$. $\Upsilon = 12.9$. As RPS_{ASG} increases to 2193, RPS_n approaches Υ. An autoscaling up event gets triggered thereby adding 25 $(= \max(1, \lfloor 170 \times 15/100 \rfloor)$ nodes to the ASG. Subsequently, the ASG scales up until $\text{RPS}_{\text{ASG}} \times N_c \leq \text{RPS}_{\text{Peak}}$. Note that all the entries in the column seven satisfy Property 1.

During scale down, the initial $\text{RPS}_{\text{ASG}} = 4400$ and $N_c = 389$. As RPS_{ASG} decreases to 3890, RPS_n approaches $\mathcal{T}_D$. An autoscaling down event gets triggered thereby deleting 38 $(= \max(1, \lfloor 389 * 10/100 \rfloor))$ nodes from the ASG. Subsequently, the ASG scales down until $\text{RPS}_n \times N_c \geq \text{RPS}_{\min}$ or $N_c = N_{\min}$. Note that all the entries in the column thirteen satisfy Property 2.

There have been cases wherein the CPU utilization on production nodes spiked without any increase in traffic. This may happen to a variety of accidental events. To handle such cases, instituting add-on scale up policies (i.e., besides scale up policy based on RPS), as exemplified in Figure 7, helps to mitigate the impact on the end users.

Figure 7. Add-on policies to check "meltdown"

IV. Previous Work

In this section, we overview previous work in the context of evaluating cloud services with respect to, for example, performance variation on the cloud.

In [21], Dejun et al. studied the performance behavior of small instances in Amazon's EC2 and demonstrated that the performance of virtual instances is relatively stable over time with fluctuations of mean response time within at most 8% of the longterm average. In [22], Stantchev presented an approach for evaluation of cloud computing configurations

Scale Up							Scale Down					
# Nodes (Current)	Nodes Added	RPS_{ASG}	RPS_n	$\Upsilon = \mathcal{T}_U - \mathcal{R}_{RPS}$	Total Nodes	New RPS_n	# Nodes (Current)	Nodes Added	RPS_{ASG}	RPS_n	Total Nodes	New RPS_n
170	0	800	4.71	12.9	170		389		4400	11.31	389	
		2193							3890			
	25				195	11.25		38			351	11.08
		2515.5							3510			
	29				224	11.23		35			316	11.11
		2889.6							3160			
	33				257	11.24		31			285	11.09
		3315.3							2850			
	38				295	11.24		28			257	11.09
		3805.5							2570			
	44				339	11.23		25			232	11.08
		4373.1							2320			
	50				389	11.24		23			209	11.10
		5018.1							2090			

Table IV
ILLUSTRATION OF ALGORITHM 3 ($\mathcal{D} = 10, \mathcal{U} = 15, \mathcal{U}_{\min} = 1, \mathcal{A}_{\text{START}} = 30, \mathcal{R}_{\text{RPS}} = 1.1, \mathcal{T}_D = 10, \mathcal{T}_U = 14$)

with focus on non-functional properties (NPFs) of individual services in SOA. Important NFPs include performance metrics (for example, response time), security attributes, transactional integrity, reliability, scalability, and availability. Yigitbasi et al. presented a framework for generating and submitting test workloads to computing clouds [23]. The framework was used by the authors to assess the performance of Amazon's EC2 in terms of various metrics, such as the overhead of acquiring and releasing the virtual computing resources, and other virtualization and network communications overheads. Marshall et al. proposed a model of an "elastic site" that efficiently adapts services provided within a site, such as batch schedulers, storage archives, or Web services to take advantage of elastically provisioned resources in [24]. Other evaluations of cloud platforms include [25], [26], [27].

In [28], Schad et al. studied the performance variance on Amazon's EC2 using microbenchmarks and real data intensive applications. They showed that EC2 performance varies a lot and often falls into two bands having a large performance gap in-between and that the choice of availability zone also influences the performance variability. Iosup et al. analyzed the performance of cloud computing services for scientific computing workloads in [29]. Further, the authors present a comparison, through trace-based simulation, of the performance characteristics and cost models of clouds and other scientific computing platforms, for general and Many Task Computing (MTC)-based scientific computing workloads.

In [30], Weinman showed that that the advantaged of on-demand provisioning depend on the interplay of demand with forecasting, monitoring, and resource provisioning and de-provisioning processes and intervals, as well as likely asymmetries between excess capacity and unserved demand. Akin to the argument made by Weinman, one of the challenges we at Netflix face is how to determine the pool size of reserved instance and cater to different applications with different demand profiles to achieve benefits of statistical multiplexing, while managing aggregate capacity [31]. Recently, Caron [32] et al. and Islam et al. [33] presented techniques for forecasting resource requirements on the cloud. Niu et al. argue that a cloud tenant's utility depends not only on its bandwidth usage, but more importantly on the portion of its demand that is satisfied with a performance guarantee [34]. The authors addressed the problem of determining the optimal policy for pricing cloud bandwidth reservations, in order to maximize social welfare, i.e., the sum of the expected profits that can be made by all tenants and the cloud provider, even with the presence of demand uncertainty. In [35], Minarolli and Freisleben presented a a two-tier resource management approach based on adequate utility functions and consisting of local controllers that dynamically allocate CPU shares to virtual machines to maximize a local node utility function and a global controller that initiates live migrations of virtual machines to other physical nodes to maximize a global system utility function. Shen et al. employed online resource demand prediction and prediction error handling to achieve adaptive resource allocation without assuming any prior knowledge about the applications running inside the cloud [36]. Han et al. presented a fine-grain technique for scaling at the resource level itself (CPUs, memory, I/O, etc) in addition to VM-level scaling [37].

In [38] Islam et al. proposed elasticity as a property of a cloud platform with time and cost as its essential elements, and showed how to measure elasticity with respect to applications with certain Quality of Service (QoS) requirements. The authors do *not* propose mechanisms for minimizing resource usage. In [39], Villegas et al. presented empirical performance-cost analysis of provisioning and allocation policies in IaaS clouds. The various provisioning policies evaluated by the authors are different from the policies supported by Amazon's Autoscaling. We believe that studies of Islam et al. and Villegas et al. are complementary to the techniques proposed in this paper. In [40], Xu et al. presented

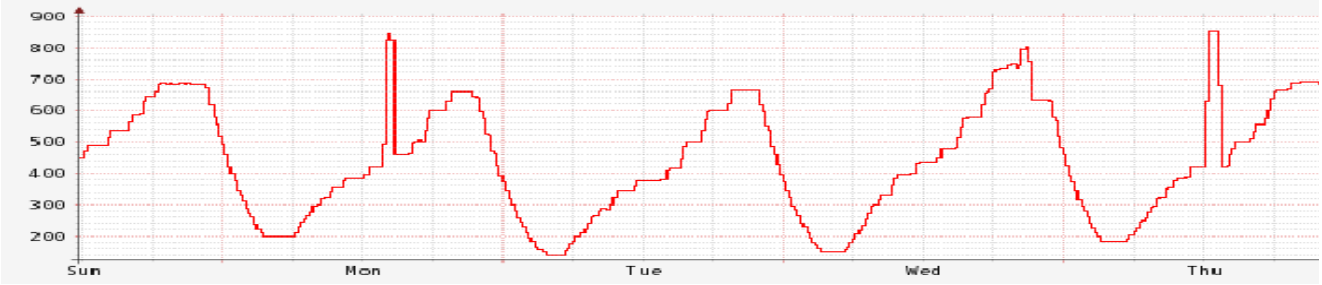


Figure 8. Illustration of spikes in load in production

a technique to adapt a VM's resource budget and application parameters to the cloud dynamics and the changing workload to provide service quality assurance.

Recently, Garrison et al. detailed the various factors for deploying cloud computing [5]. In particular, they highlight that trust between the client organization and the cloud provider is key for successful cloud deployment. Further, successful cloud deployment enable organizations to achieve greater IT economies of scale and focus on their core competencies that in turn result in competitive advantage.

There has been extensive work using control theory to address load balancing and to provide performance guarantees for Web servers. Abdelzaher et aldemonstrated how real-time scheduling and feedback-control theories can be leveraged to achieve response-time and throughput guarantees for Web servers [41]. Likewise, in [42], Abdelwahed and Kandasamy described a model-based control and optimization framework to design autonomic or self-managing computing systems that continually optimize their performance in response to the changing workload demands and operating conditions. In the context of real-time systems, Abdelwahed et astudied the feasibility of the predictive control policy for a given system model and performance specification under uncertain operating conditions [43]. To the best of our knowledge, none of the control theory-based techniques proposed earlier focus on the problem addressed in this paper. We believe that our technique is complementary to the above.

V. Conclusion

In this paper, we presented novel techniques to optimize operational efficiency on the cloud. Specifically, we presented three techniques targeted to different production scenarios. The techniques were deployed in production and resulted in up to 50% reduction in operational costs for the target Netflix applications.

At times we at Netflix observe spikes in incoming traffic. This can be happen due to a variety reasons. For instance, at the end of events such as the Superbowl we observe (as expected) a sudden rise in streaming traffic. An example traffic profile with spikes is shown in Figure 8. Going forward, we plan to extend the techniques proposed in this paper to handle such traffic spikes.

In a recent paper [44], Ford remarked the following regarding the reliability of cloud systems for SOAs:

> As diverse, independently developed cloud services share ever more fluidly and aggressively multiplexed hardware resource pools, unpredictable interactions between load-balancing and other reactive mechanisms could lead to dynamic instabilities or "meltdowns."

Our experience at Netflix has been consistent with the above as Netflix software architecture is a service-oriented architecture (SOA). Autoscaling a service down independent of the traffic upstream can potentially result in meltdowns. As future work, we plan to extend the proposed techniques earlier in this paper to capture the interaction between different services in a SOA. Outages in the cloud [45], [46], [47], [48], [49], [50], [51], [52], [53] and in data centers [54], [55], [56], [52] have been becoming increasingly more frequent. One of the ways to minimize the impact of outages is to extend the SOA to span multiple IaaS vendors. This would in turn call for extending the techniques proposed in this paper to be vendor aware.

References

[1] P. Mell and T. Grance, "The NIST definition of cloud computing, Special Publication 800-145," 2011, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.
[2] J. Feldman, "Cloud ROI Calculations," 2012, http://reports.informationweek.com/abstract/5/8702/Cloud-Computing/research-cloud-roi-calculations.html.
[3] "Worldwide and Regional Public IT Cloud Services 2012–2016 Forecast ," Aug. 2012, http://www.idc.com/getdoc.jsp?containerId=236552.
[4] A. Amies, H. Sluiman, Q. G. Tong, and G. N. Liu, *Developing and Hosting Applications on the Cloud*. IBM Press, 2012.
[5] G. Garrison, S. Kim, and R. L. Wakefield, "Success factors for deploying cloud computing," *Communications of the ACM*, vol. 55, no. 9, Sep. 2012.
[6] "Amazon Web Services," http://aws.amazon.com/.
[7] "Rackspace," http://www.rackspace.com.
[8] "GoGrid," http://gogrid.com.
[9] "Google Compute Engine," http://cloud.google.com/products/compute-engine.html.
[10] "Netflix," http://www.netflix.com/.
[11] J. B. Barney, "Firm resources and sustained competitive advantage," *Journal of Management*, vol. 17, no. 1, pp. 99–120, 1991.
[12] S. A. Alvarez and J. B. Barney, "Resource-based theory and the entrepreneurial firm," in *Creating A New Mindset: Integrating Strategy and Entrepreneurship Perspectives*, M. Hitt, R. Ireland, S. Camp, , and D. Sexton, Eds. John Wiley and Sons, 2002.
[13] G. Bhatt and V. Grover, "Types of information technology capabilities and their role in competitive advantage: An empirical study," *Journal of Management Information Systems*, vol. 22, no. 2, pp. 253–277, 2005.
[14] "Amazon EC2 Pricing," http://aws.amazon.com/ec2/pricing.
[15] "Autoscaling," http://aws.amazon.com/autoscaling.
[16] "AWS Auto Scaling Group," http://aws.amazon.com/autoscaling/.
[17] "Amazon CloudWatch," http://aws.amazon.com/cloudwatch/.
[18] "AWS: Policy," http://docs.amazonwebservices.com/AutoScaling/latest/DeveloperGuide/AS_Concepts.html#Policy.
[19] "AWS: Scaling by Policy," http://docs.amazonwebservices.com/AutoScaling/latest/DeveloperGuide/scaling_plan.html#scaling_policies.
[20] "Apache JMeter," http://jmeter.apache.org/.
[21] J. Dejun, G. Pierre, and C.-H. Chi, "EC2 performance analysis for resource provisioning of service-oriented applications," in *Proceedings of the 2009 International Conference on Service-Oriented Computing*, Stockholm, Sweden, 2009.

[22] V. Stantchev, "Performance evaluation of cloud computing offerings," in *Proceedings of the 2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*, 2009, pp. 187–192.

[23] N. Yigitbasi, A. Iosup, D. H. J. Epema, and S. Ostermann, "C-meter: A framework for performance analysis of computing clouds," in *nternational Symposium on Cluster Computing and the Grid*, Shanghai, China, May 2009, pp. 472–477.

[24] P. Marshall, K. Keahey, and T. Freeman, "Elastic site: Using clouds to elastically extend site resources," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 43–52.

[25] S. L. Garfinkel, "Commodity grid computing with amazon's s3 and ec2," *;login*, vol. 32, no. 1, pp. 7–13, 2007.

[26] ——, "An evaluation of amazons grid computing services: Ec2, s3 and sqs," Harvard University, Technical Report 08-97, 2007.

[27] E. Walker, "Benchmarking Amazon EC2 for High-Performance Scientific Computing," *;login*, vol. 33, no. 5, pp. 18–23, 2008.

[28] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of VLDB Endowment*, vol. 3, no. 1-2, Sep. 2010.

[29] A. Iosup, N. Yigitbasi, and D. H. J. Epema, "On the performance variability of production cloud services," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Newport Beach, CA, May 2011, pp. 104–113.

[30] J. Weinman, "Time is Money: The Value of "On-Demand"," Jan. 2011, http://www.joeweinman.com/Resources/Joe_Weinman_Time_Is_Money.pdf.

[31] ——, "The 10 Laws of Cloudonomics," 2008, http://gigaom.com/2008/09/07/the-10-laws-of-cloudonomics/.

[32] E. Caron, F. Desprez, and A. Muresan, "Forecasting for cloud computing on-demand resources based on pattern matching," INRIA, Technical Report, 2010.

[33] S. Islam, K. Lee, A. Fekete, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," in *Proceedings of Future Generation Computer Systems*, 2011, pp. 155–162.

[34] D. Niu, C. Feng, and B. Li, "Pricing cloud bandwidth reservations under demand uncertainty," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, London, England, 2012, pp. 151–162.

[35] D. Minarolli and B. Freisleben, "Utility-based resource allocation for virtual machines in cloud computing," in *IEEE Symposium onComputers and Communications*, July 2011, pp. 410 –417.

[36] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, pp. 5:1–5:14.

[37] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012, pp. 644–651.

[38] S. Islam, K. Lee, A. Fekete, and A. Liu, "How a consumer can measure elasticity for cloud platforms," in *Proceedings of the Third Joint WOSP/SIPEW International Conference on Performance Engineering*, Boston, MA, 2012, pp. 85–96.

[39] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 612–619.

[40] C.-Z. Xu, J. Rao, and X. Bu, "Url: A unified reinforcement learning approach for autonomic cloud management," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 95 – 105, 2012.

[41] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: a control-theoretical approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 1, pp. 80–96, jan 2002.

[42] S. Abdelwahed and N. Kandasamy, "A control-based approach to autonomic performance management in computing systems," in *Autonomic Computing: Concepts, Infrastructure, and Applications*, S. Hariri and M. Parashar, Eds. CRC Press, 2006, pp. 149–167.

[43] S. Abdelwahed, J. Bai, R. Su, and N. Kandasamy, "On the application of predictive control techniques for adaptive performance management of computing systems," *IEEE Transactions on Network and Service Management*, vol. 6, no. 4, pp. 212 –225, december 2009.

[44] B. Ford, "Icebergs in the Clouds: the Other Risks of Cloud Computing," http://arxiv.org/abs/1203.1979.

[45] "Amazon EC2 Outage Wipes Out Data," Oct. 2007, http://www.datacenterknowledge.com/archives/2007/10/02/amazon-ec2-outage-wipes-out-data/.

[46] "Major Outage for Amazon S3 and EC2," Feb. 2008, http://www.datacenterknowledge.com/archives/2008/02/15/major-outage-for-amazon-s3-and-ec2/.

[47] "Lightning Strike Triggers Amazon EC2 Outage," 2009, http://www.datacenterknowledge.com/archives/2009/06/11/lightning-strike-triggers-amazon-ec2-outage/.

[48] "Outage for Amazon Web Services," 2009, http://www.datacenterknowledge.com/archives/2009/07/19/outage-for-amazon-web-services/.

[49] "Brief Power Outage for Amazon Data Center," Dec. 2009, http://www.datacenterknowledge.com/archives/2009/12/10/power-outage-for-amazon-data-center/.

[50] "AWS Outage," 2011, http://aws.amazon.com/message/65648/.

[51] "AWS Outage," 2012, http://aws.amazon.com/message/67457/.

[52] "Twitter Is Down, Again," 2012, http://techcrunch.com/2012/07/26/communication-breakdown-twitter-is-down-again/.

[53] "List of web host service outages," http://en.wikipedia.org/wiki/List_of_web_host_service_outages.

[54] "Twitter Outage," 2011, http://status.twitter.com/post/2369720246/streaming-outage.

[55] "Twitter Outage," 2012, http://blog.twitter.com/2012/06/todays-turbulence-explained.html.

[56] "Google Talk Is Down: Worldwide Outage Since 6:50 AM EDT," 2012, http://techcrunch.com/2012/07/26/google-talk-is-down-worldwide-outage-since-650-am-edt.