Enhancing Amazon Web Services Auto Scaling

Abhinav Mishra (109838041)

Naman Mittal (109888028)

Shashi Ranjan (109974495)

CSE 691: ENERGY EFFICIENT COMPUTING

FALL - 2014

Under the guidance of

Professor Anshul Gandhi

Department of Computer Science

State University of New York, Stony Brook

Contents

1. **Introduction** **1**

2. **Background** **3**

3. **Experimental Setup and Proposed Approach** **6**

   3.1 Load Balancer's Latency Based Auto Scaling 6
   3.2 System Temperature Based Auto Scaling.. 7
   3.3 Thread Count Based Auto Scaling 8

4. **Experimental Results** **9**

   4.1 Load Balancer's Latency Based Auto Scaling 9
   4.2 CPU Temperature Based Auto Scaling 10
   4.3 Thread Count Based Auto Scaling 11

5. **Prior Work** **12**

6. **Conclusion** **13**

References **14**

Chapter 1

Introduction

Cloud computing has emerged as one of the most discussed and adopted IT paradigm in recent years. Cloud computing enables organizations to exploit flexible, secure, cost effective IT infrastructure, freeing them from the complexity of establishing, operating and improving the IT infrastructure on their own. Liberated from the intricacies of managing an underlying IT infrastructure, organizations can concentrate on core competencies of their business and needs of their customers. Adoption of cloud computing by key online service providers such as Netflix [1], Pinterest [2], Code.Org [3], Coursera [4], Expedia[5] etc. corroborate the huge acceptance of cloud computing by industry in various business domains.

This wide acceptance of cloud computing led into the formation of a new market which provides IT infrastructure services. One key service provider which started in 2006 and has transformed into a market leader is Amazon Web Service (AWS). Since its launch in 2006 AWS has evolved tremendously and now holds 21 amazing and high utility products under its belt. Two of the key characteristics of AWS is elasticity and scalability. In conventional IT infrastructure, scalability and elasticity equate to investment and infrastructure respectively. However in cloud infrastructure, elasticity and scalability are means to achieve savings and improved return of investment (ROI). AWS provides elasticity and scalability through Elastic Load Balancing and Auto Scaling. These two key aspect of AWS helps clients avoid provisioning the resources upfront for products or services with variable consumption rate or short life span. For instance, when a pharmaceutical company needs to run drug simulations (a short-term job), it can use AWS to spin up resources in the cloud, and then shut them down when it no longer needs additional resources [6]. In this manner, AWS allows clients to follow the application demand curve closely, obviating the need to manually provision the AWS resource capacity a priori.

AWS auto scaling employs various CPU based, I/O (disk read/write) based, network based and virtual instance health check based metric to scale up resources to meet unprecedented demand and then to scale down the resources as demand decreases [7]. This metric list is exhaustive and encompasses almost every aspect pertinent to AWS operation. For instance, CPU based metrics such as CPUUtilization covers processing power required by running application or network based metrics such as NetworkIn and NetworkOut cover data transfer on all network interfaces of a virtual instance.

Even though this metric list is exhaustive. Auto scaling configuration policy of AWS can be further improved by introducing some novel metrics which can capture some crucial aspects of AWS operation which remain uncaptured by existing AWS metrics. In this work, we have introduced two addition auto scaling metrics- system temperature and thread count, which can be utilized to perform auto scaling. System temperature and thread count are two important indicator of systems overall performance. System temperature is directly proportional to CPU load, memory load and network load. If the system temperature persistently crosses a certain threshold, over the course of operation it may result into components failure or wear and tear of certain components. Additionally, if the CPU temperature goes beyond a certain threshold, it reduces the clock frequency automatically to mitigate the temperature rise and reduction in clock frequency affects the request processing time. Thread count is another performance metric which can be used to execute auto scaling. Application servers are configured to spawn a fix number of thread. In the face of very high load server may reach the permissible thread count limit and will not be able to spawn new thread to process incoming requests. Such scenario would result into request drop and performance degradation. Additionally, when multiple threads on CPU,

CPU scheduler keeps on switching the threads based on remaining time slices for each threads which is an overhead for CPU. These issues can be obviated if auto scaling is performed when a server starts hitting the permissible thread count threshold.

We provided three contributions in this work. In first contribution, we utilized system temperature as auto scaling metric and, proposed and implemented an architecture in which a physical server's load can be delegated to AWS Elastic Compute Cloud (EC2) instances when physical server's system temperature reaches a certain threshold for a given time. In this way we can extend the physical server capacity as needed to stabilize the system performance with increasing load by using available cloud capacity. In our second contribution, we used thread count as auto scaling metric on EC2 instances. We successfully demonstrated thread count based auto scaling with EC2 instances. Our third contribution was the demonstration of auto scaling based on load balancer's latency. Most of the prior work considered the possibility of auto scaling based on application server's request processing latency. Contrary to previous work, we were able to trace the correspondence between load balancer's aggregated latency and incoming request and were able to perform auto scaling successfully based on this metric.

Chapter 2

Background

In this chapter, we would provide a brief description of Amazon Web Service (AWS) architecture and its key components which were used in this work [6].

- **Amazon Web Services (AWS)**
  AWS is a comprehensive cloud services platform that offers compute power, storage, content delivery, and other functionality that organizations can use to deploy applications and services cost-effectively—with flexibility, scalability, and reliability. AWS self-service means that you can proactively address your internal plans and react to external demands when you choose. Figure 2.1 below illustrate the key services provided by AWS across computation, storage and data base related cloud computing services.

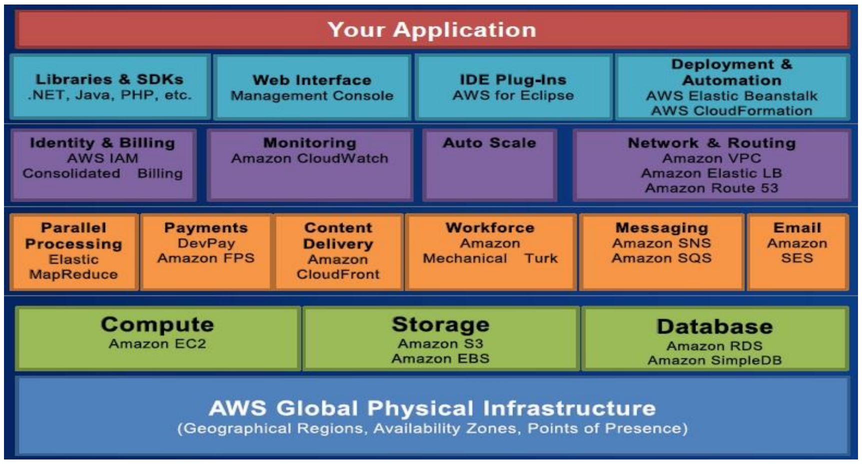
Your Application
Libraries & SDKs .NET, Java, PHP, etc.
Web Interface Management Console
IDE Plug-Ins AWS for Eclipse
Deployment & Automation AWS Elastic Beanstalk AWS CloudFormation
Identity & Billing AWS IAM Consolidated Billing
Monitoring Amazon CloudWatch
Auto Scale
Network & Routing Amazon VPC Amazon Elastic LB Amazon Route 53
Parallel Processing Elastic MapReduce
Payments DevPay Amazon FPS
Content Delivery Amazon CloudFront
Workforce Amazon Mechanical Turk
Messaging Amazon SNS Amazon SQS
Email Amazon SES
Compute Amazon EC2
Storage Amazon S3 Amazon EBS
Database Amazon RDS Amazon SimpleDB
AWS Global Physical Infrastructure
(Geographical Regions, Availability Zones, Points of Presence)

Figure 2.1: AWS Global Physical Infrastructure

- **Amazon Elastic Compute Cloud (Amazon EC2)**
  Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers and system administrators. Amazon EC2's simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon's proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. Amazon EC2 provides developers and system administrators the tools to build failure resilient applications and isolate themselves from common failure scenarios.

- **Auto CloudWatch**
  Amazon CloudWatch provides monitoring for AWS cloud resources and the applications customers run on AWS. Developers and system administrators can use it to collect and track metrics, gain insight, and react immediately to keep their applications and businesses running smoothly. Amazon CloudWatch monitors AWS resources such as Amazon EC2 and Amazon RDS DB Instances, and can also monitor custom metrics generated by a customer's applications and services. With Amazon CloudWatch, you gain system-wide visibility into resource utilization, application performance, and operational health. Amazon CloudWatch provides a reliable, scalable,

and flexible monitoring solution that you can start using within minutes. You no longer need to set up, manage, or scale your own monitoring systems and infrastructure. Using Amazon CloudWatch, you can easily monitor as much or as little metric data as you need. Amazon CloudWatch lets you programmatically retrieve your monitoring data, view graphs, and set alarms to help you troubleshoot, spot trends, and take automated action based on the state of your cloud environment.

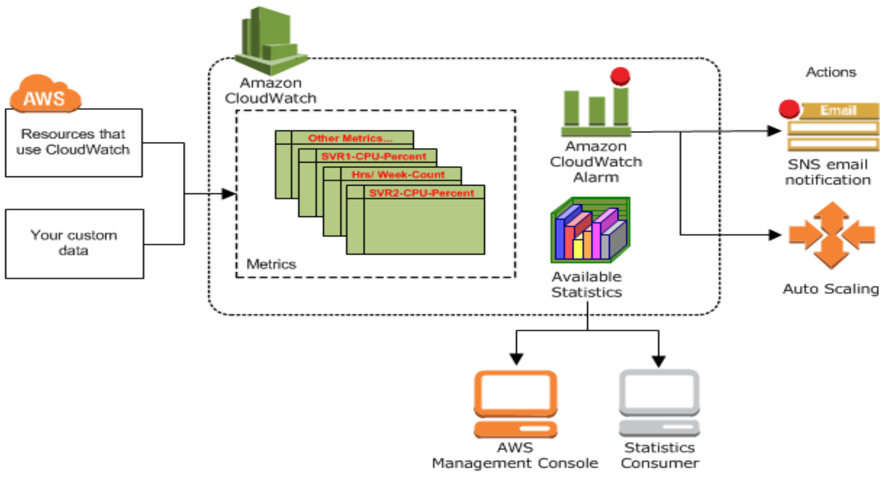

Figure 2.2: Amazon CloudWatch

As shown in Figure 2.2, metrics from the CloudWatch can be consumed to calculate statistics and present data graphically in the CloudWatch console. Alarms can be configured stop, start or terminate an Amazon EC2 instance when criteria are met. Additionally, alarms can be created to trigger Auto Scaling and Amazon Simple Notification Service (Amazon SNS) on our behalf.

- **Elastic Load Balancing**

  Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances. It enables you to achieve even greater fault tolerance in your applications, seamlessly providing the amount of load balancing capacity needed in response to incoming application traffic. Elastic Load Balancing detects unhealthy instances and automatically reroutes traffic to healthy instances until the unhealthy instances have been restored. Customers can enable Elastic Load Balancing within a single Availability Zone or across multiple zones for even more consistent application performance.

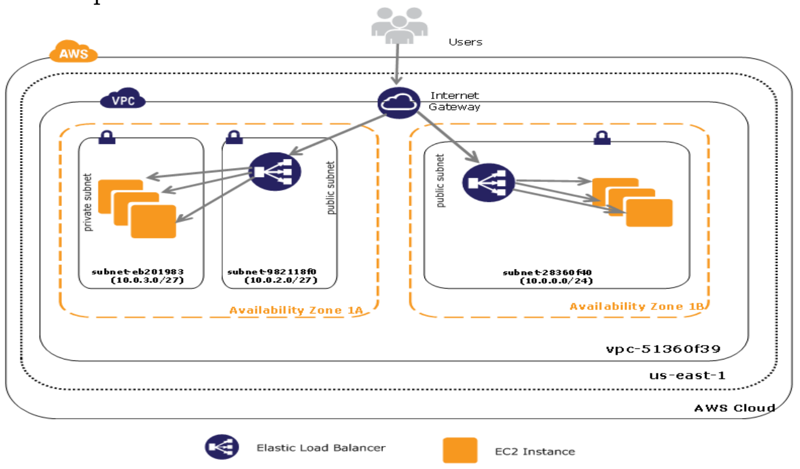

Figure 2.3: Amazon Elastic Load Balancing

- **Amazon Auto Scaling**

  Auto Scaling allows you to scale your Amazon EC2 capacity up or down automatically according to conditions you define. With Auto Scaling, you can ensure that the number of Amazon EC2 instances you're using increases seamlessly during demand spikes to maintain performance, and decreases automatically during demand lulls to minimize costs. Auto Scaling is particularly well suited for applications that experience hourly, daily, or weekly variability in usage.

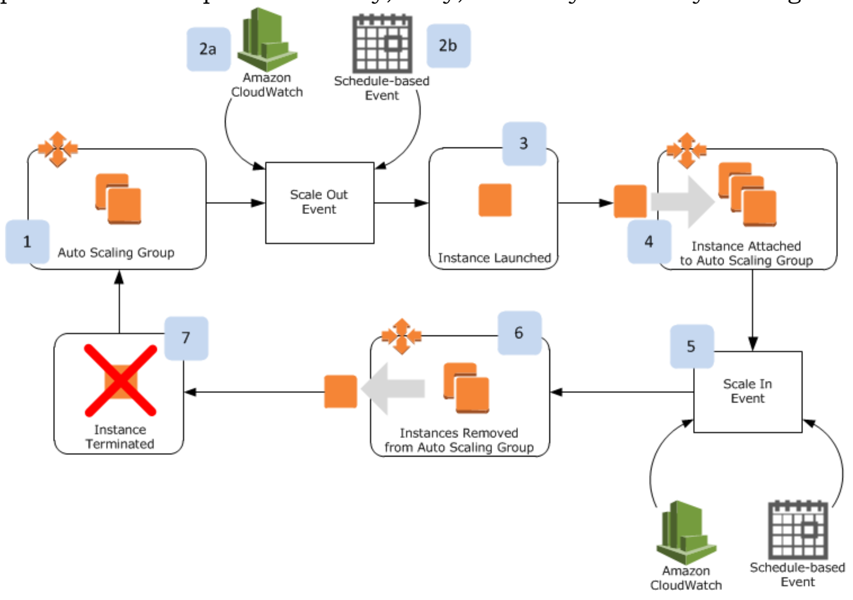

Figure 2.4: Amazon Auto Scaling

Here's how Auto Scaling works in a typical AWS environment.

1. We'll start with an Auto Scaling group set to have a desired capacity of two instances.
2. A scale out event occurs. This is an event that instructs Auto Scaling to launch an additional instance. A scale out event could be something like a CloudWatch alarm (item 2a in the diagram), or a schedule-based scaling policy (item 2b in the diagram) that launches instances at a specific day and time.
3. Auto Scaling launches and configures the instance.
4. When the instance is fully configured, Auto Scaling attaches it to the Auto Scaling group.
5. Eventually, a corresponding scale in event occurs. A scale in event is like a scale out event, except that these types of events instruct Auto Scaling to terminate one or more instances. See Auto Scaling Basic Lifecycle for some examples of scale in events.
6. Auto Scaling removes the instances from the group and marks it for termination.
7. The instance terminates.

Chapter 3

Experimental Setup and Proposed Approach

3.1 Load Balancer's Latency Based Auto Scaling

In this experiment, we demonstrated auto scaling based on Amazon Elastic Load Balancer's latency. Latency of a load balancer measures the time elapsed in seconds after the request leaves the load balancer until the response is received. In the experiment, AWS EC2 t2.micro instances were used as application servers. t2.micro instance consists of high frequency Intel Xeon processors operating at 2.5 GHz with Turbo up to 3.3 GHz, 1 GB memory and AWS EBS storage. Along with EC2 instance Amazon Elastic Load Balancer was used for load balancing between active EC2 instances. For auto scaling Amazon Auto Scaling service was employed and AWS Elastic Load Balancer's latency was monitored by Amazon CloudWatch. We used Httperf to generate request.

Experimental set up is depicted in Figure 3.1.

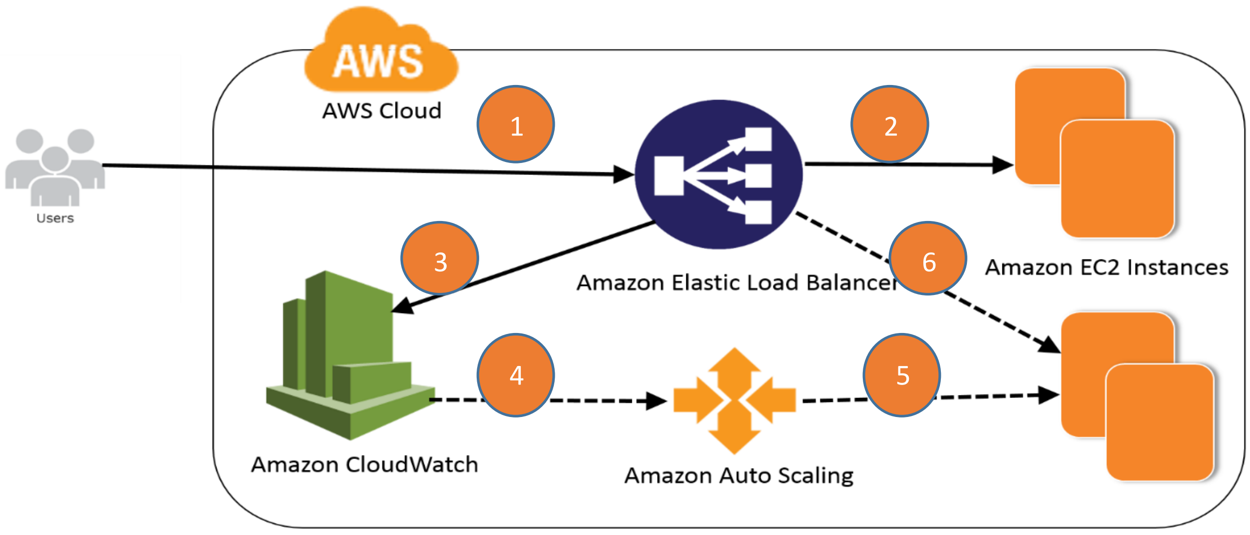

Figure 3.1: Auto Scaling Based on Load Balancer's Latency

Above experimental set up works as follows:

1. Users generate request.
2. Amazon Elastic Load Balancer (ELB) forwards request to active EC2 instances in uniform manner.
3. Latency of load balancer is monitored by Amazon CloudWatch continuously.
4. If the latency of load balancer exceeds a certain threshold for a specified interval of time, Amazon CloudWatch initiate auto scaling.
5. Amazon Auto Scaling performs scale up operation and create new EC2 instance at a time.
6. Once new EC2 instances are in action, ELB starts forwarding requests to newly created instances as well.

Scale down operation is also performed in similar manner. If the latency of ELB reduces below a certain threshold for a specified time interval, Amazon CloudWatch initiates auto scaling and Amazon Auto Scale terminates one of the active instances.

3.2 System Temperature Based Auto Scaling

In this experiment, we implemented a mechanism to extend the processing capacity of physical server in the wake of rising temperature through auto scaling. Our experimental set up consisted of a physical server which was not part of AWS cloud. To augment the processing capacity, we employed AWS EC2 t2.micro instances. We also configured two load balancers- primary and secondary load balancers. Primary load balancer was configured as Apache2 Load Balancer on one of the AWS EC2 t2.micro instance and secondary load balancer was the Elastic Load Balancer provided by AWS. Objective of primary load balancer was to distribute the load among the physical server and secondary load balancer which in turn distributed the load uniformly between scaled up and active EC2 instances. Experimental set up is illustrated in Figure 3.2.

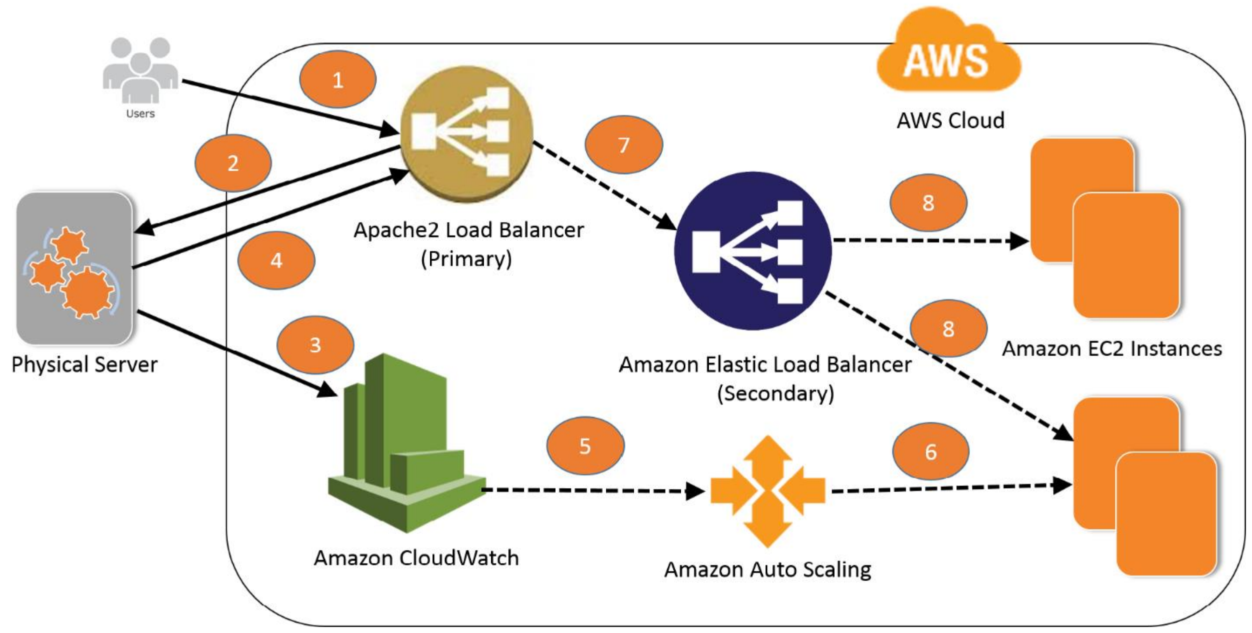

Figure 3.2: CPU Temperature Based Auto Scaling

We simulated user generated requests by executing Httperf. Apart from user generated request, physical server was running local applications which were contributing in raising system temperature. To simulate the load caused by local application, we employed stress test. In this experiment, we intended to obtain aggregated information regarding system temperature. However due to limited sensor option availability, we were able to capture only CPU temperature. Thus, in this experiment auto scaling was performed using CPU temperature of physical server.

Above mentioned experimental set up performs as follows:

1. Users generate request which is targeted at primary load balancer.
2. At the beginning of the experiment, primary load balancer is configured to forward the request to only physical server.
3. A process running in background constantly monitors CPU temperature and publish this information in the form of custom metric into Amazon CloudWatch repository.
4. Another background process constantly monitors CPU temperature to detect if CPU temperature crosses scale-up threshold for specified time interval. If this condition becomes true, primary load balancer is informed to start transferring the load to Amazon EC2 instances by adjusting load distribution weight.
5. Amazon CloudWatch also detects for similar condition through auto scaling policy. If scale up condition is met, it initiates auto scaling.

6. Based on trigger from Amazon CloudWatch, Amazon Auto Scaling instantiate EC2 instances from EC2 instance image.
7. Due to modified load distribution weights, primary load balancer initiates delegating the request to secondary load balancer as well.
8. Secondary load balancer which is an Elastic Load Balancer distributes the delegated load from primary load balancer uniformly between all active EC2 instances.

Scale down operation is also performed in similar manner based on the scale down policy. If the temperature of physical server drops below a certain threshold, primary load balancer load distribution weight is adjusted as well as one of the active EC2 instances is terminated.

3.3 Thread Count Based Auto Scaling

In this experiment, we executed auto scaling based on active thread present on applications servers. Experiment set up consisted of Amazon EC2 t2.micro instances as application servers. We used Amazon Elastic Load Balancer for uniform load distribution between active EC2 instances. In order to simulate user requests, we used Httperf. Experimental set up is illustrated in Figure 3.3.

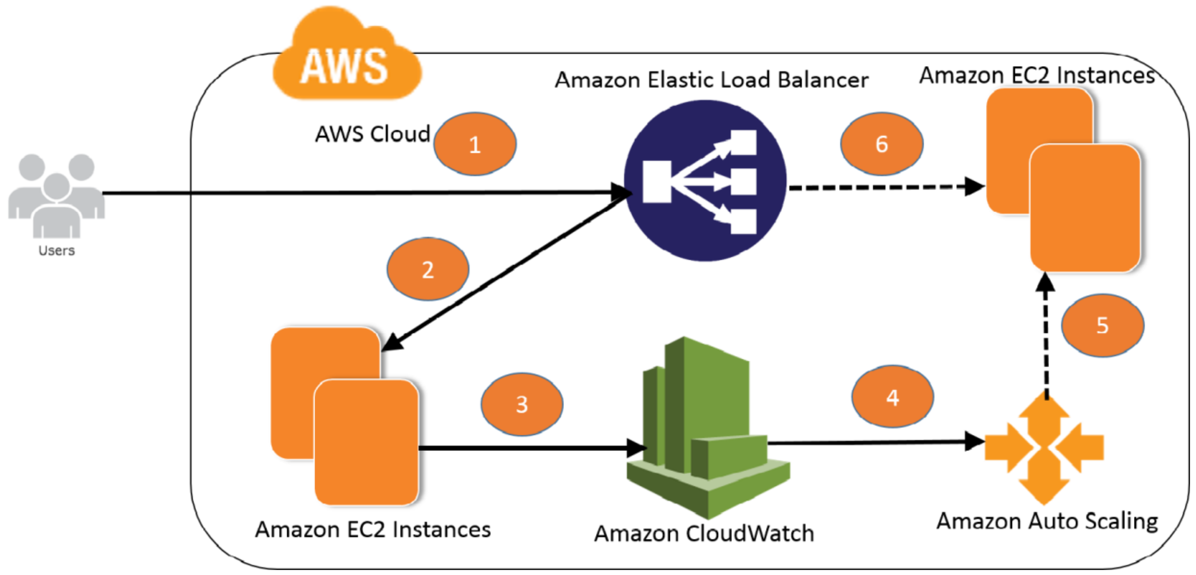

Figure 3.3: Thread Count based Auto Scaling

Above experimental set up works as follows:

1. Users generate requests which is intended for Amazon ELB.
2. Amazon ELB delegates the incoming requests to active EC2 instances.
3. EC2 instances process incoming request which results into spawning of new threads. The thread count information is publish to Amazon CloudWatch in the form of custom metric.
4. Based on auto scaling policy if scale up threshold is crossed by the application servers, Amazon CloudWatch initiates auto scaling.
5. Amazon Auto Scaling launches new instances on alarm raised by Amazon CloudWatch.
6. Amazon ELB starts sending new incoming requests to newly launched Amazon EC2 instances.

Chapter 4

Experiment Results

4.1 Load Balancer's Latency Based Auto Scaling

In this experiment, we used Httperf to generate the request which generated request between 200 to 800 requests every two minutes. Following auto scaling policies were followed in the experiment,
Scale Up:
Increase the number of instance by 1, if latency in seconds >= 1 for 1 consecutive period of 60 seconds and respond to next scaling activity only after 50 seconds.

Scale Down:
Decrease the number of instance by 1, if latency in seconds < 1 for 2 consecutive periods of 60 seconds and respond to next scaling activity only after 50 seconds.

The result of this experiment is presented below in Figure 4.1.

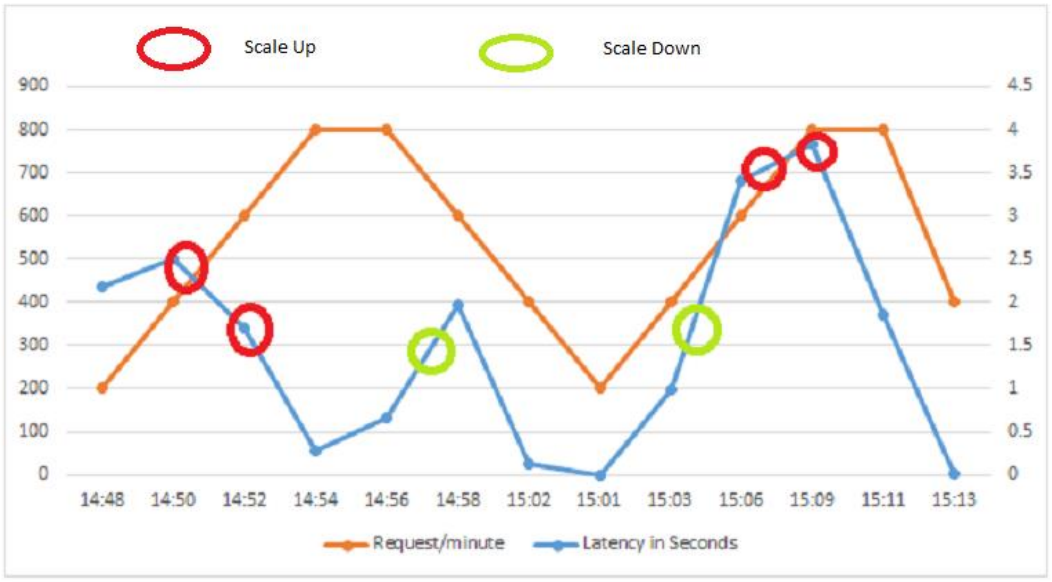

Figure 4.1: Latency Based Auto Scaling

Initially at 14:48, there was only one active instance present. As the request count increased from 200 to 400, latency increased from 2.1 sec to 2.4 sec. As one complete interval of 60 seconds elapsed with latency >= 1, following the scaling up policy number of instances were increased from 1 to 2. As soon as number of instances were increased from 1 to 2, even with increase in the request rate latency dropped from 2.4 sec to 1.67 sec. Nevertheless, latency was again more than 1 sec, scaling up policy was executed and number of instances were increased from 2 to 3. With increased number of instances latency further dropped even with rise of request rate. As the latency dropped and remained below 1 for two consecutive period, scaling down policy was triggered and number of instances were reduced from 3 to 2 at 14:57. There was slight delay in initiation of scale down operation by Amazon Auto Scaling. Again as no if instances were reduced, latency rose. Nevertheless, it remained more than >1 sec only for brief period and went down again due to reduction in request rate. Again latency remained below 1 for more than two consecutive period of 60 sec. This resulted in scale down operation with slight delay. As request rate increased and number instances decreased, latency went up again remaining above 1 for many consecutive periods which resulted into scaling up operation.

During this experiment, we observed that scaling up based on latency provided immediate impact in the face of increasing load. Additionally, we also noticed that there was always a slight delay in execution of scaling operation from the moment the alarm was triggered by Amazon CloudWatch.

4.2 CPU Temperature Based Auto Scaling

In this experiment, we employed stress test which was executed in oscillating manner to cause CPU temperature to rise and fall between ~65° C to ~37° C. Following auto scaling policies were followed in the experiment,

Scale Up:
Increase the number of instance by 1, if metric_temperature in count >= 50 for 1 consecutive period of 60 seconds and respond to next scaling activity only after 50 seconds.

Scale Down:
Decrease the number of instance by 1, if metric_temperature in count <= 45 for 1 consecutive period of 60 seconds and respond to next scaling activity only after 50 seconds.

Figure 4.2 illustrates the result of this experiment. In this experiment, since the stress test was the major contributing application for temperature raise, request rate information is not relevant enough. The points marked by red circle signify the point of scale up operation. All the peaks similar to few marked ones are basically point of scale up operation. Similarly all points marked by green circle are points of scale down operation.

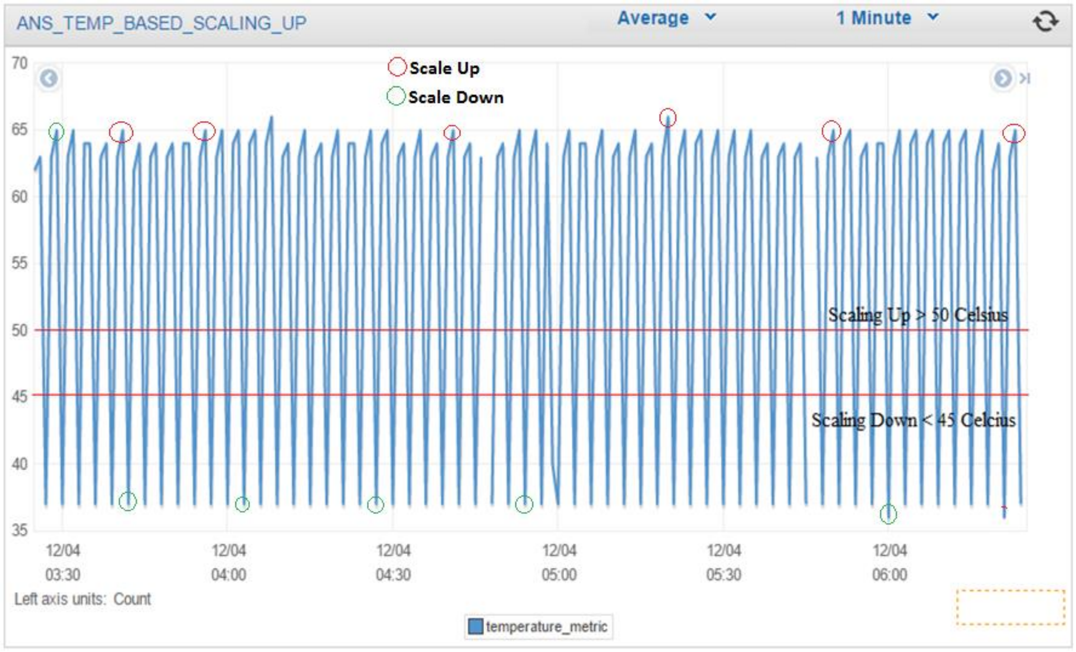

Figure 4.2: CPU Temperature Based Auto Scaling

During the experiment, we noticed that scaling operation closely followed the periodic variation in the temperature which was being published as customer metric in Amazon CloudWatch.

4.3 Thread Count Based Auto Scaling

In this experiment, we employed Httperf to generate request ranging from 50 to 200 with an increment of 50 every two minutes. Background job running at each application server (EC2 instance) also pushed the data, periodically every two minutes. Application server created 5 thread for each request received. So a request of 50 would result into 250 thread at application server. Following auto scaling policy was employed in the experiment,

Scale Up:
Increase the number of instance by 1, if metric_thread_count in count >= 300 for 2 consecutive period of 60 seconds and respond to next scaling activity only after 60 seconds.

Scale Down:
Decrease the number of instance by 1, if metric_temperature in count <= 150 for 2 consecutive period of 60 seconds and respond to next scaling activity only after 60 seconds.

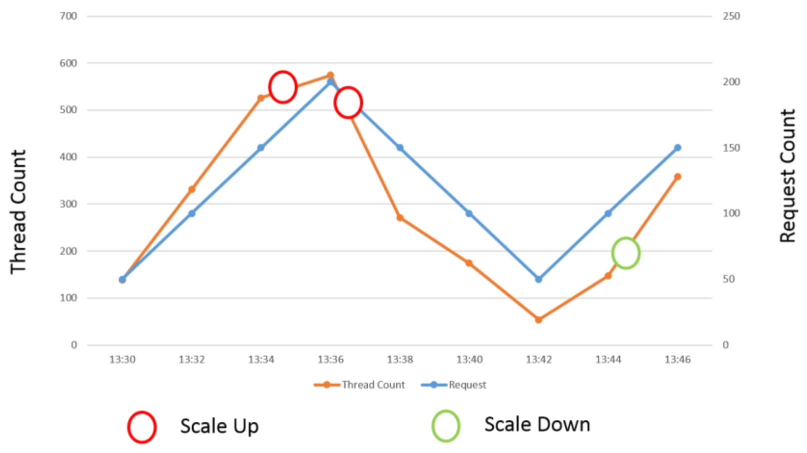

Figure 4.3: Thread Count Based Scaling

This experiment was initiated with one instance. Initially 50 requests were sent which result into 250 threads. However, only 138 threads were reported to Amazon Cloud due to little mis-synchronization between receipt of request and publishing of thread count to Amazon CloudWatch. During this brief interval some of the thread executed completely. After first request, a request count of 100 was received as a result of this thread count 331 was published into Amazon CloudWatch. Published thread count was greater than scale up threshold. However, due to scale up policy of wait for 2 minutes, no scaling performed. On receiving 150 request, thread count rose to 526. This time scaling up was performed and a new instance was launched. Request continued to rise to 200. Nevertheless, due to additional instance thread count increased to only 574. Notice that this thread count was average across two instances. A thread count greater than 300 again result into scale up operation and a new instance was created. Request count dropped to 150 and thread count also dropped to 271. With the reducing request rate and three instance thread kept falling and when thread count remained below 150 twice, a scale down operation was performed. With one instance dropped and increasing request rate, average thread count increased again.

In this way, thread based auto scaling resulted into sharing of incoming requests across multiple EC2 instances. For example, at 13:34 with a request of 150, thread count was 526 as there was only one instance present. However, at 13:46, with same request count, thread count was 358 as there were two instances present.

Chapter 5

Prior Work

There have been lot of work on dynamic resource provisioning in virtualized environments using reactive Auto-Scaling techniques, which was based on CPU Utilization, Latency, Network utilization ([8][9][10][11]). These researches tried to achieve the performance goals through the dynamic resource provisioning. The auto-scaling idea has also been extended to cloud environments like EC2 ([12][13]).

These researches tried to handle the increasing workload by acquiring more VM instances in the cloud whenever the parameter used to scale crossed the threshold mark for a defined amount of time due to insufficient amount of computing resources. For an example Marshall [13] used more instances to increase the computing power of a torque cluster when the submitted jobs cannot be finished on time.

One category, in research related to Auto-Scaling, is to have cost-efficiency in the cloud while maintaining the performance. For the cloud provider side there were many researches, [12][14] which discussed the resource allocation and instance consolidation strategies for the cloud data centers where the goal is to maximize the profit by saving maximum of energy cost and utilizing the resources to the maximum of their potential without breaking the agreement with the customer.

In cloud auto-scaling many cloud providers offer API's which allow the users to control number of VM instance programmatically which facilitates user-defined auto-scaling. AWS and some third-party cloud management services (RightScale [15], enStratus[16]) offer schedule-based and rule-based auto-scaling mechanisms. Schedule-based auto-scaling allow user to add and remove instance at a given time for example “run 10 instances between 6 AM to 10 PM each day and 4 instances rest of the time”. Rule based mechanisms allow user to define simple trigger by specifying the scaling threshold and actions for example “remove an instance when response time is below 100ms or increase an instance if response time is more than 400ms”. The problem with using response time, latency as the scaling factor, is that not all the applications have time-based workload pattern and it is not straightforward. RightScale[15] and AzureScale[17], Amazon EC2 provide some more metrics to scale like memory utilization, queue size. Pending request to scale up and down these scaling indicators made dynamic scaling easier for web applications.

In our work, we have further enhanced the scaling metrics provided by Amazon Web Services. We proposed two metrics- system temperature and thread count in application server.

Chapter 6

Conclusion

Cloud computing has become a new de facto to establish, operate and improve the IT infrastructure. Many organizations are thriving in their business domains by exploiting this technology. Two key characteristics of such offering are elasticity and scalability. Consequently, a robust mechanism is required to achieve these functionalities without manual intervention. This requirement has led to the foundation for various auto scaling techniques. In this work, we analyzed the auto scaling approach followed by Amazon Web Services (AWS), a market leader in providing IT infrastructure through cloud computing. We observed that auto scaling configuration policy provided by AWS is exhaustive and caters various aspect of auto scaling on cloud platform adequately. Nevertheless, the scheme followed by AWS could be further improved. We proposed two new auto scaling metrics – system temperature and thread count of application server, to improve existing Amazon Auto Scaling. Our primary objective was to demonstrate the feasibility of auto scaling using these two metrics, which we achieved successfully. We were able to extend the processing capacity of a physical server using temperature based scaling. With thread count based scaling, we targeted the primary issues faced by application server in presence of high thread count. Additionally, we analyzed the auto scaling based on Amazon Elastic Load Balancer's latency. Most of the prior works, focused on auto scaling based on latency of application server. In contrast to previous work, we demonstrated auto scaling with load balancer's latency and we were able to see the improvement achieved due to auto scaling done based on aggregated latency of load balancer.

References

[1] https://www.netflix.com/

[2] https://www.pinterest.com/

[3] http://code.org/

[4] https://www.coursera.org/

[5] http://www.expedia.com/

[6] https://media.amazonwebservices.com/AWS_Overview.pdf

[7]http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/- ec2-etricscollected.html

[8] H. Lim, S. Babu, J. Chase, and S. Parekh. Automated Control in Cloud Computing: Challenges and Opportunities In 1st Workshop on Automated Control for Datacenters and Clouds. June 2009.

[9] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. 2007. Adaptive Control of Virtualized Resources in Utility Computing Environments. EuroSys, 2007.

[10] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In 2nd International Conference on Autonomic Computing, Seattle, WA, USA, June 2005.

[11] Q. Zhang, L. Cherkasova, E. Smirni. A RegressionBased Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In Proceedings of the Fourth International Conference on Autonomic Computing, Jacksonville, Florida, USA, June 2007.

[12] P. Ruth, P. McGachey and D. Xu. VioCluster: Virtualization for Dynamic Computational Domains. Cluster Computing, 2005. IEEE International Sep 2005

[13] P. Marshall, K. Keahey and T. Freeman. Elastic Site Using Clouds to Elastically Extend Site Resources. In the 10th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Melbourne, Victoria, Australia, 2010.

[14] Y. Yazir, C. Matthews, R. Farahbod, S. Neville. Dynamic Resource Allocation in Computing Clouds using Distributed Multiple Criteria Decision Analysis. In 3rd International Conference on Cloud Computing, Miami, Florida, USA, 2010

[15] RightScale. http://rightscale.com

[16] enStratus. http://www.enstratus.com

[17] AzureScale. http://archive.msdn.microsoft.com/azurescale

[18] Ming Mao, Marty Humprey. Auto-Scaling to minimize Cost and Meet Application Deadlines in Cloud Workflows. Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. Seattle, Washington, 2011