Chapter 19
Scalability and Performance Management of Internet Applications in the Cloud

Wesam Dawoud
University of Potsdam, Germany

Ibrahim Takouna
University of Potsdam, Germany

Christoph Meinel
University of Potsdam, Germany

ABSTRACT

Elasticity and on-demand are significant characteristics that attract many customers to host their Internet applications in the cloud. They allow quick reacting to changing application needs by adding or releasing resources responding to the actual rather than to the projected demand. Nevertheless, neglecting the overhead of acquiring resources, which mainly is attributed to networking overhead, can result in periods of under-provisioning, leading to degrading the application performance. In this chapter, the authors study the possibility of mitigating the impact of resource provisioning overhead. They direct the study to an Infrastructure as a Service (IaaS) provisioning model where application scalability is the customer's responsibility. The research shows that understanding the application utilization models and a proper tuning of the scalability parameters can optimize the total cost and mitigate the impact of the overhead of acquiring resources on-demand.

1. INTRODUCTION

With virtually limitless on-demand resources, cloud offers a scalable and fault tolerant architectures that enable the hosted Internet application to cope with unpredicted spikes in workload. In the current cloud world, Software as a Service (SaaS) and Platform as a Service (PaaS) scalability is the provider responsibility. On the other hand, Infrastructure as a Service (IaaS), which offers more flexible environment, delegates the scalability management to the customers. In this chapter we dedicate our analysis to scalability of Infrastructure as a Service (IaaS) layer; therefore we define the entities that are involved in our study as follows:

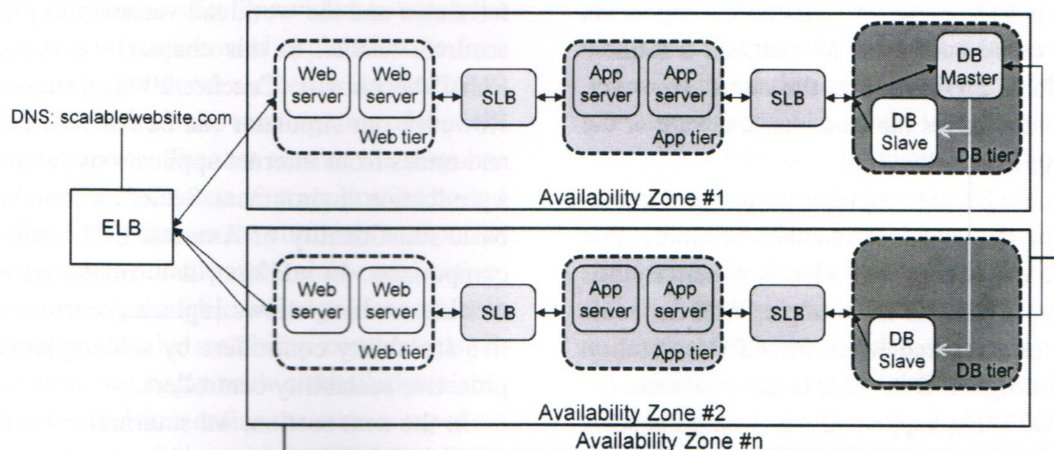
DOI: 10.4018/978-1-4666-4522-6.ch019

Copyright © 2014, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

- **Internet Application:** An Internet application, sometimes called an Internet service, is an application delivered to users from a server over the Internet (Urgaonkar, 2005). Examples of Web applications are wikis, discussion forums, blogs, online retails, and news Websites. The number of users of these applications varies according to the time of the day and rises sharply on occasions. For example, video games discussion forums endure a cyclical demand variation, depending on the day or night times hours, but experience high traffic with each release of a new game. Similarly, online retail faces a daily cyclical variation in demand and high spikes at special occasions such as Christmas. In fact, these applications existed even before the cloud emergence. However, the cloud infrastructure provides elastic environment for these applications to scale up and down according to workload variation.
- **Internet Application's User:** An Internet application's user is a user who interacts with the Internet application through a Web browser. The interaction may include browsing, submitting text, or uploading files.
- **Internet Application's Provider:** Typically, Internet application's providers are a company or organization that runs the Internet application for profit purposes, such as online retails, or non-profit purposes, such as Wikipedia. In this chapter, we assume that the Internet application is hosted in the cloud infrastructure; therefore, we refer to the owner as the cloud customer.
- **Cloud Provider:** A cloud provider is the company that offers the Infrastructure and the tools for the cloud customers to host and maintain the performance of their applications in the cloud (e.g., Amazon EC2).

Typically, Internet applications are implemented as a multi-tier architecture as seen in Figure 1. However, in some applications, the business logic (i.e., application tier) and the data representation (i.e., Web tier) are merged into one tier. In fact, multi-tier implementation enables simpler scalability for Internet applications especially for the Web tier and the application tier. The scalability of the database tier is not as simple as Web and application tier. However, it also can be scaled out into many instances using the Master/Slave architecture as we discuss in Section 3.2.

Figure 1. A detailed multi-tier scalable architecture



In multi-tier architecture, each tier provides a particular functionality, while the type of the incoming request determines the participating tiers in the request handling. As an example, a request of a static page can be handled by only a Web tier. On the other hand, a search for items in an online retail store will result in interactions between all tiers including the Web tier, application tier, and database tier (Urgaonkar, 2005). To cope with the incoming workload variation, the application at each tier may be replicated to many servers. To keep load balancing, the incoming workload will be distributed among replicas using a dispatcher. The emergence of pay-as-you-go concept in the cloud environment allows customers to dynamically specify the number of replicas that cope with workload demand while keeping the total cost to the minimum. To control the number of replicas, IaaS providers (e.g., Amazon EC2) offer the customers an online monitoring of specific metrics utilization (e.g., CPU, Memory, and Network). A simpler approach to provision adequate VMs at any time is to determine a static upper threshold (e.g., 70% CPU utilization) as a trigger for increasing the number of Virtual machines (VM) instances at high workload, and another static lower threshold (e.g., 30% CPU utilization) as a trigger for decreasing the number of VM instances at low workload.

In fact, as we will discuss in Section 2.2, current IaaS providers, especially Amazon EC2, provide a wide range of tools and components helping cloud customers to maintain a reliable and scalable application in the cloud. However, the provider leaves the customer unaware of the following facts:

First, the current implementation of the scalability in the cloud is reactive, meaning that the scale out is triggered when a predetermined condition is met. As an example, Web tier scale out can be triggered when the CPU utilization is 70% or higher. But, what is the best scale out threshold? What happens if we rise up the scale out threshold to 80%? Will this reduce cost? And what is the application performance for each scalability threshold?

Second, provisioning resources in the cloud do not happen instantly due to the technology and the virtual machine (VM) image size (i.e., network throughput). Accordingly, each provider can initialize VMs with different overhead (Mao, 2012). If the cloud customer did not consider the overhead of provisioning resources, the application can approach periods of performance degradation with each trigger for scale out.

Our contributions in this chapter are as follows:

- Study and compare the scalability implementation in some of well known IaaS providers.
- Define the scalability parameters that have crucial impact on an Internet application's performance.
- Analyze the impact of the network overhead on the scalable applications performance.
- Find the best values of scalability parameters that optimize the cost while maintaining a high performance.

For these purposes, we developed ScaleSim. It is a simulator built on top of CloudSim (Calheiros, 2011) to observe and optimize the Internet application scalability parameters at large-scale realistic environment. Our simulator is fed with realistic measurements of the application performance and the workload variation to provide accurate results. In this chapter, we simulated RUBiS benchmark (Cecchet, 2002) at large scale. However, our simulator can be fed with models and traces from Internet applications running in a production environment. ScaleSim contains the basic functionality of Amazon EC2 scalability components. In addition, it is implemented as modules, which allows replacing current reactive scalability controllers by self-implemented proactive scalability controllers.

In the next section, we summarize the basic concepts that face any researcher or engineer deal-

ing with the scalability in the cloud infrastructure. In Section 2.2 we define the main components of a scalable application. In Section 2.3, we shed the light on the scalability parameters that have a direct impact on Internet application's performance. Afterwards, in Section 3 we suggest a methodology for calculating performance thresholds of each tier. In Section 4, we implement a simulator on the light of the available scalability components in Amazon EC2 to observe and analyze Internet application scalability at large scale. In Section 5, we evaluate the current reactive scalability. In Section 6, we study the possibility of tuning the scalability parameters for lower cost and less Service Level Objectives (SLOs) violations. Finally, in Section 7, we conclude our work and point out to the future work.

2. INTERNET APPLICATION SCALABILITY

Dynamic scalability is not only crucial for IaaS customers, but also for PaaS and SaaS providers who host their systems into IaaS layer. In fact, IaaS providers offer architectures and components that ease scaling applications in IaaS. On the other hand, a non-trivial part of the task lies on the customer's side. Due to the vast number of the hosted applications into IaaS and the variant behaviors and demand of each application, customers should not expect IaaS providers to watch the performance of each hosted application. In fact, what an IaaS provider describes in Service Level Agreement (SLA) is the running time of the VM machine instance. For that reason, the response time and the performance of an application hosted in IaaS will stay forever the customer's responsibility. To have a reliable and scalable application, customers need enough knowledge of the scalable architecture and the scalability components. In this section, we begin with a detailed scalable architecture and then define the main component of the scalability followed by related scalability parameters that require customer's awareness.

2.1. Scalable Applications' Architecture

In addition to VMs instances as computation units, as seen in Figure 1, there are many auxiliary components that can be accessed as Web services. Consuming these services may include added cost. However, the offered services are manageable, reliable, and fault-tolerant compared to customers self-created components, as we explain in the next section.

2.2. Scalability Components

With the advance of cloud computing infrastructure, many services and concepts have emerged to ease and support scalable and reliable Internet applications. As a matter of fact, Amazon EC2 (Amazon EC2, 2012) is considered a pioneer in cloud computing infrastructures. Therefore, we explain the services and concepts that are related to application scalability in terms of Amazon EC2 but also map them to other providers' terms.

1. **Amazon Machine Image (AMI):** AMI is a pre-configured operating system image that can be used to create a virtual machine instance. Windows Azure (Windows Azure, 2012) as well as Amazon EC2 (Amazon EC2, 2012) allow the clients to upload their own image or select from a list of available images. Different providers and communities offer images with software stacks to deliver ease in running of their software. These images are stored in a non-volatile repository. Most of the IaaS providers also allow users to customize virtual machine (VM) images and create their own images as snapshots.
2. **Amazon Simple Storage Service (S3):** S3 is a simple Web service that provides a fault-tolerant and durable data storage. The data is stored as objects redundantly across different geographical regions for higher availability. The stored objects can be accessed by URL. S3 is optimal for storing static data that

will be delivered to users directly without manipulation. In addition, it is used to store virtual machine images. According to the used technology, the rate of data transfer varies from one provider to another. For example, at the time of writing this book, RackSpace (RackSpace, 2012) displayed a data transfer rate of (22.5 MB/s). This fact results in lower overhead for running a VM instance compared with Amazon EC2 and Windows Azure (Mao, 2012).

3. **Amazon Elastic Block Store (EBS)**: EBS is a block level storage volume that persists independently from the VM instance life. Unlike the local storage that can be lost after a failure or a planned termination of a VM instance, the EBS volume lasts permanently. Consequently, it is used for applications that need permanent storage like databases. At any time a VM instance fails, the EBS volume can be re-attached to another healthy VM instance. Despite the fact that the EBS volumes are stored redundantly, to reduce the recovery time from a failure, users can periodically take snapshots of these volumes. In addition, for high durability, the snapshots also could be stored in S3 storage. As an example, in Figure 1, a best practice is to map the Master database to EBS storage. Whenever the database instance fails, we can remap the storage volume to another instance and restore the database to operational mode quickly. To balance the workload on database tier a user can dump the database into S3 storage that can be used to initiate Slave instance, as we explain in details in Section 5.2.
4. **Regions and Availability Zones:** Cloud infrastructure is designed to offer a fast and reliable service globally. As a result, data centers of a cloud provider are distributed to span more geographical location areas (i.e., Regions). Within each Region, there are many Availability Zones that are engineered to be isolated from failures propagation. The networking between the Availability Zones within the same Region is inexpensive and induces a low networking latency. On the other hand, the networking between VM instances within Availability Zones located in different Regions implies networking through the Internet. As a result, even for reliability and fault-tolerance, the cloud customers are not advised to split application tiers into different Regions.
5. **Static Load Balancer (SLB):** Also referred to as Load Balancer (LB) or a dispatcher. Usually, it is a VM running a third-party software (e.g., HAProxy, Nginx, and Apache-proxy) to distribute workload across many back-end VM instances (i.e., replicas). The re-direction of the request to the back-end instances follows a specific algorithm. *Round robin* is a widely used algorithm in Load Balancers. In case of unequal size of back-end instances, *weighted round robin* can be used to direct a quantity of the traffic proportional to back-end instance capacity. In case of databases, especially when the majority of the requests dispatch read queries, a load balancer can be stood in front of the database tier. The database tier itself can be split into a Master database instance, and one or more Slave instances. The write quires are directed by the load balancer to the Master database, while the read queries are directed to the Slave database instances. Again, to be sure about the data consistency, the Master and Slave instances should not be located in different Regions to avoid high latency synchronization through the Internet. Nevertheless, to scale database into different Regions other techniques like database sharding (Curino, 2010; Agrawal, 2011) can be used.
6. **Elastic Load Balancing:** The challenge with the SLB is that they are in need to have up to date lists of the available healthy replicas

behind it. Whenever a replica fails or does not work properly, it should be excluded from the list to avoid losing or delaying the routed traffic to it. On the other hand, whenever a new replica is initiated it has a new IP address that is unknown to SLB. For the third party load balancers, it is the Internet application owner's job to mange registering and de-registering instances to the SLB. This implies running an additional component to interface with the load balancer and update the replicas list with each exclusion or addition of a replica. Moreover, SLB owners should run additional components to allow the balancer to distinguish between healthy and non-healthy replicas. Alternatively, Elastic Load Balancer of Amazon EC2 is supported with additional control component that keeps watching the status of the replicas. Whenever a VM instance does not respond properly, it is discarded from the replicas to prevent routing traffic to it. When the instance recovers to healthy mode, ELB can consider it in the possible replicas again. It is important to note that registering and de-registering instances to the Load Balancer are not part of the Elastic Load balancer job. They are done by what is called an Auto Scaling Group, which is explained next. On the contrary to ELB as a software load balancer, GoGrid offers a hardware load balancer that has a rich interface with many functions. Some of the distinguishing features of the GoGrid load balancer are the ability to have log files formats similar to apache style access log. Furthermore, it has an important feature called connection throttling, which allows the load balancer to accept only a pre-defined number of connections per an IP address. By this way, the load balancer can mitigate malicious or abusive traffic to Internet application.

7. **Auto Scaling Group:** It is a concept by Amazon EC2 which keeps a healthy group of instances running under a unique name. At the creation time of the group, the user can specify the minimum number of the healthy instances that should be available all the time. Whenever a VM instance does not work correctly, the group controller replaces it automatically with a new one. Connecting the auto scaling group with an ELB is necessary to provide the ELB an updated list of the available running replicas within the scaling group.
8. **Auto Scaling Policies:** Auto scaling policies should be attached to a specific scaling group. They describe how the scaling group should behave whenever it receives a scale out or down trigger.
9. **CloudWatch:** A Web service that enables monitoring various performance metrics, as well as configuring alarm actions based on the status of the monitored metrics. For example, the user can set up CloudWatch to send an email or trigger scalability when the CPU utilization of a database instance goes over 70%. More details can be seen in the example at the end of this section.
10. **Route 53:** In reality, ELB is limited to one region. As a result, Amazon offers Route 53 (Route 53, 2012) as a scalable and highly available Domain Name System (DNS). It allows scaling an Internet application globally for less latency and higher reliability. With Route 53, Internet application users can be directed to the closest region according to their geographical location. In this way, the users will be served from the closest datacenter. This allows a geographical distribution of the load and a high reduction of latency.

In Table 1, we summarize scalability components that are implemented by some of the significant public cloud providers.

As shown in Table 1, Amazon EC2 has all the components that are necessary for efficient scalability. For the other providers, third parties like RightScale (RightScale, 2012), open source management tools like Scalr (Scalr, 2012), or self implemented controllers are necessary to implement dynamic and automated scalability.

How to Configure Auto Scaling in Amazon EC2 IaaS?

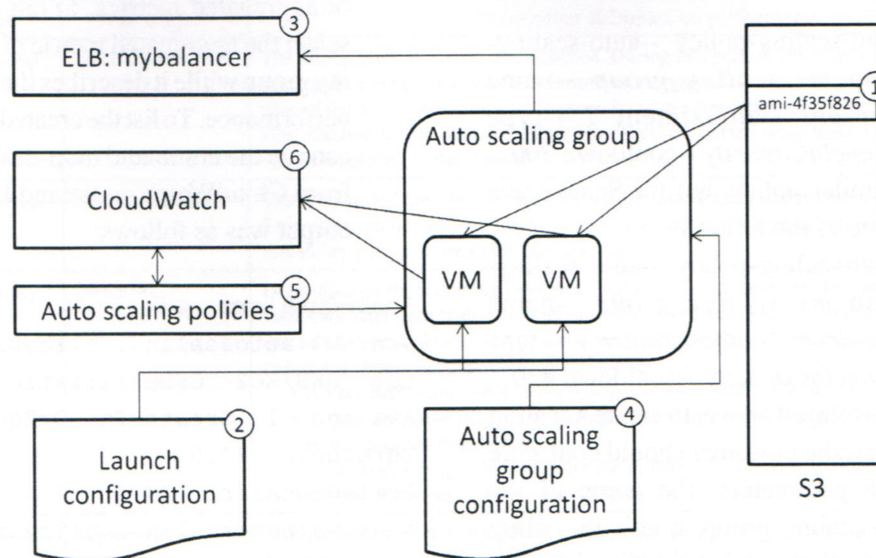
In the following example, we summarize how a cloud customer can enable the scalability to specific tier using the components offered by Amazon EC2. The purpose of this example is to express the parameters that are required to be determined by the customer and have high impact on the application performance. The same concepts are applicable to the other providers either by third party scaling systems or by the provider self-developed tools. In this example, we assume a Web tier that should maintain running at least two VMs and can scale up to fifteen instances of type *m1.small*. The group adds one instance per a scale out, and terminates one instance per a Scale down. The scale out is triggered when the aggregated CPU utilization of all the instances in the scalability group goes over 70%. On the other hand, the Scale down is triggered when the aggregated CPU utilization of all the instances in scalability group goes under 30%. The system will not scale out before five minutes of the last scale out, and will not Scale down before seven minutes of the last Scale down (see Figure 2).

1. **Prepare an Image to Run:** As described before, customers can create their own instance or pick one of the available images offered by the provider. For example, we consider running instances from the image ami-4f35f826, where the customer is supposed to install a Web server and configure it with the IP of the load balancer of the application tier. The customer can copy the html pages to the Web folder within the image. A more efficient practice in case of high frequently html code updating is to keep the code in external storage (e.g., S3) and retrieve it at the VM initializing time.
2. **Launch Configuration:** For auto scaling, the customer should pre-determine the launch configurations. It is important to mention that not all scalability configurations can be done through the Web dashboard. Until the time of this writing, to create a launch configuration, customers should have Amazon auto scaling command line package (AutoScaling, 2012). To create launch configuration, customers should determine a unique name of the configuration, a valid

Table 1. Scalability components of some of public IaaS providers

Amazon EC2	Windows Azure	Rackspace	GoGrid
AMI	Images	N/A	GoGrid Server Images (GSIs) and PartnerGSIs
ELB	N/A	Cloud Load Balancer	F5 Load Balancer (Hardware)
EBS	Windows Azure Drives	Only a local storage	N/A
S3	Azure Blob Storage	Rackspace (cloud files)	Cloud Storage
Regions and Availability zones	Regions but no Availability zones	Regions but no Availability zones	Regions but no Availability zones
Scalability Group	N/A	N/A	N/A
Scalability Policies	N/A	N/A	N/A

Figure 2. Main components of a dynamic scalable system in Amazon EC2



instance id, the type of the instance to be run, the name of the key pair, and the security group. An example of creating a launch configuration is as follows:

a. as-create-launch-config my_launch_conf_group --image-id ami-4f35f826 --instance-type m1.small --key my_key --group my_group --monitoring-enabled
b. The key and security group can be created using the dashboard. More details can be found in Amazon (Amazon EC2, 2012).

3. **Running a Load Balancer:** If a user decided to run ELB, the CNAME of the Internet application should point to the DNS name of the ELB not the IP. It appears that, Amazon EC2 does not dedicate a public IP for each ELB. In our example, we consider running an ELB called *mybalancer*. It is necessary to determine both the incoming port of the ELB and the forward port that the replicas are waiting on. The elastic load balancer also should be configured with metrics that help it to abandon non-healthy replicas depending on predetermined criteria. Running and managing an ELB can be done either by command line (ELB APIs, 2012) or through the dashboard.
4. **Auto Scaling Group Configuration:** To create a scaling group, customers should determine a unique name for it, a launch configuration, an availability zone, a minimum number of instances, a maximum number of instances, and a grace period in seconds. The purpose of the grace period is to give the system time to stabilize after each initialization of a VM instance within the group. The command of creating a scaling group can be as follows:
   a. as-create-auto-scaling-group *my_scaling_group* --launch-configuration *my_launch_conf_group* --availability-zones *us-east-1a* --min-size *2* --max-size *15* --load-balancers *mybalancer* --health-check-type *ELB* --grace-period *120*
5. **Auto Scaling Policies:** In our example, to create a scale out policy that should be triggered by the CloudWatch whenever a specific

condition is fulfilled, we run the following command:

a. as-put-scaling-policy --auto-scaling-group *my_scaling_group* --name *scale-out* --adjustment *1* --type *ChangeInCapacity* --cooldown *300*
b. A similar policy, but for Scale down can be as the follows:
c. as-put-scaling-policy --auto-scaling-group *my_scaling_group* --name *scale-down* "--adjustment=-1" --type *ChangeInCapacity* --cooldown *420*
d. As displayed above, to create a scaling policy, the customer should configure these parameters: the name of the auto scaling group, a unique scaling policy name, the size of the scaling step, the type, and the cooldown time in seconds. The positive scaling step (i.e., *adjustment*) means adding the specified number of instances to the scaling group, while negative adjustment means removing the specified number from the scaling group. Another important parameter is the cooldown time. It describes how many seconds the auto scaling group should wait after each scaling before going into another scaling. The cooldown time is used to give the scaling group a time to stabilize after triggering any scaling policy. More details about the command parameters can be found in AutoScaling, (2012).

6. **CloudWatch:** Provides monitoring service allowing customers to watch their application performance and react immediately for workload variation. To trigger scaling policies, CloudWatch should be configured either with CloudWatch command line (CloudWatch Command Line, 2012) or through the Web interface. Amazon offers an easy Web interface that enables creating metric alarms. There are many metrics to monitor, including single instance metrics or aggregated metrics. In our example, we select the aggregated metric of an auto scaling group while it describes the whole group performance. To list the created metric a user can run the command mon-describe-alarms from CloudWatch command line tool. The output was as follows:

```
ScaleOutAlarm  OK
arn:aws:autoscalin...olicyName/scale-
out  AWS/EC2  CPUUtilization  300
Average  1  GreaterThanOrEqualTo-
Threshold  70.0
ScaleDownAlarm  OK
arn:aws:autoscalin...olicyName/scale-
down  AWS/EC2  CPUUtilization  420
Average  1  LowerThanOrEqualToThresh-
old  30.0
```

Currently, CloudWatch provides a free mode where the metrics are measured at five-minute frequency. From our experience, free mode is not efficient for those applications that have frequent changes in the workload. The other choice offers more frequent measurement (i.e., one-minute frequency) by setting what is called a detailed monitoring of an instance; however it is charged monthly per an instance. Furthermore, for both modes, customers will be charged monthly per alarm and per thousand API requests. More details about the CloudWatch can be found in CloudWatch, (2012).

2.3. Scalability Parameters

To summarize, in Table 2, we list the parameters that are required to be set by the customer and have crucial impact on the dynamic scalable application performance.

It is clear that cloud providers have many efficient components that help the customer to build a scalable application. Nevertheless due to the fact that provider cannot understand each hosted

Table 2. The scalability parameters that have most impact on scalability performance

Component	Parameter	Description & Impact on performance
Auto Scaling Group	grace-period	The period after an instance is launched. During this period, any health check failure of that instance is ignored. In our experiments we noticed that a value greater than the booting time of the VM instance (e.g., 120 seconds) works fine while a very high value causes a slow reaction to VM instance failure after first run. On the other hand, a very low value causes instability to the system.
Auto Scaling Group	default-cooldown	The time period that should pass after a successful scaling activity to consider a new one. This value can be determined globally per scalability group or individually per each scalability policy, as we did in our example.
Auto Scaling Policy	cooldown	Depending on the incoming workload fluctuation, customers can determine the best cooldown after each scale either up or down. In our example, we set this value to be 300 seconds for the scale up but 420 seconds for the Scale down. While setting these values, customers should keep in mind the following: • A very high value causes slow reaction to workload variation. • A very low value may cause adding or removing many instances quickly which will result in an instable system • In general, wrong values increase the probability of having periods of over-provisioning or under-provisioning.
Auto Scaling Policy	adjustment	This parameter determines the size of the scaling step. The positive values means scaling out, while negative values means scaling down. In Section 6, we study the impact of the size of the step on both the cost and performance.
CloudWatch	metric-name	The name of the metric to be watched. Depending on the application, a customer should determine the metric that has the most impact on application performance. In Section 5, we concentrate on the CPU utilization and analysis how it can impact the application performance.
CloudWatch	threshold	The threshold which the metric value will be compared. Each application has its specific performance thresholds. In Section 3, we study the impact of these values on application performance and explain the practical way to determine these thresholds.
CloudWatch	period	Number of consecutive periods for which the value of the metric needs to be compared to the threshold.
CloudWatch	evaluation-periods	Number of consecutive periods for which the value of the metric needs to be compared to threshold. The multiplication of period by evaluation-periods should be higher than or equal to *cooldown* value at Auto Scaling Policy. In other words, it is meaningless to have very frequent triggers for a scalability group while it scales one time per five minutes.

application requirements and demand variation they leave tuning the scalability parameters to the customers. Any misconfiguration due to the lack of knowledge can have an impact on the application performance and may not achieve cost savings expected by moving to the cloud infrastructure. In the rest of this chapter, we develop the methodology, the environment, and the tools that allow both researchers and cloud customers to investigate directly the Internet applications scalability parameters.

3. MODELING AN INTERNET APPLICATION

Understanding the application model is crucial for maintaining the Internet application performance by avoiding bottlenecks in system performance. What makes modeling an Internet application behavior complex is fact that each tier in the Internet application runs different software that has different requirements. The dependency between the Internet application tiers propagates the degradation in performance of one tier to the other tiers (Iqbal, 2010; Urgaonkar, 2005a; Zhang, 2007). As an example, a database server is known to be an I/O intensive application that requires a big RAM.

At any time the allocated RAM exceeds 90%, the operating system starts paging to the virtual memory allocated at hard disk. This swapping results in more I/O operations and consumes much of the CPU time, which consequently, degrades the whole Internet application performance dramatically. The solution is to keep the memory allocation less than 90% as a threshold either by scaling vertically as described by Heo et al. (2009) and Dawoud et al. (2011) or horizontally by determining a scale out threshold (e.g., 90%) for all tiers considering the memory as metric.

Actually, the memory model is relatively simple compared to CPU model where the response time is not a function of the CPU utilization only but also the incoming requests rate (Heo, 2009). To understand the CPU model of software, response time should be examined with different request rates. In our research, we selected RUBiS benchmark (Cecchet, 2002) as an Internet application. It is an online auction site developed at Rice University to model basic functions of ebay.com system. We selected the RUBiS implementation which consists of Apache as a Web server, Tomcat as an application server, and MySql as a database.

3.1. State of the Art

Towards avoiding bottlenecks in multi-tier systems, Iqbal et al. (2010) implemented a prototype using multi-instances scaling architecture. This approach considers scaling database layer horizontally, but it did not discuss associated challenges (e.g., data replication and synchronization). Using analytical models to describe different tiers behavior, Urgaonkar et al. (2005b) presented a multi-tier model based on a network of queues, while each queue represents a different tier. The scalability of this model is implemented by dispatching new instances at each tier except database tier which is not replicable in their model. In fact, implementing Urgaonkar's approach in a production environment is challenging, as it requires monitoring low level metrics. The more efficient modeling for applications is the black box models (Dawoud, 2012; Iqbal, 2011). The black box models can be less accurate, but more efficient to implement in production environments. Using regression analysis of CPU utilization and service time to predict the bottlenecks, Dubey et al. (2009) demonstrated an approach for performance modeling of two-tier applications (Web and database). Even though the approach does not imply dynamic scaling, it aids in understanding application behavior for optimum capacity planning. Using queuing theory models along with optimization techniques, Jung et al. (2008) presented off-line techniques to predict system behavior and automatically generate optimal system configurations. Nevertheless, the authors considered scaling resources vertically, which limits a VM scaling into one physical host.

3.2. Physical Setup

To reduce the experiment budget, we setup both the workload generator and the load balancer inside Amazon EC2 infrastructure. Both the Web and the application are run on instances bundled to Amazon S3. Write only database (i.e., Master database) is created from an instances mapped to EBS storage for permanent storage, while read only a database (i.e., Slave database) is created from a bundled image stored at Amazon S3. The type of images was *Small instance (m1.small)* for the Web, the application, the slave databases, and the load balancers. For master database and load generator we run *Medium instances (m1.medium)*. To avoid the other tiers' impact on the tier under analysis, we create many replicas in the other tiers that keep the CPU utilization around 30%. As an example, to model CPU utilization of Web tier, we run four instances of application tier and two instances of slave database.

The generated workload is step traffic that increases the number of simultaneous clients gradually. In our experiments we consider the 95th percentile of transaction response times, which means that 95% of the measured response times of all requests is less than or equal to a given value (e.g., 95% of the requests is less than 100 milliseconds).

3.3. Web, Application, and Database Tier Thresholds

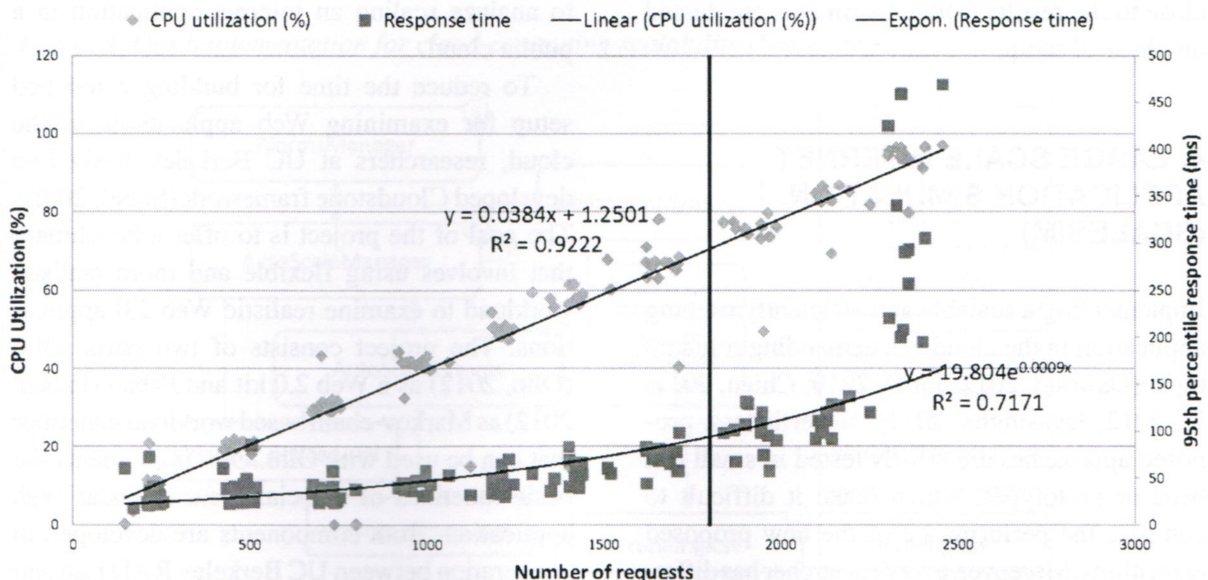
As the number of requests increases, the CPU utilization also increases. The relation between the number of requests and the CPU utilization is linear most of the run time, as seen in Figure 3. On the other hand, the response time increases exponentially with the CPU utilization. At some high values of the CPU utilization, the response time increases dramatically while the requests spend long time in the queue waiting for processing. Our goal of this analysis is to determine the CPU threshold that keeps the response time within a specific limit. In our system, we consider 100 ms as a higher limit of response time, while a response time around this value gives the user the feeling that the system is reacting instantaneously (Nielsen, 1993). Figure 3 demonstrates the following: to keep the response time of 95% of the requests less than or equal to 100 ms, the CPU utilization of each instance at Web tier should be less than or equal to 70%.

Repeating the experiment with the same workload and different number of instances, one application instance and four Web instances shows the following: to keep the response time of 95% of the requests less than or equal to 100 ms the CPU utilization of each instance at application tier should be less than or equal to 62%.

For the same setup but with five instances at Web tier and four at application tier, the result shows the following: to keep the response time of 95% of the requests less than or equal to 100 ms the CPU utilization of each instance of read only databases should be less than or equal to 72%.

Our experiment shows that each tier has a different performance threshold. If the customer failed to determine the right threshold for each tier, the whole Internet application performance is exposed to degradation. As an example, in our setup, the auction Website owner (i.e., RUBiS) has a Service Level Objective (SLO) to keep the response time of 95% of the requests less than 100 milliseconds. In this case, the periods of time at which the CPU utilization of application tier are higher than 62% but less than 70% (i.e., High, but not high enough to scale out) will result in SLO violation even though the other tiers' utilization is still under the threshold. In Section 6, we discuss

Figure 3. Web tier performance thresholds



the possibility of tuning the scale out threshold for lower cost and SLO violation.

3.4. ARX Model Extractions

To have online measurements of the instance under analysis, we built a java client that continuously monitors basic metrics: CPU utilization, Memory allocation, Network IN/OUT rate, and Disk read/write rate. In addition to monitored metrics, we enabled the load balancer to log URL requests. The monitored metrics are synchronized with the log file. For each monitoring interval (i.e., one minute in our case) we count the number of requests of each type. As we have done in (Dawoud, 2012), we consider 18 types of requests in RUBiS benchmark depending on the URL. The vector of requests rate is the input for our MISO Autoregressive model with eXogenous inputs (ARX), while the output is the modeled metric (e.g., CPU of the Web tier). Extracted models for each resource are necessary to run more realistic simulation of Internet applications in a large scale environment. To have an accurate model, we had only samples from areas showing no spikes in response time. The fitness of Web and application tiers models was higher than 94%, as explained in (Dawoud, 2012). The high fitness of models leads to simulation results close to the results from experiments conducted on physical setup.

4. LARGE SCALE INTERNET APPLICATION SIMULATION (SCALESIM)

Implementing a scalable and efficiently running application in the cloud is a demanding research topic (Dawoud, 2012; Iqbal, 2010; Chieu, 2011; Li, 2012; Jayasinghe, 2011). Nevertheless, proposed approaches are mostly tested in small test beds or prototypes, which make it difficult to compare the performance of the new proposed algorithms. Moreover, every researcher has different experimental setup with certain assumptions and specific workload. To ease the comparison and offer an environment for large scale running of Internet application in the cloud, we developed (ScaleSim). It is a simulator built on top of the CloudSim framework (Calheiros, 2011). We build it to examine the current implementation of the scalability in production environments (e.g., Amazon EC2). The simulator is built into components that can be customized by other researchers to compare their algorithms with the current running algorithms in a production environment and also with the other researchers' scalability algorithms.

4.1. State of the Art

CloudSim (Calheiros, 2011) is a framework for modeling and simulating cloud computing infrastructures and services. It allows scalable, repeatable, and fast evaluation of the new developed algorithms and polices before implementing them to a production environment. CloudSim supports modeling and simulation of federated cloud, data center network topologies, energy aware computation, and most importantly it supports user defined polices for provisioning virtual machines. Therefore, it was the best framework for us to implement our components and algorithms to analyze scaling an Internet application in a public cloud.

To reduce the time for building a test-bed setup for examining Web applications in the cloud, researchers at UC Berkeley RAD Lab developed Cloudstone framework (Sobel, 2008). The goal of the project is to offer a benchmark that involves using flexible and more realistic workload to examine realistic Web 2.0 applications. The project consists of two parts. Olio (Olio, 2012) as a Web 2.0 kit and Faban (Faban, 2012) as Markov-chain based workload generator that can be used with Olio. Olio implements the basic functions of a social-event calendar Web application. Both components are developed in cooperation between UC Berkeley RAD Lab and

Sun Microsystems Inc. The framework is scalable at the real infrastructure. However, it can be costly to have large scale experiments at physical infrastructure. Nevertheless, it is very useful to use it to get real measurements that can be fed to our simulator for large scale simulation.

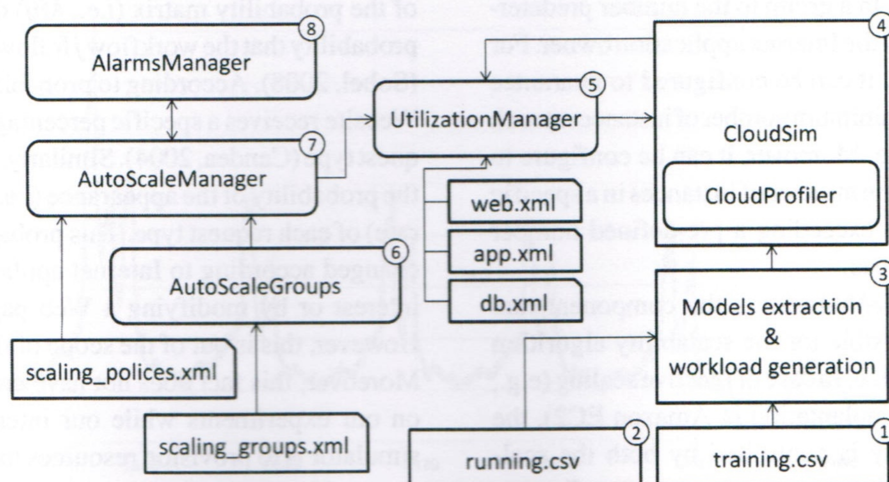
Another developed benchmark to examine Wikipedia similar Web applications is WikiBench (Van Baaren, 2009). It is a trace based benchmark able to create thousands of requests per a second. The traces are realistic while they are anonymous real traces from Wikimedia foundation. The benchmark has the ability to control the traffic intensity without affecting the traffic properties like inter-arrival time and distribution of page popularity. Again, this benchmark, as same as RUBiS benchmark that is used in this research, is designed to run in physical environment. We are looking forward to taking these measurements at physical environment to our simulator for more experiments on differ application like Wikipedia.

4.2. Developed Simulator (ScaleSim)

In our simulator, we implemented the functionality of the main components of scalability in the cloud. To make configuring the system more flexible, we depend on meta-data files (i.e., xml files) for configuration. With each new run, components fetch the attached configuration files. Figure 4 explains in details the components of our developed simulator (ScaleSim) and the interaction between the components as follows:

1. The training file (e.g., *training.csv*) contains real measurements of an application in a physical environment. The measurements consist of the rate of each considered URL request and the utilization of the monitored resources. The file is built as described in Section 3.4. However, same procedures can be implemented to any other Internet application.
2. The running file (e.g., *running.csv*) contains an artificial generated workload. To have more realistic results, the workload should be generated to mimic the real behavior of Internet applications. We explain in details the workload generation for our experiment in Section 4.3.
3. Models extraction and workload generation module do two tasks: first, it read the training files to extract models. Second, it calculates the consumption of resources. The expected

Figure 4. Our implementation for cloud computing scalability (ScaleSim)



consumption of the resources is passed to *CloudProfiler* object. It is an object resides in CloudSim to build the datacenters and the brokers that manage the coming workload.

4. In CloudSim we implemented a simple Datacenter to avoid internal optimization of resources (e.g., VM migration) that might influence our simulation. At the start of the simulation, a new object of *UtilizationManager* is created.
5. Whenever *UtilizationManager* is started, it creates an object of *AutoScaleManager* class. In fact, *UtilizationManager* is considered as an actuator for the scalability commands, which are received from *AutoScaleManager*. *UtilizationManager* has a direct monitoring of resources in CloudSim environment. It passes these measurements to *AutoScaleManager that* decides about scaling out or down to cope with incoming workload. Usually, starting a new VM commands are passed to CloudSim including the profile of the VM image to be started. In our simulator, the VM profile also includes the total required time to put a VM to operational mode. More details will be shown in Section 5.1.
6. *AutoScaleGroups* is an object implementing the same concept of the scalability groups in Amazon EC2. It maintains the number of the VMs in a group to the number predetermined by the Internet application owner. For example, it can be configured to guarantee that the minimum number of instances at each tier is one. Moreover, it can be configure to prevent the number of instances in a specific tier from exceeding a pre-defined number of instances.
7. *AutoScaleManager* is the component that is responsible for the scalability algorithm intelligence. In case of reactive scaling (e.g., current implantation in Amazon EC2), the scalability is controlled by both the scaling policies as input from the application owners (i.e., *scaling_polices.xml*) and the *AutoScaleGroups*. To employ proactive scaling algorithm, *AutoScaleManager* can be developed to consider historical measurements for coming workload prediction.
8. *AlarmsManager* is a queue receiving a stream of alerts. The alerts are initiated at *AutoScaleManager* whenever the utilization matches any of scaling polices. Each alert contains attributes (e.g., timestamp, scaling group, scale direction, and evaluation periods) that help the *AlarmsManager* manager to group the alerts and pass the scalability decision to *AutoScaleManager* at the proper time.

4.3. Workload Generation

To have real measurements, our experiment is conducted on physical environment to extract the application models. At the simulation environment, the input to the models is the rate of each URL request type. So, to simulate a realistic and large scale running of Internet applications, we need to preserve the real user behavior (e.g., flow pattern and thinking time), otherwise results can be inconsistent (Menascé, 2002). RUBiS benchmark, as same as Faban workload generator, has flow probability matrix *M* which is *NxN* matrix describes *N* states of the system. Each element of the probability matrix (i.e., *Mij*) describes the probability that the workflow *j* follows workflow *i* (Sobel, 2008). According to probability matrix, a Website receives a specific percentage of each request type (Candea, 2004). Similarly, we calculate the probability of the appearance (i.e. the requests rate) of each request type. This probability can be changed according to Internet application users' interest or by modifying a Web page contents. However, this is out of the scope of this research. Moreover, this fact does not have crucial impact on our experiments while our intention in this simulator is to provision resources that cope with the workload in each tier.

To mimic a realistic arrival rate of users and workload variation, we used the world cup 1998 workload (Arlitt, 1999) traces. They are apache log style traces of 1.35 billion requests initiated to world cup 1998 official Website over three months period. Even, they are traces of different applications; they have a real arrival rate of requests that can be mapped to RUBiS bench mark requests' rate. For each period of time (i.e., one minute in our case) we multiply the number of requests by the probability of each of the RUBiS benchmark requests. The result is the rate of each considered request of RUBiS benchmark (i.e., 18 requests in our case), which is stored in *running.csv* file, as explained in Figure 4. The rates vector, for each time window, is used to calculate the consumption of resources at each tier (see Figure 5).

As expected, using world cup request rates causes over-utilization of a single VM. Our simulator, depending on the scalability policies, finds the best number of VMs that should run in each tier to cope with the coming workload. This is what we study in details in next section.

5. DYNAMIC SCALABILITY

Elasticity is one of the great characteristics that attract many customers to move into the cloud infrastructure. It enables dynamic scalability where cloud customers can acquire more VM instances dynamically to handle the workload surges. Actually, there is a delay between initiating a request for a VM until having it ready. In this section, we study the source of these delays and their impact on the current reactive implementation of the scalability in the cloud.

5.1. Initializing a VM in the Cloud

In this section, we explain the stages of running a VM in the cloud. The goal is to point out the sources of overhead in running the VM instance. Networking overhead is the main source of the delay in running a VM instance. As an example, Mao (2012) shows that running a Linux instance at Rackspace takes half the time to run the same size instance at Amazon EC2. It is understandable when we know the data transfer rate between a VM and Image Store is 22.5 MB/s at Rackspace, compared to 10.9 MB/s at Amazon EC2.

Figure 5. Requests' rate to world cup 1998 official website for one week started at June 15th, 1998