**Introduction**

The interest in cloud computing is increasing rapidly in the IT world. The term cloud computing refers to a set of technologies which provide access to resources as storage units, servers, and applications over the Internet. There are three main categories of cloud computing services: SaaS (Software as a Service), PaaS (Platform as a Service) and laaS (Infrastructure as a Service). Our analysis concerns laaS and, in particular, is focused on the performance of the auto scaling capabilities of the underlying infrastructure. Solutions which provide scalable and self-adaptive systems are key factors in the selection of a cloud service provider. In a cloud computing environment resources are allocated on-demand, allowing users to access to a potentially infinite amount of resources in a dynamic way. However, the resource demand usually is not uniform, but it changes according to end-user requests. Thus, the application should have the necessary resources to ensure adequate performance even in case of traffic peaks. In contrast, when the traffic decreases, it would be better to release the unnecessary resources in order to reduce cost. Therefore, dynamic scalability is a critical key point to the success of cloud computing [[1](http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=6481061#ref_1)].

Performing scaling activities manually is time consuming, so cloud systems provide solutions to automatically scale out or scale in the infrastructure on the basis of predefined policies. Policies

**AUTO-SCALING Solutions**

For an Autoscaling solution to work, it must be combined with at least load balancing and monitoring services. Monitoring services can be used to retrieve some relevant metrics on which alarms and triggers can be defined to execute custom actions on certain conditions. This allow to define scaling policies tailored to the current system status. Furthermore, an increment (scale out) or a decrement (scale in) of the resource pool size must be hidden from the users. This is achieved by using a load balancer that allows to have a static single-entry point for the application. Following are the auto-scaling solutions we currently have integrated into our system:

**AWS Auto Scaling**

Auto Scaling [[6](http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=6481061#ref_6)] is part of the services offered by Amazon in its IaaS public cloud. The core concept of Amazon Auto Scaling is the Auto Scaling Group (ASG). An Auto Scaling Group is a set of different Amazon Elastic Compute Cloud (EC2) instances sharing similar characteristics and subject to the same scaling policies. Therefore, every machine in the group have the same AMI and the same hardware characteristics. Auto Scaling groups are defined with a minimum and maximum number of EC2 instances taken from an EC2 Availability zone. Availability Zones are distinct locations within a Region that are engineered to be isolated from failures in other Availability Zones. Regions represent the geographic locations where Amazon's data centers are placed. CloudWatch [[7](http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=6481061#ref_7)] is a Web service offered by Amazon to monitor the cloud system and to keep track of various performance metrics. CloudWatch allows to define alarms for each metric to be checked. An Amazon CloudWatch Alarm is a trigger associated with a threshold on a specific metric. The threshold is associated with a comparison operator defining when to fire the alarm, e.g., when the metric is greater than or equal to the specified threshold. An alarm is always associated to an action executed when the alarm fires. This mechanism allows to define, for example, a simple notification system: it is sufficient to specify the action of sending emails whenever an alarm fires. Moreover, CloudWatch Alarms can be used to define scaling policies on instances belonging to the same ASG. For example, it is possible to specify a constant increment of one instance or a percentage increment of 100% of the current number of instances within the group, that is equivalent to doubling the group size. It is also possible to specify the cooldown period within which no other scaling activities can take place, so that the effects of a scaling activity can become visible avoiding collisions with other conflicting scaling policies. Amazon also provides load balancing (*Elastic Load Balancing*- ELB) service.

**Kubernetes Horizontal Pod Auto scaling**

Kubernetes 1 provides the means to support container based deployment within Platform-as-a-Service (PaaS) clouds, focusing specifically on cluster-based systems. Kubernetes enables deployment of multiple “pods” across physical machines, enabling scale out of an application with dynamically changing workload. This part autoscaling process of Kubernetes is called as Horizontal Pod Autoscaling. Each pod can support multiple Docker containers, which are able to make use of services (e.g. file system and I/O) associated with a pod. With Horizontal Pod Autoscaling [7], Kubernetes automatically scales the number of pods2 in a replication controller3, deployment or replica set based on observed CPU utilization (or, with alpha support, on some other, application-provided metrics). The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user.

The Horizontal Pod Autoscaler is implemented as a control loop, with a period controlled by the controller manager’s **--horizontal-pod-autoscaler-sync-period** flag (with a default value of 30 seconds). During each period, the controller manager queries the resource utilization against the metrics specified in each HorizontalPodAutoscaler definition.

The HorizontalPodAutoscaler controller fetch metrics through direct Heapster4 access. The HorizontalPodAutoscaler queries Heapster directly through the API server’s service proxy sub resource. Heapster needs to be deployed on the cluster and running in the kube-system namespace. The Autoscaler accesses corresponding replication controller, deployment or replica set by scale sub-resource. Scale is an interface that allows to dynamically set the number of replicas and examine each of their current states.

There are two different types of load balancing in Kubernetes

Internal – aka “service” is load balancing across containers of the same type using a label. These services generally expose an internal cluster ip and port(s) that can be referenced internally as an environment variable to each pod.

External –Services can also act as external load balancers if you wish through a NodePort or LoadBalancer type. NodePort will expose a high level port externally on every node in the cluster. By default somewhere between 30000-32767.

As Kubernetes uses underneath Containers, which provide the most appropriate mechanism for cloud native applications, enabling rapid spawning and termination compared to Virtual Machines (VMs).

**Docker-Swarm Autoscaling**

Docker Swarm is native clustering for Docker. It turns a pool of Docker hosts into a single, virtual host. As of know it doesnot support Autoscaling but we have developed a prototype of it. The load balancer we used is Crossroads [[19](http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=6481061#ref_19)], a widely used open-source load balancer, that processes all the incoming requests and redirects them to active VMs. Then we ran the cadvisor metrics collection system to collect the necessary metrics (like Avg CPU Utilization, Container start/stop time) based upon which we used the swarm replica feature to add or remove containers.

**Aws Autoscaling Merged with Kubernetes Horizontal Pod Autoscaling**

This solution is a merged solution of both the AWS Autoscaling(IaaS) and Kubernetes Horizontal Pod Autoscaling(PaaS). With the combination of both, this virtually guarantees that the desired # of instances will always be running. Even if some instance goes down, AWS will create a new one.   Kubernetes minions can be configured such that they automatically join a cluster on boot-up.This guarantees that the cluster will keep having resources it needs to continue to run workloads. In the case of the Kubernetes master running in an ASG, the master node can restart even if it goes down making the cluster more available.  Or, the master can be scaled up when the load on the cluster is high.Metrics from both the system can be utilized to see the performance of the application on such a system.

SECTION III

**System Architecture**

This section describes the overall architecture of the system along with the detailed individual Autoscaling solution deployment architecture. The system comprises of 4 main parts as shown in the diagram. Each part is built upon the design of microservices and can be individually scaled up or down based upon the requirement.

Application Type

Load Generator

NoSQL

Database

Autoscaling Solution 1

Monitoring Service

Autoscaling Solution N

Monitoring Service

Following is the brief description of each part:

1. **Type of Application:** For the performance test of different Autoscaling solutions we have added different types of application but not limited to these in the following categories:
   1. CPU-Intensive Application
   2. I/O Application
   3. High Memory Usage Application

These applications allow to test the different Autoscaling policies of the solutions. The user can select any of the above based upon his/her need to check the performance of the Autoscaling solution.

1. **Load Generator:** The workload is generated by the Loadtest application built in nodejs and customized and integrated it with this system. Now based upon the application selection the user has the choice to configure what real world load he want to generate. We have added the feature that if the load generated is high enough then automatically some slave nodes are created to serve the load, so that a single node does not becomes the bottleneck. Also, to simulate the real world load we have already added some pre-readymade load configurations which the user can directly select and test the Autoscaling solution. However, it’s not limited to those predefined configuration, the user has the choice to configure following parameters and make his/her own configuration file:
2. Number of Concurrent Clients: Number of clients to start in parallel.
3. maximum number of Requests: A max number of requests after they are reached the test will end.
4. maximum Seconds: Maximum seconds for which the test is to run
5. Each request timeout: Timeout for each generated request in milliseconds.
6. method: The method to use: POST, PUT. Default: GET.
7. body: The contents to send in the body of the message, for POST or PUT requests.

Can be a string or an object (which will be converted to JSON).

1. content Type: The MIME type to use for the body. Default content type is text/plain.
2. requests Per Second: The number of requests each client will send per second.
3. **Autoscaling Solutions:** Currently we have considered 4 Autoscaling solutions, which are integrated into system. These solutions can be deployed with different configurations and terminated automatically from the system’s easy to use user interface. As part of each Autoscaling solution monitoring service is attached with them to collect the performance metrics and store in database. The deployment of each solution is done on Amazon Cloud with the parameters configurable from the UI. Following is the deployment architecture of each solution along with the configurable parameters.
   1. **Amazon Autoscale Solution:** As part of this solution following parameters are configurable:
      1. Type of Instance
      2. Minimum and Maximum No of Instances to scale in and out.
      3. Scaling Decision Metric (Among the following choices):
         1. CPU Utilization
         2. Disk Reads
         3. Disk Read Operations
         4. Disk Writes
         5. Disk Write Operations
         6. Network In
         7. Network Out
      4. Threshold for scale in and scale out
      5. Auto-Scaling Policy
         1. Simple Scaling
         2. Step Scaling

Based upon these parameters Autoscaling group on AWS is created and deployed with all the necessary. As part of this solution we use Amazon Cloud Watch service to collect the metrics for the whole Autoscaling group and individual EC2 instances. Our system after periodic time fetches the data from the Cloud Watch service and store in the Database.

* 1. **Kubernetes Horizontal Pod Scaling:** In Kubernetes the VMs don’t Autoscale so the total number of VMs are fixed in the Cluster. Also, as of now only the CPU Utilization parameter is supported as the Autoscaling decision metric in Kubernetes. As part of this solution following parameters are configurable:
     1. Type of Instance
     2. Number of Instances to include in the Kube Cluster.
     3. Scaling Decision Metric
        1. CPU Utilization
     4. Minimum and Maximum Number of Pods for scale In and Out
     5. Threshold for scale in and scale out

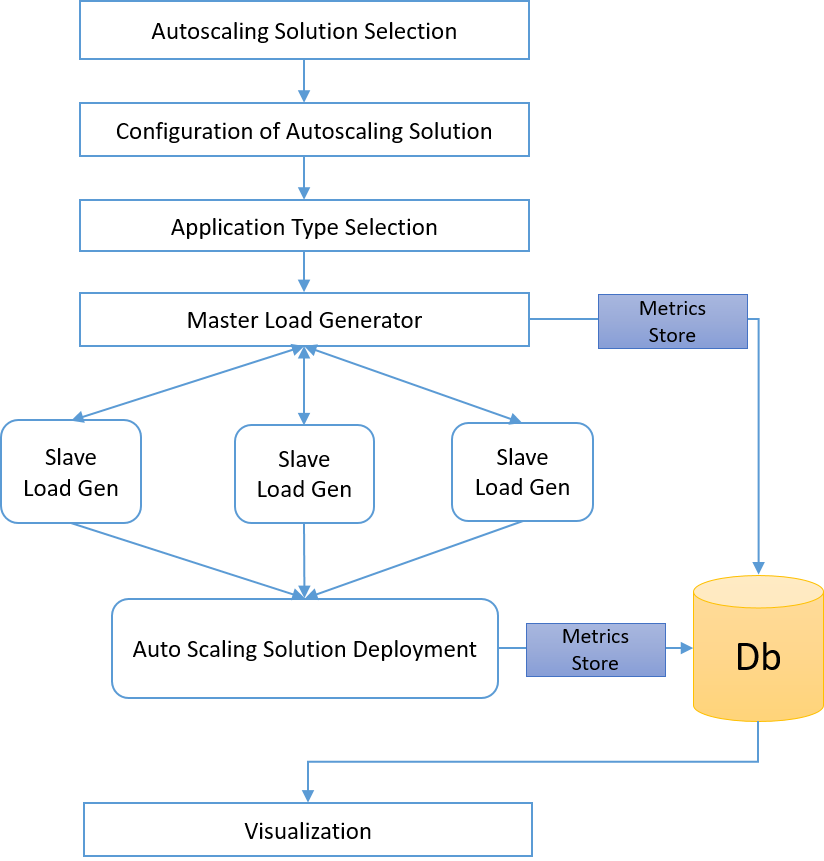
Based upon these parameters Kubernets cluster is created on the AWS using the EC2 instances. For collecting the metrics, we use the Heapster service along with the influx db to store the metrics natively on the machine. These both services are deployed as part of cluster. Our system after periodic time fetches the data from the Heapster service and store in the Database.

* 1. **AWS Autoscaling merged with Kubernetes Horizontal Pod Scaling:** As Kubernetes was having the limitations of VMs scaling, however with the merging of AWS Autoscaling with the new instance which is started as part of it is configured to get connected to Kubernetes cluster and terminate when scale down happen. This solution has the mix of the parameters to configure both the solutions together. As part of this solution following parameters are configurable:
     1. Type of Instance
     2. Minimum and Maximum No of Instances to scale in and out.
     3. AWS Autoscaling Scaling Decision Metric (Among the following choices):
        1. CPU Utilization
        2. Disk Reads
        3. Disk Read Operations
        4. Disk Writes
        5. Disk Write Operations
        6. Network In
        7. Network Out
     4. Threshold for scale in and scale out for AWS Autoscaling
     5. Aws Auto-Scaling Policy
        1. Simple Scaling
        2. Step Scaling
     6. Kubernetes Autoscaling Decision Metric
        1. CPU Utilization
     7. Minimum and Maximum Number of Pods for scale In and Out for Kubernetes
     8. Threshold for scale in and scale out for Kubernets horizontal pod scaling

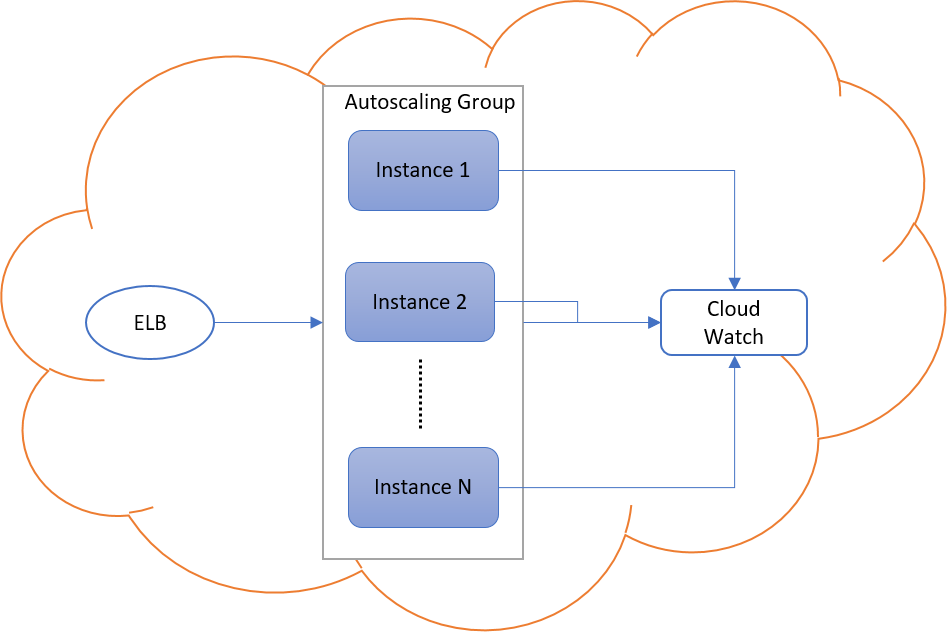
Based upon these parameters Kubernets cluster along with the AWS Autoscaling group is created on the AWS cloud. For collecting the metrics, we use both the Amazon Cloud Watch service (to get the metrics for AWS Autoscaling group) and the Heapster service along with the influx db to get the metrics about the Kubernets cluster. Our system after periodic time fetches the data from the Heapster service and store in the Database.

1. **NoSQL Database:** We use MongoDB to store all the performance metrics collected as part of the process. These Performance metrics are then used to visualize Autoscaling process along with the other performance metrics.

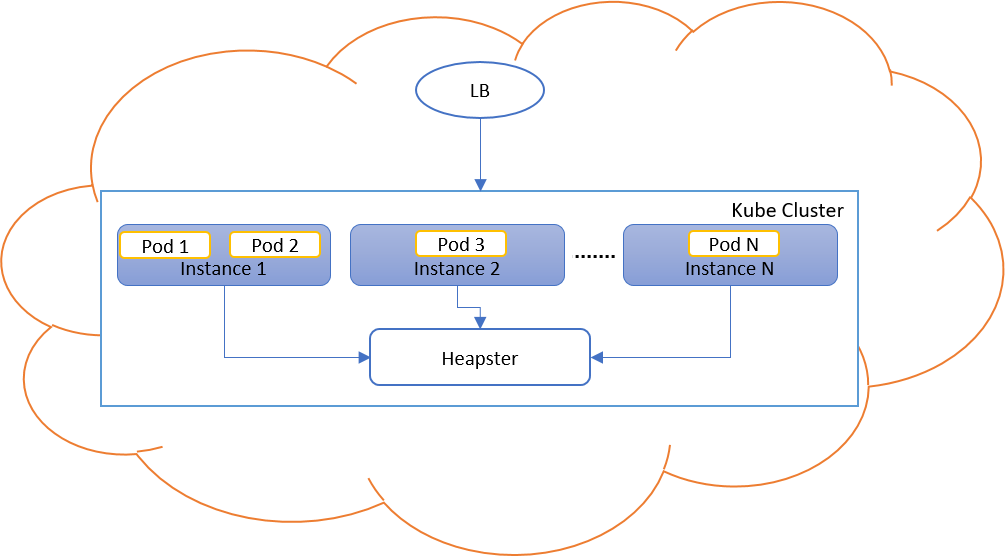
**System’s Functioning**

****

AWS Autoscaling



Kubernetes Horizontal Pod Autoscaling



**Performance Metrics/ Result**

1. **AWS Autoscale**
   1. CPU usage per VM instance
   2. Memory usage per instance.
   3. Start and Stop time duration for the VM instance.
   4. Success, Failure, and Restart rate of each VM
   5. Number of Instances
   6. CPU-Utilization for Autoscaling group
   7. Number of requests served per second (IOPS).
   8. Min, Max, and Mean Latency for the requests
   9. Error rate for the requests.
   10. Time Taken to complete each request
2. **Kubernetes**
   1. CPU usage per VM instance
   2. Memory usage per instance.
   3. CPU usage per pod
   4. Memory usage per pod
   5. Pod Timeline
      1. Assignment to VM
      2. Mount Volume
      3. Image Pulling
      4. Image Pulled
      5. Container Creation Request
      6. Container Created
      7. Killing
   6. Hpa schedule timeline
   7. CPU utilization for cluster
   8. Number of requests served per second (IOPS).
   9. Min, Max, and Mean Latency for the requests
   10. Error rate for the requests.
   11. Time Taken to complete each request
3. **AWS Autoscale and Kubernetes Together**