Autoscaling for Hybrid Cloud

[ECE 792] Linux Networking Project

Tapas Mallick (trmallic) *Mandhani Kushal (mkushal)*
Pardhakeswar Pacha (ppacha) *Suhas NG (snaramb)*

.

1.Introduction

1a. Background

In the present world where every application is expected to be agile at all times, the problem of handling the resources for these applications often raises a big concern. A heavy outburst of traffic on a physical server hosting an application will lead to high latency and poor performance of the servers thereby reducing the agility of the application. These performance issues affect both single-tenant and multi-tenant deployment models and thus raise a serious concern.

1b. Problem Statement

For instance, consider a movie booking application server hosted on a Virtual Private Cloud residing within an enterprise's network. On a weekend we often find surge in the number of requests to the server. This may cause the load on the CPU to increase and the performance of the server may go down and it is also possible that there could be downtime if the surge in traffic load is massive.

One solution to this problem could be to increase the number of servers hosting the application. However, maintaining the additional resources will cost more to the enterprise. Additionally, the new servers will be idle or underutilized after the traffic surge reduces. To solve this, we should have a mechanism to add resources only when there a reduction in performance. Also, it is of paramount importance that this mechanism should be automated. It should be able to monitor the certain performance metrics and automatically scale the resources accordingly.

1c. Project Description

We have provided a solution to this problem by implementing Autoscaling on a Hybrid Cloud environment. If an application within a Virtual Private Cloud environment is experiencing a heavy traffic, a decision will be taken to create more resources for the application on another virtual private cloud. Load Balancing servers now shares the traffic with the newly collected VM(‘s) to bring the load to a limit. These new resources could be a Virtual Machine or a Container. Note that these resources could now reside on different cloud platforms and they can scaled in a hybrid cloud environment.

Auto-scaling also removes additional resources when they are no longer essential. For instance, whenever the traffic is back to normal limits, a decision can be taken to remove these additional resources. It is a cost effective scheme as the enterprise only needs to pay for additional resources during the time of need.

1d. Summary of the Report

As part of this report, we will start with discussing about some of the existing solutions and the need for a new solution. We will then proceed to discuss about this solution in detail covering the functional and management features, deployment topology and environmental constraints. In the Implementation architecture section, we will discuss VM and Container deployment models along with a detailed description about Northbound, Southbound and Logic Layer. This section also has a User and Developer guide. Next section discusses the results and showcases the solution. Last section summarizes the solution and discusses future scope.

2. Related Work

This service is inspired from some of the existing solutions in the industry. Companies such as amazon, cisco and sap have developed their own version to support auto-scaling for tenants on their infrastructure. We will discuss about these solutions and their shortcomings.

2a. SAP Autoscale Service

SAP has deployed an application autoscaler, which is a service designed to scale up or scale down the bound on the application instance based on user-defined policies. This service observes the performance and spontaneously scales the resource and prevents the application from crashing. The process of using the autoscaler here involves 3 steps. First is the creation of the application. Second we create a policy for autoscaling. Lastly, we will bind the application to the instance of the autoscaler.

The limitation of this autoscaling solution is that it does not allow the tenant to connect to his custom VPC. This limits the tenant's ability to utilize the cloud resources in a flexible way. The auto-scaling service works mostly with applications which are owned by SAP. It might be difficult to find support for applications other than these.

Apart from this, customized policy configuration is very tedious and the user needs to provide lot for scripts in JSON to deploy the policy. Also, the auto-scaler is not that flexible as it does not monitors most of the parameters. This probably restricted the application only within sap customers.

2b. Amazon Auto Scaling

Amazon EC2 has rich features for auto-scaling to monitor a variety of metrics such as cpu, memory and so on. It also has policies which allow both dynamic and static schedulability. It offers benefits for fault tolerance, better availability and cost management.

With all these features available there are certain scenarios where the autoscaler cannot meet the tricky needs of the customers. The first is that, these feature are offered only if the application runs on the amazon web services cloud. If there is any tenant who wants to use his own VPC then it cannot scale it onto these subnets. This actually would be a cheaper option to port the application to may be another cheaper cloud during the peak load.

This would not work well for a tenant who would like to burst to public cloud only during need of excess resources. Even though AWS provides a lot of image customization options, very few of them are effective and the option to *build your own image* is not really useful due to the challenges it presents. This is a big limitation for tenants with very specific image needs.

Another point we would like to highlight is that Amazon Auto Scaling is loaded with almost all the features we can think of in a cloud environment. But not all of these features are highly flexible. The services offered by Amazon Autoscaling aren't flexible enough as opposed to the flexibility of tweaking these features enjoyed in a linux environment.

2c. Cisco Intercloud Fabric Service:

Back when Cisco still offered this service, Intercloud fabric offered services to extend a network over public cloud securely. There was a flexibility to choose the some of the popular providers, Amazon Aws and Microsoft Azure but it did not provide services for other providers. If a tenant needs to extend their private network (VPC) to a different cloud provider or a different data center of their own, this solution does not work. This solution is strictly for cloud-bursting from a enterprise network to a cloud environment.

Also, the solution provided by Cisco for achieving auto-scaling requires the tenants to bear high costs of setup and maintenance. Cisco intercloud fabric service requires a customer to have one dedicated Virtual machine for Intercloud fabric extender that runs in private cloud and one in provide cloud for Intercloud Fabric switch. This is an overhead in the Tenant's VPC.

It was probably due to these reason that the Cisco's solution was never able to onboard a large number of tenants and has been shut down. However, this solution provided all the elements essential for the enablement of auto-scaling in the tenant's infrastructure.

3. Project description

3a. Objective:

This solution focuses on improving the performance aspect among the Cloud Service Attributes. This service has been designed with the intention to take smart decisions governed by a set of predefined parameters. The idea is to leverage the information available in the form performance metrics and maximize the performance of the complete cloud infrastructure. Another major objective of this work is to interact and experiment with different cloud deployment models like multi-cloud environments. Services like these open up opportunities to build on the existing infrastructure.

3b. Proposed Features

This project enables a VPC hosted application to leverage the resources of another VPC residing on either a public cloud or a private cloud. This happens when the load is in excess of the capacity of the on-premise servers. The following is the list of features included as part of the Project.

a. Functional features

- The load is defined as excess when the performance metrics show a decline in resource performance.
- We consider the following three aspects as the key to making wise scaling decisions:

    - When to scale up
    - How much to scale up
    - Where to scale resources
- We define maximum and minimum thresholds to make scaling decisions on when to scale.
    - Scale up: Load>MaxTh
    - Scale Down: Load<MinTh.
    - Idle: MaxTh > Load > MinTh

      MaxTh:Maximum Threshold
      MinTh:Minimum Threshold
- Once the scaling decision is made, there are two factors to consider:
    - How many additional resources are required or to be removed.
    - Where to scale: Find the best hypervisor based on it's performance. VM's on this hypervisor with:
        - Preferably Load<minTh
        - Atleast Load<MaxTH
- Auto-scaler captures the performance data and monitors against the configured thresholds. If excess load is determined, this thread configures the additional resources to be leveraged by the Tenant's application. Ex: It creates Server VMs/Containers on the standby VPC hypervisor with less load.
- Primary VPC contains the application servers(VMs/Containers), Network devices(VMs), Load Balancer(VM), DNS Server(VM).
- Standby VPC should be ready with minimum required components (required packages). The tenant's can use the linux image provided to them. This VPC could be located on any data center as long as there is connectivity to the Primary VPC hypervisor.
- Cooldown period for ongoing requests before removing the newly added resources.
    - A tenant configurable grace period enables graceful shutdown of the newly added resources.
    - The auto-scaler waits for this period before destroying the VMs. However, no new requests are given to these resources in grace period.

b. Management Features

- **Self Healing** - inspired by Kubernetes - is a service that helps restart Containers when they fail the health check.
    - This service monitors all the containers on all hypervisors to identify failed containers and respawns them on a free host.
    - This service will detect any container that does not have a valid performance metric value. This will happen usually when the container is shutdown or deleted.
    - This will also respawn all containers on a affected Host onto other Hosts. This is because when the Host is down, the Containers will not have a valid performance metric value and will be respawned.

- **Logging –** Logging all the events that occur in the system.
    - The logged events include:
        - Scale Up
        - Scale Down
        - Failed Containers and Healing
    - Two formats of logs available:
        - Csv format
        - Syslog format
- **Configurability:** Tenants has the flexibility to design their network and provide the maximum and minimum threshold values.
    - Network Design Input: Tenant administrators can provide an input json with the subnet details as per their needs.
- **Manageability**: Tenants have the ability to manage and maintain their system on the go.
    - CLI interface to update the Minimum and Maximum Threshold and Cooldown periods.
- **Dynamic Monitoring:** This feature is designed to give a snapshot of the Tenant's system to both Provider and the Tenant himself.
    - The complete service is designed and tweaked in such a way as to store complete information in a single json file.
    - It stores the state of the Tenant's system. Ex: status is scaled up and 6 additional VMs residing on two different hypervisors.
    - In multi-cloud deployments, there is a need for building trust between Provider and Tenant using smart-contracts.
    - This feature can be the building block for devising a standard in multi-cloud deployments.

Steal Time vs Average CPU Load:

We would like to discuss an interesting trade-off regarding the performance evaluation metrics. In a VM based environment, steal time indicates how much the resources have been stolen from the VM by Hypervisor. In other words, it shows the load on Hypervisor. Although this looks like a very good performance metric for making scaling decisions, it has a limitation. Consider a case of multi-tenant deployment on a hypervisor. If Tenant 1 has low load but sees more steal time due to more load on Tenant 2, the auto-scaler would create resources to avoid performance issues. Even though there was no load on Tenant 1, it had to scale due to the shared environment. Please note that this does not apply to single-tenant deployments.

4.Implementation architecture

In this project we have two deployment models, one is container based and another is a VM based environment.

VM-based Deployment

In case of VM based environment, the network infrastructure comprises of two hosts, host-1 and host-2. Host 1 represents the main VPC and the Host 2 represents the standby VPC. Each tenant is provided a Controller VM with a bridge in NAT/DHCP mode. The tenant is provided with the functionalities of creating his own topology

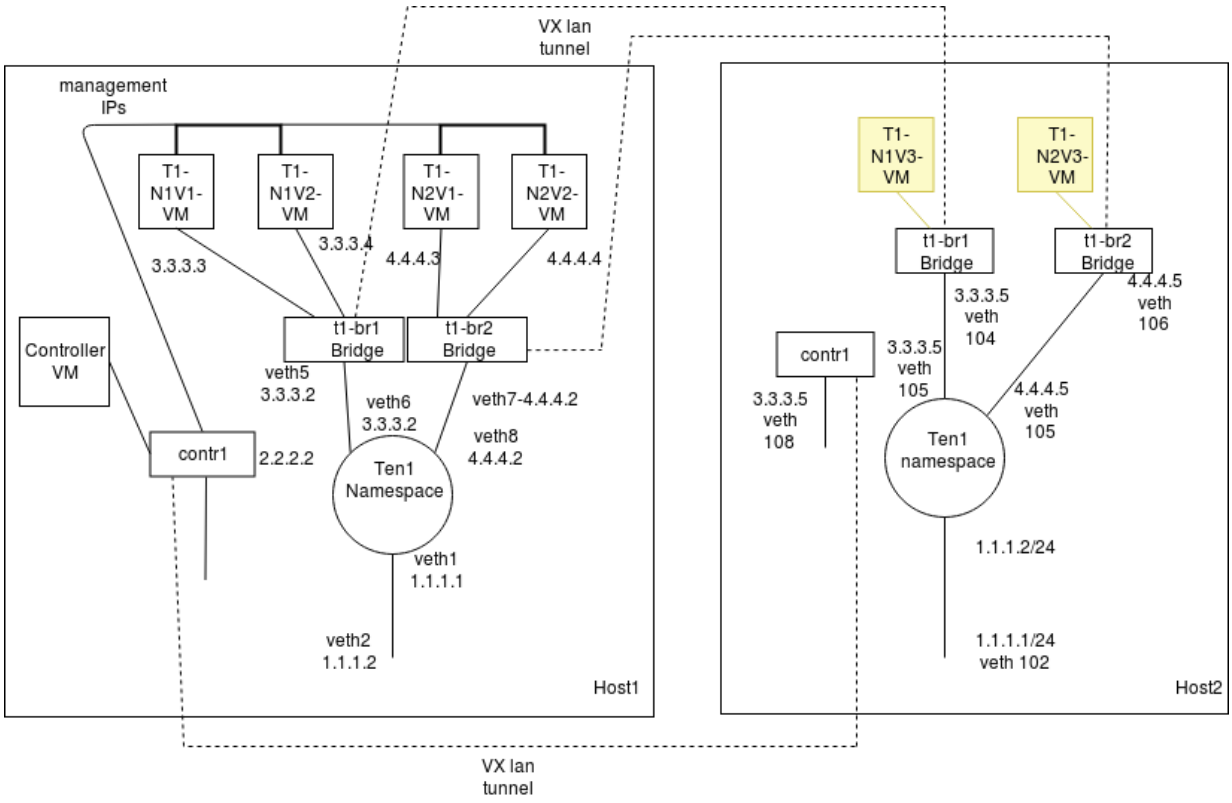


Figure representing the network infrastructure for the VM based environment.

using the input json file. There are four key aspects in this overview topology: L3 Isolation using Namespaces, L2 Isolation using Bridges within a Tenant's subnets, VxLAN connectivity to provide tunnelling across hypervisors and a management network for management activities.

L3 ISOLATION BETWEEN TENANTS:
To provide L3 isolation for each tenant a dedicated namespace is created and all network bridges of the same tenant are connected to this namespace. The namespace also acts as a load-balancer for each tenant. This is accomplished using iptables in the namespace.

L2 ISOLATION BETWEEN SUBNETS:
To provide L2 isolation between subnets of the same tenant, a dedicated bridge is used for each subnet of the tenant.

VxLAN Connectivity:

The second host is configured with a Controller bridge and the Controller bridge from host 1 is connected to the controller bridge on host 2 via VxLAN. This provides logical isolation for the Controller VM to span the VM's across hosts under the same tenant. VxLANs are also created between the bridges of applications within the tenant to provide L2 connection between the VM's belonging to the same subnet across different hypervisors.

MANAGEMENT NETWORK & CONTROLLER:
A dedicated management network per Tenant is created and each tenant has a dedicated Controller VM to host the auto-scaling service. Each VM has two interfaces - One for hosting its applications/services and the other for management activities. All the VMs within a Tenant can be reached from the Controller VM over the management network.

Container-based Deployment

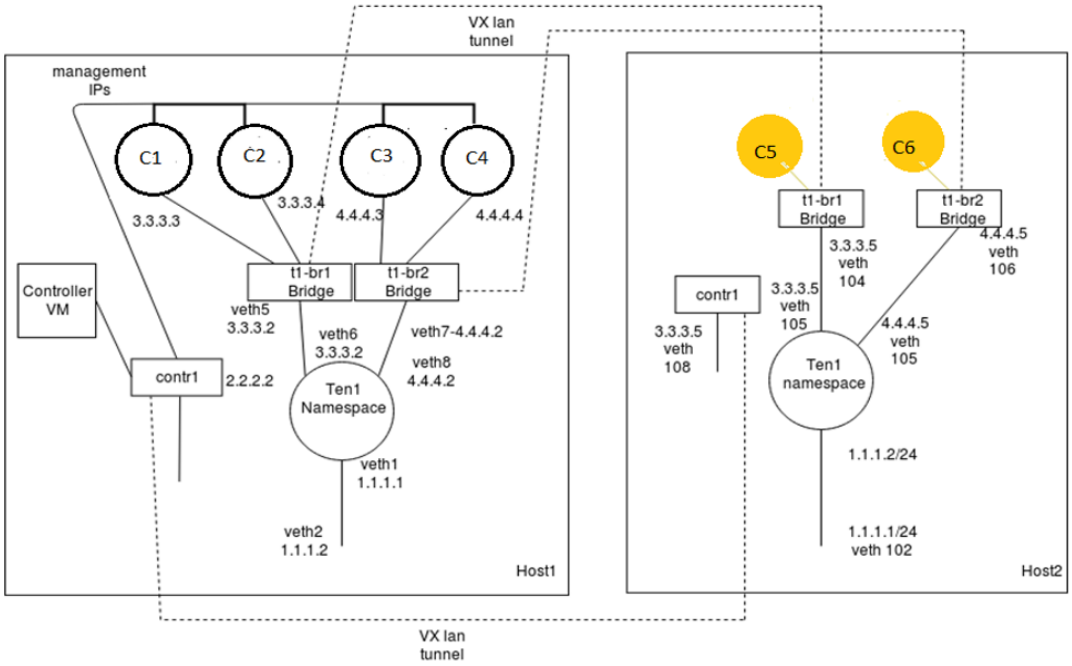
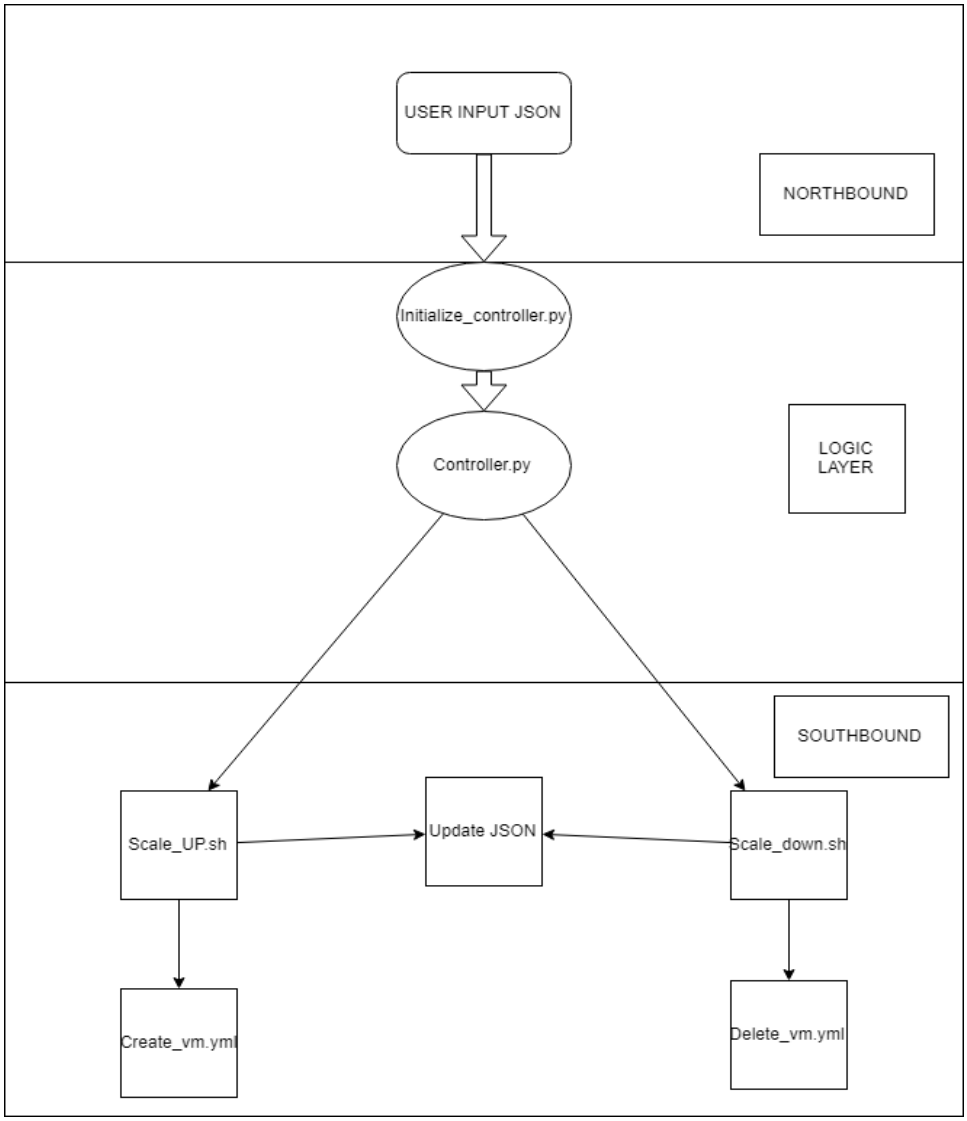


Figure representing the network infrastructure for the Container based environment.

The deployment model is similar to VM-based deployment. We have used dockers to create and manage the Containerized environment. Instead of connecting the VMs to the Networks, we now connect the Containers to the Bridges using veth pairs.

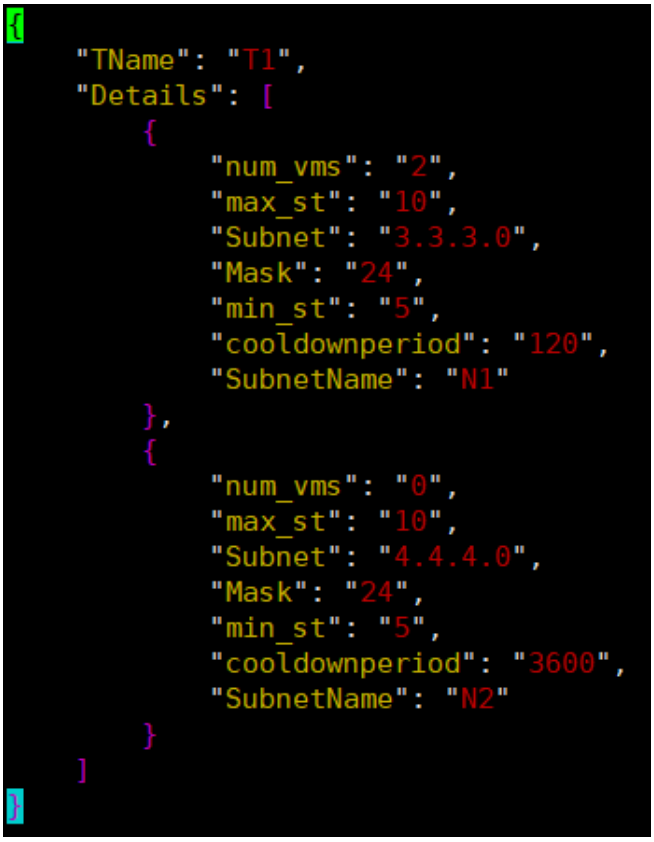


Logical Layer Separation

4a. Northbound Interface

VM:

The Northbound Interface in this architecture includes the input JSON file provided by the Tenant. This file has the subnet details of the tenant and this is used as an initial schema to build the Tenant's network from scratch. The Northbound Interface gives the Tenant Administrator the flexibility to configure auto-scaling logic.



Sample Input JSON

Containers:

In case of VM and Container the northbound remains almost same. The only change is in some of the threshold values like cpu thresholds instead of steal time thresholds. These parameters are passed to the lower layer by the north bound interface some of the inputs are as follows:

a. Initial Number of VMs: This refers to the initial number of VMs that tenant want to have in each subnet.
b. Threshold values: It includes the values for minimum steal time and maximum steal time to monitor in the Guest VM spawned in the host to scale down and scaleup respectively. The steal time is in terms of percentage.
c. Cooldown Period: This is time to wait after scale up or scale down before another scaling action is done. The input time is given in seconds.
d. Subnet Details: The User needs to provide details such as subnet address and mask.

These parameters are stored in a JSON format. A python application parses these JSON file.

Core logic : The fundamental questions when it comes to autoscaler is when to scale, where to scale and by how much. The figure below represents the flow chart for the core logic.

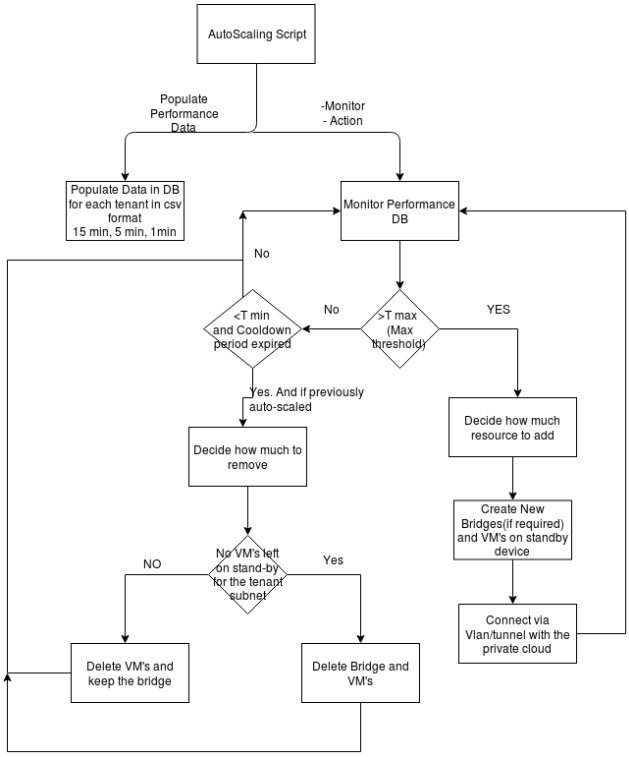


Fig 2.1: flowchart for autoscaling

VM:

<u>When to scale</u>

There are two types of scaling; scale up and scale down. When the steal time in the VM increases above the maximum threshold it is going to scale up. Also, the autoscaler checks if the other host is overloaded and not above the max threshold. When the steal time is below the minimum threshold then it is eligible for scale down. Additionally, while scaling down the cooldown time needs to expire, and only then can the scale down can start. There is also a third state where no scaling occurs. This is when the steal time is between maximum and minimum value.

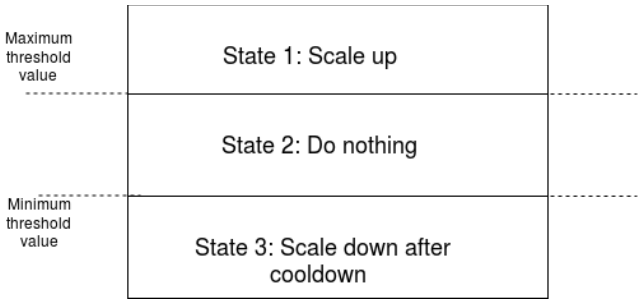


Figure 2.2: Describing the possible states of operation of the autoscaler.

A python script is used to implement this core logic. When the script gets the steal time data it makes the decision to scale up or scale down. An ansible script is called for scale up or scale down. In a nutshell, the ansible script for scaling up creates a new VM in Standby VPC, attach it to the bridge and it will configure the IP for the VM and the scale down script undefine the VM and remove the link from the bridge. The script is also going to modify the entries in the load balancer accordingly while scale up and scale down such that the requests are direct to the VMs in the subnet.

Where to scale :

The scaling can be done within any of the possible host. The major criteria for deciding on which hypervisor to create new VM is through the steal time. The steal time of all the VM of the host is collected and then it determines whether the steal time is greater than the maximum threshold. If it exceeds the value it means that the hypervisor is overloaded. The autoscaler will not create the VM for that host and instead moves on to the next host. But if the hypervisor is not overloaded above the threshold value then the autoscaler will create a new VM and does the necessary configuration changes at load balancer and bridge.

How much to scale:

Only 1 VM is created or destroyed for a single scale up and scale down respectively. Even if the load is high the subsequent iteration of the script execution ensures that appropriate number of VMs are created to distribute the load and bring the steal time within the threshold values.

Containers:

The major difference between the core logic of VMs and Containers is the metric of performance evaluation. Since Container is a process, steal time cannot be used as a metric anymore. In this case, a slight design change was made to include this. The average load of Container CPU usages per hypervisor per subnet has been used to take scaling decisions. The thresholds have been defined accordingly. All the other logic layer parts remain same as VMs.

Southbound interface:

VM:

The Southbound Interface mainly consists of three components:

- Create Resources
- Delete Resources
- Collect Performance Metrics

In this deployment model, we have used collectd to measure steal time and configured the VMs to report their cpu steal time percentages to the Controller VM. The management network is utilized in this flow.

Create Resources:
Scale_up.sh is a southbound script that takes all the required parameters and creates a VM on the specified Hypervisor. This is script also updates the JSON file to report this change in the state (Dynamic Monitoring). Apart from this, the scale_up.sh script sets up collectd on the newly created VM. Load Balancer is also setup to accommodate this change.

Delete Resources:
Scale_down.sh is another southbound script that takes care of removing an excessive resource. This script also updates the JSON file to record the event to support Dynamic Monitoring. Load Balancer is configured accordingly.

Other Miscellaneous scripts:
1) Creation of new VM: scaleup.sh calls create_vm.yaml
2) Delete VM. scaledown.sh delete_vm.yaml
3) Updation of Load Balancing rules. LoadBalancer.sh
4) Enabling Collectd. start_collectd.sh collectd.yaml

Containers:
Dockers were used to manage the creation and deletion of VMs.
Create Resources:
Scale_up.sh is a southbound script that takes all the required parameters and calls docker to create a Container on the specified Hypervisor. This is script also updates the JSON file to report this change in the state (Dynamic Monitoring). Load Balancer is also setup to accommodate this change.

Delete Resources:
Scale_down.sh calls docker to remove an excessive resource. This script also updates the JSON file to record the event to support Dynamic Monitoring. Load Balancer is configured accordingly.

Other Miscellaneous scripts:
1) Creation of new Container: scaleup.sh calls docker commands
2) Delete VM. scaledown.sh
3) Updation of Load Balancing rules. LoadBalancer.sh
4) Getting Performance Data from Dockers: Used Docker stats to calculate average CPU load.

User Guide:
This service is a very easy to use cli based service. The user needs to provide an input JSON and call main.sh. This JSON has all details about the subnets required in the Tenant's network.
Usage Steps: Run this command from the directory after extracting the contents.
./main.sh <schema.json>

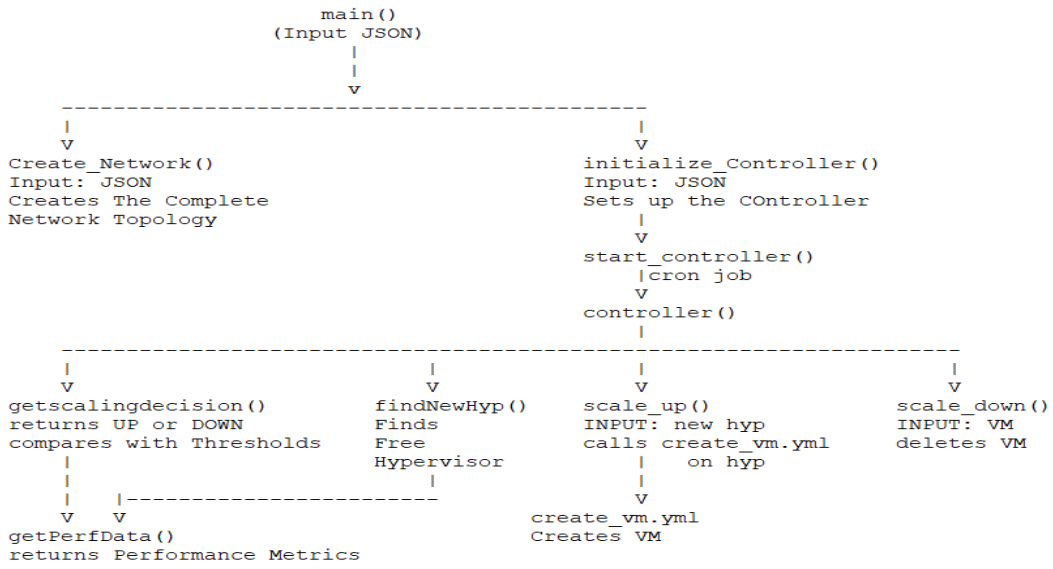
Controller VM:
This VM is your all-in-one access to your complete network. You can reach all the VMs in your network from this VM. Note that you will not be communicating with the VMs over the default datapath. There is a dedicated management network for you to access the VMs.

The CLI Interface can be used to modify the Threshold values and cooldown period.

Dynamic Monitoring:
You can monitor the status of your network by simply checking the schema.json located at /root/autoscaler directory. Look for the fields:
>status: “idle” or “scaledup” indicates the overall status
>scaleduptime: Shows time of last scale up event.
>num_scaled_vms: Number of additional VMs/Containers created.
>VM-specific flags like “scaledupresource” and “scaleduporder” help in identifying whether the resource is scaled up or permanent.

Developer Guide:

The initial provisioning(southbound interface) works as per the following functional diagram:



create_network():
Brings up the tenants complete infrastructure (Bridge , network, VMs DHCP, VxLANs and Namespaces) and configures Load Balancers. Also adds all tenant related data to JSON. specifically standby host details.
initialize_controller():
Creates the Controller VM and creates the initial VMs/containers based on parameters. This function also sets the timer on controller() by calling start_controller().

getScalingDecision():
The Decision maker function reads the Thresholds and current performance metrics and decides whether to scale or not. Calls getPerfData()
getPerfData():
This method returns steal time in case of VMs and returns average of CPU loads on Containers. This is also used by findNewHyp()
findNewHyp():
Decides where to scale when the controller decides to scale.
Scale_up() and scale_down():
Creates/Deletes VMs/Containers on the given hypervisor. These methods also update the Load Balancing rules in the Namespace.

Dynamic Monitoring for developers:

As mentioned earlier, the schema.json keeps the current snapshot of the system. This snapshot can be used to keep track of the system by keeping track of changes to this file. This file can be accessed as an API by hosting a static resource on the preinstalled tomcat server on the controller VM.

Scripts can be built to call this API on a periodic basis to keep track of changes to the system and log any events that may occur.

5. Evaluation:

Test Case 1:
Feature: Tenant Onboarding
Input: JSON File provided by the Tenant.
Expected Output: Successful creation of Tenant Infrastructure including Namespace, Management Network, Subnets, Controller VM.
Actual Output:

```
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/M3Code/VM/onboarding$ sudo ./main.sh T2schema.json
Current Directory: /home/ece792/M3Code/VM/onboarding
Using Input JSON file: T2schema.json
Tenant ID: 1
Creating mgmt Network
Network C-T2 defined from XMLs/C-T2.xml

Network C-T2 started

Found main hypervisor in hypervisor list

PLAY ***************************************************************************

TASK [Create the bridge] *******************************************************
changed: [192.168.50.66]

TASK [Bring up the bridge] *****************************************************
changed: [192.168.50.66]

TASK [Define the network 'C-T2'] ***********************************************
changed: [192.168.50.66]

TASK [Start the Network] *******************************************************
changed: [192.168.50.66]

TASK [Create the VXLAN interface] **********************************************
changed: [192.168.50.66]

TASK [Bring up the VXLAN interface] ********************************************
changed: [192.168.50.66]

PLAY RECAP *********************************************************************
192.168.50.66              : ok=6    changed=6    unreachable=0    failed=0

Created mgmt Network
Creating NS
Created NS
Creating subnets
Network T2N1 defined from XMLs/T2N1.xml

Network T2N1 started
```

```
Created mgmt Network
Creating NS
Created NS
Creating subnets
Network T2N1 defined from XMLs/T2N1.xml

Network T2N1 started

Found main hypervisor in hypervisor list

PLAY ***************************************************************************

TASK [Create the bridge] *******************************************************
changed: [192.168.50.66]

TASK [Bring up the bridge] *****************************************************
changed: [192.168.50.66]

TASK [Define the network 'T2N1'] ***********************************************
changed: [192.168.50.66]

TASK [Start the Network] *******************************************************
changed: [192.168.50.66]

TASK [Create the VXLAN interface] **********************************************
changed: [192.168.50.66]

TASK [Bring up the VXLAN interface] ********************************************
changed: [192.168.50.66]

PLAY RECAP *********************************************************************
192.168.50.66              : ok=6    changed=6    unreachable=0    failed=0

Network T2N2 defined from XMLs/T2N2.xml

Network T2N2 started

Found main hypervisor in hypervisor list

PLAY ***************************************************************************

TASK [Create the bridge] *******************************************************
```

```
Found main hypervisor in hypervisor list

PLAY ***************************************************************************

TASK [Create the bridge] *******************************************************
changed: [192.168.50.66]

TASK [Bring up the bridge] *****************************************************
changed: [192.168.50.66]

TASK [Define the network 'T2N2'] ***********************************************
changed: [192.168.50.66]

TASK [Start the Network] *******************************************************
changed: [192.168.50.66]

TASK [Create the VXLAN interface] **********************************************
changed: [192.168.50.66]

TASK [Bring up the VXLAN interface] ********************************************
changed: [192.168.50.66]

PLAY RECAP *********************************************************************
192.168.50.66              : ok=6    changed=6    unreachable=0    failed=0

Created subnets
```

```
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/M3Code/VM/onboarding$ sudo virsh list
 Id    Name                           State
----------------------------------------------------
 188   Controller-VM-T1               running
 189   Controller2-VM                 running
 190   Controller-VM-T2               running

ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/M3Code/VM/onboarding$
```

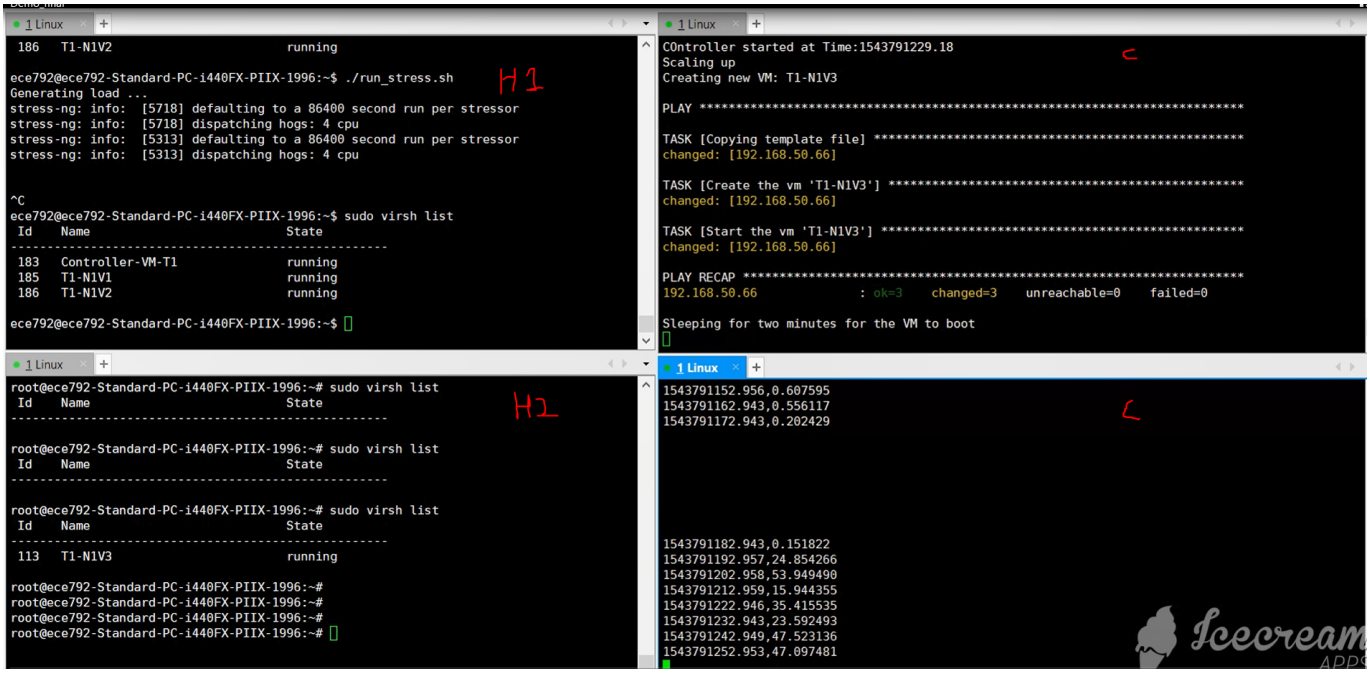
Test Case 2:
Feature: VM Scale Up during high steal time
Input: Stress-ng tool to generate cpu load
Output: New VM created on the free Hypervisor.
Actual Output: Showing steal time log(collectd) and calling controller.py

```
root@controller:~/autoscaler# python controller.py
COntroller started at Time:1543791229.18
Scaling up
Creating new VM: T1-N1V3

PLAY ***************************************************

TASK [Copying template file] ***************************
changed: [192.168.50.66]

TASK [Create the vm 'T1-N1V3'] *************************
changed: [192.168.50.66]

TASK [Start the vm 'T1-N1V3'] **************************
```

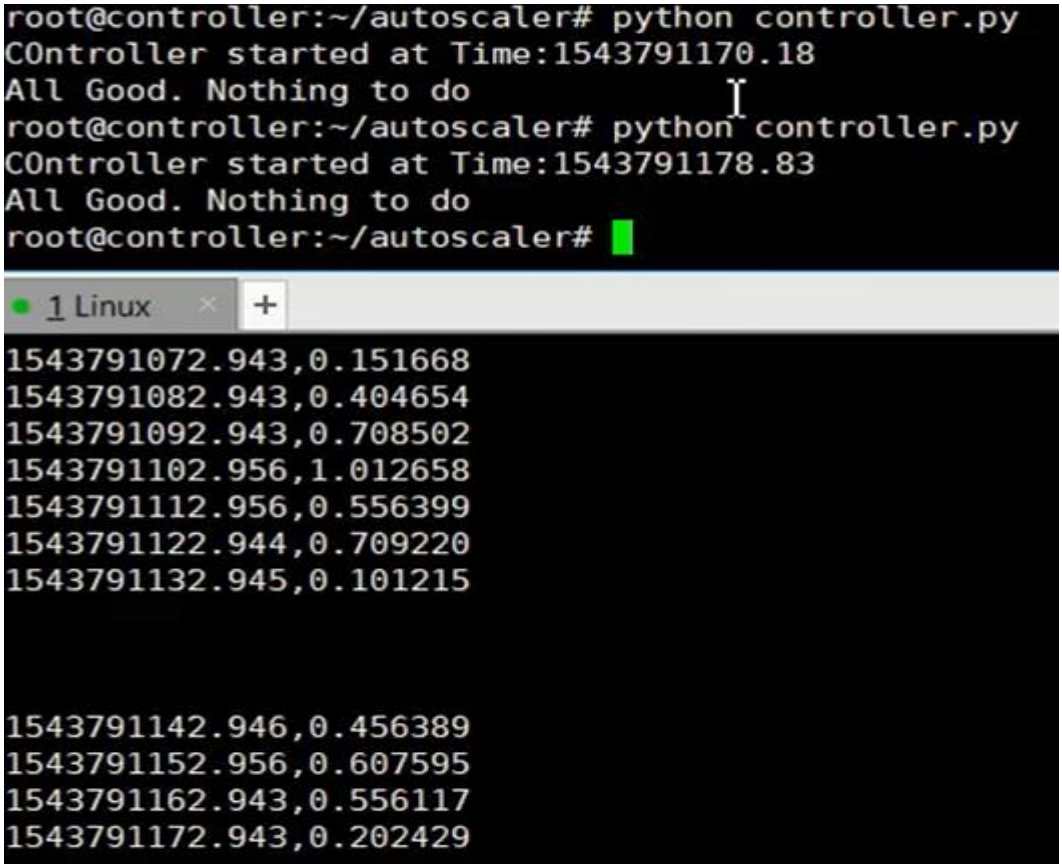
1 Linux +

```
1543791152.956,0.607595
1543791162.943,0.556117
1543791172.943,0.202429
1543791182.943,0.151822
1543791192.957,24.854266
1543791202.958,53.949490
1543791212.959,15.944355
1543791222.946,35.415535
1543791232.943,23.592493
1543791242.949,47.523136
1543791252.953,47.097481
```

H1 - Main Hypervisor
H2 - Standby Hypervisor
C- Controller
Showing new VM created on H2 when H1 is overloaded.



Test Case 3:
Feature: VM Idle when load is low
Input: No/low load on the VMs
Output: No action taken by Controller.
Actual Output: Showing steal time log(collectd) and calling controller.py.



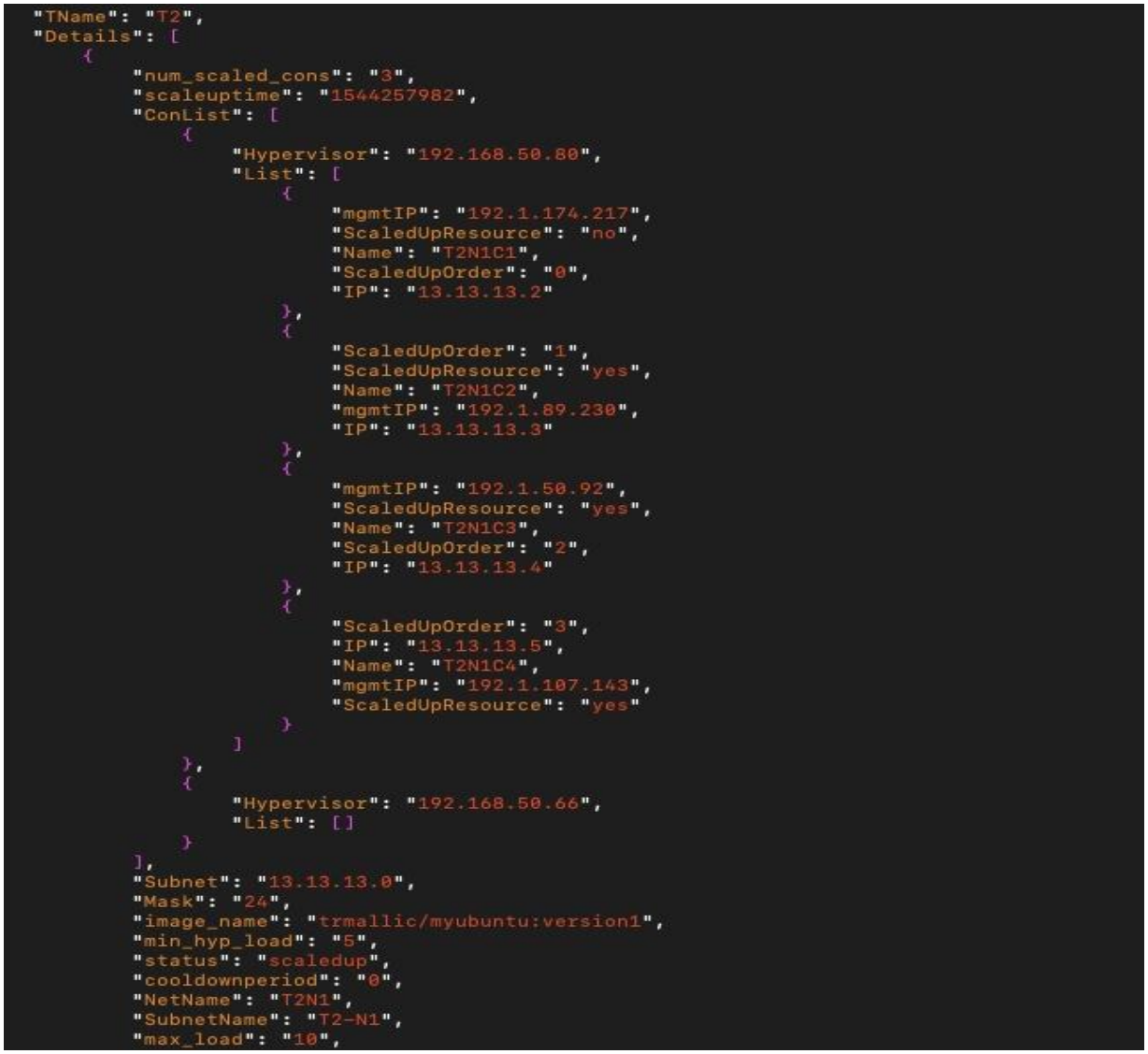
Test Case 4:
Feature: VM Scale Down
Input: Previously scaled up system with reduced load
Output: Deletion of Scaled up VM
Actual Output:

```
root@controller:~/autoscaler# python controller.py
Controller started at Time:1544259276.64
Hypervisor: 192.168.50.80
Scaling down
No overloaded hypervisors. Scaling down.
Destroying VM: T1-N1V3

PLAY ***************************************************************************

TASK [Destroy the vm 'T1-N1V3'] ************************************************
changed: [192.168.50.66]

TASK [Undefine the vm 'T1-N1V3'] ***********************************************
ok: [192.168.50.66]

PLAY RECAP *********************************************************************
192.168.50.66              : ok=2    changed=1    unreachable=0    failed=0

Updating json
Updating json done
root@controller:~/autoscaler#
```

```
root@controller:~# tail -f /var/lib/collectd/T1-N1V3/
4
1543962898.335,0.202532
1543962908.335,0.151592
1543962918.335,0.353001
1543962928.336,0.303183
1543962938.336,0.303030
```

Test Case 5:
Feature: Container Scale Up
Input: Heavy CPU Usages on Containers on a Hypervisor
Output: New Container Creation on free Hypervisor
Actual Output:

```
root@controller:~/autoscaler_con# python controller.py
Controller started at Time:1544257891.48
cont_stats.py
Scaling up
f3bc4031d03b4135cb8a12d0c3ea3410faa0b8856f4cfa798ae928f7d38e7b66
mv: cannot move '/etc/resolv.conf.dhclient-new.28' to '/etc/resolv.conf': Device or resource busy
MGMT IP: 192.1.107.143
Provisioned
13.13.13.5
ip: 13.13.13.5 mask: 24 gw: 13.13.13.1
Updating json
Updating json done
setting load balancer
setting load balancer
root@controller:~/autoscaler_con#
```



```
"TName": "T2",
"Details": [
    {
        "num_scaled_cons": "3",
        "scaleuptime": "1544257982",
        "ConList": [
            {
                "Hypervisor": "192.168.50.80",
                "List": [
                    {
                        "mgmtIP": "192.1.174.217",
                        "ScaledUpResource": "no",
                        "Name": "T2N1C1",
                        "ScaledUpOrder": "0",
                        "IP": "13.13.13.2"
                    },
                    {
                        "ScaledUpOrder": "1",
                        "ScaledUpResource": "yes",
                        "Name": "T2N1C2",
                        "mgmtIP": "192.1.89.230",
                        "IP": "13.13.13.3"
                    },
                    {
                        "mgmtIP": "192.1.50.92",
                        "ScaledUpResource": "yes",
                        "Name": "T2N1C3",
                        "ScaledUpOrder": "2",
                        "IP": "13.13.13.4"
                    },
                    {
                        "ScaledUpOrder": "3",
                        "IP": "13.13.13.5",
                        "Name": "T2N1C4",
                        "mgmtIP": "192.1.107.143",
                        "ScaledUpResource": "yes"
                    }
                ]
            },
            {
                "Hypervisor": "192.168.50.66",
                "List": []
            }
        ],
        "Subnet": "13.13.13.0",
        "Mask": "24",
        "image_name": "trmallic/myubuntu:version1",
        "min_hyp_load": "5",
        "status": "scaledup",
        "cooldownperiod": "0",
        "NetName": "T2N1",
        "SubnetName": "T2-N1",
        "max_load": "10",
```


Test Case 6:
Feature: Container Scale Down
Input: Previously scaled up system with reduced cpu load averages and cooldown expiry
Output: Deletion of newly added Container.
Actual Output:

```
root@controller:~/autoscaler_con# python controller.py
Controller started at Time:1544258587.66
Scaling down
No overloaded hypervisors. Scaling down.
Destroying container: T2N1C4

PLAY ***************************************************************************

TASK [Destroy the container 'T2N1C4'] ******************************************
changed: [192.168.50.80]
 [WARNING]: Consider using 'become', 'become_method', and 'become_user' rather than running sudo


PLAY RECAP *********************************************************************
192.168.50.80              : ok=1    changed=1    unreachable=0    failed=0

Updating json
Updating json done
```


```
{
    "TName": "T2",
    "Details": [
        {
            "num_scaled_cons": "2",
            "scaleuptime": "1544257982",
            "ConList": [
                {
                    "Hypervisor": "192.168.50.80",
                    "List": [
                        {
                            "ScaledUpOrder": "0",
                            "ScaledUpResource": "no",
                            "Name": "T2N1C1",
                            "mgmtIP": "192.1.174.217",
                            "IP": "13.13.13.2"
                        },
                        {
                            "ScaledUpResource": "yes",
                            "IP": "13.13.13.3",
                            "Name": "T2N1C2",
                            "ScaledUpOrder": "1",
                            "mgmtIP": "192.1.89.230"
                        },
                        {
                            "ScaledUpOrder": "2",
                            "ScaledUpResource": "yes",
                            "Name": "T2N1C3",
                            "mgmtIP": "192.1.50.92",
                            "IP": "13.13.13.4"
                        }
                    ]
                },
                {
                    "Hypervisor": "192.168.50.66",
                    "List": []
                }
            ],
            "Subnet": "13.13.13.0",
            "Mask": "24",
            "image_name": "trmallic/myubuntu:version1",
            "min_hyp_load": "5",
            "status": "scaledup",
            "cooldownperiod": "0",
            "NetName": "T2N1",
            "SubnetName": "T2-N1",
            "max_load": "10",
            "min_load": "5",
            "num_cons": "3",
            "max_hyp_load": "10"
```

6. Summary and Future Scope

Summary

In this Project report, we discussed about the performance issues that an application can face due to a unforeseen increase in traffic and have proposed a solution to this problem. This solution helps improve the performance of the application and helps in reducing the expenditures by the enterprises. We discussed some related solutions and each of them has their own drawbacks. Our solution neatly defines the need for scaling with all the performance metrics available today. The solution defines Thresholds and Cooldown periods to handle steady and sudden increase in traffic load. The auto-scaler then makes an informed decision on scaling. We also discussed how the auto-scaler reduces the scaled up resources once the overall load decreases. We described the system architecture and discussed the management features. We also discussed about the layer architecture and how each component delivers for the success of the overall solution. Some notes for Users as well as developers have also been covered. We closed the discussion by showing some experimental results using some very practical testcases.

Future Scope

1. Building further on the Dynamic Monitoring tool. Building trust between the Provider and the Tenant on a multi-cloud platform can be achieved by further pursuing this. Open-source Blockchain technologies like Hyperledger look promising in achieving this.
2. This service has been developed and deployed on two Hypervisors within the University Labs. This service is flexible enough to be deployed on different cloud platforms with very little modifications.
3. Integration with Public Cloud platforms like Amazon Web Services and Google Cloud Platform. Integrating their APIs into our ecosystem would enable this service to leverage resources of their platform along with the one offered as part of this service.
4. Self Healing Management Feature can be enhanced further to allow user-defined health checks to identify a failed Container. Ex: TCPSocketCheck or HTTPGetAction can allow user to specify a TCP port / HTTP Get of the suspect container to be inspected and if the port is closed, the Container can be marked as failed and it can be respawned.