DevOps Project

Infrastructure

Terraform was used to create a virtual infrastructure running a cluster of 3 VMs in AWS. The architecture consists of one VPC with an IP range of 10.0.0.0/16, an internal gateway, and a route table to forward the traffic between VMs and Internet, and 3 subnets, each in a different availability zone. The instances were separated in two groups: masters (master1) and workers (worker 1, worker 2). Also, security group rules were added to allow ICMP and SSH from anywhere and allow all traffic from the VMs to the outside. Each EC2 instance has an EBS volume attached to it. I used Elastic IP Addresses instead of DNS records because of the high costs for register a domain name. Terraform files can be found under *tf-config/aws* and the infrastructure can be provisioned with the following commands:

```
terraform init
terraform plan
terraform apply
```

After the provisioning phase, an Ansible playbook was created to update and upgrade the packages, to configure the hostnames of VMs and create a Docker Swarm or a Kubernetes cluster. I created a variable under *vars/main.yaml* namely *kubernetes.flag* from where you can choose between Swarm (fasle) or Kubernetes (true). The playbook can be executed by running the followind command:

```
ansible-playbook -i inventory.yml main.yml
```

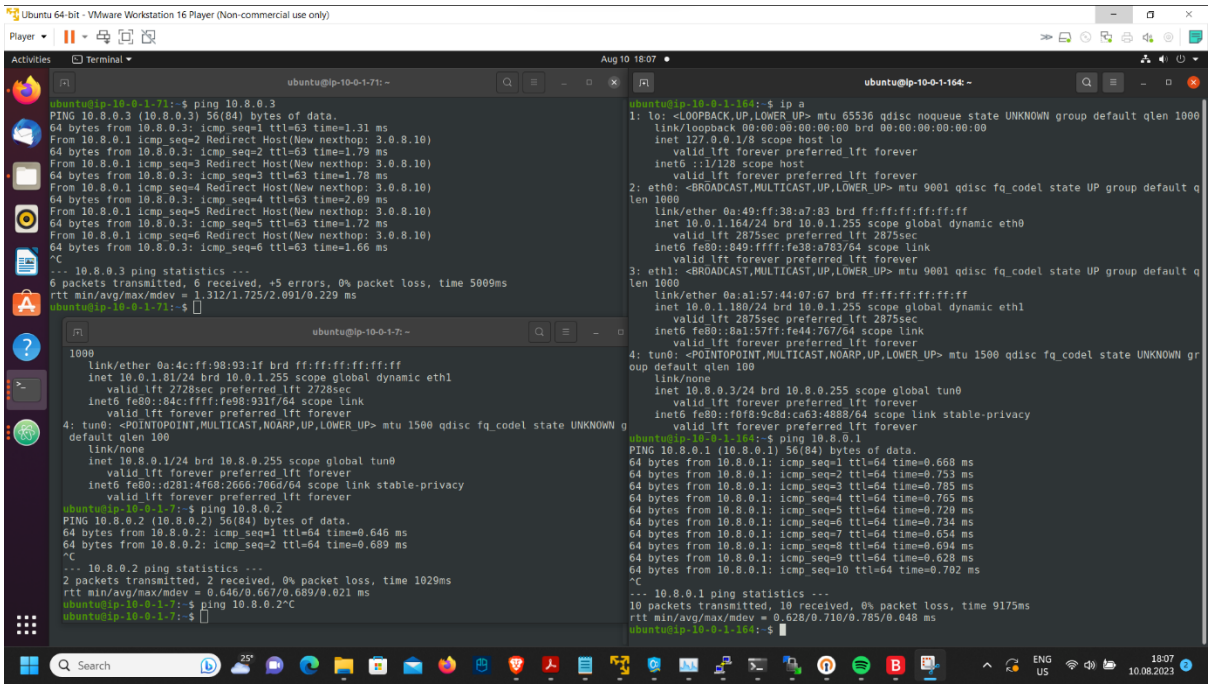
Password authentication was already deactivated by default, but I included a task to deactivate it just in case. The keypair was created during provisioning step (see Terraform files). After these steps a Docker Swarm or a Kubernetes cluster will be up and running with 1 master and 2 workers.

Networking

Firewall rules using iptables were set to allow all traffic between VMs, SSH access from local machine and HTTP/HTTPS from everywhere. This step was also automated using another playbook namely iptables.yml and can be run using:

```
ansible-playbook -i inventory.yml iptables.yml
```

Then, to configure a VPN network between instances I used OpenVPN which can be installed also through a playbook vpn.yml. Next, I configured master1 to be the server by running openvpn-install.sh script inside it. This script generated .ovpn config files that can be used further by the clients. To generate two files, the script should be run twice. The traffic between EC2 instances was then tunnelled through 10.8.0.0/24 network using the tun0 interface attached by the OpenVPN to the instances as illustrated in the next figure.



Monitoring

In this step a monitoring step consisting of Prometheus, Alertmanager, Blacbox exporter, Pushgateway and Grafana was created. I also included cAdvisor and node-exporter to collect as many metrics as possible. The stack is under monitoring directory where I added installation scripts and configuration files for each service. All the scripts and configuration files should be run and copy to the master1 node. The following metrics were gathered and displayed using Grafana (http://18.198.73.192:3000/):

VM CPU Usage

```
100 - (avg by(instance) (rate(node_cpu_seconds_total{mode="idle"}[1m])) * 100)
```

VM Memory Usage

```
node_memory_Active_bytes/node_memory_MemTotal_bytes*100
```

VM Free Space Usage

```
topk(1,              node_filesystem_avail_bytes              /
node_filesystem_size_bytes * 100) by (instance)
```

VM Network Transmit Bytes

```
irate(node_network_transmit_bytes_total{device="eth0"}[1m])
```

VM Network Receive Bytes

```
irate(node_network_receive_bytes_total{device="eth0"}[1m])
```

VM Network Transmit Bytes Through Tunnel

```
irate(node_network_transmit_bytes_total{device="tun0"}[1m])
```

VM Network Receive Bytes Through Tunnel

```
irate(node_network_receive_bytes_total{device="tun0"}[1m])
```

Container CPU Usage

```
(rate(container_cpu_usage_seconds_total{name!=""}[1m])) * 100
```

Container Memory Usage

```
container_memory_working_set_bytes/container_spec_memory_limit
_bytes * 100
```

VM Free Space Usage

```
(container_fs_usage_bytes{name!=""}/container_fs_limit_bytes{n
ame!=""}) * 100
```

VM Network Transmit Bytes

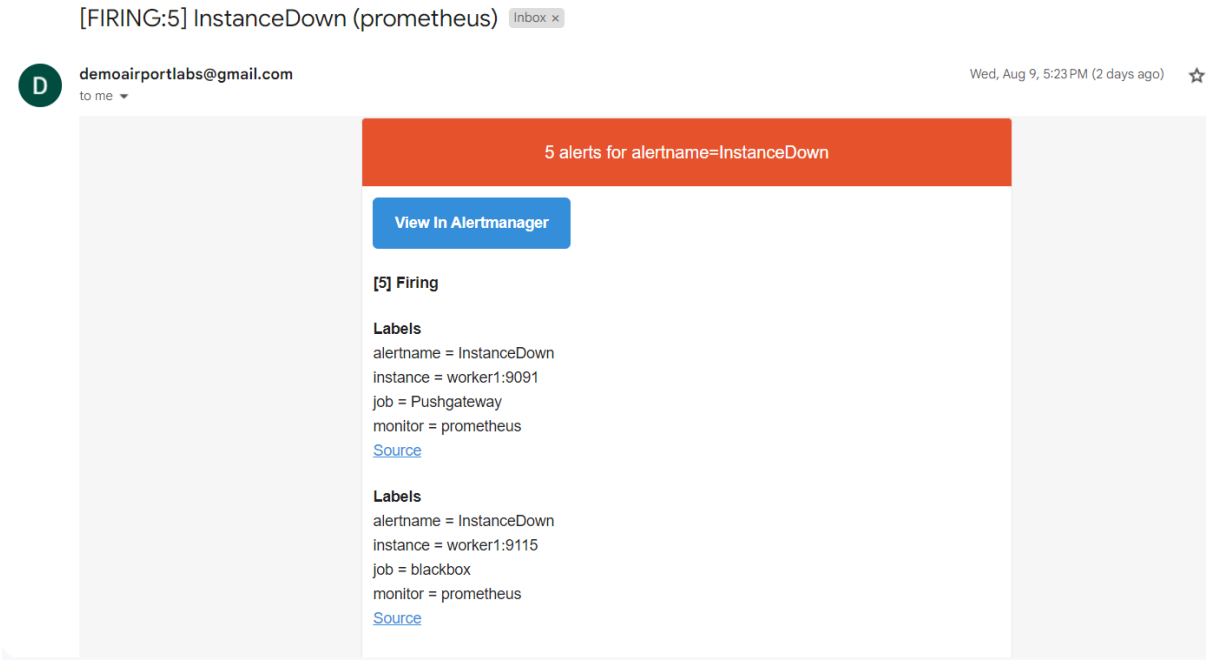
```
irate(container_network_transmit_bytes_total{name!=""}[1m])
```

VM Network Receive Bytes

```
irate(container_network_receive_bytes_total{name!=""}[1m])
```

Jobs can be analyzed in monitoring/prometheus/Prometheus.yml.

I also created alerts for critical metrics () which send emails in case of node failure as in the following figure.



CI/CD

In this step I deployed on the previous infrastructure an open source application, where the user first requests the index.html and then it can interact with it (it is only for demo purposes so the application is not responsive).

Two shell scripts were created in demo-app directory: build-pipeline.sh which builds the docker images for each microservice and deploy.pipeline.sh which deploys the application as containers on top of Swarm cluster. Replicas were set to 3 and each replica is hosted on a different node. The application can be accessed at: http:// 3.121.205.77/.

For the CI/CD pipeline I opted for Jenkins which was installed on the master1 node. From Jenkins GUI I created a job which is trigger manually and it builds and deploy the app hosted at: https://github.com/robertbotez/demo-app.

Extra

The application was deployed also using Kubernetes. Moreover, I created a Helm Chart for deploying the app more easily. The infrastructure cost report was attached in the repo.