What Autoscaling should react to?

{*adiprakash, algupta, ksasmit, vsahu*}*@cs.stonybrook.edu*

***Abstract*— Today, in this data oriented world, efficient Data Centers are indispensable to large storage and computation requirements. They have become the mainstay which offer on-demand computing and storage capacity with minimal cost. However, with the data explosion, keeping the cost within operational limit for the storage and processing has become the focal point of the Cloud Service Providers (CSP). Besides, power consumption and cooling add another 30%-50% extra cost. To get away with these, CSPs rely on the dynamic scaling of the underlying infrastructure in response to business volume, performance desire and other dynamic behavior. Most of the CSPs over-provision the servers to keep the system ready for peak demands. However, this situation leads to the wastage of the resources, energy and cost of the systems if the load on the system is low most of the time. Moreover, dynamic scaling policies are not monitored too often to get optimal cost under different loads.**

In this paper, we present a detailed analysis of Auto-scaling metrics and experimental study to find under which load pattern, which metric suits well. We have chosen three metrics namely, Response Times (*rt*), Request Rate (*rr*) and Queue Length (*ql*). Each of these metrics is studied well under static and dynamic loads with the application of conservative and aggressive policies. Under dynamic load *ql* comes out to be the most dominant metric for scaling while under static load, choice can be made between *rt* and *rr* depending on cost and performance requirement.

Keywords - Cloud Computing, Data Center, Auto-scaling, Load Balancing

I. INTRODUCTION

Cloud computing has become the mainstay of the bigger enterprises. It offers off-loading heavy computation and storage to the Data Centers and helps in concentrating on the core fundamentals of the services being offered by the enterprises. Off-loading the administrator's burden to the cloud helps in saving manpower, cost and management effort. However, merely off-loading doesn't alleviate all of the problems. It needs to be properly managed by the Cloud Service Providers (CSPs) so as to provide best possible facility to the enterprises in terms of availability, management, scalability, cost etc. *Auto-scaling* is one of the main techniques employed by the CSPs to autoscale VM capacity or number horizontally and vertically based on present loads and user defined policies. Using auto-scaling, users can define triggers by specifying the performance metrics and thresholds. Whenever the observed performance metric is above or below the threshold, a predefined number of resource instances will be added to or removed from the system. For example, a user can define a trigger like Add 2 instances when CPU usage is above 60% for 5 minutes.

Scaling based on the load, CPU utilization, response time or queue length has been implemented for quite a long time. But, tuning to the exact metric under different situation has relatively been ignored. And consequently, we see over-provisioning which results in under-utilization of the servers and therefore incur more cost. Besides, energy is another factor which makes it more severe. Choosing appropriate performance metric and finding precise threshold is not a straightforward task, and the problem becomes harder in case of dynamic workloads.

In this paper we present the analysis of various metrics and their effect on scaling under various load and modes of thresholds. We also design and implement a custom java scalar that helps in monitoring the various metrics and perform auto-scaling based on the threshold set by the administrator. Our java scalar sits on top of *HAproxy*, a high availability load-balancer, and peeks into the statistics accumulated and accordingly adds or removes the servers gracefully.

The rest of the paper is organized as follows. Section II provides the motivation for our research followed by related work done in the past in section III. Section IV gives a holistic view of our experimental setup and design for achieving the fully controlled scaling mechanism based on Java Scalar. Section V showcase the result of our experiments and provides the evaluation. It depicts the results in rich graphical fashion for multiple traces of loads under various modes of thresholds followed by bench-marking in Section VI. Finally section VII and VIII concludes the paper along with the future work.

II. Motivation

Performance and economics are orthogonal in the data Servers. Enterprises often fantasize, because we pay only for what we use, going to the Cloud automatically saves a lot of capital. However, there is one subtle problem to it. The risk of compromised performance often leads to the over-provisioning of the servers. Over-provisioning of cloud resources, in the absence of other choices, has become an epidemic. The result is that many organizations are investing in cloud resources they simply do not use. Unused resources produce, of course, zero return on investment (ROI).

Idle servers still use 60% of the peak power and therefore incur relatively significant cost. Moreover, Keeping too many idle servers up most of the time to compensate for the bursts occasionally, adds to under-utilization of the servers. Big players like Amazon and Microsoft rent out these spare machines, but small CSPs don't have options but to keep spare machines always on.

III. Related Works

1. **Load Balancer Behavior Identifier (LoBBI) for Dynamic Threshold Based Auto-scaling in Cloud.** In this paper, a rule engine based framework[1] is proposed to dynamically decide the autoscaling threshold values based on the load characteristics and current performance of the VMs in a cloud computing infrastructure. Our experiment is based on comprehensive analysis of different metrics for different threshold values for autoscaling VMs to find out which metric to use as the deciding factor to autoscale. We believe incorporating dynamic threshold values over our proposed metric can provide even better performance and cost savings.

2. **Cloud Auto-scaling with Deadline and Budget Constraints.** This paper discusses about scaling based on workload information and performance desire without violation of target temporal deadline(months or years)[2]. The goal is to reduces cost by choosing appropriate instance types (High CPU, High IO, Mixed). The autoscale metric is application based - job deadline and budget. Our experiment tests performance under different autoscaling metrics without violating the different performance constraint. We have used only one type of VM instance and only CPU intensive workload. Adding different instance types and workload(High CPU, High IO, Mixed) to our test-bed and making a choice of instance type based on the request type would be an enhancement to our work bringing added benefits.

3. **Dynamic Selection of VMs for Application Servers in Cloud Environments.** This paper suggests use of heuristics and machine learning approaches[3] to study the workload pattern and dynamically select appropriate VM type while autoscaling. Presently, our work is based on homogeneous VMs and varying CPU intensive jobs but the idea proposed in this paper can be applied to our experiment for an extensive analysis of choosing metric.

4. **BATS: Budget-Constrained Autoscaling for Cloud Performance Optimization.** This paper emphasizes on budget as the prime factor for making autoscaling decisions. Autoscaling is done based on past and instantaneous load data while ensuring the target budget[4]. Our experiment metrics indirectly ensures minimum budget by keeping minimum number of VMs active and at the same time preventing SLA violations.

5. **Adaptive, Model-driven Autoscaling for Cloud Applications.** This paper suggests employing a Kalman filtering technique[5] in combination with a queue theory based modeling to proactively determine the right scaling actions for an application deployed in the cloud. This uses Response time and Number of VMs as its performance and cost metrics same as ours.

6. **Evaluating Auto-scaling Strategies for Cloud Computing Environments.** This paper focuses on evaluation of the quality of autoscale policies using ADI(Auto Scaling demand index)[6]. It explains generation of two step size configuration strategies (Adaptive and Fixed) and three auto-scaling triggering strategies (Reactive, Conservative, and Predictive). These can be incorporated with our study to find out the best autoscale metric by measuring the performance using ADI. Also, adding 2 and 3-step size configuration strategies to the system with preferred metric can bring additional performance/cost benefits.

7. **Integrated and Autonomic Cloud Resource Scaling.** In a Cloud, resources from three separate domains, compute, storage and network, are acquired or released on-demand. This paper discusses an approach called Integrated and Autonomic Cloud Resource Scaler (IACRS) where these three domains are auto-scaled in an integrated fashion taking one or more out of CPU load, Link load, and Response time as autoscaling metric. Therefore the complete system being multi-domain with multi-metric[7]. Our experiment is somewhat similar to this system in a way that we consider different metrics (Request rate, Response time and queue length) as deciding factor for autoscaling under different scenarios.

8. **AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers.** This paper discusses about ways to deal with bursty, unpredictable, time-varying load, while meeting response time SLAs by introducing a wait period[8] before turning servers off and also maintaining spare servers for such scenarios. Implementing this technique to our study would give more appropriate results, saving unnecessary autoscaling during load spikes.

IV. Solutions /Experimental Setup

Our experimental bed consists of several components required to simulate the autoscaling of application servers at data center. Experimental setup requires significant part of the overall time and effort so to configure every other components precisely. The whole setup is based on homogeneous servers with Unix based operating systems. We used Google cloud based VMs with debian-8-jessie-v20160301 based Unix systems, 10GB SSD, 3.75 GB RAM and n1-standard-1 processor range for load generator, load balancer and application servers. Below are the list of components and their role to our experiment.

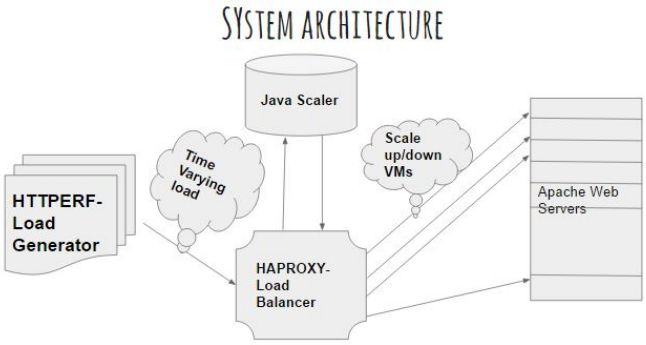

Fig. 1. System Architecture

1) Load generator (HTTPERF): We used HTTPERF, which is a tool for measuring web server performance, as a load generator to our system. It provides flexibility to generate various HTTP workloads and means to measure server performance. For our experiment, we slightly modified it to produce various time varying loads to the load balancer. HTTPERF Installed and configured on a separate VM to generate maximum workload variations which requires large memory and resources to open file descriptors for every connections. We also wrote a C based automation script to generate time varying workload traces for Httperf.

2) Load balancer (HAPROXY): Haproxy is a free, open source software that provides a high availability load balancer and proxy server for TCP and HTTP-based applications that spreads requests across multiple servers using user defined algorithm. It is written in C and has a reputation for being fast and efficient (in terms of processor and memory usage). We modified configuration file to tune different set of maximum connections which it can hold in both of its front-end and back-end interfaces and included list of available servers (VMs) which can be added/removed later depending upon the scaling requirements. In our experiments, we used round-robin load balancing mechanism to distribute loads.

3) Java Scaler: This is one of the key components of our experimental setup which controls the scaling needs of our system. It provides a controlled and manageable platform to operate the desired needs of application servers while keeping the maximum server utilization along with the desired performance. Haproxy itself provides a real time statistics daemon tool which provides numerous required parameters associated with the load balancer. We use this daemon process to fetch the real time values of request rate, queue length and response time metrics required for autoscaling. Below are the details of its input requirements
   - Haproxy web statistics url: Haproxy provides real time and structured web view of several network parameters related to it. Our Java Scaler uses this platform, parses the values from web page and applies business logic provided during the experiment to scale up/down accordingly.
   - Metric type: This input provides the metric to be monitored and scale accordingly. We focused in analyzing the behavior of request rate, queue length and response time in terms of maximum server utilization and performance.
   - Scaling thresholds: These arguments specify the metric range beyond which an existing server is added/removed by Java autoscaler.

This follows the graceful restart of load balancer.

- Polling intervals(Seconds)- It is one of the most important parameters of Java Scaler which defines how frequently autoscaler should monitor the given metric and make autoscaling decision. Choosing a polling interval too short and too long can result in VMs thrashing[1] or resource under/over utilization respectively. We recommend to maintain it above 5 seconds as Haproxy itself polls system to get the updated parameters in every 5 seconds. Our experiments spread over wide range of polling intervals from 10 to 60 seconds to understand the different scaling rates/modes and their behavior. This helped us in deciding the cap for maximum server utilization without hurting desired performance.

4) Apache Web Server: During this work, we used maximum of six Apache based web servers hosted on separate machines for autoscaling. Apache servers were configured with different CPU intensive services for client requests. We wrote task varying PHP scripts and installed it on the application servers to imitate real service providers. These servers were of homogeneous in nature. The application servers served incoming requests on *first-come-first-serve basis*.

5) Haproxy graceful restart: Haproxy needs restart as soon as it observes any change to its configuration file. Java Scaler edits the configuration file at run time and uses a small script to gracefully restart the Haproxy without loosing any request packet.
   - How it works: Whenever Java Scaler needs to scale up/down the application servers, it triggers this script which in turn updates the system IP table to drop all incoming SYN requests to the Haproxy. Meanwhile it restarts the Haproxy with the updated configuration file and then, it again updates the IP table to start allowing all the SYN packet request to the Haproxy. According to TCP protocol, HTTP request creates TCP connection via three-way SYN/ SYN-ACK/ ACK handshaking before sending any data. During graceful restart, as load balancer doesn't accept the SYN packets, sender will retry sending the same packet again after particular timeout interval. Meanwhile Haproxy would be reloaded with newer changes. This prevents dropping of any HTTP request with just 400 ms of service delay during the restart process. There is explicit delay of 1 second before restarting of Haproxy to clear all the pending requests so to avoid interrupting any past processing. Below commands are used in scripts that performs ip table reset.

```
iptables -I INPUT -p tcp --dport 80 --syn -j DROP
sleep 1
service haproxy restart
iptables -D INPUT -p tcp --dport 80 --syn -j DROP
```

[1]repeated add-remove-add autoscaling pattern because of highly dynamic metric value and/or large polling interval

V. Evaluations and Results

We performed this study on varieties of workload traces, both static and time varying. We classify these workload traces broadly into static and dynamic types:

- Static traces- Workload traces with smaller variations to some predefined request rate. This emulates the normal load over a period of time.
- Dynamic traces- Workload traces with larger/bursty variations within some range of predefined minimum and maximum request rate. This emulates the scenarios where websites experience random and unpredictable load.

We took empirical approach to evaluate each metric with several combinations of both aggressive and conservative modes to determine the threshold range. These combinations were constraint to average response time being within allowable limit. Below are the definition of modes used:

- Conservative mode- wider range of given metric value is used by Java scaler to make scaling decision. The goal of this mode is to have maximum server utilization of available VMs with less frequent scale up/down activity. It might result in performance degradation but leads to low wear and tear of VMs in long run.
- Aggressive mode- In this mode, Java scaler uses wider range of given metric value to make scaling

decision. The goal of this mode is to avoid any performance degradation by frequent scale up/down of VMs. It provides strict monitoring of given metric values to provide required scaling instantly. It provides relatively better performance in terms of response time but might result in wear and tear of VMs because of frequent switching on and off the active stack.

The key point here is that scaling action under conservative mode is subset of that under aggressive mode. The reason for this lies in the fact that threshold window for aggressive mode lies within threshold range for conservative mode. Thus, all the time instances where autoscaling is done in conservative mode, it's also done at those instances in aggressive mode given same workload and system configuration.

In this section, we explain our selective key experimental results. We evaluated each metric with different range of thresholds with constraint of not diminishing the performance by keeping response time (SLA) below 400ms. For all the experiments, we've kept run duration fixed at 20 minutes. All the graph shown in this section have x-axis as time. Fig 2 depicts behavior of request rate along with queue length and response time over time when autoscaling is done under request rate metric.

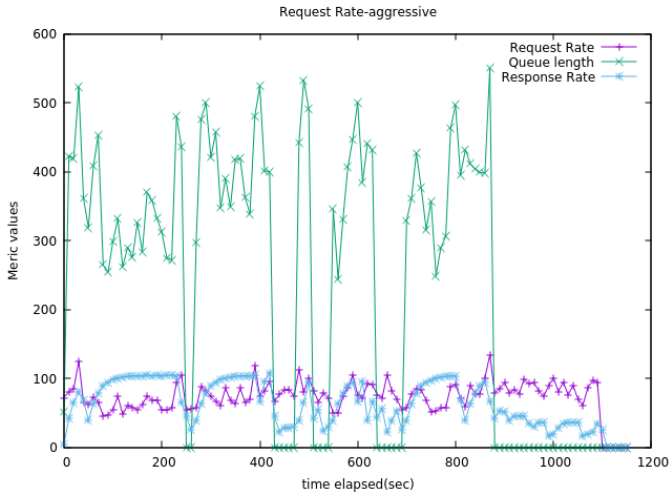

Fig. 2. Request Rate, Static trace

We observe that as request rate exceeds the threshold values, which we determined empirically, autoscaler performs graceful addition/removal of VM followed by decline in the rising metric values. Towards the end where system is operating with more number of VMs, we notice that queue length and hence response time remain at very low values.

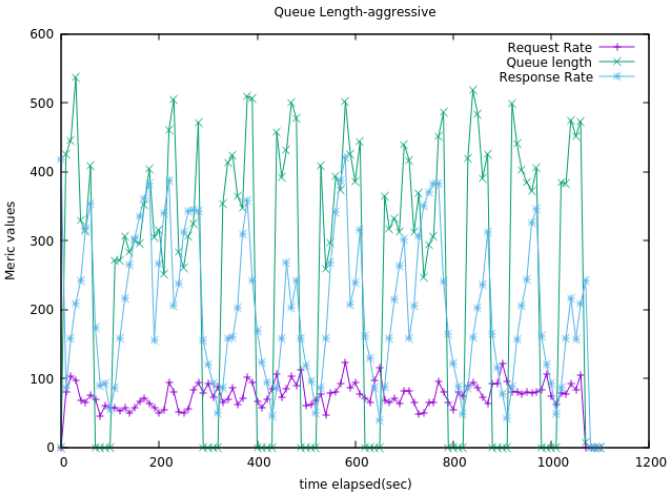

Fig. 3. Queue Length, Static trace

Similarly, Fig 3 observes the queue length, request rate and response time metric when autoscaling is performed based on Queue length. Since the monitoring metric is queue length, we observe that response time varies a lot before autoscaling is done. Same static type request rate is used in this case but threshold range of queue length is set aggressively.

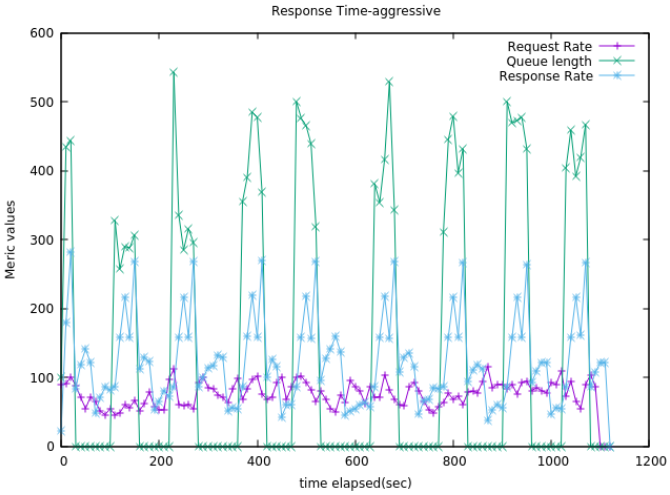

Fig. 4. Response Time, Static trace

Fig 4 shows the same evaluation as we did in case of Fig 2 and 3 but for response time. We find response time values complementary with the queue

length variations which is intuitive as increased queue length causes request to be served slowly, thus resulting in increased response time.

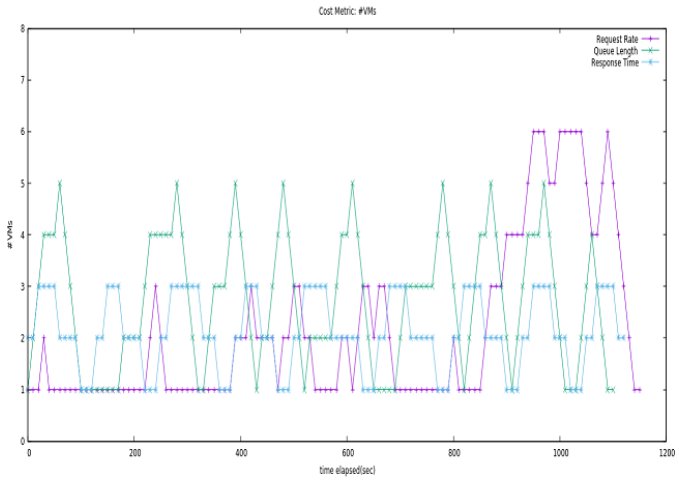

Fig. 5. Cost Metric, Static trace

In Fig 5, we perform cost analysis by consolidating the results of all metric for static trace. Response time is found to be the most economical in terms of average number of VMs used. It can be understood as the most defensive scaling approach as response time varies mostly with the queue size of Haproxy which builds up only when request rate is throttled. This is the reason why response time results in maximum server utilization in terms of VMs.

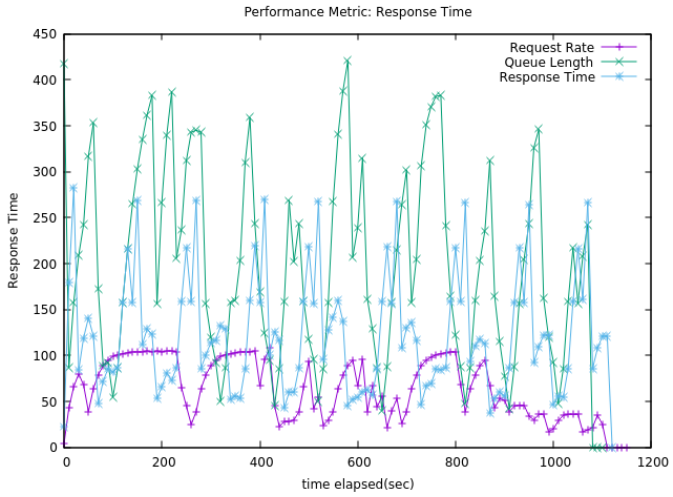

Fig. 6. Performance Metric, Static trace

Fig 6 shows the response time as performance metric for each of the three metric in action and found that autoscaling based on request rate results in minimal response time. Since request rate is observed at front end, rise and fall in request rate is responded by autoscaler before queue length and response time get chance to increase. This proactive autoscaling may result in false auto scaling because of momentary spike in incoming requests.

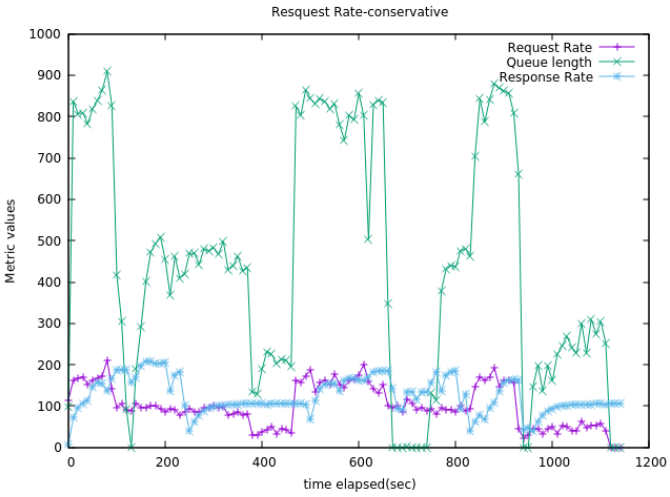

Fig. 7. Request Rate Dynamic Conservative

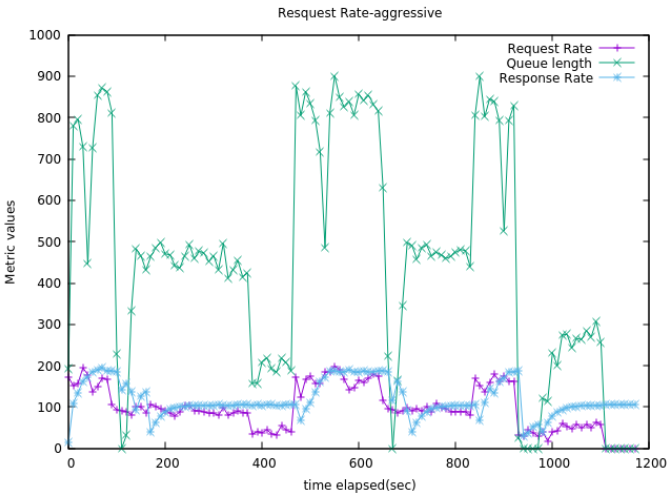

Fig. 8. Request Rate Dynamic Aggressive

We perform our second set of similar experiments for dynamic bursty workloads. We have observed each metric under 2 different autoscaling modes. Fig 7 and 8 show all three metrics when autoscaling is done

based on request rate in conservative and aggressive mode respectively. In this case, as autoscaling is done based on request rate, we see that there is not much difference in the way response time and queue length vary. The reason for this is same as mentioned earlier in performance analysis for static trace.

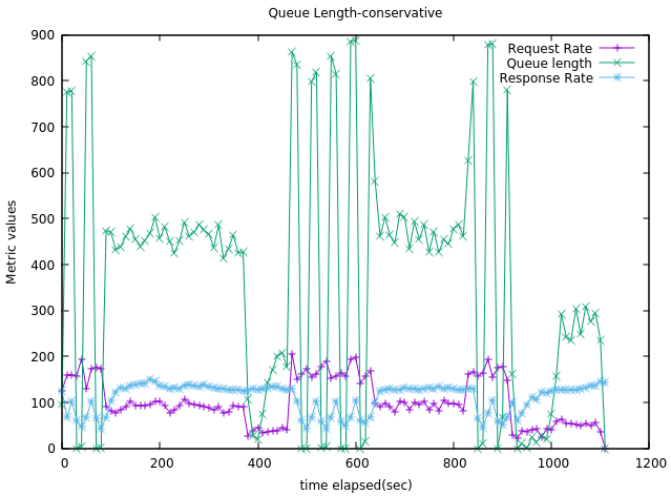

Fig. 9. Queue Length Dynamic Conservative

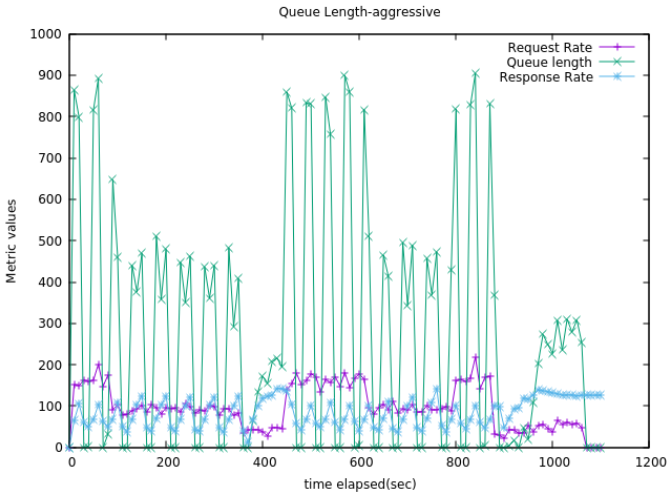

Fig. 10. Queue Length Dynamic Aggressive

Fig 9 and 10 represents all three observed metric when autoscaling is performed based on queue length for dynamic bursty traces in conservative and aggressive mode respectively. As explained earlier, time instances where autoscaling happens in conservative mode, it happens at those time instances in aggressive mode too. Because of narrow range, aggressive mode displays more number of autoscaling instances. The interesting part in these 2 graphs is in time range from 450s to 600s, where there is continuous switching of VMs. This is because the request rate remains high close to 200. Since we monitor metric value every 15 seconds, there is successive addition of VMs at each instances as queue length surges again in 15 seconds. This could be solved by provisioning addition/ removal of multiple VMs based on prevailing metric value. We see it as extension to our current work.

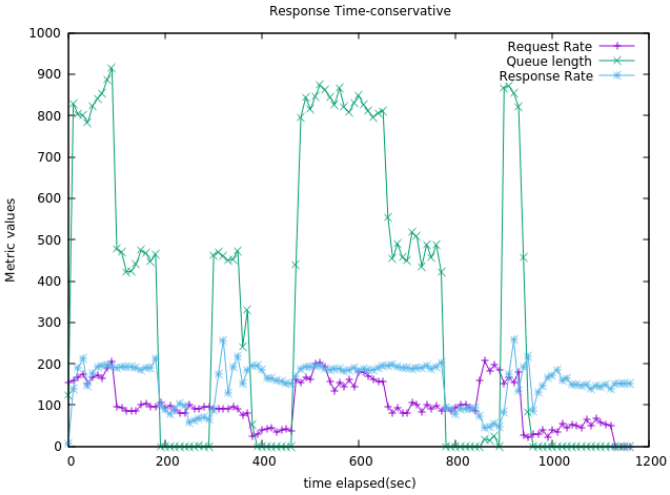

Fig. 11. Response Time Dynamic Conservative

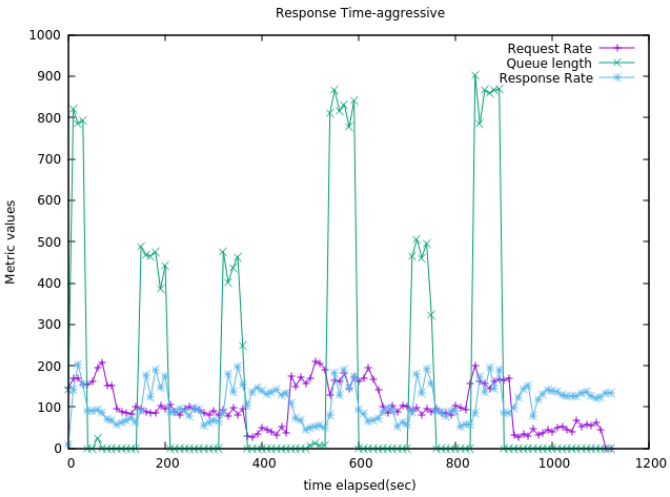

Fig. 12. Response Time Dynamic Aggressive

Fig 11 and 12 are captured under monitoring of response time with both conservative and aggressive

without violating the response time of 400 ms. It is observed that queue length and response time remain high for longer duration in conservative mode. The reason for this behavior is that in conservative mode, we allow metric to reach up to much higher threshold before doing autoscaling. This also results in better resource utilization.

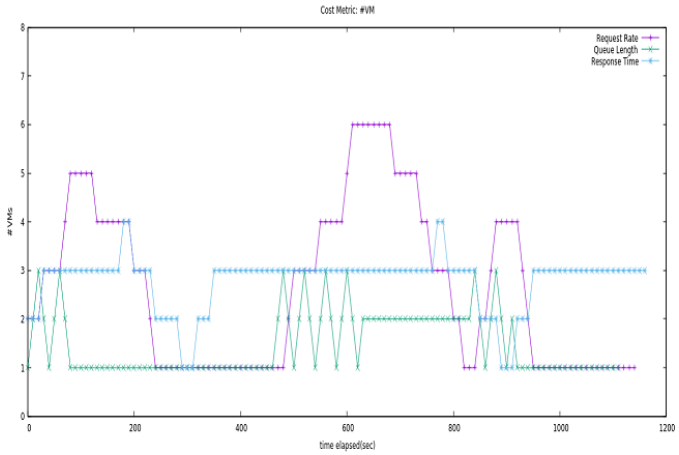

Fig. 13. Cost Metric for Dynamic load, Conservative mode

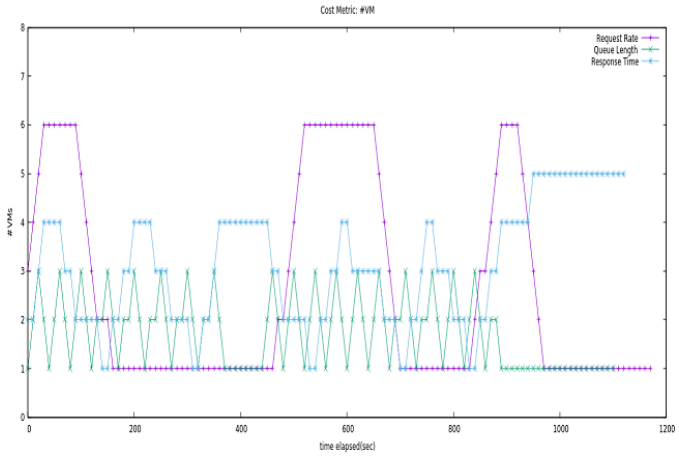

Fig. 14. Cost Metric for Dynamic load, aggressive mode

In Fig 13 and 14, we show the cost analysis for all three metrics for dynamic trace under conservative and aggressive modes respectively. Queue length is observed to be the most economical and performance driven metric. The reason for this behavior is the pattern with which queue length varies. As request rate depends on load generator and response time increases because of increase in queue length, autoscaling on the basis of queue length utilizes available resources fully before triggering the upscale/ downscale.

Autoscaling in aggressive mode for dynamic load results in minimal number of VM usage, however it comes at the cost of wear and tear in long term. From experimental results, we find that for queue length, switching rate[2] is 1.6 per min and 2.7 per min for conservative and aggressive mode respectively. For response time, switching rate stands at 0.75 per min and 1.6 per min for conservative and aggressive mode respectively.

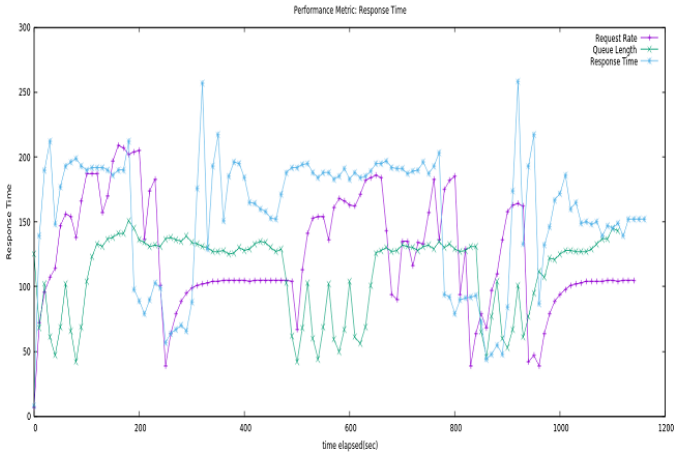

Fig. 15. Performance Metric Dynamic Conservative

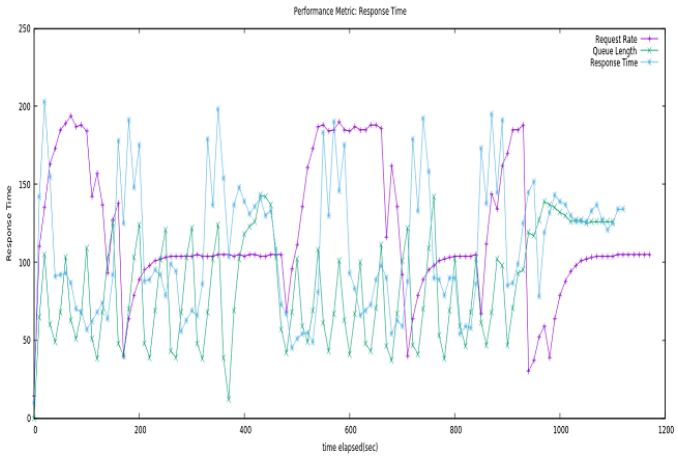

Fig. 16. Performance Metric Dynamic Aggressive

Fig 15 and 16 show the performance metric in terms of response time for all the evaluated metrics

[2]switching rate is defined as # of VM instances removed/ added per minute

for dynamic trace under conservative and aggressive mode respectively. The average response time in case of queue length is minimum among all 3 metric in both the modes. We computed this value to be 111 and 103 ms respectively from the data we recorded. The reason for queue length outperforming other metric in case of performance for dynamic trace is that in homogeneous servers, queue length directly controls the response time of requests. Hence performing autoscaling based on queue length keeps response time within limits. Also, queue length is oblivious to momentary change in request rate which prevents unnecessary addition and removal of VMs. Another important point to note here is that average response time is not minimum in case of request rate for dynamic trace. This is because the abrupt change in incoming load leads to queue build up.

	Static Trace	Dynamic trace	
		Conservative	Aggressive
Cost(#of VMs)	Response time	Queue length	Queue length
Performance	Request rate	Queue length	Queue length

Fig. 17. Summary from graphs

VI. Benchmarking

We used siege, an open source bench-marking tool to standardize cost and performance of our system. Siege allows to mimic the real-time traffic with configurable number of simulated users. We tried to test the dynamic load for the configurable URLs of the Application servers.

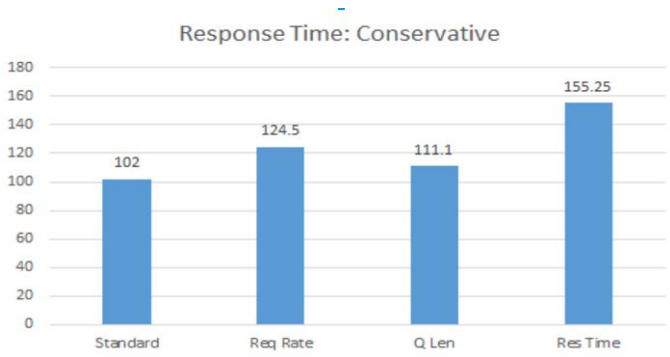

Fig. 18. Benchmarking- Conservative mode

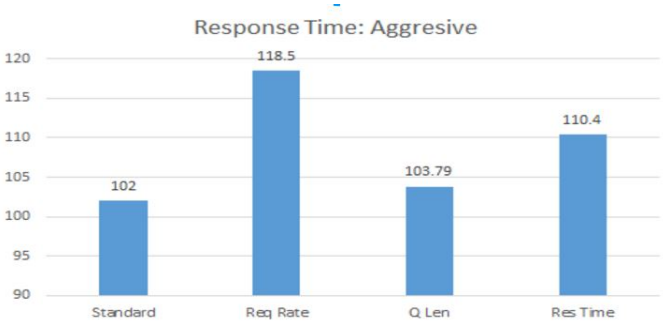

Fig. 19. Benchmarking- Aggressive mode

We ran bench marking load for 20 minutes for the set of 5 VMs and collected the data. The data corroborates the result we get from our experimental setup. Comparison of results for dynamic load in conservative and aggressive mode with the bench-marking numbers are shown in fig. 18 and 19 respectively. We observe that the response time in case of queue length is minimum for all 3 metrics and is slightly more than the ideal time.

VII. Conclusions

In this project, we analyzed various metrics and their impact on autoscaling in terms of overall cost and system performance. Our experiment results show that under dynamic bursty traces, queue length should be the metric to which autoscaling should react to whereas under static load, response time and request rate work better categorically.

We see that autoscaling based on queue length is cost effective but it comes at cost of wear and tear. From experimental data, we note that in aggressive mode, switching is close to double of that in conservative mode.

Following are the key observations from the rigorous experiments we performed using diverse dynamic traces:

1. In case of dynamic and bursty traces, autoscaling based on queue length results in better performance and economical operation in both conservative and aggressive modes. Since response time increases when queue builds up, scaling based on queue length justifies above observation. It is evident from all the graphs that response time variation is closely coupled with queue length which leads to maximum server utilization along with the desired performance without hurting any of the SLAs.

2. Response time proved to be the most economical metric in case of static type traces as it avoids scaling

of any VMs till the response time crosses acceptable range. Response time grows analogous to queue length which overflows only if request rate exceeds the processing limit of available VMs. This results in maximum server utilization of active VMs.

3. In case of static trace, we observe that Request rate is most performance driven metric among others as it restricts SLA violations in terms of response time at load balancer stage itself.

4. In some cases, both conservative and aggressive mode result in similar behavior. This is because the threshold range for aggressive mode lies completely within that of for conservative mode.

VIII. Future Work

1.Consideration of server setup time & Application of wait period. We have not considered the server setup time. Our experiment considers that servers are instantly switched on and off. In real world data centers this time can extend upto several minutes. Inclusion of server setup time in our analysis, i.e consideration of wait time before switching off the servers would fetch more stable results and would save the unnecessary wear and tear for bursty loads.

2.Adding/removing multiple VMs. In our autoscale experiments, we have considered addition/removal of single VM during each upscale/downscale instance. An extension to this behavior will be addition/removal of multiple VMs together at the same time depending on the load variation/metric value.

3.Dynamic threshold values. Like one of the papers in related works section, We can consider including dynamic threshold values. Based on the workload and requirements, we can change the threshold limits for scale-up and scale-down dynamically during the run time. This would add more flexibility to the system.

4.CPU bound and IO bound workload. We can consider different kinds of servers each able to handle different types of workloads(CPU bound and IO bound). The autoscale algorithm would consider different thresholds for each type and autoscaling would be performed independent of each other. In our experiment we have considered only CPU bound operations, adding IO bound tasks would emulate real world traces.

IX. Code and Experimental Data Repository

For more information about Java Scaler source code and different experimental traces along with their results. Click here- Code base & Experimental data

References

[1] Load Balancer Behavior Identifier (LoBBI) for Dynamic Threshold Based Auto-scaling in Cloud. M.Kriushanth, Dr. L. Arockiam. 2015 International Conference on Computer Communication and Informatics (ICCCI -2015), Jan. 08 10, 2015, Coimbatore, INDIA

[2] Cloud Auto-scaling with Deadline and Budget Constraints. Ming MaoJie LiMarty Humphrey, Department of Computer Science, University of Virginia, Charlottesville, VA, USA 22904

[3] Dynamic Selection of Virtual Machines for Application Servers in Cloud Environments. Nikolay Grozev and Rajkumar Buyya, Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Australia

[4] BATS: Budget-Constrained Autoscaling for Cloud Performance Optimization. A. Hasan Mahmud, Yuxiong He, Shaolei Ren. IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems.

[5] Adaptive, Model-driven Autoscaling for Cloud Applications. Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang, IBM Research

[6] Evaluating Auto-scaling Strategies for Cloud Computing Environments. Marco A. S. Netto, Carlos Cardonha, Renato L. F. Cunha, Marcos D. Assuncao. 2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems

[7] Integrated and Autonomic Cloud Resource Scaling. Masum Z. Hasan, Edgar Magana, Alexander Clemm, Lew Tucker, Sree Lakshmi D. Gudreddi

[8] AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. ANSHUL GANDHI, MOR HARCHOL-BALTER, and RAM RAGHUNATHAN, MICHAEL A. KOZUCH.

[9] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. Performance Evaluation Review, Volume 26, Number 3, December 1998, 31-37. (Originally appeared in Proceedings of the 1998 Internet Server Performance Workshop, June 1998, 59-67.)

[10] Cost Optimal Elastic Auto-Scaling in Cloud Infrastructure. Mukhopadhyay, S.; Sidhanta, S.; Ganguly, S.; Nemani, R. R.;American Geophysical Union, Fall Meeting 2014, abstract #IN31B-3719

[11] AUTO-SCALING THRESHOLDS IN ELASTIC COMPUTING ENVIRONMENTS. Document Type and Number: US Patent Application 20160117180

[12] Axe: A Novel Approach for Generic, Flexible, and Comprehensive Monitoring and Adaptation of Cross-Cloud Applications. Jrg Domaschka, Daniel Seybold , Frank Griesinger, Daniel Baur