Ignite19-US-HOW-14-Protecting-Kubernetes

Hands-On-Workshop

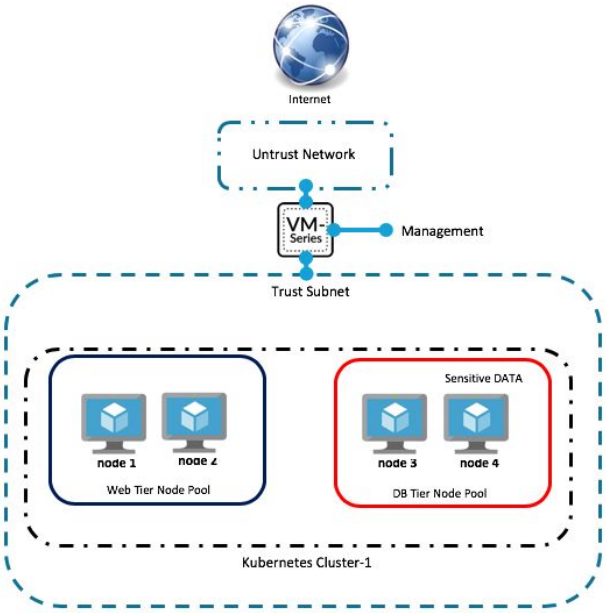
Protecting your container workloads in Kubernetes

Terraform Version

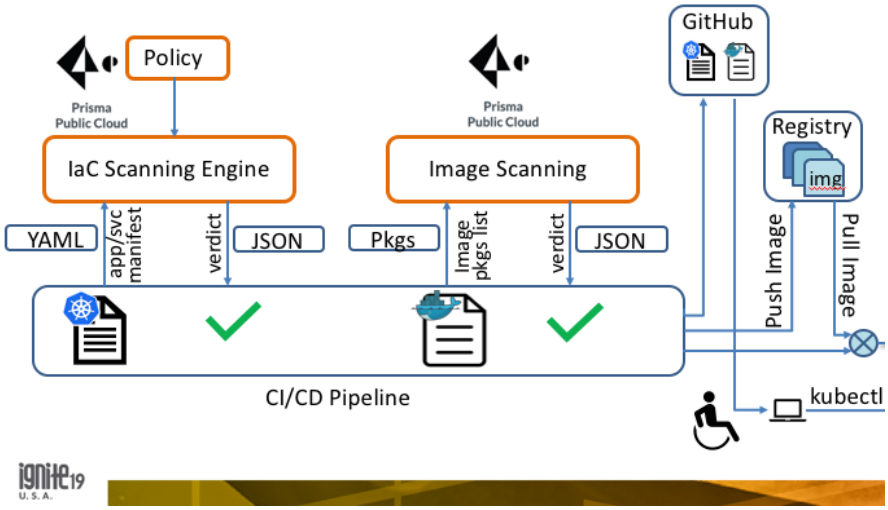
Table of Contents

Lab Overview

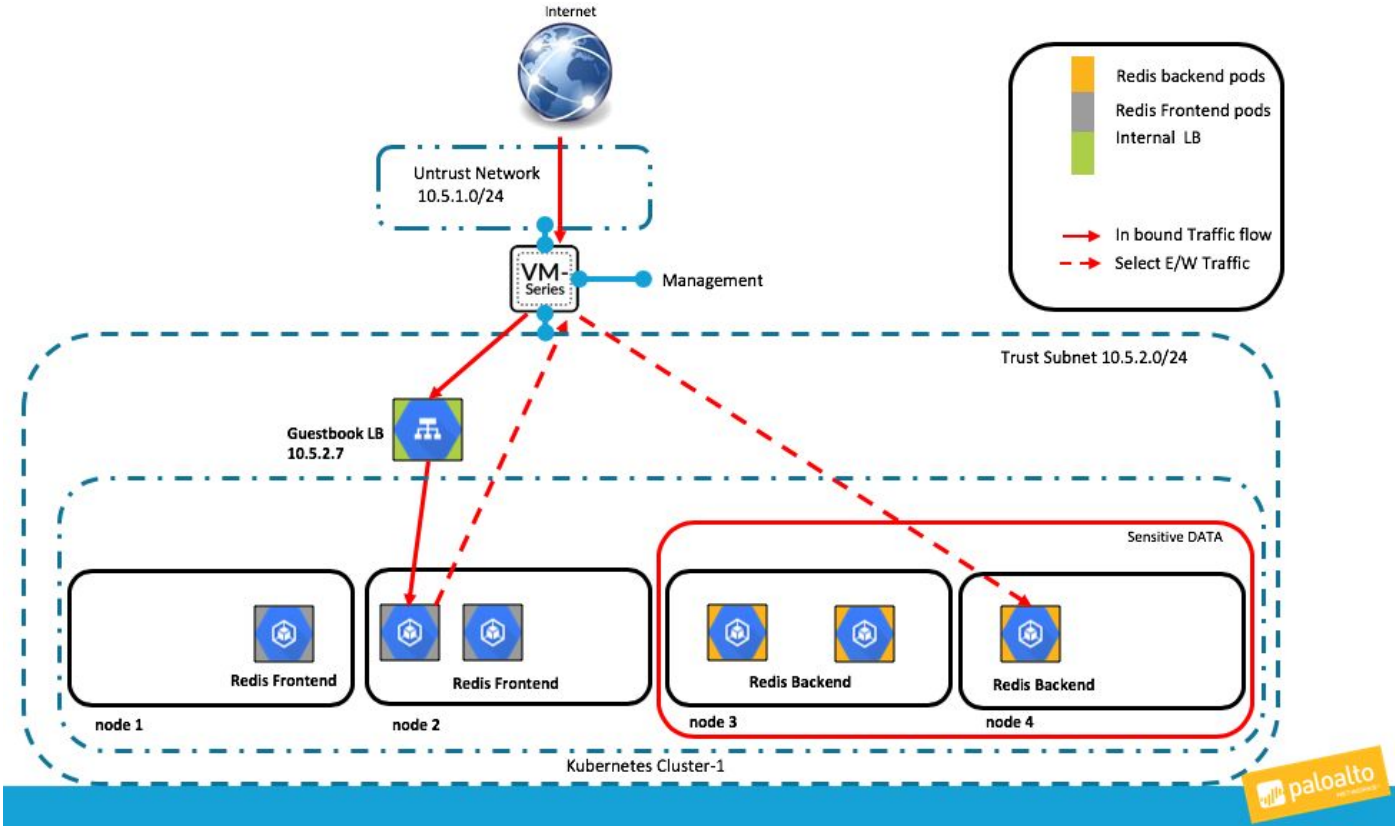
This lab will walk through deploying a Terraform template that deploys a Kubernetes(k8s) cluster, Palo Alto Networks VM-Series firewall into an existing GCP project. After successfully deploying the terraform template, the following infrastructure will be instantiated:



Then the lab will then guide through the use of the Prisma Public Cloud API Scanning of a container image and K8s Manifest file.



Finally the lab will deploy the manifest file to the k8s cluster and walk through visibility of both North/South and select East/West traffic flows. The following diagram shows the deployed pods with the traffic flows:



About GCP Terraform Templates

GCP Terraform Templates are files that can deploy, configure, and launch GCP resources such as VPC networks, subnets, security groups, firewall rules, route tables, Kubernetes clusters, and more. These templates are used for ease of deployment and are key to any cloud deployment model.

For more information on Templates refer to Google's documentation
https://cloud.google.com/community/tutorials/managing-gcp-projects-with-terraform

There are also many Terraform template s available here:
https://github.com/GoogleCloudPlatform/terraform-google-examples

Support Policy

This template is released under an as-is, best effort, support policy. These scripts should be seen as community supported and Palo Alto Networks will contribute our expertise as and when possible. We do not provide technical support or help in using or troubleshooting the components of the project through our normal support options such as Palo Alto Networks support teams, or ASC (Authorized Support Centers) partners and backline support options. The underlying product used (the VM-Series firewall) by the scripts or templates are still supported, but the support is only for the product functionality and not for help in deploying or using the template or script itself.

Instances Used

When deploying this Terraform template the following machine types are used:

Instance	Machine Type	QTY
PayGo Bundle 1 – VM-Series Firewall	n1-standard-4	1
Kubernetes Ubuntu Cluster Nodes	n1-standard-1	4
Internal Load Balancer		1

Note: There are GCP costs associated with each machine type launched, please refer to the Google instance pricing page https://cloud.google.com/compute/pricing

Prerequisites

Here are the prerequisites required to successfully launch this template:

- Terraform application - Instructions on the installation can be found here: https://www.terraform.io/intro/getting-started/install.html
- GCP account- Account creation can be done here: https://cloud.google.com/free/
- Google Cloud SDK- GCP template installations in this guide are performed from the CLI. Install the SDK/CLI by selecting the relevant platform from the following link and following the installation instructions: https://cloud.google.com/sdk/

Activity 1 – GCP Setup

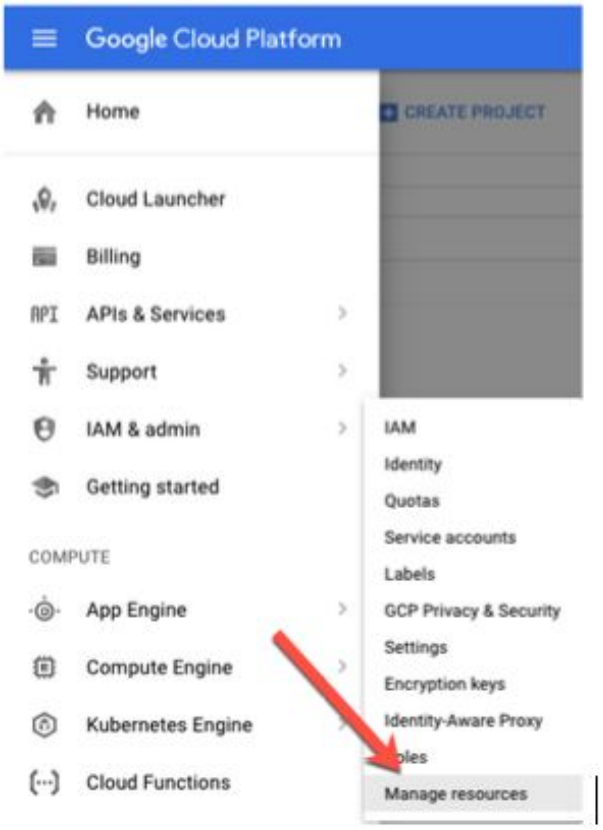
In this activity, you will:

- ***Create a project***
- ***Enable the APIs needed for this lab***

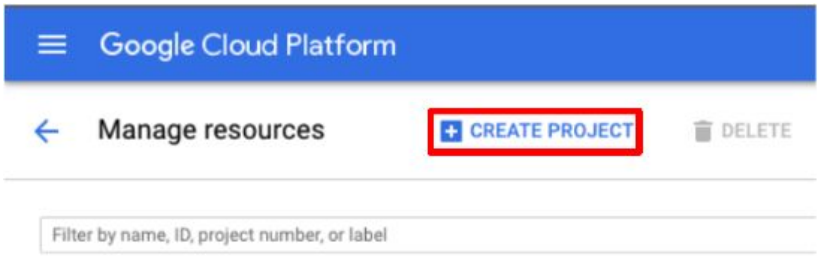
Task 1 – Create a Project

All GCP resources in this guide are deployed to a single project, which is an organizational boundary that separates users, resources, billing information, etc. It is similar to an AWS VPC or an Azure Resource Group. By default, GCP will create a project upon creation of an account.

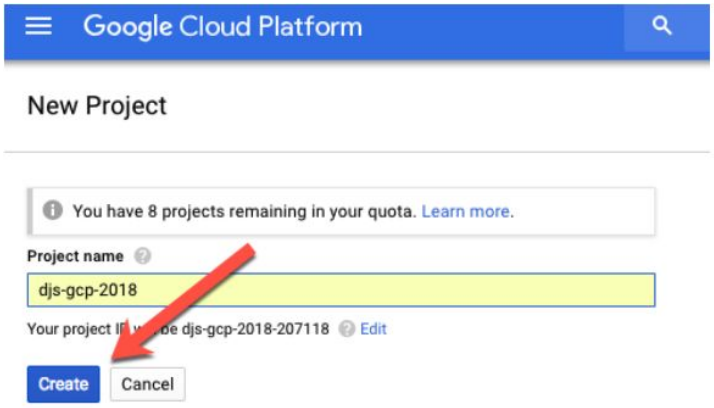
➔ To create an additional dedicated project, use the drop-down on the left and select **IAM & admin > Manage Resources:**



➔ Click **Create Project:**



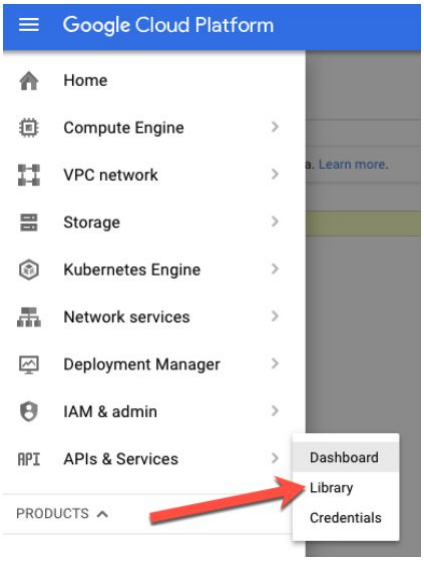
➔ Specify a name for the project and click **Create**:



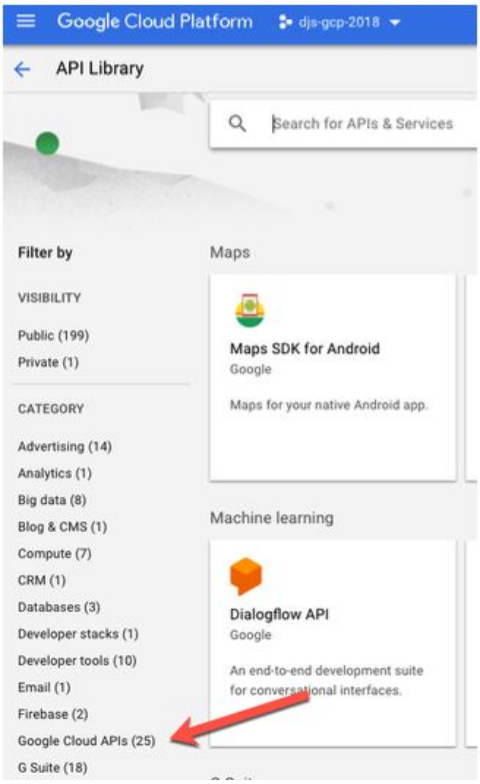
Note that project creation will take a few minutes.

Task 2 – Enable the Needed APIs

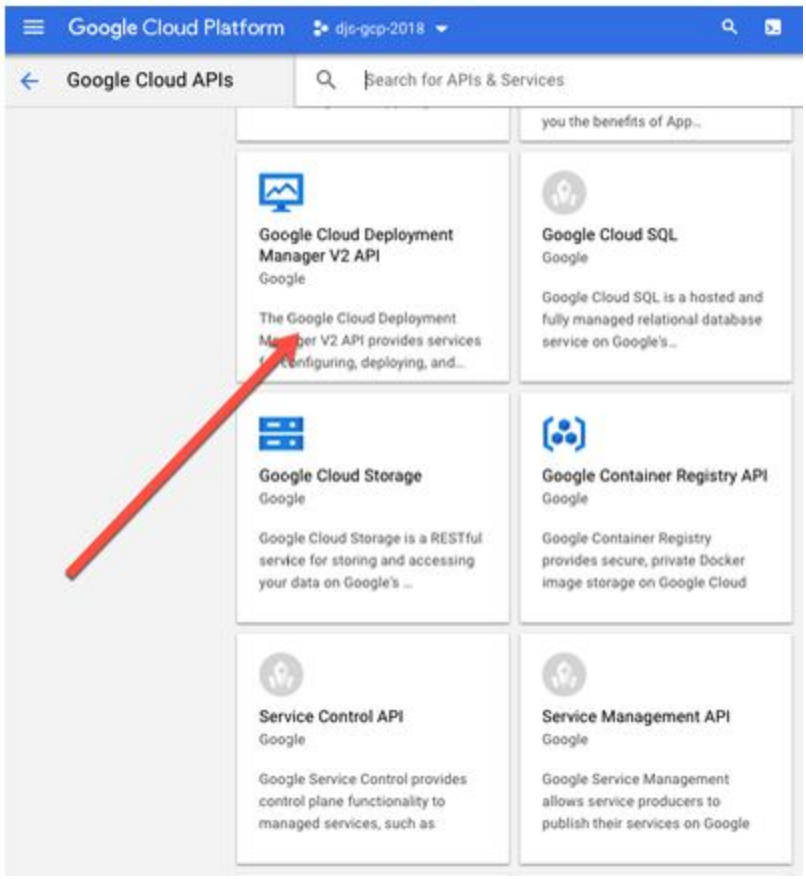
Deploying a template requires the Cloud Deployment Manager API be enabled on the project. Navigate to **APIs & Services >Library**:



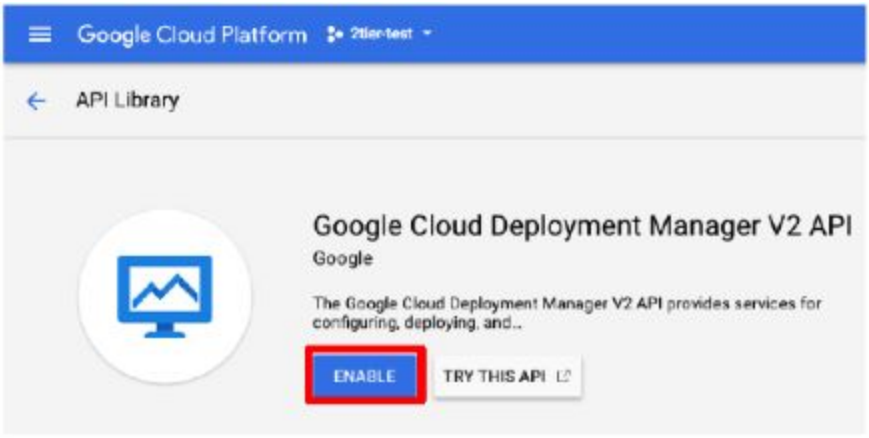
➔ Select Google Cloud APIs on the Left-Hand-Side:



➔ Select Google Cloud Deployment Manager V2 API:



➔ Select **Enable**:



Enabling the API for the project will take a few minutes to complete.

End of Activity 1

Activity 2 – Terraform File Setup

In this activity, you will:

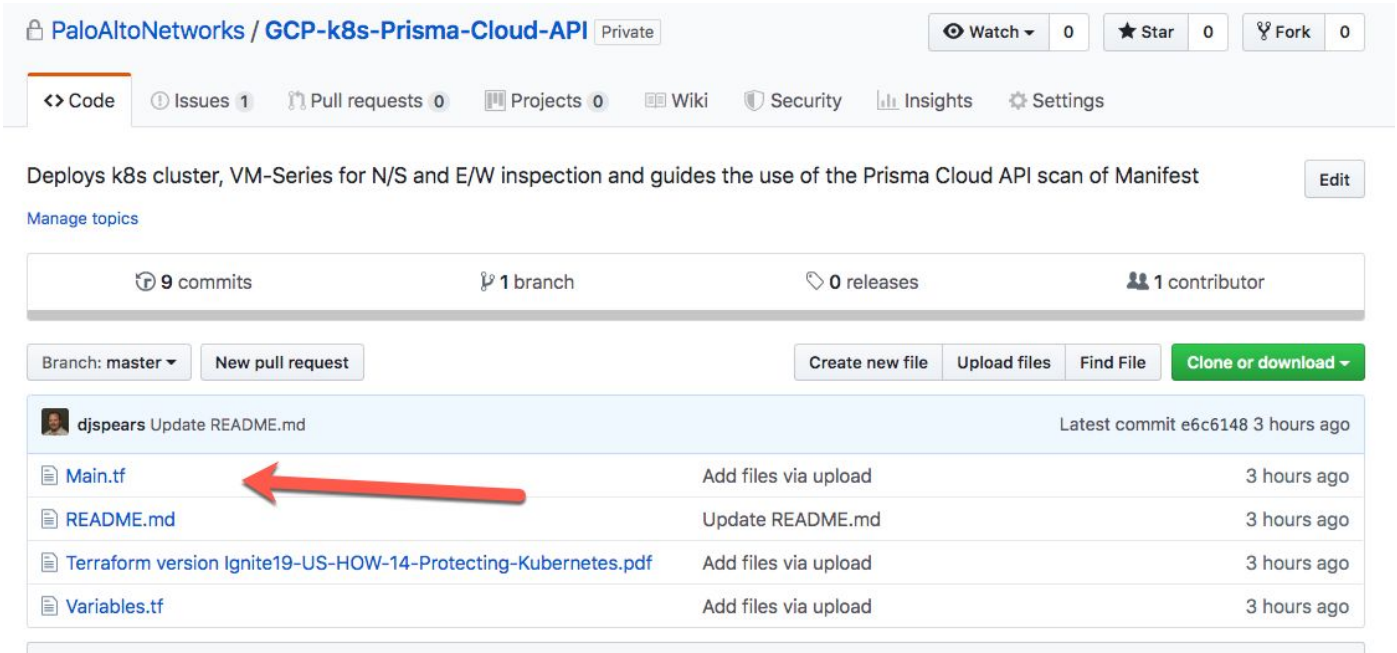
- ***Download the Terraform Files needed to instantiate the environment***
- ***Identify the variable parameters needed for Terraform and configure***

Task 1 – Download Terraform files

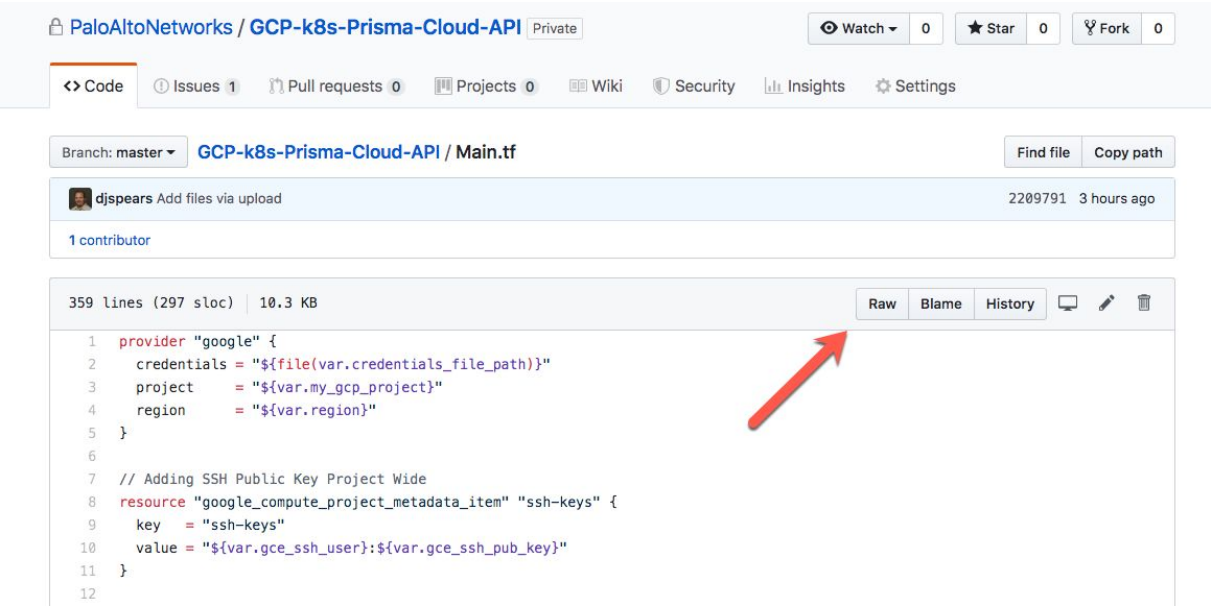
➔ Navigate to the following GitHub repository to get the terraform files: https://github.com/PaloAltoNetworks/GCP-VM-Series-k8s-Prisma-Cloud-API

For this lab, the Main.tf and Variables.tf files will need to be downloaded to your local machine.

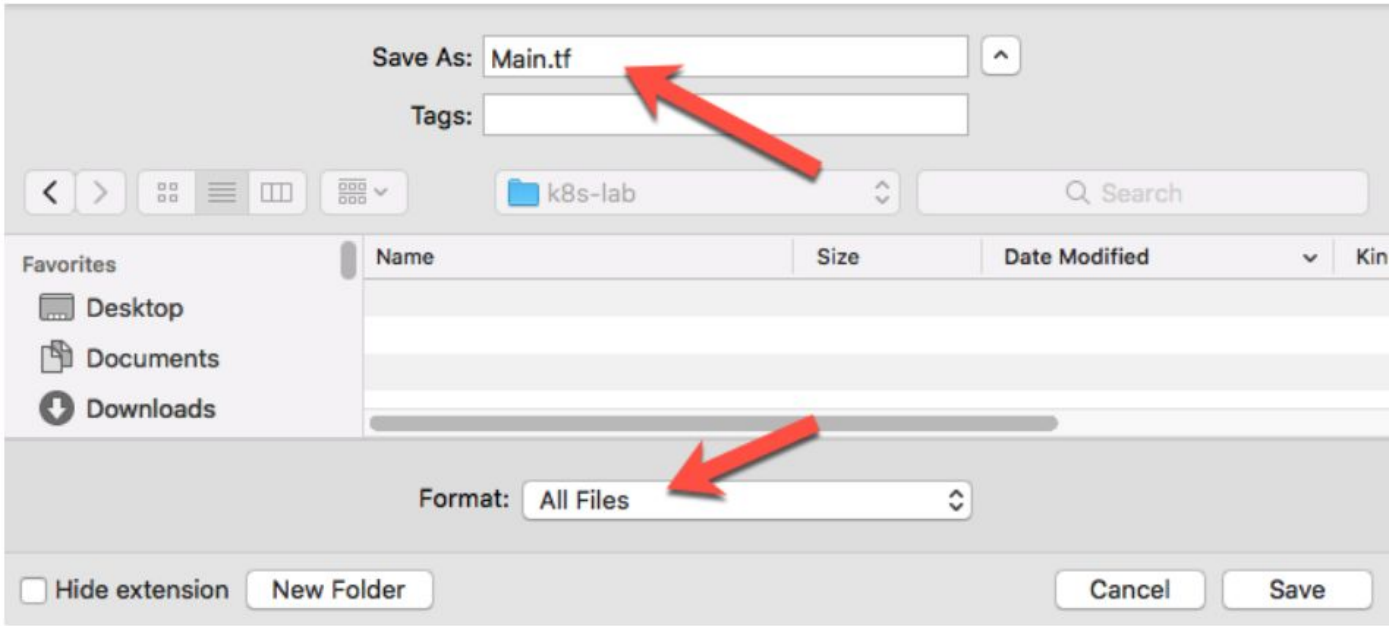
➔ Click on the Main.tf file:



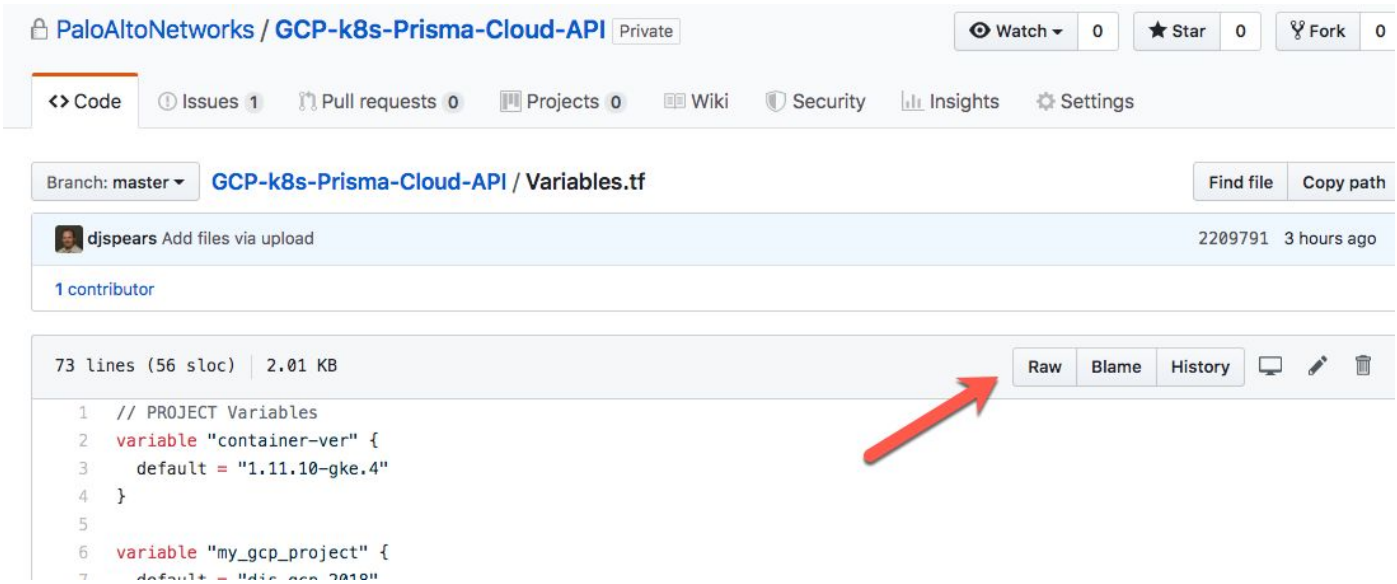
➔ Next Select Raw and save to a known local location:



Make sure the .tf file extension is added to the file. The following screenshot shows the mac save as dialogue box where the extension needs to be added and the Format needs to be changed to All Files:

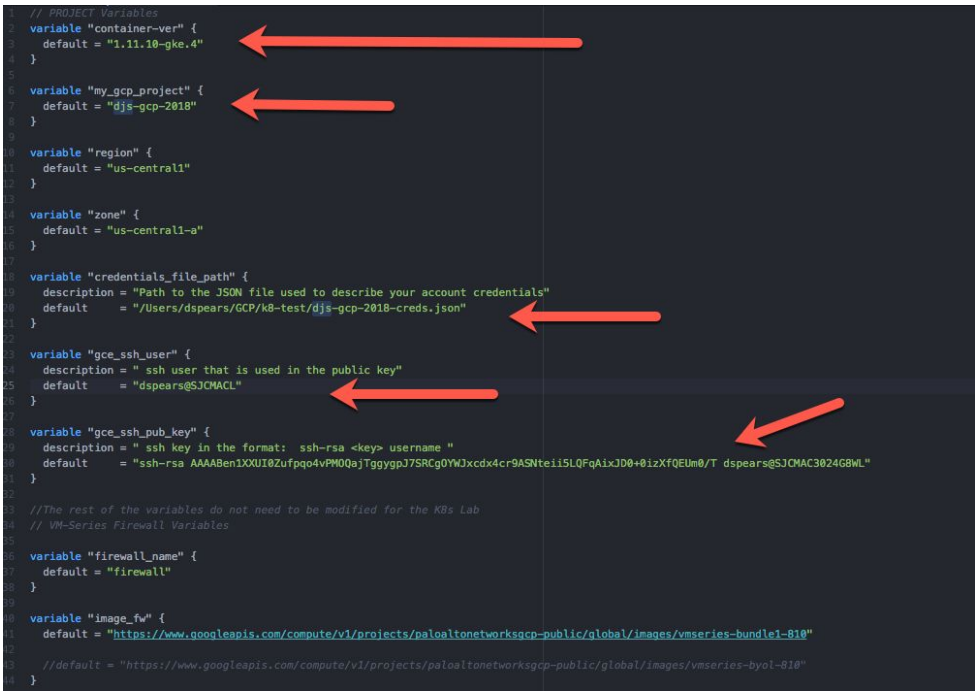


➔ Return to the repository and follow the previous steps to download the Variables.tf file to the same location as the Main.tf:



Task 2 – Gather Information and Update the Variables.tf file

The Terraform deployment files consist of a main file and a variables file. The Variables.tf file contains information that is easily modified and commonly changed for various situations. The variables in the Variables.tf file are used by the Main.tf file during deployment. Deploying this Terraform template in GCP does require modification of the template Variable.tf file to include deployment-specific information. The fields that must be updated are shown below:



A description of the fields that need to be updated are:

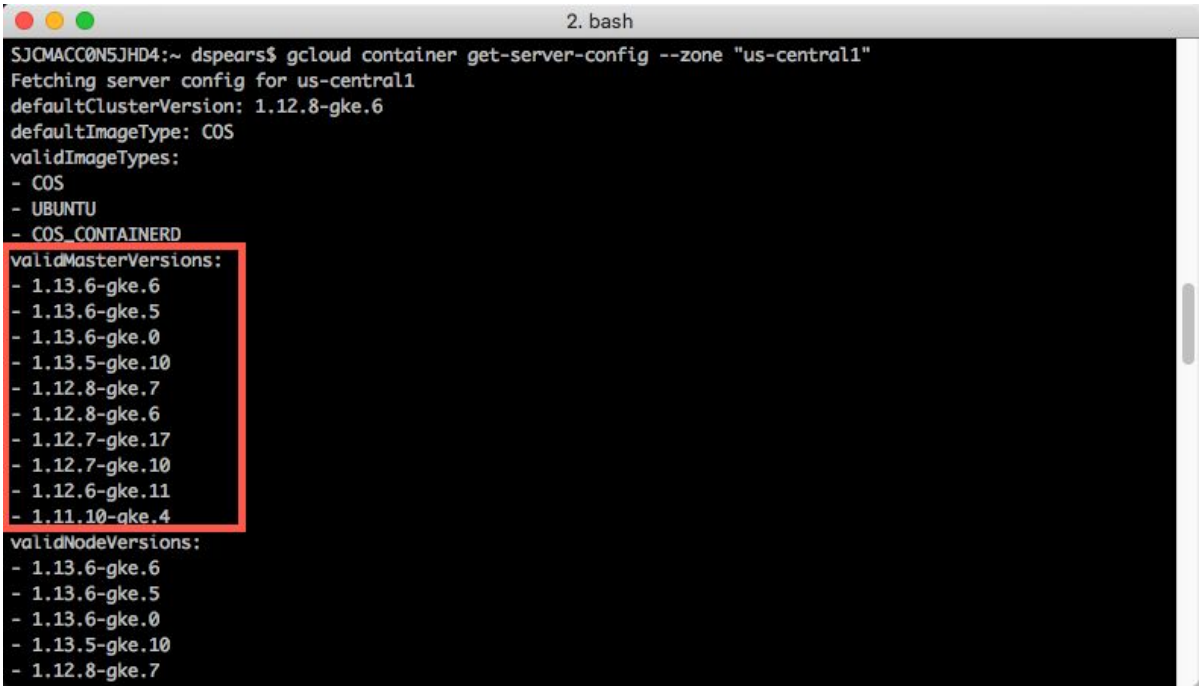
Container-ver This is the K8s Master version. Google updates versions often so it is likely that the version is no longer available.

Project ID is the id that is associated with the GCP project that was created previously.

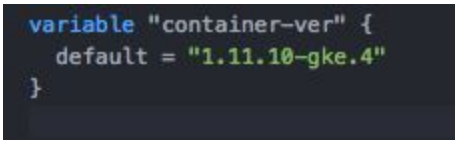
Credential file path is the path to the JSON file that has service credentials that will be used to deploy the template.

SSH User and SSH Pub Key are the SSH keys that can be used to access VM resources in the environment. In this lab the Firewall will be bootstrapped from an existing public GCP bucket. However, if that process fails, this SSH user can be used to access the firewall CLI.

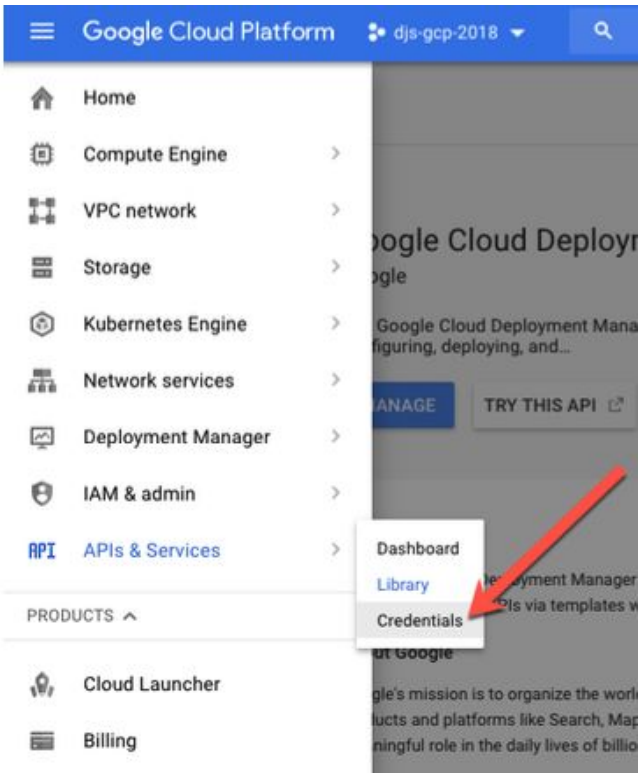
➔ To validate the master k8s version run the following command:
gcloud container get-server-config --zone "us-central1"



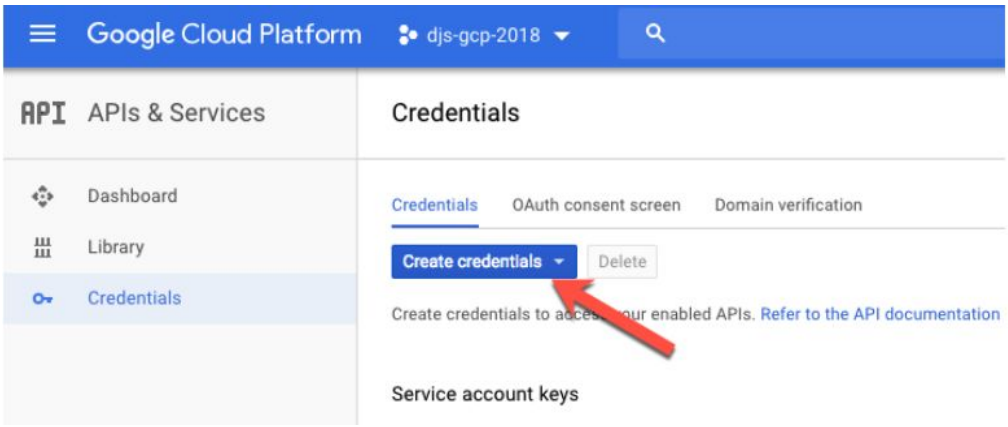
➔ Make sure a valid Master Version listed in the output is in the variables.tf file:



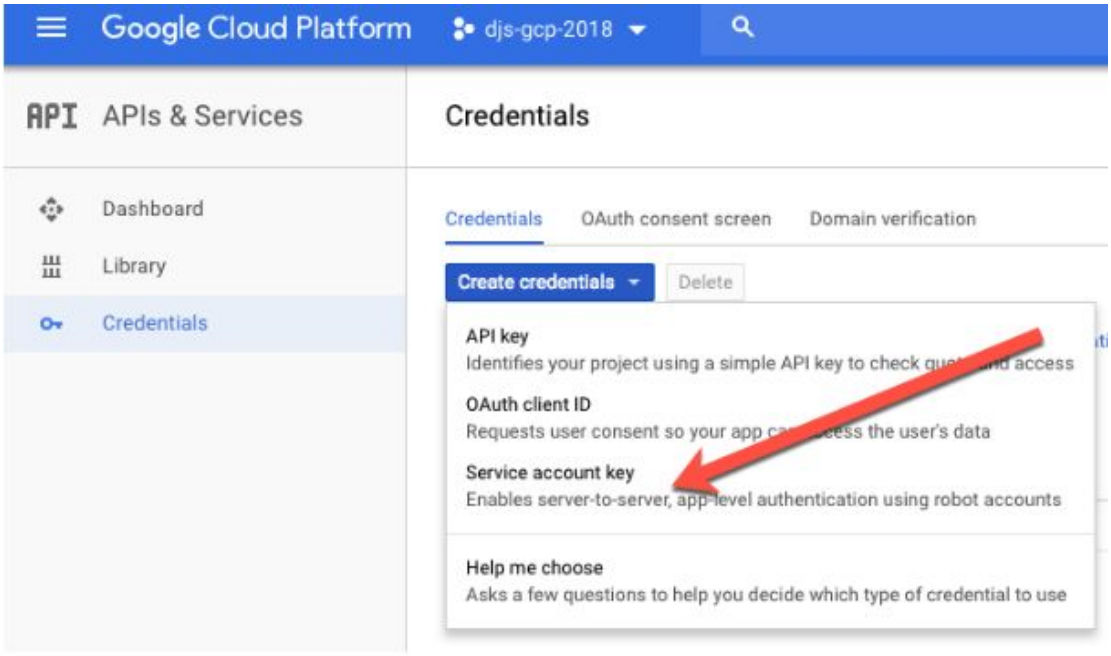
➔ To create the credentials to access the APIs in JSON format. In GCP console go to (APIs & Services > Credentials:



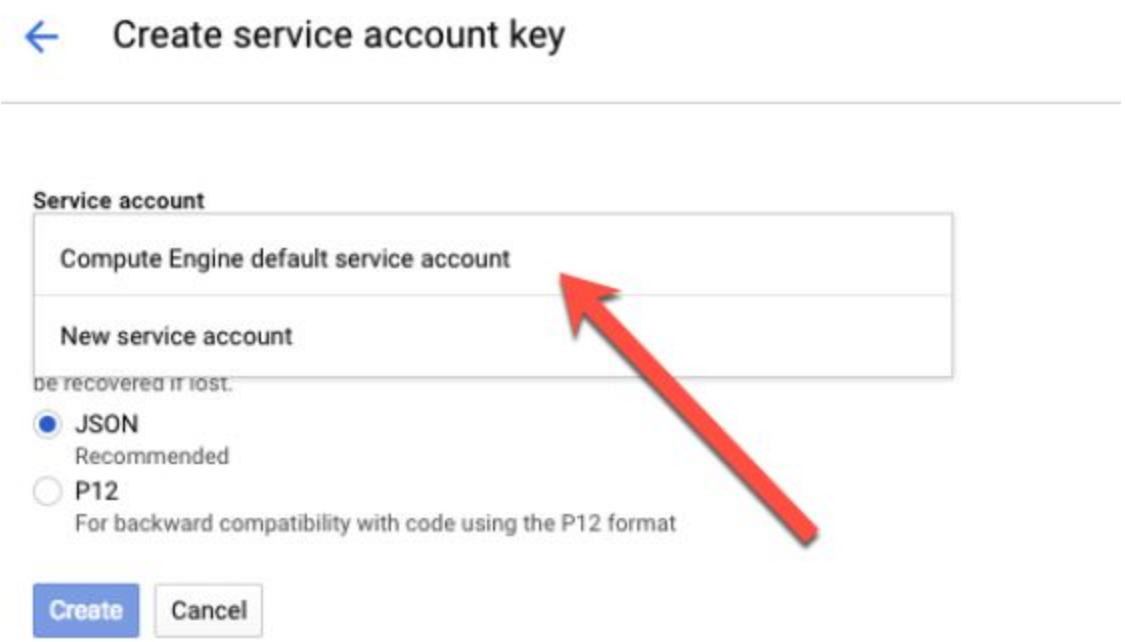
➔ Click Create Credentials:



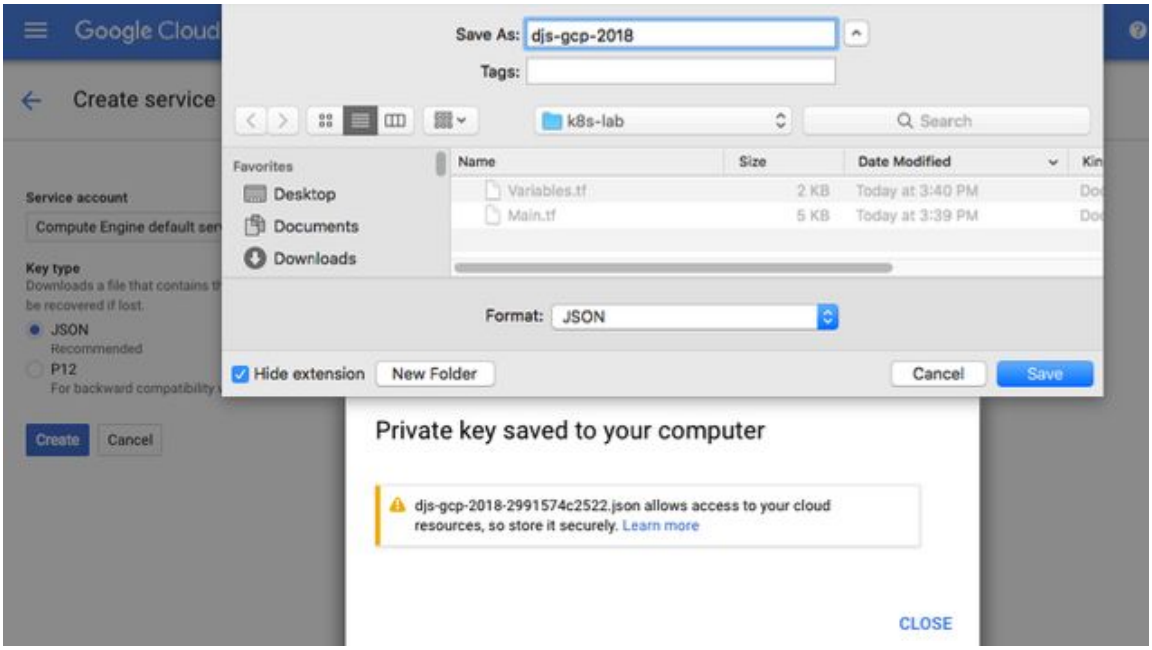
➔ Then select Service Account Key:



➔ Pick Compute Engine default service account and make sure the JSON format is ticked. Click Create:



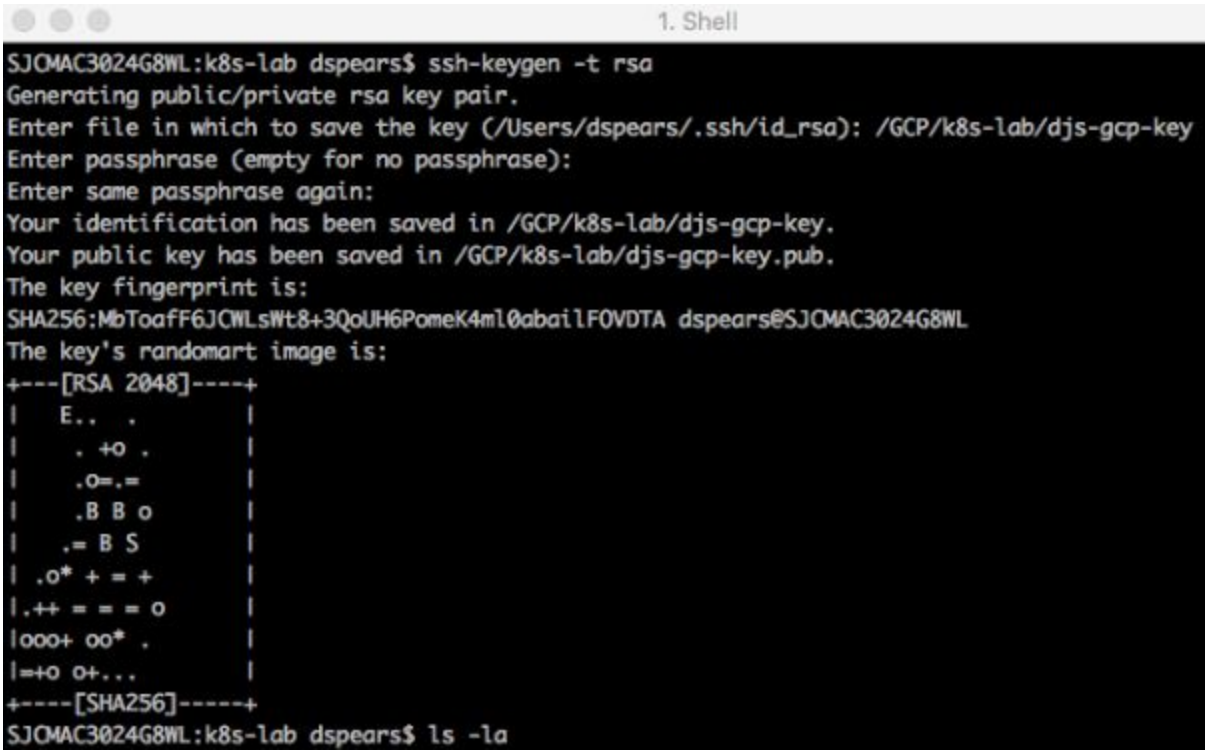
➔ Download the file to your computer. It is easy to put the credential file in the same folder as the terraform Main.tf and Variables.tf files:



➔ Use an editor of your choice to update the Variables.tf file with the appropriate path:

```
// PROJECT Variables
variable "my_gcp_project" {
  default = "Your_Project_ID"
}

variable "region" {
  default = "us-central1"
}

variable "zone" {
  default = "us-central1-a"
}

variable "credentials_file_path" {
  description = "Path to the JSON file used to describe your account credentials"
  default     = "/GCP/k8s-lab/djs-gcp-2018.json"
}
```

➔ If you do not already have an SSH key, the following example shows how to create a SSH key on a Mac using the ssh-keygen -t rsa command:

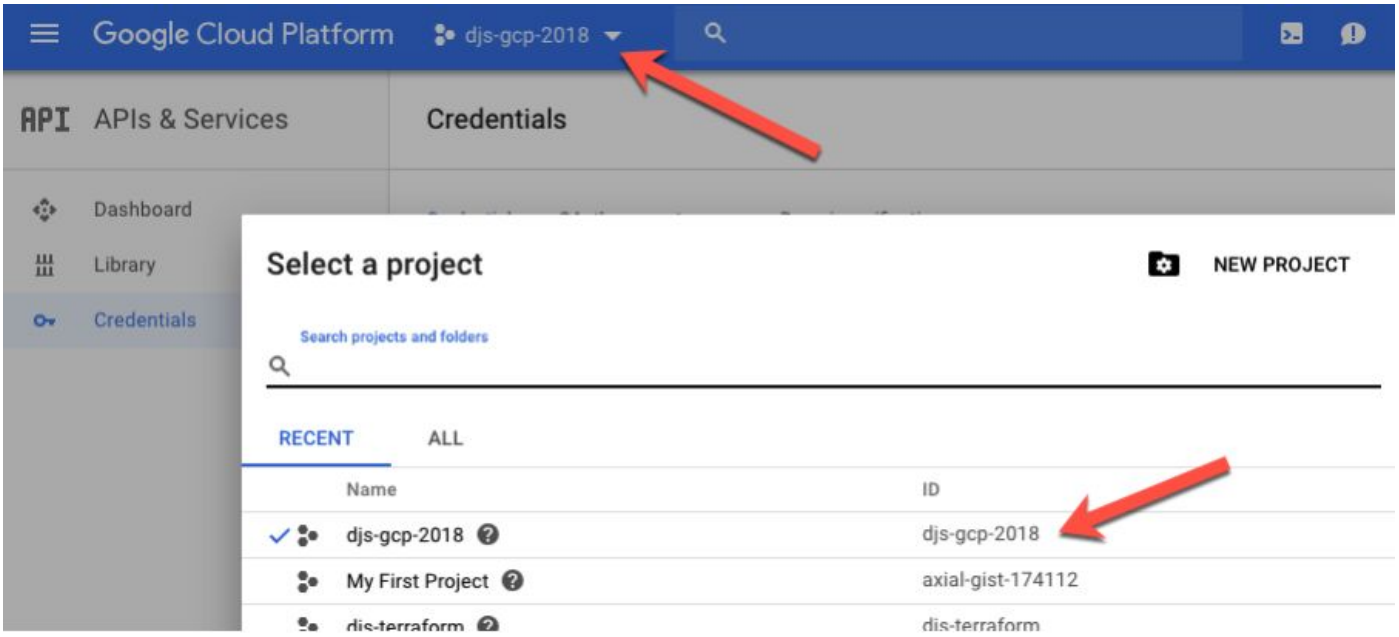


In the previous example the keys were generated and stored in the same directory as the other lab files. The public and private keys can be seen using the ls -la command. I have also displayed the public key via the more command in the following screenshot:

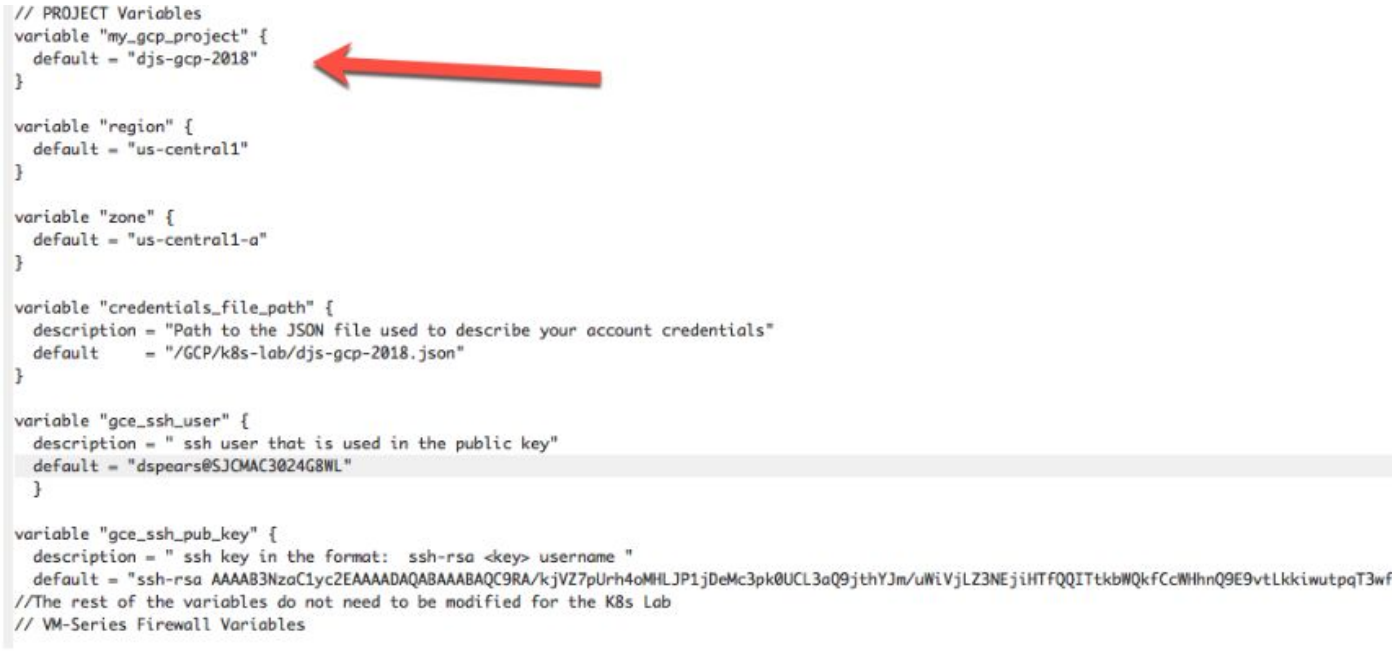
➔ Retrieve the username and key from the public key file. Then update the Variables.tf file:

```
// PROJECT Variables
variable "my_gcp_project" {
  default = "djs-gcp-2018"
}

variable "region" {
  default = "us-central1"
}

variable "zone" {
  default = "us-central1-a"
}

variable "credentials_file_path" {
  description = "Path to the JSON file used to describe your account credentials"
  default     = "/GCP/k8s-lab/djs-gcp-2018.json"
}

variable "gce_ssh_user" {
  description = " ssh user that is used in the public key"
  default = "dspears@SJCMAC3024G8WL"
  }

variable "gce_ssh_pub_key" {
  description = " ssh key in the format:  ssh-rsa <key> username "
  default = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC9RA/kjVZ7pUrh4oMHLJP1jDeMc3pk0UCL3aQ9jthYJm/uWiVjLZ3NEjiHTfQQITtkbWQkfCcWHhnQ9E9vtLkkiwutpqT3wfm1nUeSA0rcTSFGMCB0tqVJds/u/KIAxYpS5I2Z/GYk1qcT9oXHSf/mr
//The rest of the variables do not need to be modified for the K8s Lab
// VM-Series Firewall Variables
```

➔ To get the GCP Project ID click the project selection drop down at the top of the GCP Console. The Project ID is displayed in the project selection window:



➔ Update the Variables.tf file with the GCP ID:



```
// PROJECT Variables
variable "my_gcp_project" {
  default = "djs-gcp-2018"
}

variable "region" {
  default = "us-central1"
}

variable "zone" {
  default = "us-central1-a"
}

variable "credentials_file_path" {
  description = "Path to the JSON file used to describe your account credentials"
  default     = "/GCP/k8s-lab/djs-gcp-2018.json"
}

variable "gce_ssh_user" {
  description = " ssh user that is used in the public key"
  default = "dspears@SJCMAC3024G8WL"
  }

variable "gce_ssh_pub_key" {
  description = " ssh key in the format:  ssh-rsa <key> username "
  default = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC9RA/kjVZ7pUrh4oMHLJP1jDeMc3pk0UCL3aQ9jthYJm/uWiVjLZ3NEjiHTfQQITtkbWQkfCcWHhnQ9E9vtLkkiwutpqT3wf
//The rest of the variables do not need to be modified for the K8s Lab
// VM-Series Firewall Variables
```

End of Activity 2

Activity 3 – Deploy the Terraform Template

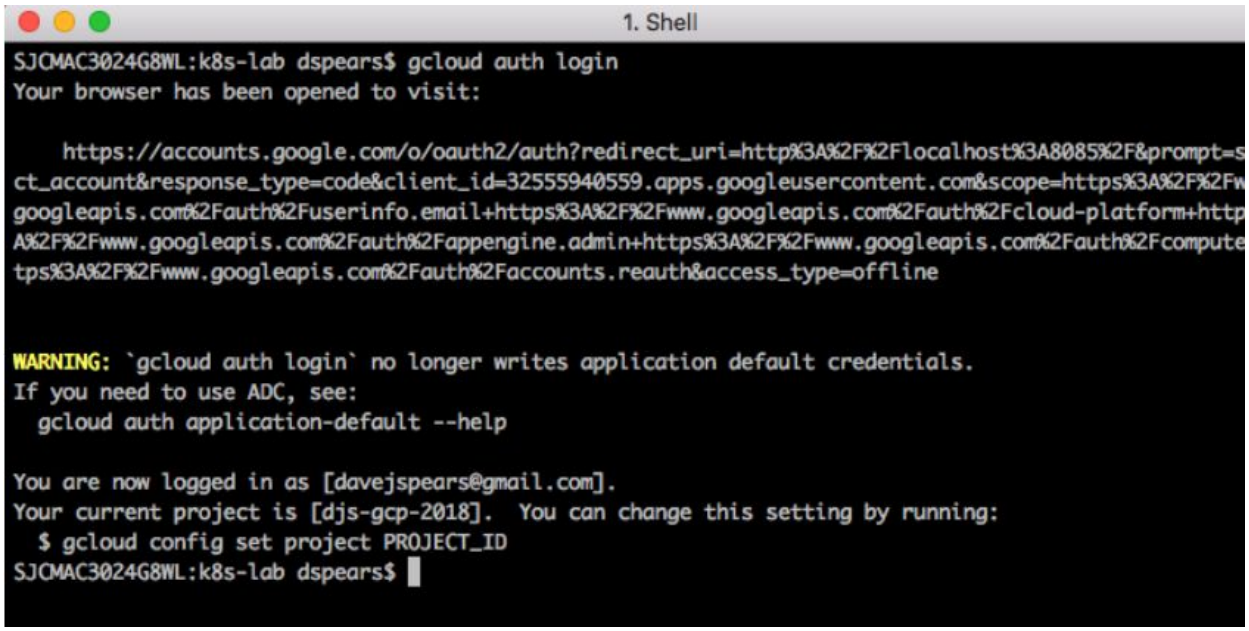
In this activity, you will:

- ***Authenticate Google Cloud SDK***
- ***Initialize and deploy the terraform template***

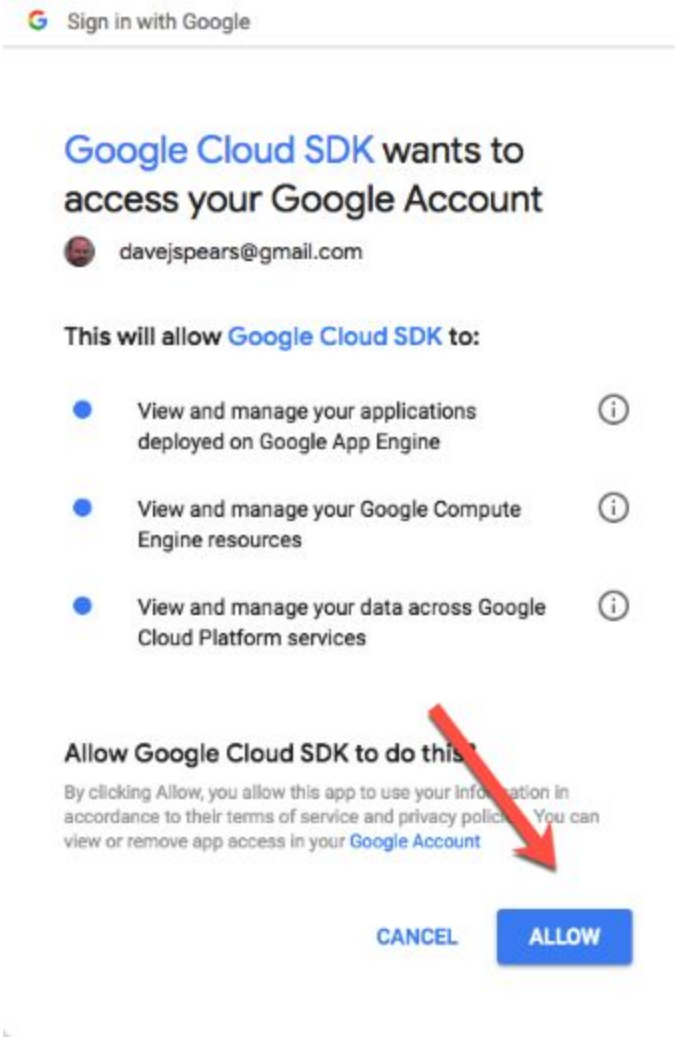
Task 1 – Authenticate Google Cloud SDK

➔ Open a terminal shell and navigate to the directory containing the Terraform template files.Authenticate to the GCP environment from the command line with the command: gcloud auth login

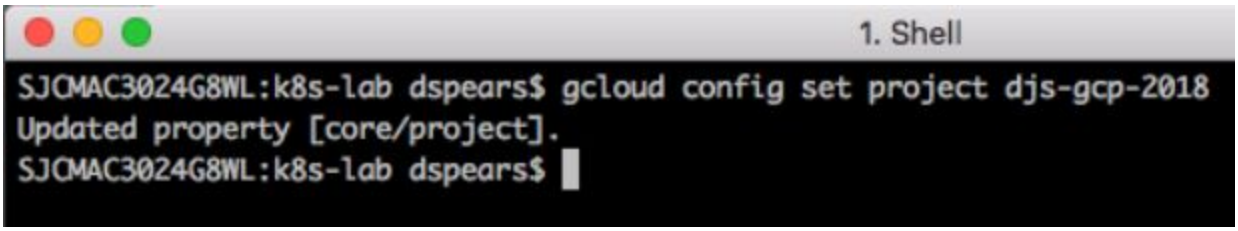
➔ Copy/paste the link into a browser(if needed) and select the account to authenticate if a browser does not automatically launch:



➔ Review the requested permissions and click Allow:

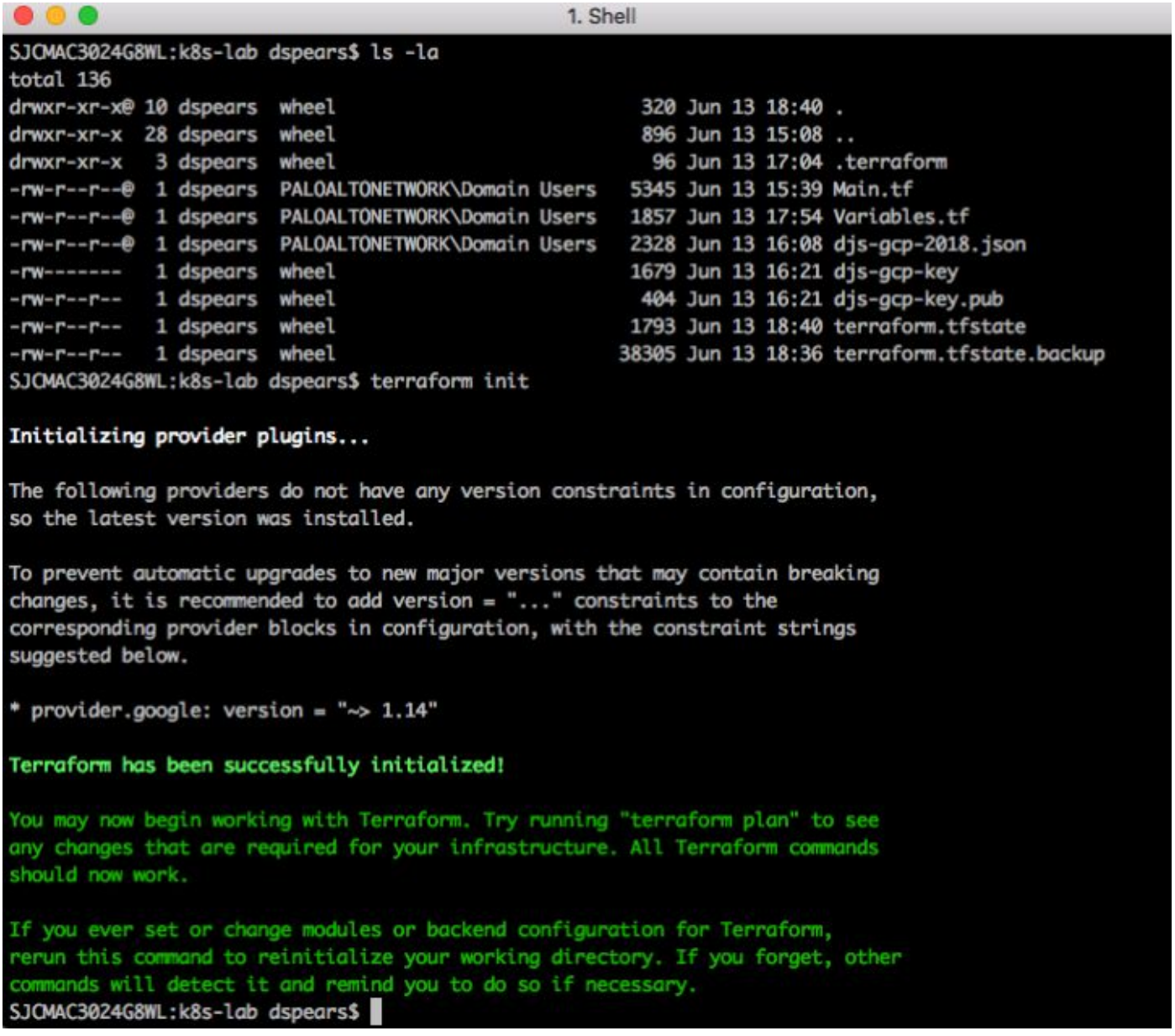


➔ Note the Project ID. If it is not the project that was created for the lab, use the **$ gcloud config set project my_Project_id** command to change the project:

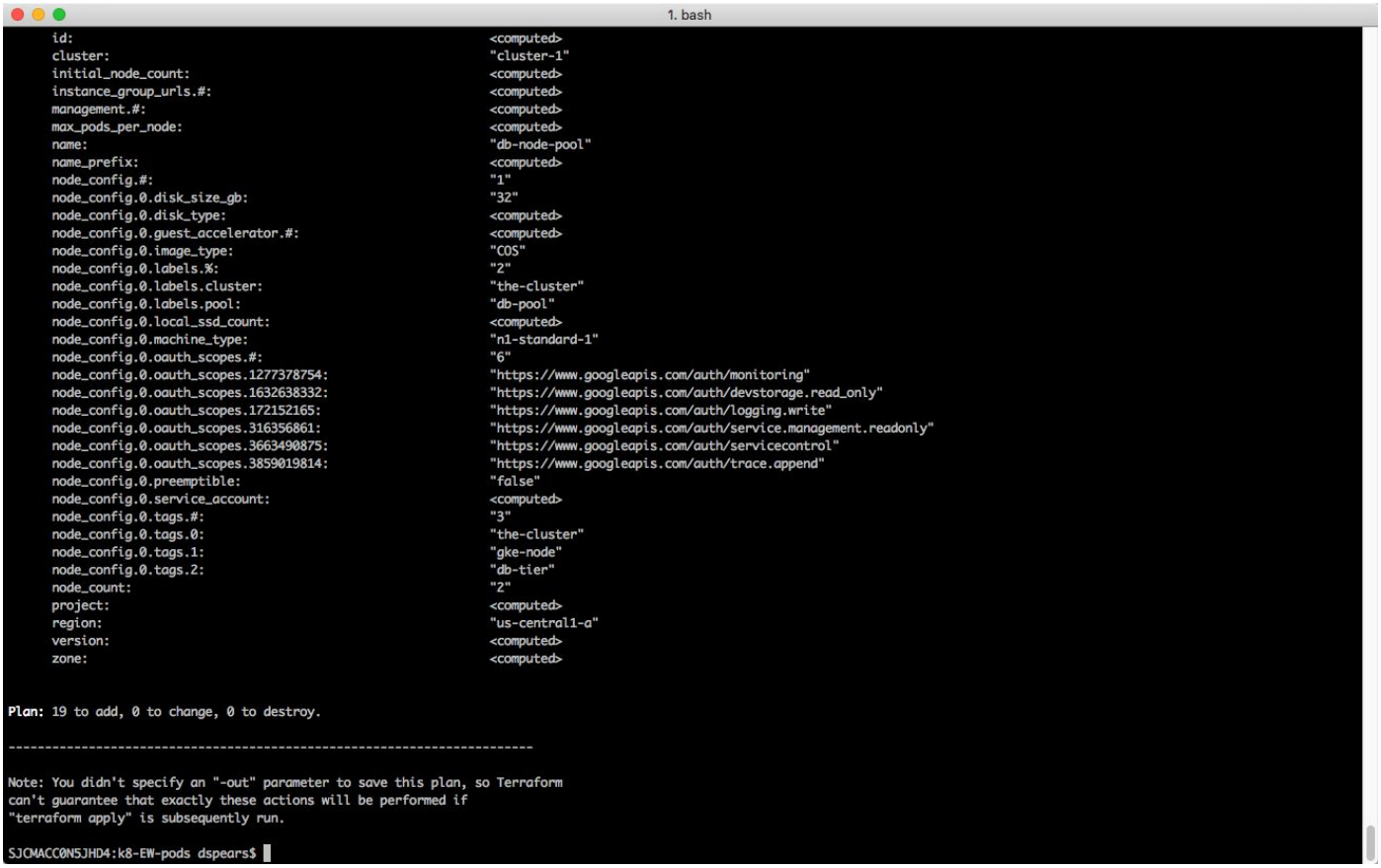


Task 2 – Deploy the Terraform Template

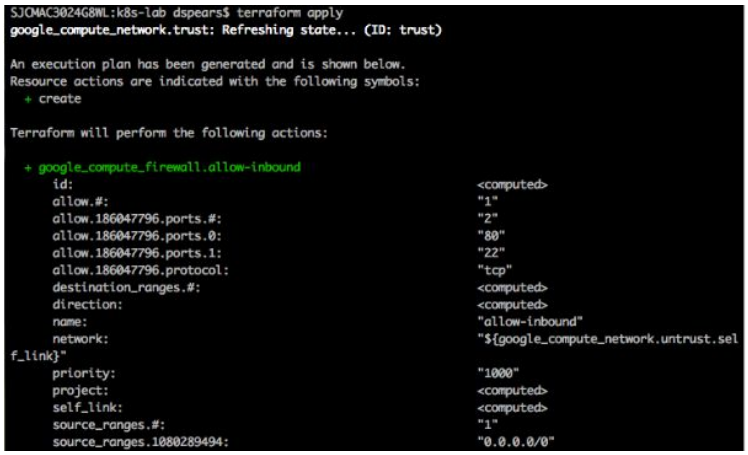
➔ Ensure you are in the directory with the Main.tf and Variables.tf files and execute the **terraform init** command which will initialize terraform:

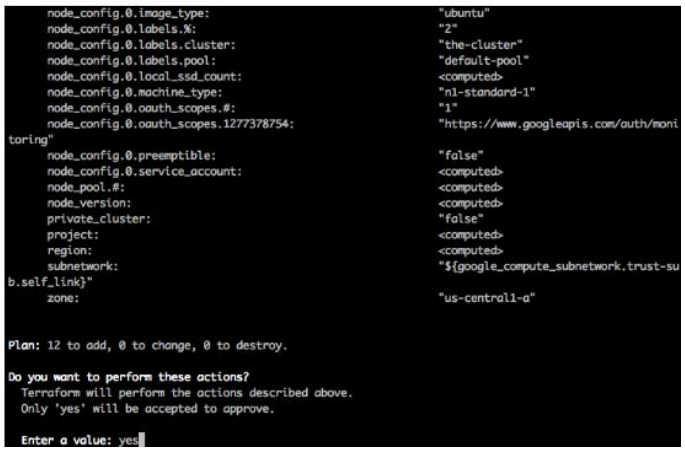


Once the terraform init has completed run the **terraform plan** command. This will show what changes will be implemented with the terraform script. This will also identify if there are any errors detected with the terraform files:

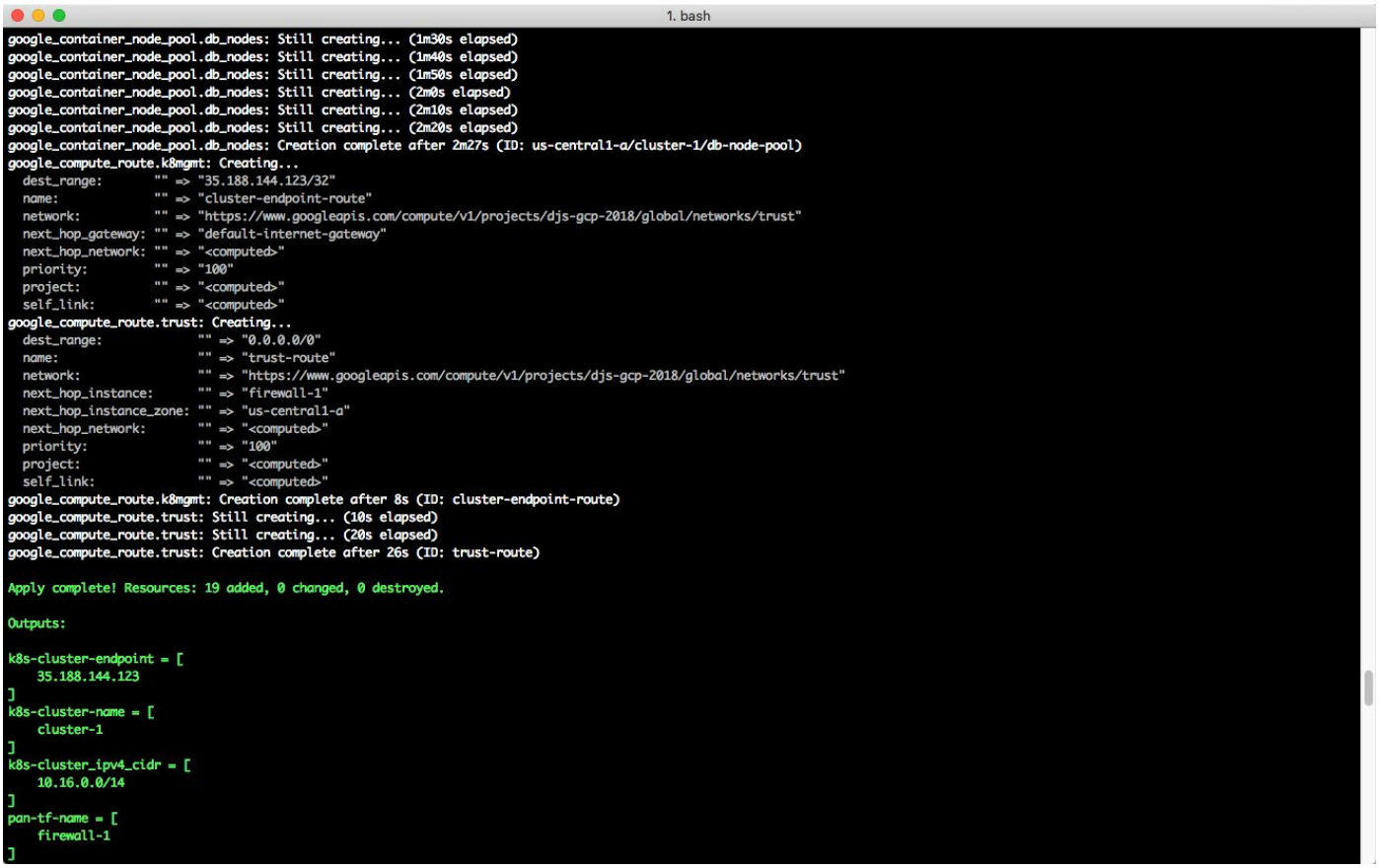


➔ Now run the **terraform apply** command to deploy the template. At the action prompt enter **yes**.





It will take a few minutes to complete. If all goes well, Terraform will output; “Apply Complete!” and provide some additional output information about the resources deployed:



End of Activity 3

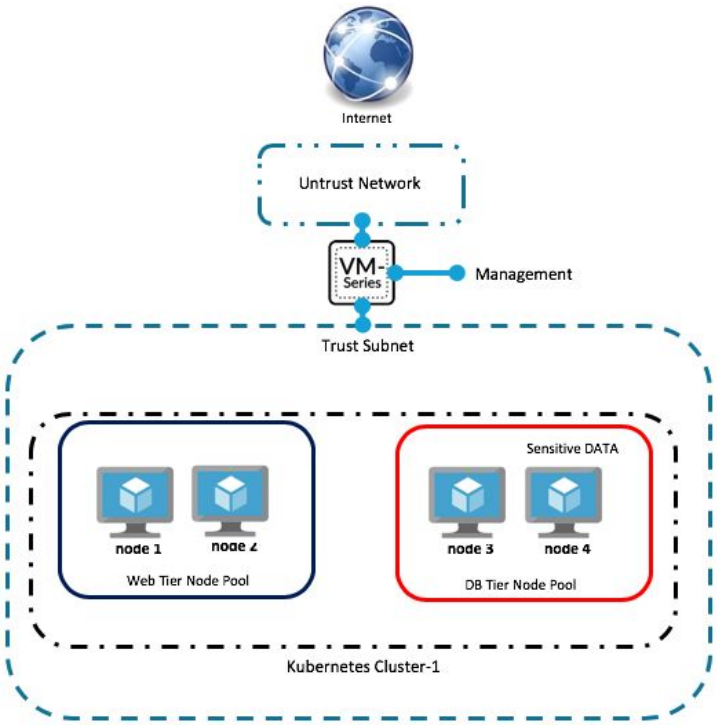
Activity 4 – Review what was deployed

In this activity, you will:

- ***Review the resources that have been launched***
- ***Log into the VM-Series firewall***
- ***Confirm bootstrap success and firewall licensing***

Task 1 – Understand what has been initially deployed

During the lab environment creation a number of things have been deployed automatically. The following diagram shows the initial lab deployment:

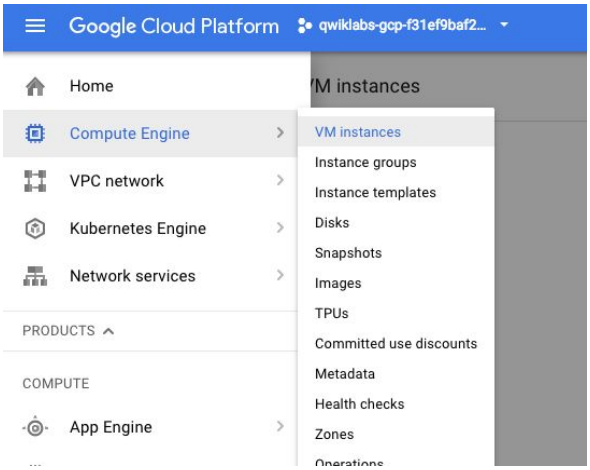


Some things to Note:

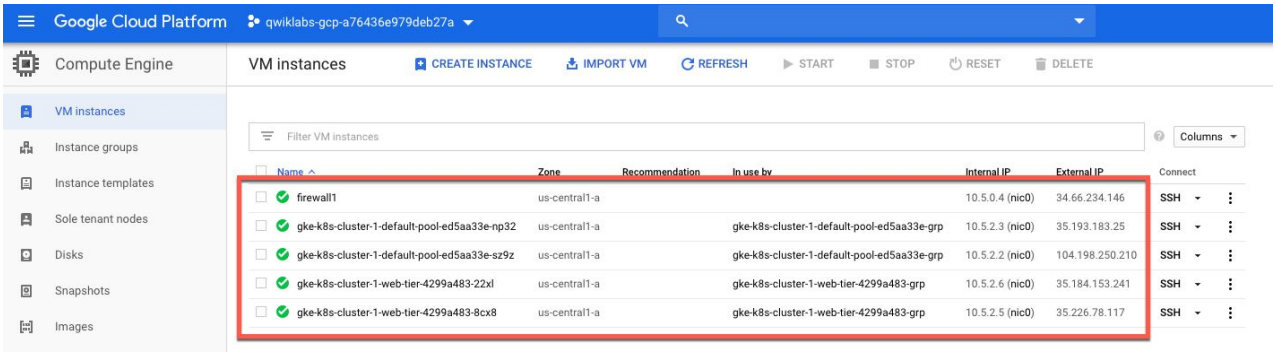
- A VM-Series Firewall has been bootstrapped with an initial configuration
- A K8s cluster has been created with 4 nodes in two separate node pools
- The VM-Series will be used for both North/South and East/West Inspection.

Task 2 – Look around GCP console

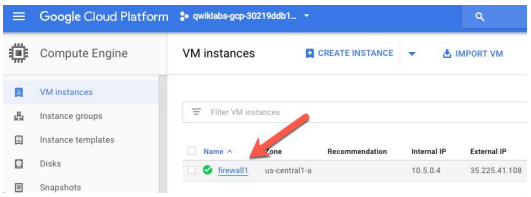
➔ Navigate to **Compute Engine > VM Instances** to see that VM-Series firewall has already been launched when the lab was started.



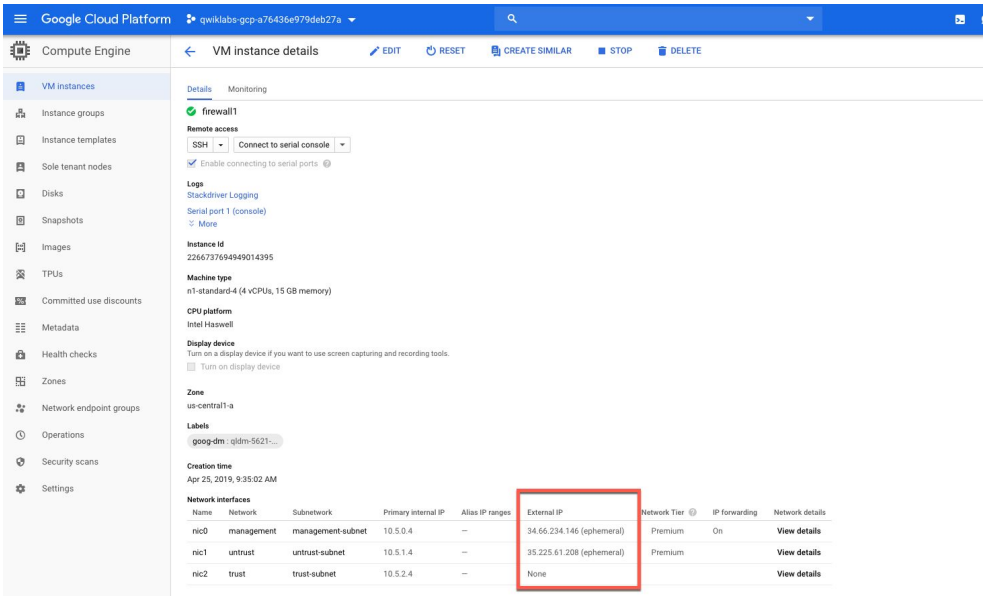
You should see a firewall and 4 nodes that were deployed as part of the k8's cluster:



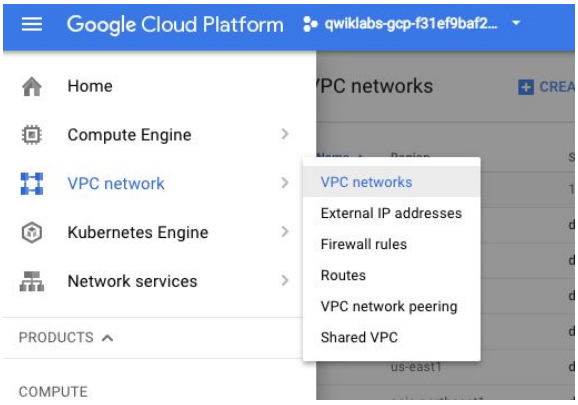
➔ Clicking on the **firewall1** opens a detailed view of the deployed firewall:



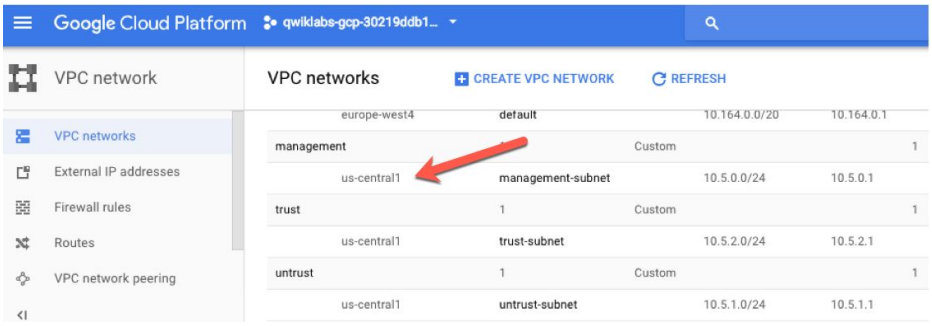
Note the **External IP** Addresses of the firewall on the firewall VM Instance details screen. These will be used later in the lab to connect the firewall and test application functionality:



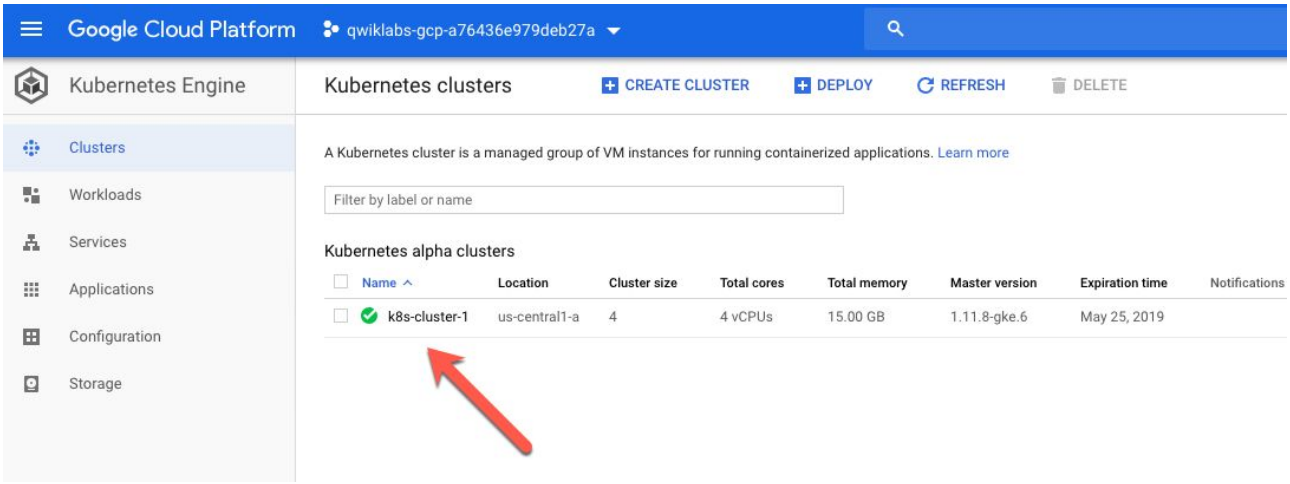
➔ Navigate to **VPC Network > VPC Networks** to see the different networks that have been created as part of the lab.



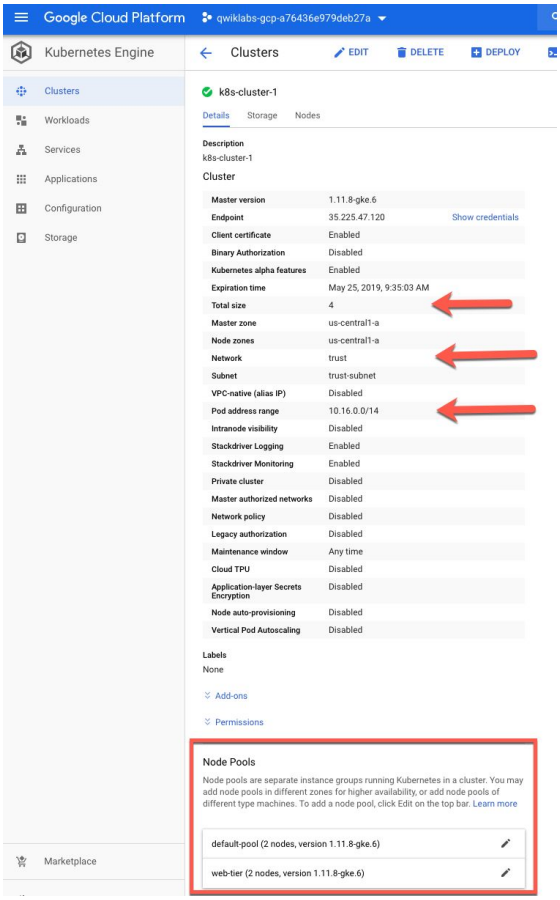
You should see 3 non-default VPC networks: management, untrust and trust.



➔ Navigate to **Kubernetes Engine > Clusters** and validate that the K8s cluster was successfully built and is running:

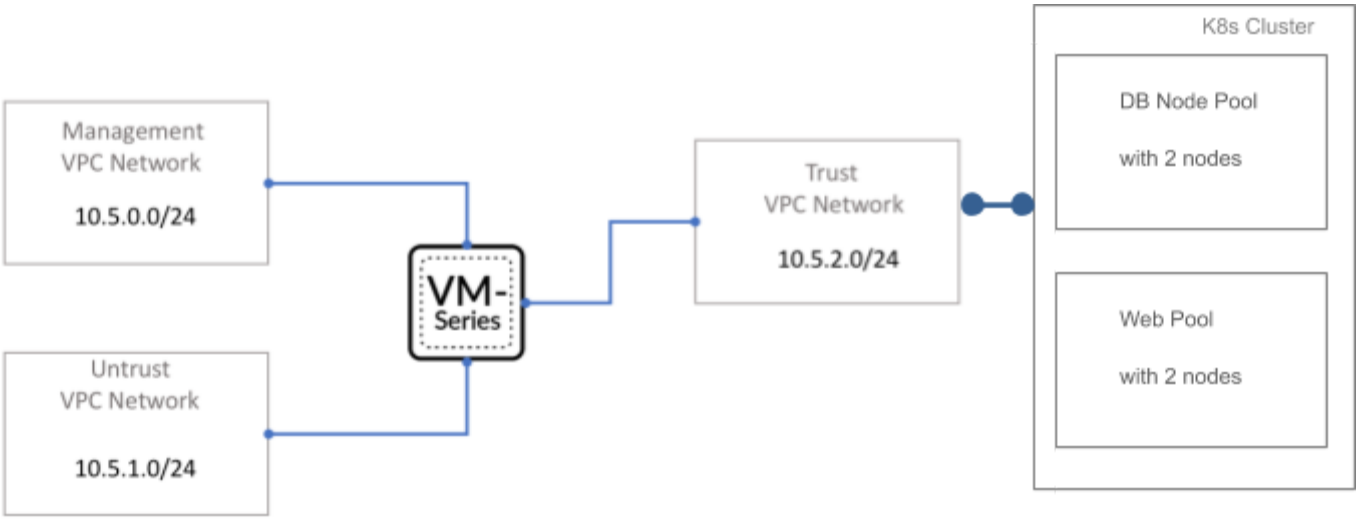


➔ Clicking on the cluster will open a more detailed view of the k8s cluster that was deployed:



Note: the **number of nodes (Total size)**, **node networks**, and **Pod IP address ranges** in the K8s Cluster details.

The following diagram describes the topology that has been deployed:

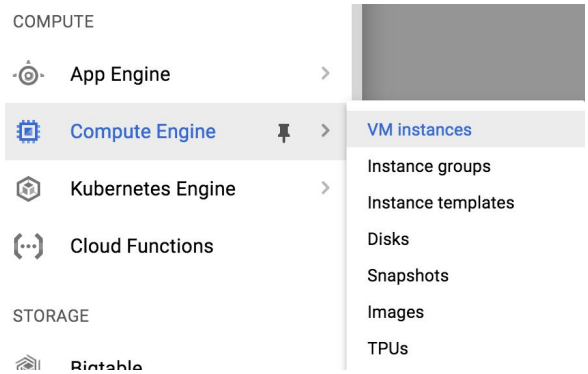


Feel free to navigate through other parts of the Google Cloud Console such as **VPC Networks > Routes**, **Firewall Rules**. This will come in handy in activities later on.

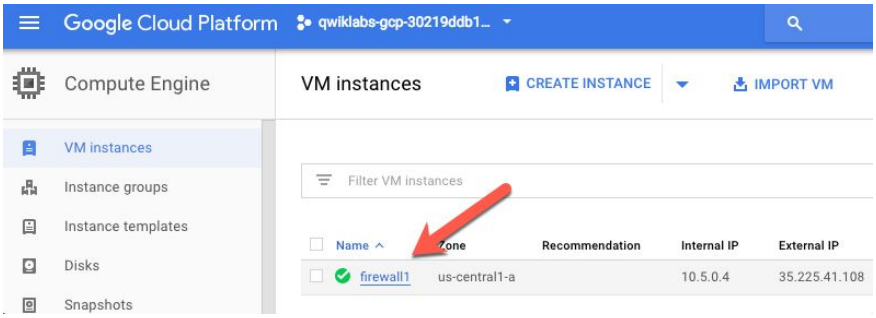
Task 3 – Login into the firewall

The VM-Series firewall deployed as part of the lab has been bootstrapped. Bootstrapping is a feature of the VM-Series firewall that allows you to load a predefined configuration into the firewall during boot-up. This ensures that the firewall is configured and ready at initial boot-up, thereby removing the need for manual configuration. The bootstrapping feature also enables automated deployment of the VM-Series.

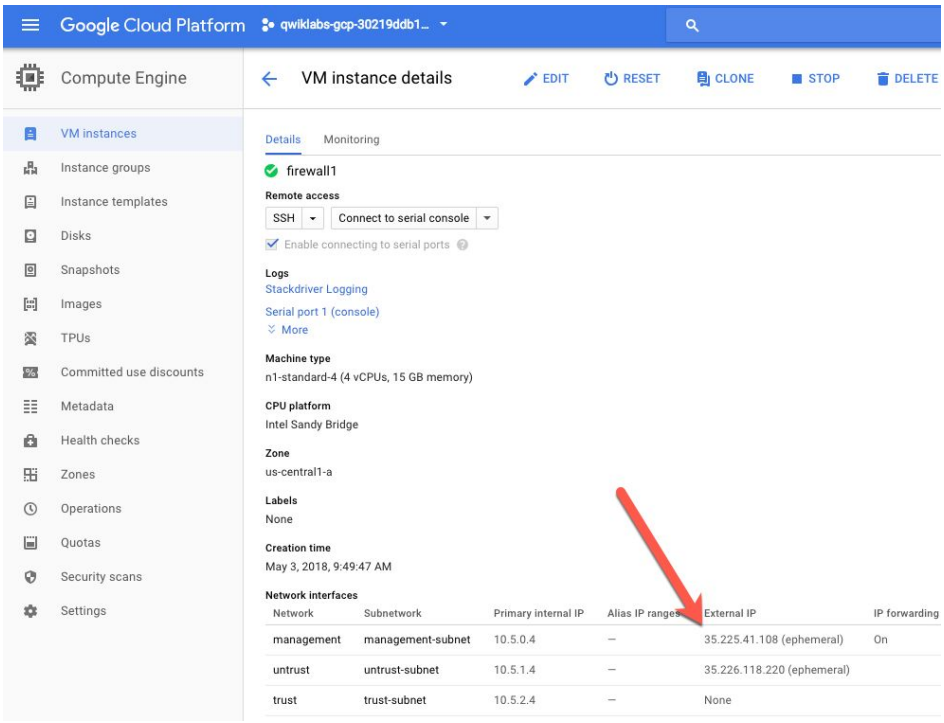
➔ Navigate to **Compute Engine > VM Instances**



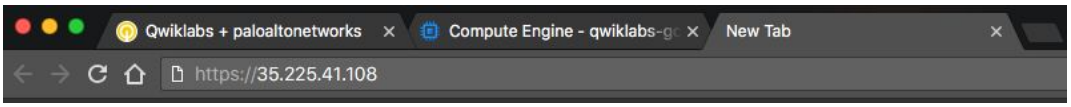
➔ Click on **firewall1** instance name to get more information and identify the management interface IP.



➔ Copy the **External public IP** of the management interface



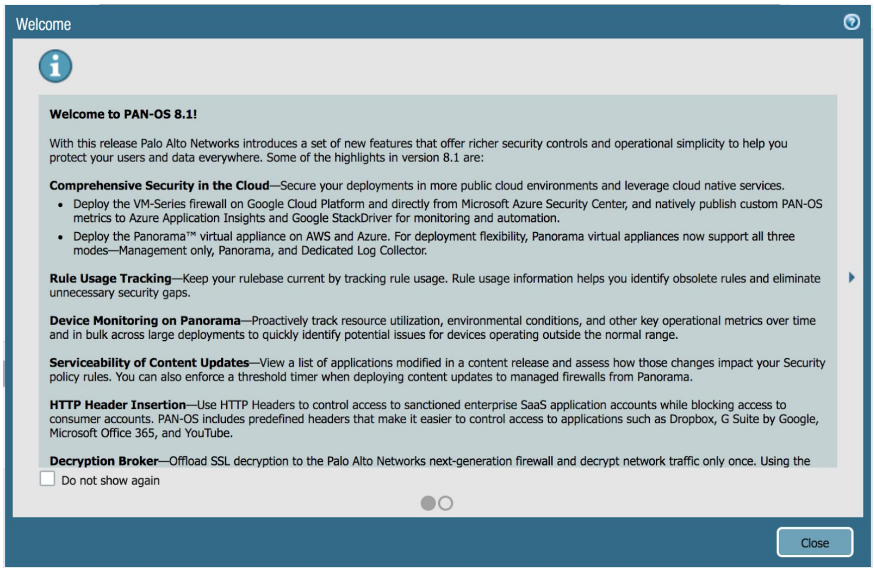
➔ Open another browser tab and navigate to the firewall management interface:



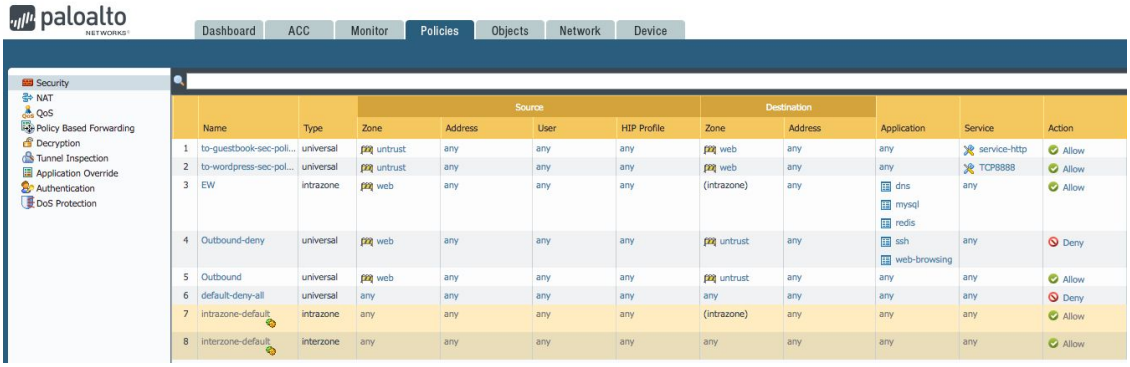
If you get a security exception, please ignore for this lab and proceed to the firewall login page. We are using a self-signed certificate which causes the exception. When presented with the login screen you should be able to login to the firewall using (Hint: It's a good idea to jot this password down or save it to a notepad as you will regularly need it):

username: **admin**
password: **Pal0Alt0@123** **Note:Those are zeros not capital O's.**

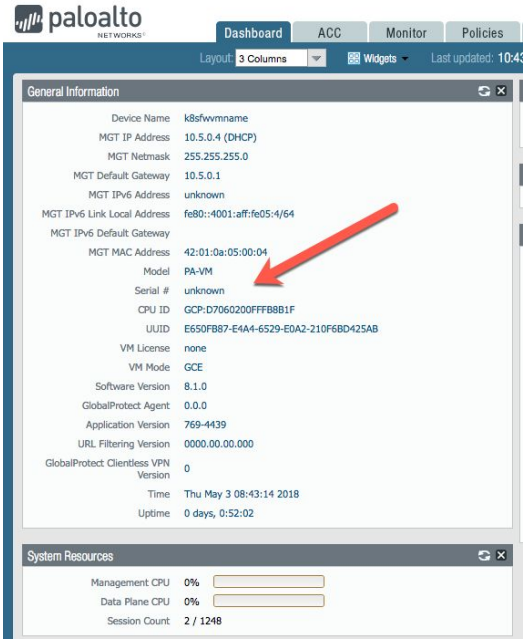
Once logged in you will see a **Welcome** screen, dismiss the welcome dialog box by clicking **Close**.



➔ Click the **Policies** tab and you will notice a predefined security policy which was imported using the bootstrapping feature. There are also some predefined NAT policies:



➔ Click on the **Dashboard** tab, check to verify that the firewall has a serial number.



We are using the PayAsYouGo image from GCP Cloud Launcher. A license will be required to view the logs later in the lab.

End of Activity 4

Activity 5 – Container Image Scanning for Vulnerabilities

In this activity, you will:

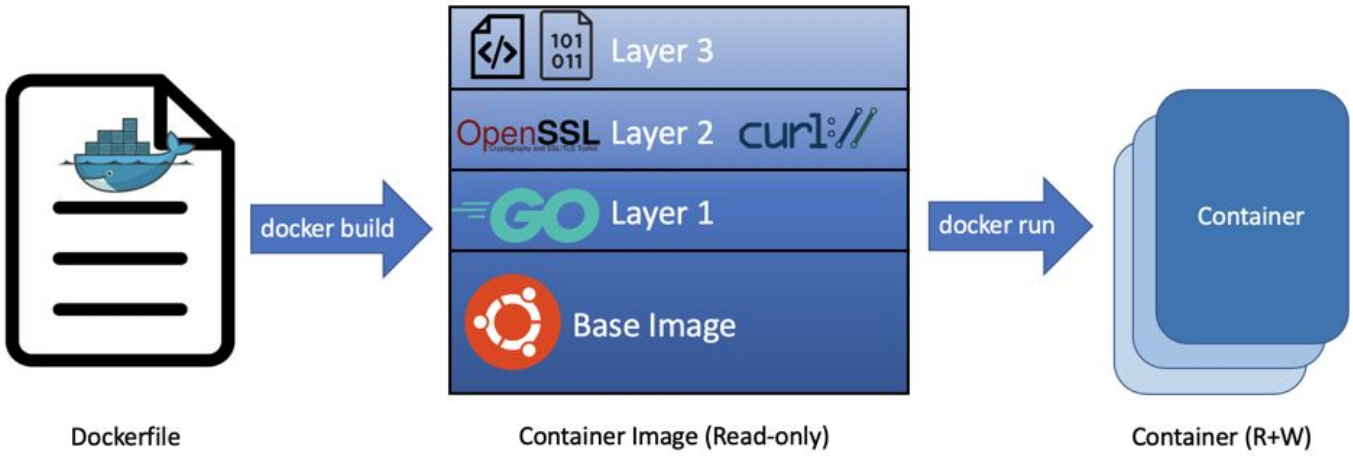
- ***Scan your container images for security vulnerabilities using Prisma Public Cloud (formerly Redlock) free public APIs***
- ***Scan publicly available container images for security vulnerabilities***
- ***Patch the images***
- ***Push the patched image to your container registry***

What are container images?

A container image is a lightweight, standalone, executable packaging of software which includes everything needed to run an application: code, runtime, libraries and local configuration.
Container images are made up of different layers. Every container image has a base layer (parent layer) which is usually an Operating System. The subsequent layers are built on top of it which might include language runtimes, libraries, and code (file, executables), etc.
Each of the layers are immutable and built on top of the previous layer. These layers are independent of each other. For example OpenSSL can be installed on many different base images (OS). Most of the layers are reusable such as base layer, libraries, language runtimes, which are pulled from internal or external shared repositories such as DockerHub, GitHub, npm, etc.

How are containers built?

Dockerfile:
Dockerfile is the manifest with build instructions on how to build a specific container image.
Image:
Container images are read-only templates from which containers are launched. Each image contains a series of layers as explained above.
Container:
A container is a running and mutable (Read + Write) form of the image.



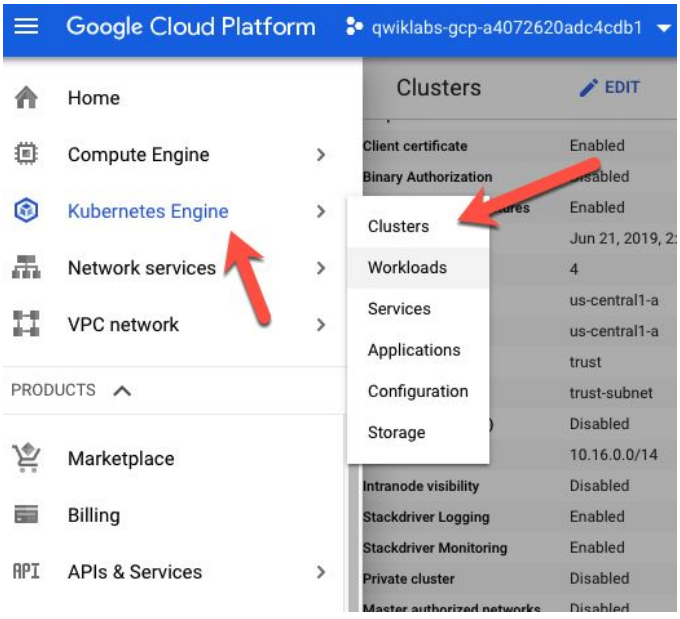
What does “scanning” my image mean?

Prisma Public Cloud (formerly Redlock) Image Scanning service provides free public API to scan your container images. When you “scan” an image, you are getting a list of all the vulnerabilities from all the packages and base OS installed in the image across all the layers. Each layer could contain multiple packages. The scan result will give you all the known vulnerabilities grouped by severity or package.

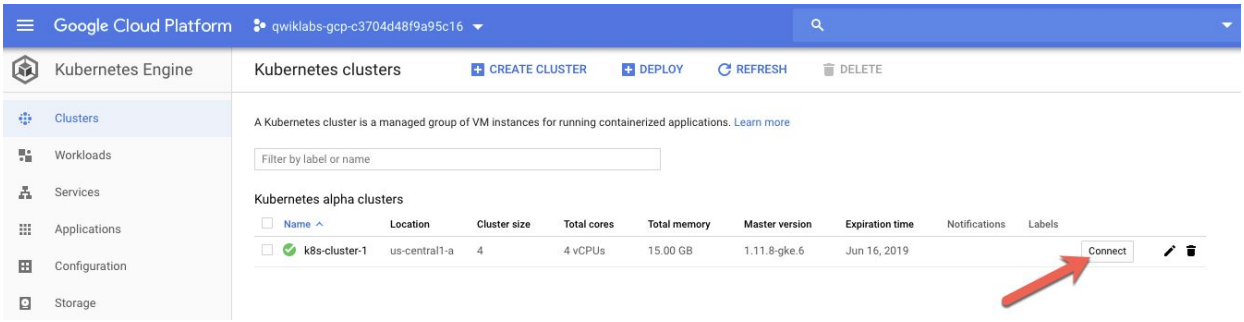
Task 1 – Connect to a Cloud Shell

NOTE: This is going to be done through the Kubernetes cluster so kubectl commands can be run later.

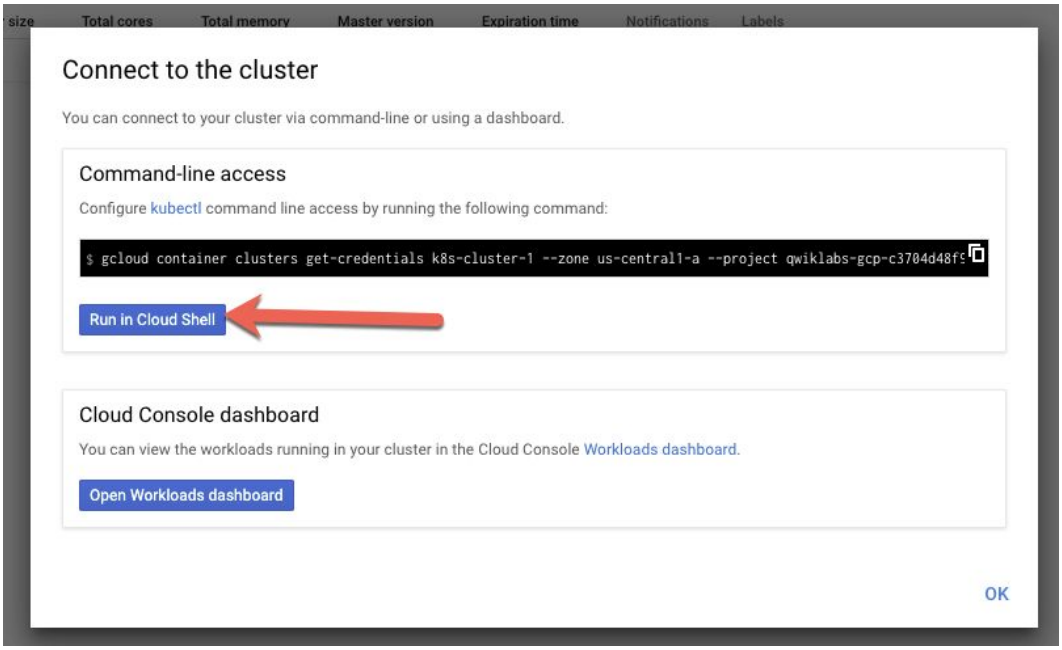
➔ Click on **Kubernetes Engine > Clusters**



➔ Click on the **Connect button** next to your cluster

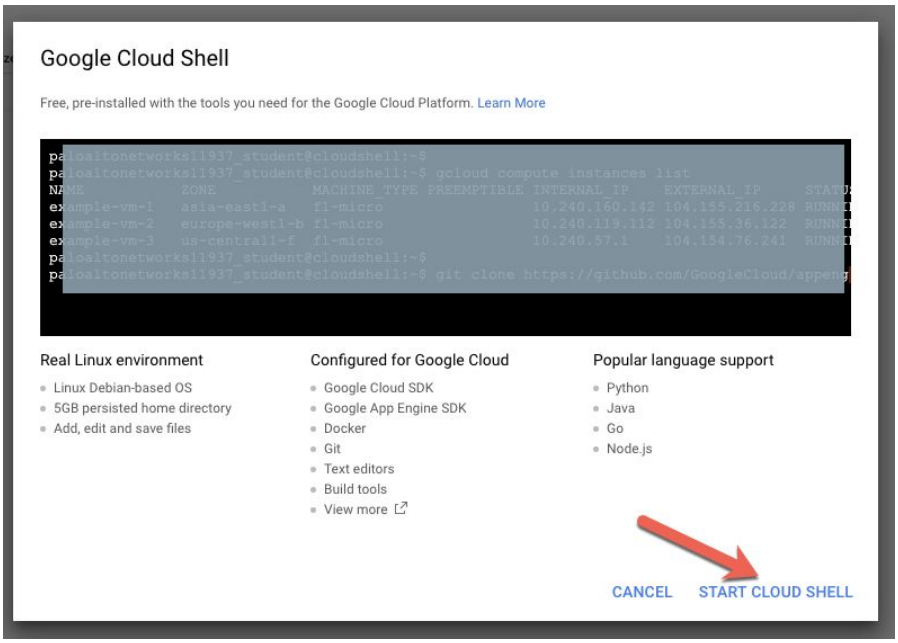


➔ Hit **Run in Cloud Shell** in the popup window



Note: If you're doing this for the first time, you will also see the following popup window.

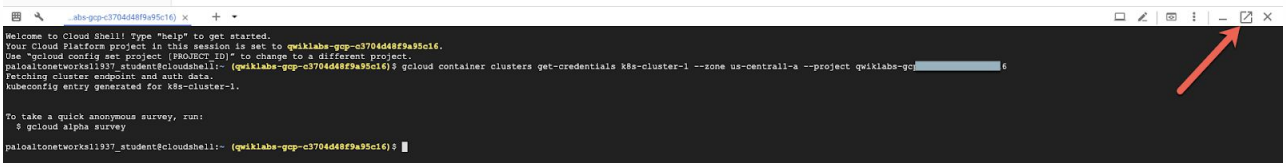
➔ Hit **START CLOUD SHELL** in that window



➔ Hit enter to execute the pre-populated command in the Cloud Shell to connect to the Kubernetes cluster

gcloud container clusters get-credentials k8s-cluster-1 --zone us-central1-a --project <project-id>

➔ Open the Cloud Shell in a new tab for convenience (DO NOT CLOSE THE NEW TAB)



Task 2 – Build and Scan the Application Container Image

In this activity we will start by building our app container image, then we will scan it for security vulnerabilities.

We will build the frontend service for our Guestbook app. The Development team wrote the code, and now we are packaging the code in a container.

This is the Dockerfile, which describes what we are including in this image (base OS/image, code and code dependencies/libraries):

```
1  # Copyright 2016 The Kubernetes Authors.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #     http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14
15 FROM php:5-apache
16
17 RUN apt-get update
18 RUN pear channel-discover pear.nrk.io
19 RUN pear install nrk/Predis
20
21 # If the container's stdio is connected to systemd-journald,
22 # /proc/self/fd/{1,2} are Unix sockets and apache will not be able to open()
23 # them. Use "cat" to write directly to the already opened fds without opening
24 # them again.
25 RUN sed -i 's#ErrorLog /proc/self/fd/2#ErrorLog "|$/bin/cat 1>\&2"#' /etc/apache2/apache2.conf
26 RUN sed -i 's#CustomLog /proc/self/fd/1 combined#CustomLog "|/bin/cat" combined#' /etc/apache2/apache2.conf
27
28 # Add the application code to the image
29 ADD guestbook.php /var/www/html/guestbook.php
30 ADD controllers.js /var/www/html/controllers.js
31 ADD index.html /var/www/html/index.html
```

In this Dockerfile…

- We are using PHP:5-apache ***base image*** for our frontend app container (line 15) https://hub.docker.com/_/php
- Installing and updating ***dependencies*** (line 17-19)
- Adding the frontend ***app code***, PHP, JavaScript, and HTML to the image (line 29-31)

Now, let's build and scan this image.

➔ Download the lab repo in Cloud Shell by running below cmd:

git clone https://github.com/PaloAltoNetworks/ignite2019-how14.git

```
paloaltonetworks11946_student@cloudshell:~ (qwiklabs-gcp-161afff642864f2c)$ git clone https://github.com/PaloAltoNetworks/ignite2019-how14.git
Cloning into 'ignite2019-how14'...
remote: Enumerating objects: 23, done.
remote: Counting objects: 100% (23/23), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 23 (delta 5), reused 20 (delta 5), pack-reused 0
Unpacking objects: 100% (23/23), done.
paloaltonetworks11946_student@cloudshell:~ (qwiklabs-gcp-161afff642864f2c)$ ls
ignite2019-how14  README-cloudshell.txt
```

➔ Access the directory with the Dockerfile by running the following command:

cd ignite2019-how14/code

```
paloaltonetworks11946_student@cloudshell:~ (qwiklabs-gcp-161afff642864f2c)$ cd ignite2019-how14/code/
paloaltonetworks11946_student@cloudshell:~/ignite2019-how14/code (qwiklabs-gcp-161afff642864f2c)$ ls
controllers.js  Dockerfile  Dockerfile.withScanning  guestbook.php  index.html  Makefile
```

➔ Edit the Dockerfile to add Prisma Public Cloud image scanning API call

➔ Open the Dockerfile in your favorite editor

nano Dockerfile

➔ Go to the end of the file and append the following two lines at the end

ARG rl_args

RUN SCAN_CMD=$(eval "curl https://vscanapidoc.redlock.io/scan.sh 2>/dev/null") && echo "$SCAN_CMD" | sh

➔ Save and exit

Press `Ctrl + o` to save and then `Ctrl + x` to exit

After making the changes Dockerfile should look like this:

```
paloaltonetworks11946_student@cloudshell:~/ignite2019-how14/code (qwiklabs-gcp-161afff642864f2c)$ tail Dockerfile
# them. Use "cat" to write directly to the already opened fds without opening
# them again.
RUN sed -i 's#ErrorLog /proc/self/fd/2#ErrorLog "|$/bin/cat 1>\&2"#' /etc/apache2/apache2.conf
RUN sed -i 's#CustomLog /proc/self/fd/1 combined#CustomLog "|/bin/cat" combined#' /etc/apache2/apache2.conf

ADD guestbook.php /var/www/html/guestbook.php
ADD controllers.js /var/www/html/controllers.js
ADD index.html /var/www/html/index.html
ARG rl_args
RUN SCAN_CMD=$(eval "curl https://vscanapidoc.redlock.io/scan.sh 2>/dev/null") && echo "$SCAN_CMD" | sh
```

What we're doing here by adding the r1_args and SCAN_CMD is, we are listing all the packages installed in this image and getting the list of all the vulnerabilities associated with those packages from the **Prisma Public Cloud free public image scanning API**. The **Prisma Public Cloud** Infrastructure as Code Scanner will provide a pass/fail for the build based on the list of vulnerabilities we get back.

ARG rl_args is for passing the build arguments to configure when to pass/fail the build and how to group/see the scan result. See https://vscanapidoc.redlock.io/ for more information.

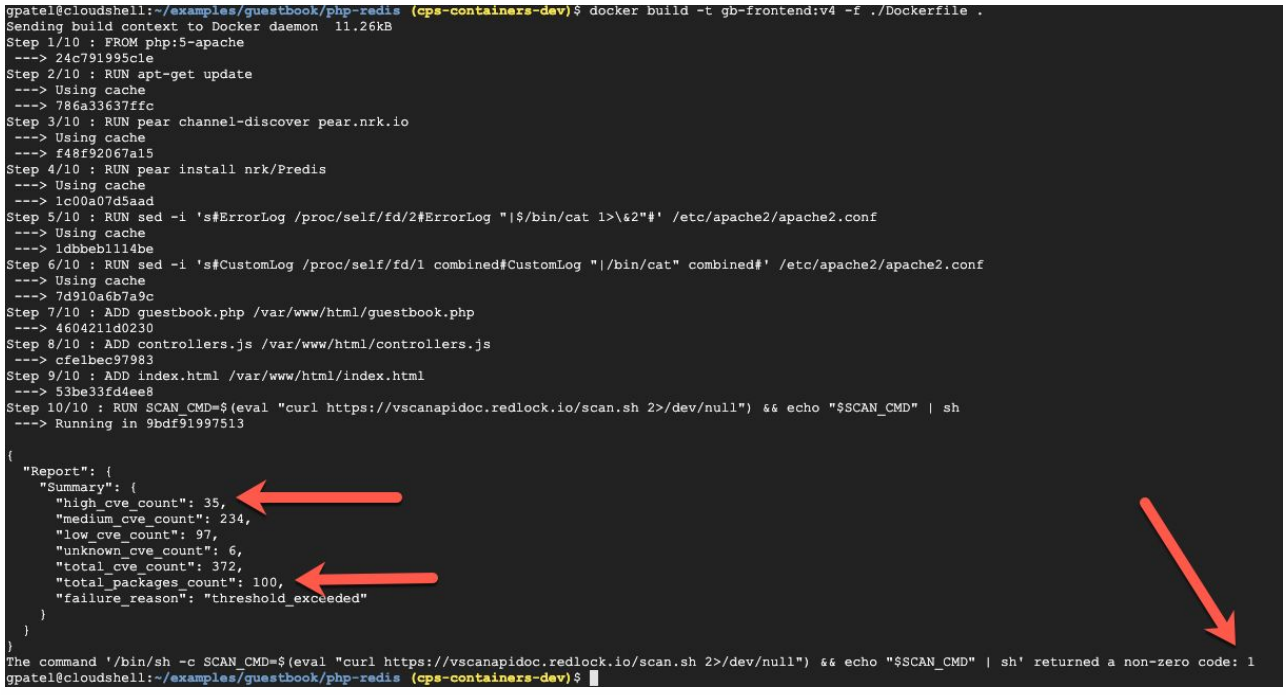
Note: *For your convenience, we have placed the final Dockerfile as* `Dockerfile.withScanning` in the `ignite2019-how14/code` folder.

➔ *You can copy that one using the following command:*

cp Dockerfile.withScanning Dockerfile

➔ Build the Docker image using the following command. This will make the actual API call during the build and display the scan result.

docker build -t gb-frontend:v4 . -f ./Dockerfile



➔ Next, analyze the completed results and take note of the following:

a. Notice the docker build failing with a non-zero exit code

b. It fails because the vulnerability scan result received from the Prisma Public Cloud image scan API endpoint indicate more than one packages have known vulnerabilities

c. Notice that the final image would have had 38 high severity CVEs, 248 medium and 102 low severity CVEs, totaling 394 CVEs.

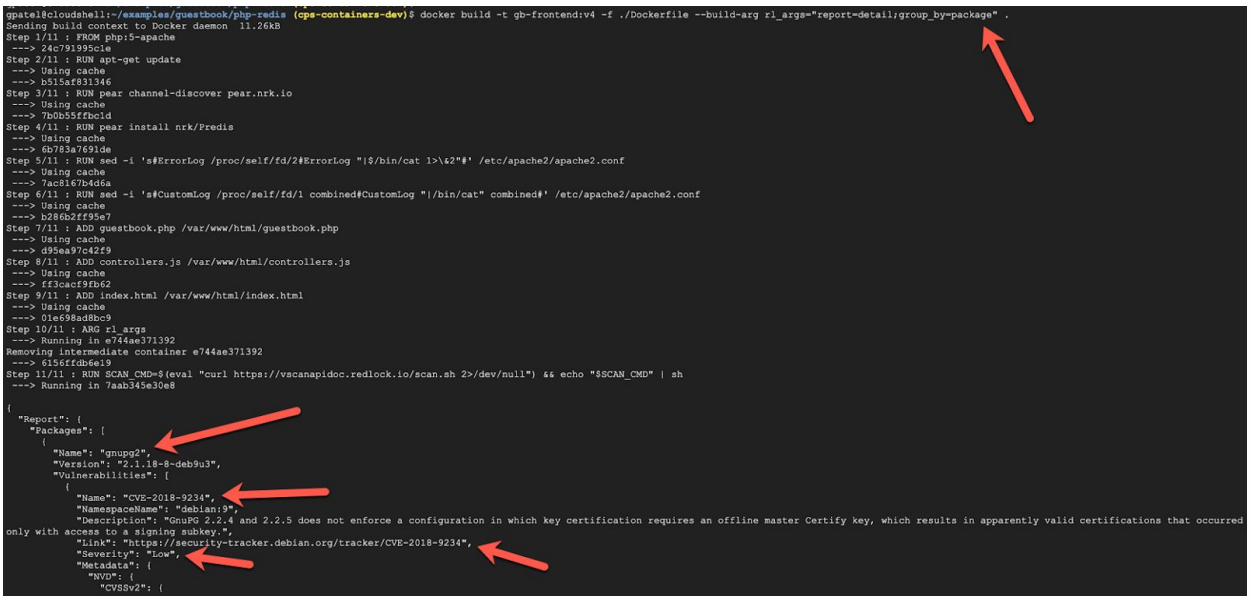
Note: Your results may be different as new CVEs are being identified.

d. The number of packages analyzed are 100

e. Failure reason is the number of CVEs exceeded the threshold (by default 1)

➔ Next, get the list of CVEs grouped by the packages by passing the ***--build-arg rl_args="report=detail;group_by=package"*** argument to the docker build command

docker build -t gb-frontend:v4 . -f ./Dockerfile --build-arg rl_args="report=detail;group_by=package"



Note: The output might look different

End of Activity 5

Activity 6 – Kubernetes App Manifest Scanning for Security Misconfigurations

In this activity, you will:

- ***Scan your kubernetes application deployment manifest using Prisma Public Cloud Infrastructure-as-Code (IaC) public API for security best practices***
- ***Analyze the result***
- ***Fix all the applicable misconfigurations***

In this activity we will start using Kubernetes specific terms such as Pods, Services, etc.
Here is a good primer: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

What is Kubernetes?

Kubernetes is an open-source container-orchestration system for automating application deployment, scaling, and management.

What is a Kubernetes Manifest?

Kubernetes manifest file describes how your containerized application is deployed in kubernetes. There can be one or more objects in a manifest file such as Deployment (replicated group of Pods), Services (proxy), Volumes, and ConfigMaps (configuration for the application pods/containers). Manifest files can be in JSON or YAML format. YAML format is more common in kubernetes world, so we will use that in this lab, but Prisma Public Cloud Infrastructure-as-Code (IaC) API supports both JSON and YAML format.

What does “scanning” the manifest file mean?

When you scan your kubernetes manifest files using the free Prisma Public Cloud Infrastructure-as-Code (IaC) API, you get back the analysis result that points of any configuration which is vulnerable to exploitation. The scan result will have severity associated with each of the rule violations.

You can include this scan into your CI/CD (Continuous Integration/ Continuous Delivery) pipeline, so all your kubernetes manifests go through an automated sanity check before they are applied to production. CI Build should fail if any of your manifest has a high security security misconfiguration.

This API also allows you to scan Terraform and CFT files for security best practices violations. Detailed documentation can be found here: https://iacscanapidoc.redlock.io/

Task 1 – Scan the Application Manifest for Security Best Practices

➔ Back in the Cloud Shell, explore the manifest files. First, go back to the repo base folder *ignite2019-how14/* by executing the following command:

cd ..

➔ Next view the guestbook application manifest by executing the following command:

cat guestbook-ew.yaml

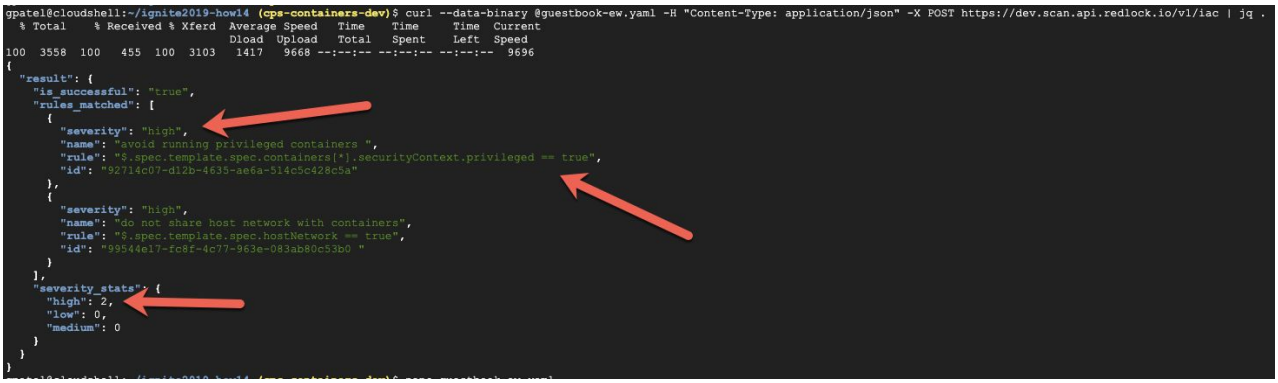
```
paloaltonetworks11951_student@cloudshell:~/ignite2019-how14/code (qwiklabs-gcp-a4072620adc4cdb1)$ cd ..
paloaltonetworks11951_student@cloudshell:~/ignite2019-how14 (qwiklabs-gcp-a4072620adc4cdb1)$ ls
code  guestbook-ew.yaml  README.md
paloaltonetworks11951_student@cloudshell:~/ignite2019-how14 (qwiklabs-gcp-a4072620adc4cdb1)$ cat guestbook-ew.yaml
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    app: redis
    tier: backend
    role: master
spec:
  #type: LoadBalancer
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    tier: backend
    role: master
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: redis-master
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: master
        tier: backend
    spec:
      hostNetwork: true
      containers:
      - name: master
        image: gcr.io/google_containers/redis:e2e  # or just image: redis
        securityContext:
          privileged: true
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 6379
      nodeSelector:
        pool: db-pool
```

➔ Scan the guestbook app manifest with Prisma Public Cloud IaC (Infrastructure-as-Code) Scan API using the following command:

curl --data-binary @guestbook-ew.yaml -H "Content-Type: application/json" -X POST https://scanapi.redlock.io/v1/iac | jq .

Note: Do not forget the period at the end.

➔ Analyze the results after the previous curl call:



As you can see from the scan result, we have 2 potential security misconfigurations in our manifest:

a. A container is running in privileged mode which can be dangerous
b. Pods in a deployment are sharing network namespace with the host

Both of these are classified as high severity security best practice violations, as you can see from the severity field for both of the violations.

Task 2 – Update the Manifest to Fix the Policy Violations

If these configuration lines are not absolutely necessary then we should remove them. You should work with your developer and security team to discuss other options to avoid these offending configurations which can be potentially exploited. In our case, we will assume we have consulted with our dev and security team and decided to remove both offending violations.

➔ Open the manifest in your favorite text editor :)

nano guestbook-ew.yaml

➔ Remove the following lines (line 32)

```
hostNetwork: true
```

and (line 36-37)

```
securityContext:
     privileged: true
```

Use your choice of editor (vi/nano) to modify the `guestbook-ew.yaml` file. To delete a line in the nano editor, you can move your cursor to the line you want to delete and then press `Ctrl + k` to delete that line

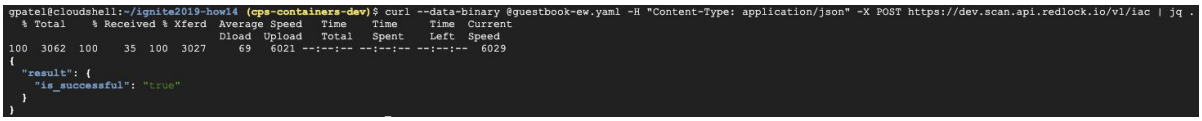
➔ Save and exit

Press `Ctrl + o` to save and then `Ctrl + x` to exit

➔ Rescan the guestbook app manifest again to make sure the policy violations are cleared by executing the following command again:

curl --data-binary @guestbook-ew.yaml -H "Content-Type: application/json" -X POST https://scanapi.redlock.io/v1/iac | jq .

➔ Validate that the policy violations are gone!



You can also scan your Terraform and CFT template files with the same Prisma Public Cloud IaC public API endpoint, and they're all provided for free.

For more details on Kubernetes manifest scanning and IaC API documentation, checkout the documentation page: https://scanapidoc.redlock.io/

End of Activity 6

Activity 7 – Launch a two tiered application

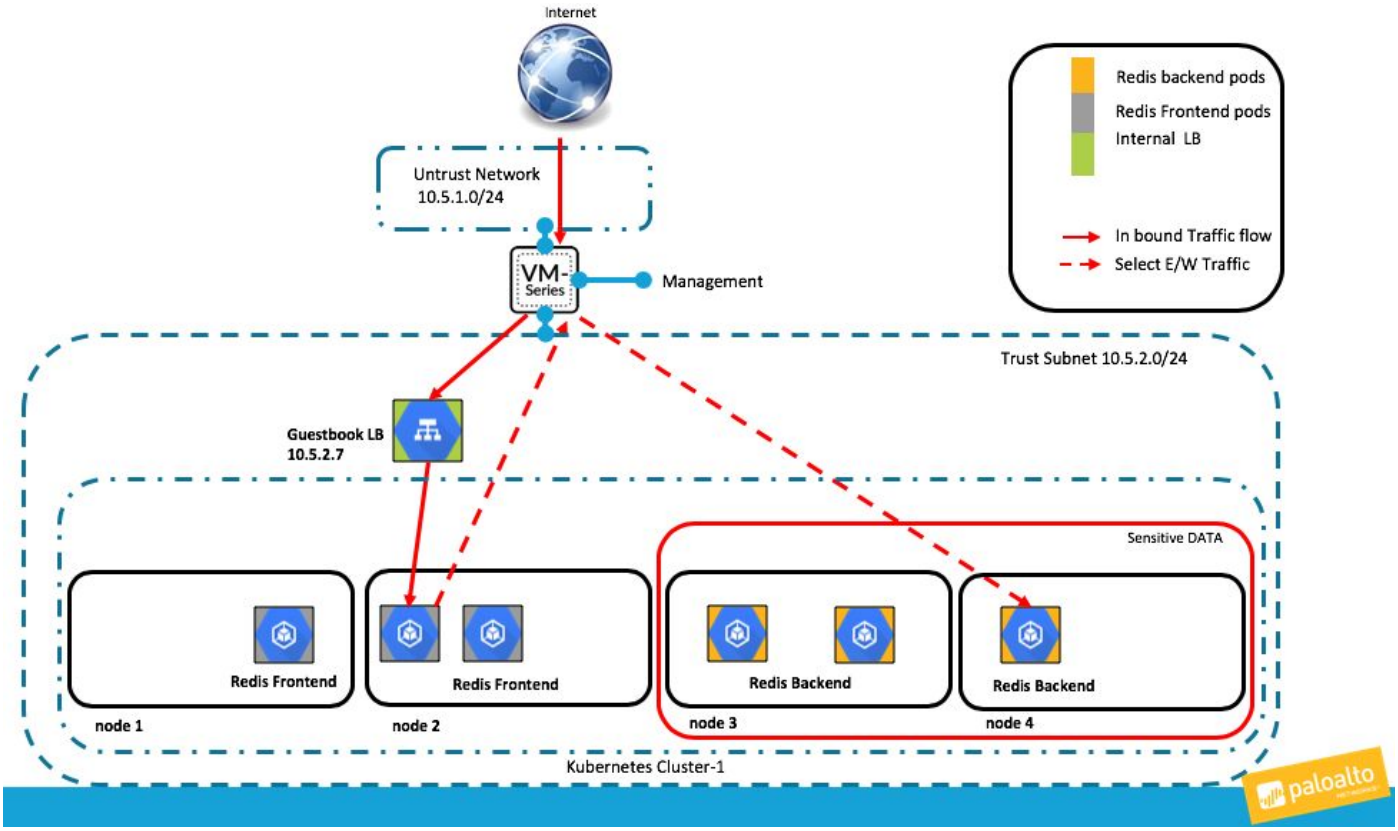
In this activity, you will:

- ***Optionally: Explore the application's manifest file that was scanned in Activity 4***
- ***Launch a two tier application within your cluster***
- ***Explore aspects of the application and how a VM-Series Firewall can be used to inspect both North/South and East/West traffic.***

Task 1 – Inspect the Guestbook Manifest file

In Activity 4 a Guestbook Manifest file was scanned for CVEs and corrected based on the results of the Prisma Public Cloud IaC public API. Guestbooks have been used by businesses for many years as a way to connect with customers and obtain contact information for future events and promotions. Today, businesses such as popular retail stores, 5-star hotels and even small family-owned B & B's are turning to iPad guestbook apps to help them gather information and enhance the customer's "in-biz" experience. Acquiring email addresses and a **social media** following is a crucial part of any marketing plan. With much of the population using computers on a daily basis, an email marketing plan is of the utmost importance. Using a guest book app in your store makes collecting email addresses a snap and offers enticing features with which the traditional paper and pen guestbook just can't compete. The guestbook application we will build and secure today could be used for Hotel website visits, shopping sites or any other business that wants to keep track of their customers and provide them with promotions or advertisements.

During the initial deployment a K8s cluster with a VM-Series firewall was deployed. Now the guestbook manifest file that was used in Activity 4 will be deployed to the K8's environment. Once deployed the environment will look like the following diagram:



As you can see this is a two-tiered application with Pods that are dedicated to front-end web services and backend DB services. By selecting which pods are placed in the Node Pool designated for sensitive data, it is possible to analyze select intrapod traffic. This results in sensitive information benefiting from increased protections over cloud native port/protocol access lists.

If interested, the following section dives a bit deeper into the templates being used to create this application. This is a link to the application manifest.

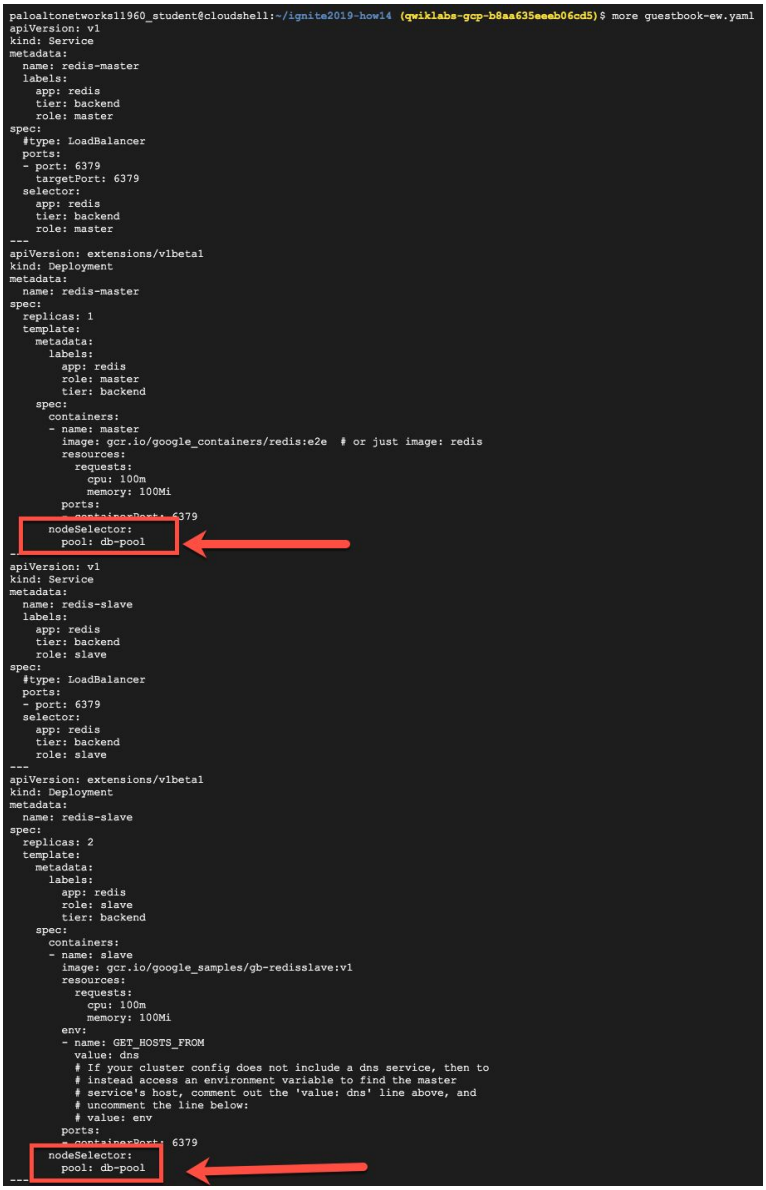
➔ Optionally, click the link below and open it in a browser of your choice.

https://github.com/PaloAltoNetworks/ignite2019-how14/blob/master/guestbook-ew.yaml

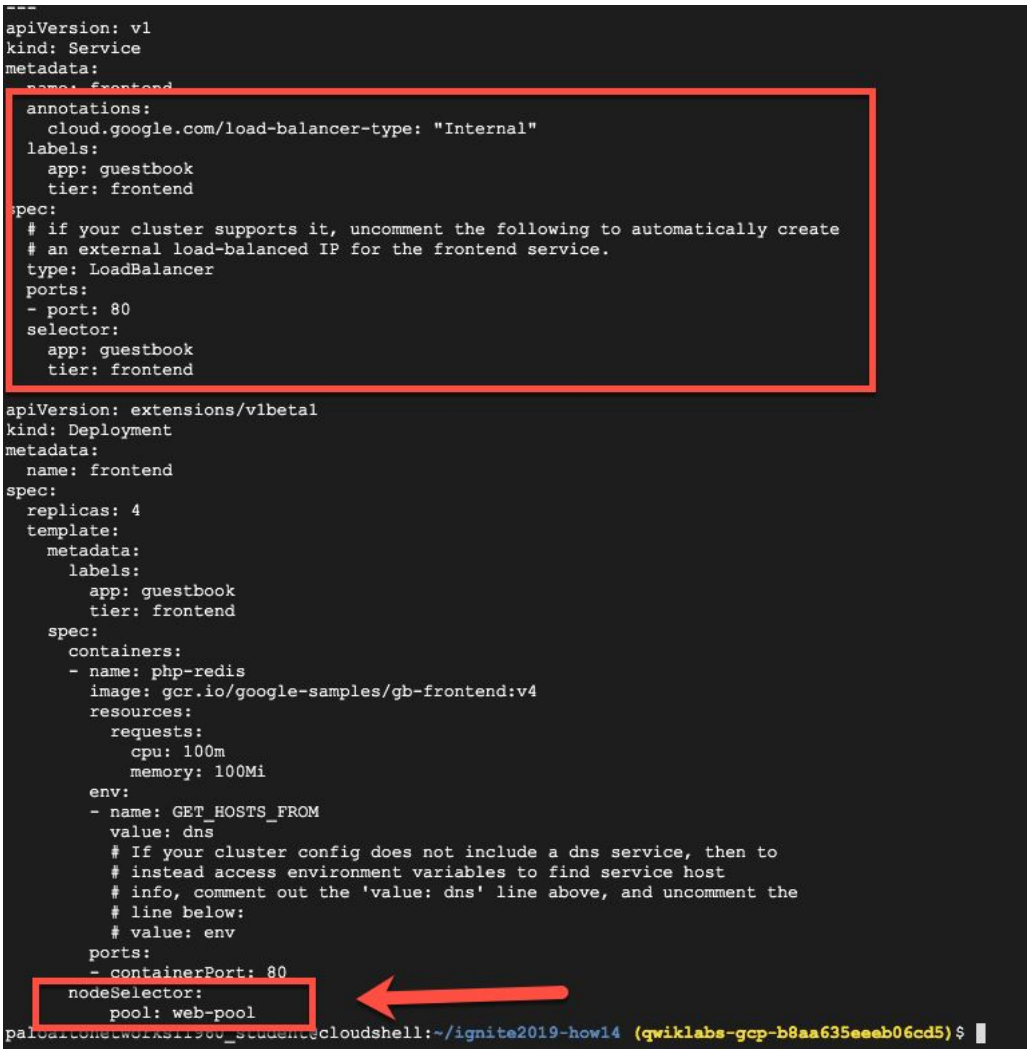
➔ The manifest can also be viewed by executing the following command in the cloud console:

more guestbook-ew.yaml

The manifest file declares various aspects of the application. For instance, it tells the orchestrator what type of resources you intend to deploy. In this case we will deploy a 2-tier simple redis application with a frontend and backend tier. The backend tier will consist of a redis-master and redis-slave for db redundancy. **Node Selectors** are used to make sure the web frontend and DB backend are placed on the desired nodes:



Looking at the frontend section of the Manifest file shows the Internal load balancer definition that is being deployed and along with the node selector for the frontend web services:

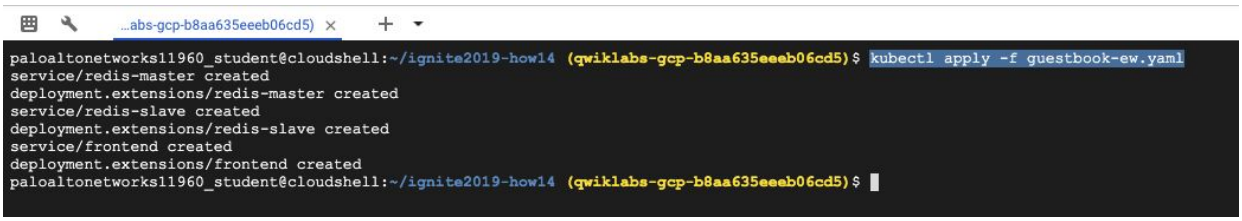


Even though we have two tiers in our application, only one (the frontend service) is exposed to the outside world via a load balancer. The annotation listed above tells GCP and Kubernetes that the load balancer would be of type: Internal.

Task 2 – Launch the Application

➔ Back in your cloud shell type the following command to create the application:

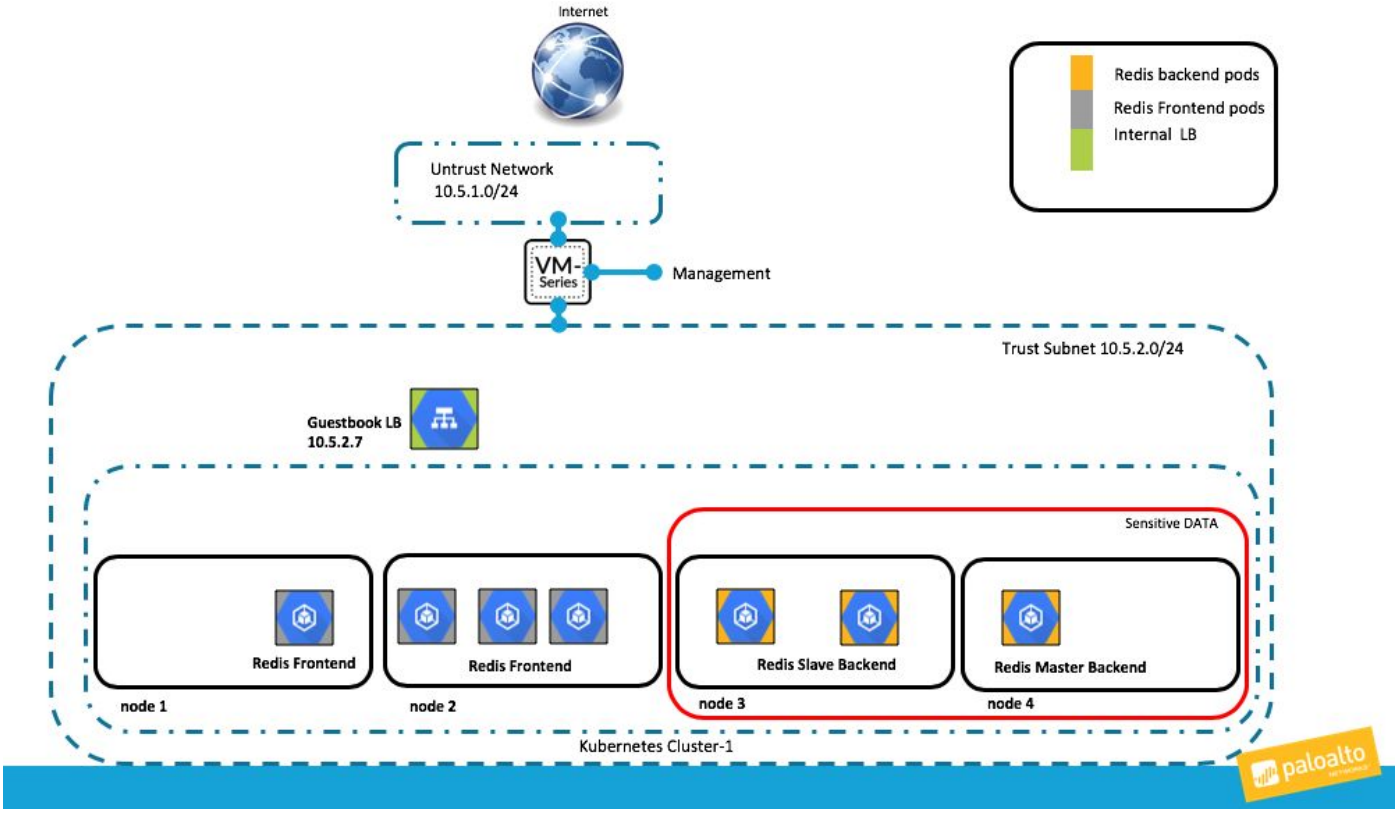
kubectl apply -f guestbook-ew.yaml



You should see the services and deployments being created.

Task 3 – Explore what was just deployed

This is now the environment with the newly deployed pods added:



➔ Let's validate this by list the new pods in your cluster. In your cloud shell type:

kubectl get pods -o wide

```
paloaltonetworks11985_student cloudshell:~/ignite2019-how14 (qwiklabs-gcp-e41669c7b0cce627)$ kubectl get pods -o wide
NAME                           READY   STATUS              RESTARTS   AGE   IP          NODE                                                NOMINATED NODE
frontend-549dd7fd98-dvnmb      0/1     ContainerCreating   0          20s   <none>      gke-k8s-cluster-1-web-tier-be1e76da-82qh            <none>
frontend-549dd7fd98-k5g98      0/1     ContainerCreating   0          20s   <none>      gke-k8s-cluster-1-web-tier-be1e76da-82qh            <none>
frontend-549dd7fd98-p9zjd      0/1     ContainerCreating   0          20s   <none>      gke-k8s-cluster-1-web-tier-be1e76da-cvk3            <none>
frontend-549dd7fd98-qr9bm      0/1     ContainerCreating   0          20s   <none>      gke-k8s-cluster-1-web-tier-be1e76da-cvk3            <none>
redis-master-85d458569f-f8bt2  0/1     ContainerCreating   0          21s   <none>      gke-k8s-cluster-1-default-pool-4e9e74e8-8dp7        <none>
redis-slave-7dcc7fb5dd-4s84g   1/1     Running             0          21s   10.16.0.8   gke-k8s-cluster-1-default-pool-4e9e74e8-8dp7        <none>
redis-slave-7dcc7fb5dd-qm46h   1/1     Running             0          21s   10.16.1.11  gke-k8s-cluster-1-default-pool-4e9e74e8-93hs        <none>
paloaltonetworks11985_student cloudshell:~/ignite2019-how14 (qwiklabs-gcp-e41669c7b0cce627)$
```

You can see that the Ready and Status of pods as they start up. Verify that all the pods get to created and running. Also notice that the fronted and redis pods are on different node pools.

```
paloaltonetworks11960_student@cloudshell:~/ignite2019-how14 (qwiklabs-gcp-b8aa635eeeb06cd5)$ kubectl get pods -o wide
NAME                           READY   STATUS    RESTARTS   AGE   IP          NODE                                                NOMINATED NODE
frontend-549dd7fd98-4jwnw      1/1     Running   0          8m    10.16.2.4   gke-k8s-cluster-1-web-tier-1a23b339-1ts3            <none>
frontend-549dd7fd98-9fcvh      1/1     Running   0          8m    10.16.2.3   gke-k8s-cluster-1-web-tier-1a23b339-1ts3            <none>
frontend-549dd7fd98-w4sdc      1/1     Running   0          8m    10.16.3.3   gke-k8s-cluster-1-web-tier-1a23b339-knc1            <none>
frontend-549dd7fd98-w9w7k      1/1     Running   0          8m    10.16.3.4   gke-k8s-cluster-1-web-tier-1a23b339-knc1            <none>
redis-master-85d458569f-lqwn7  1/1     Running   0          8m    10.16.1.6   gke-k8s-cluster-1-default-pool-e2467308-sxml        <none>
redis-slave-7dcc7fb5dd-9dqdk   1/1     Running   0          8m    10.16.1.7   gke-k8s-cluster-1-default-pool-e2467308-sxml        <none>
redis-slave-7dcc7fb5dd-z6g98   1/1     Running   0          8m    10.16.0.12  gke-k8s-cluster-1-default-pool-e2467308-81nd        <none>
paloaltonetworks11960_student@cloudshell:~/ignite2019-how14 (qwiklabs-gcp-b8aa635eeeb06cd5)$
```

➔ Next, let's look at the load balancing service for the front-end pod. Execute the following command in the shell:

`kubectl get svc`

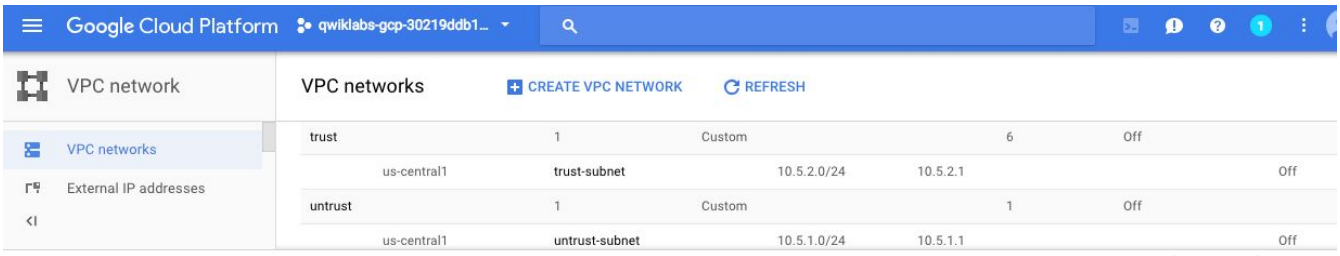
If you notice the "external-ip" of the frontend service's load balancer you will see that initially this may show as pending:

```
paloaltonetworks11985_student@cloudshell:~/ignite2019-how14 (qwiklabs-gcp-e41669c7b0cce627)$ kubectl get svc
NAME           TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
frontend       LoadBalancer   10.19.248.199   <pending>     80:30261/TCP     18s
kubernetes     ClusterIP      10.19.240.1     <none>        443/TCP          12m
redis-master   ClusterIP      10.19.240.183   <none>        6379/TCP         18s
redis-slave    ClusterIP      10.19.249.193   <none>        6379/TCP         18s
```

Repeating the “kubectl get svc” command will eventually show a private ip address for the internal load balancer.

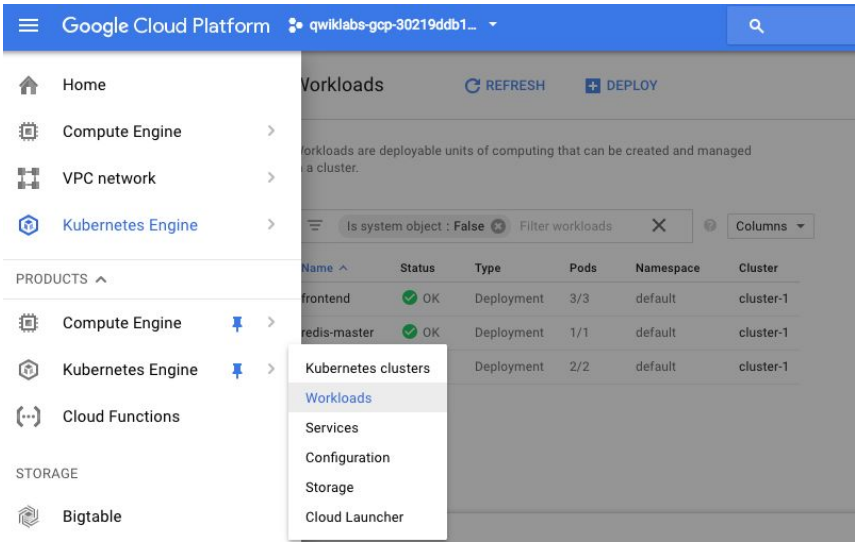
The private IP address is from the trust subnet’s CIDR block.

➔ navigate to **VPC network > VPC** networks to see the GCP config

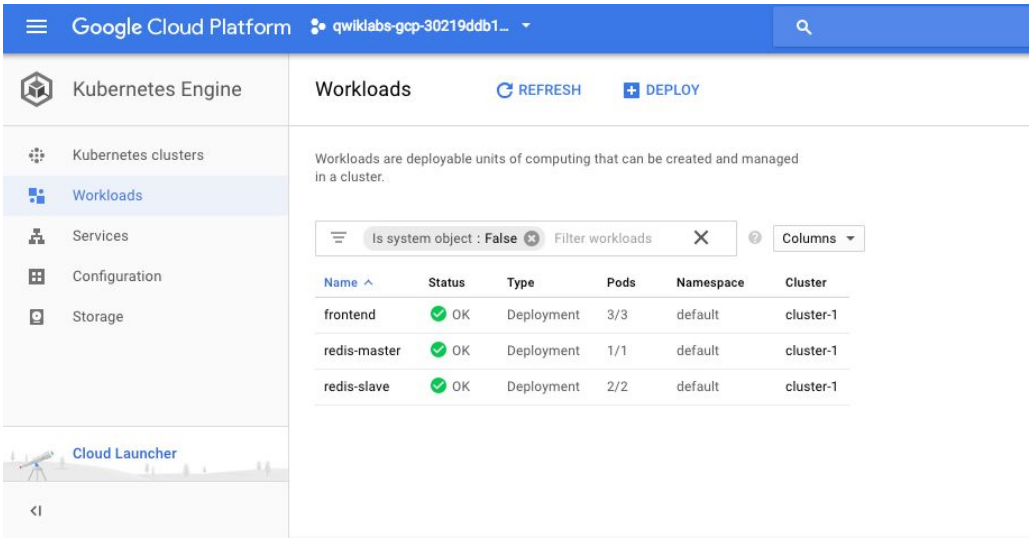


To see the new Kubernetes resources in the GCP console

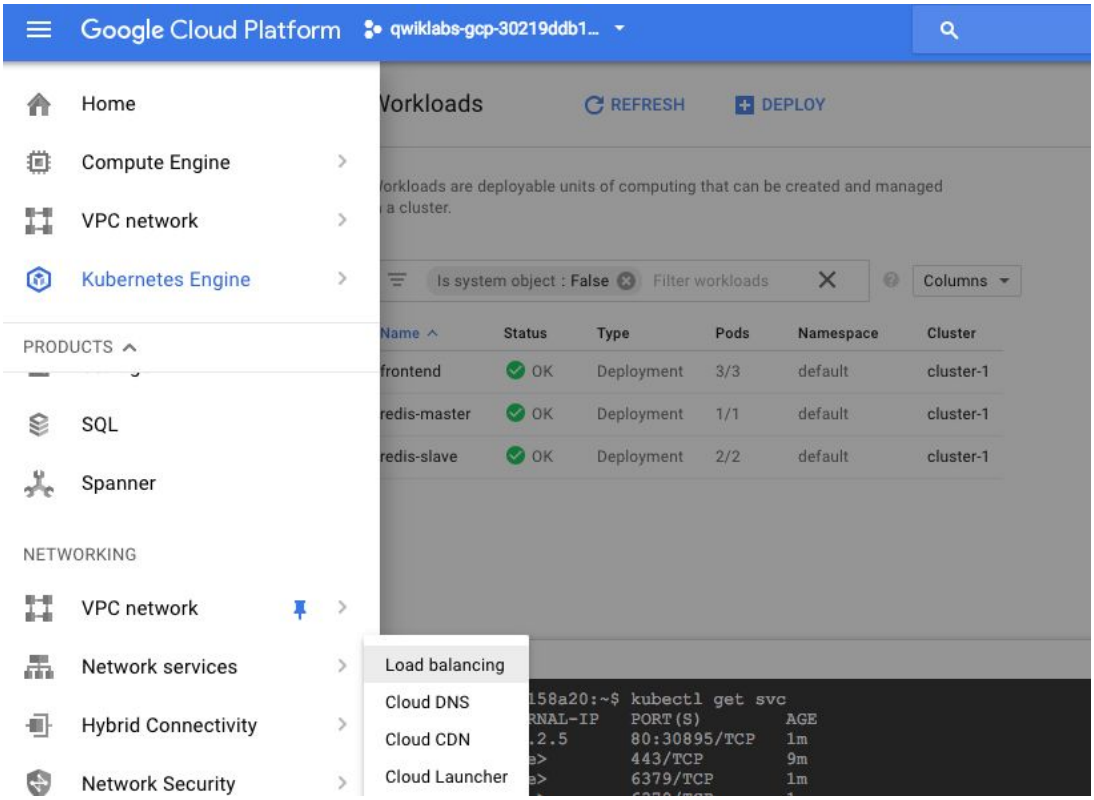
➔ navigate to **Kubernetes Engine > Workloads**:



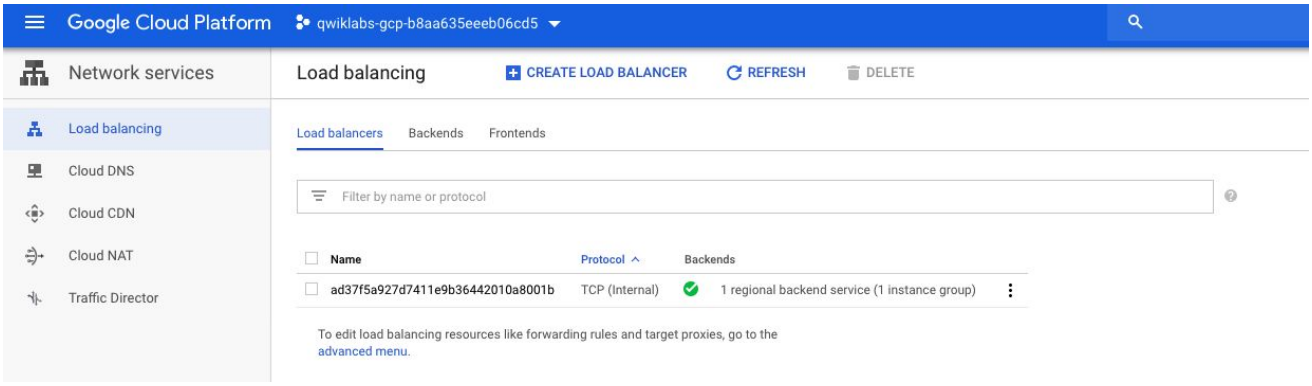
Here you can see the workloads that were just deployed. You can click on any of these to get more information:



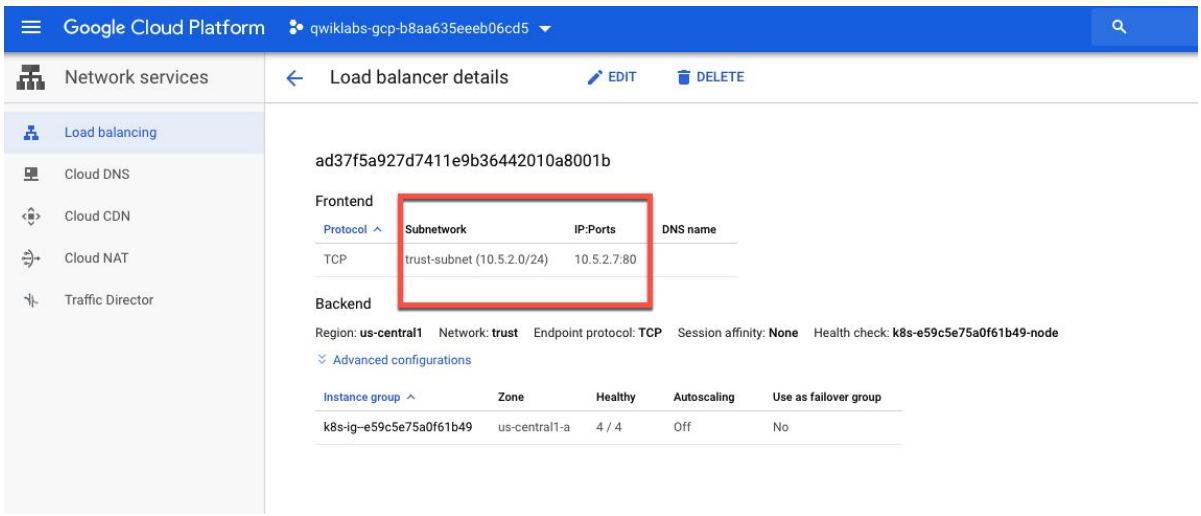
➔ The Internal Load Balancer can be seen by navigating to **Network Services > Load Balancing**



➔ Click on the load balancer to get more details:



This also shows that the internal load balancer is on the trust network and has an internal IP address that will be used later in the firewall configuration:



End of Activity 7

Activity 8 – Securing Inbound Traffic

In this activity, you will:

- ***Secure traffic that is inbound to your frontend service***
- ***Validate that traffic is visible in the Firewall logs***

Task 1 – Note the Internal Load Balancer's IP Address

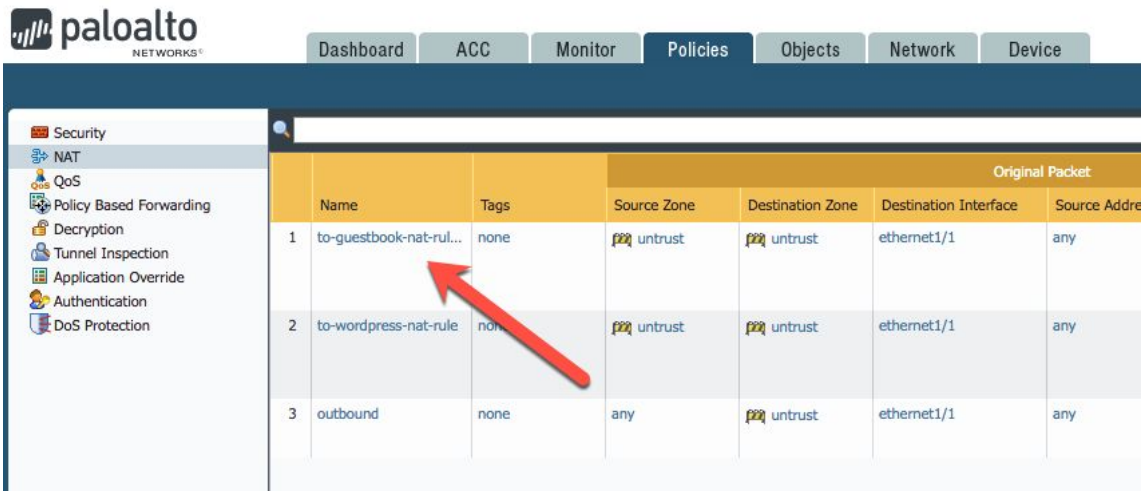
From the previous activity: make a note of the internal load balancer's ip address. (very probable it is 10.5.2.7)

Task 2 – Update the Firewall's NAT Policy

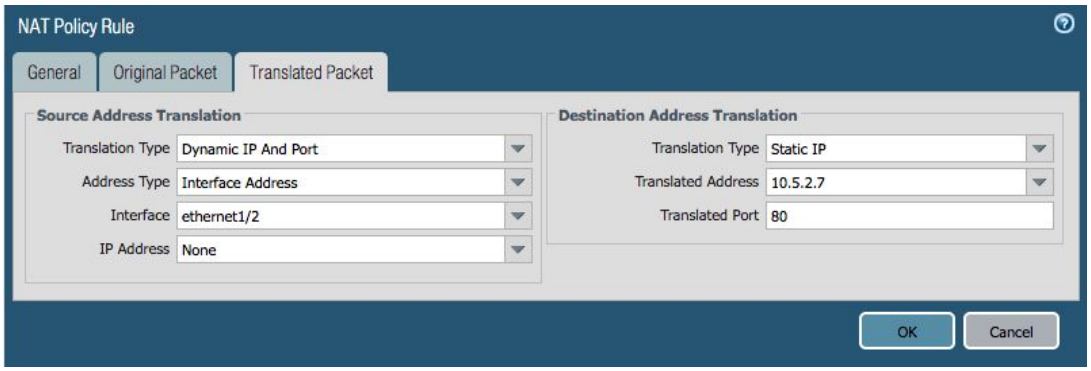
➔ Login in to the VM-Series firewall using the management interface's ip address. *https://<firewall mgmt ip-addr>*

Hint: Check Activity 4, Task 2 if you don't have ip address jotted down

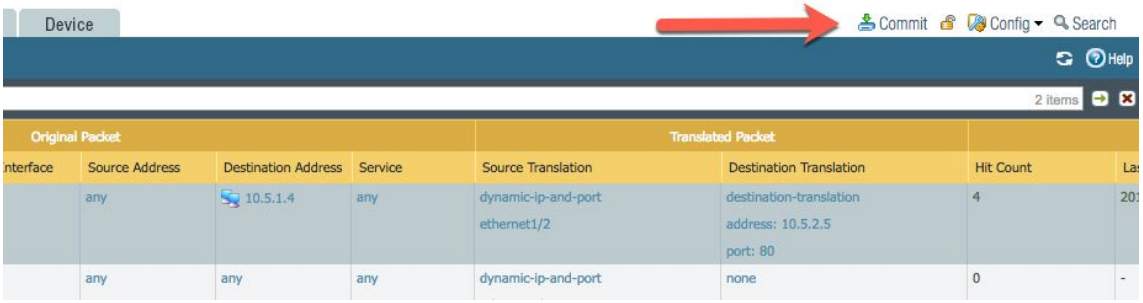
➔ Click the **Policies** Tab and navigate to **NAT** on the left. Click on the **to-guestbook-nat-rule** to edit it:



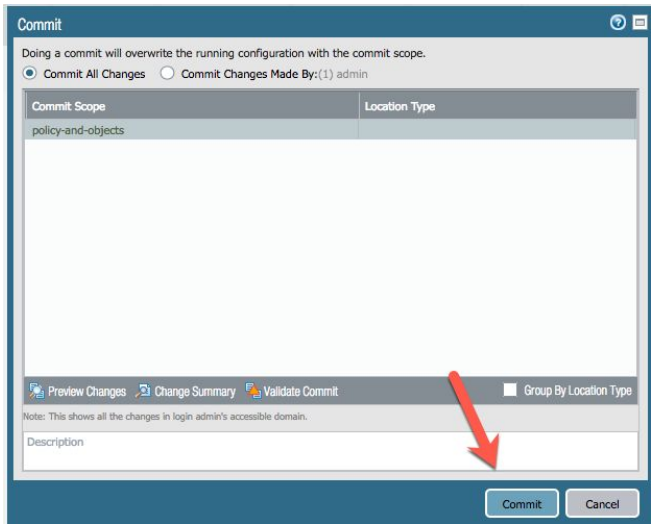
➔ Click on the **Translated Packet** tab. In the **Translated Address** field, enter the load balancer's ip address from Task 1 and click **OK**. It might not need to be changed.



➔ If a change was needed, Click the **Commit** link on the top right



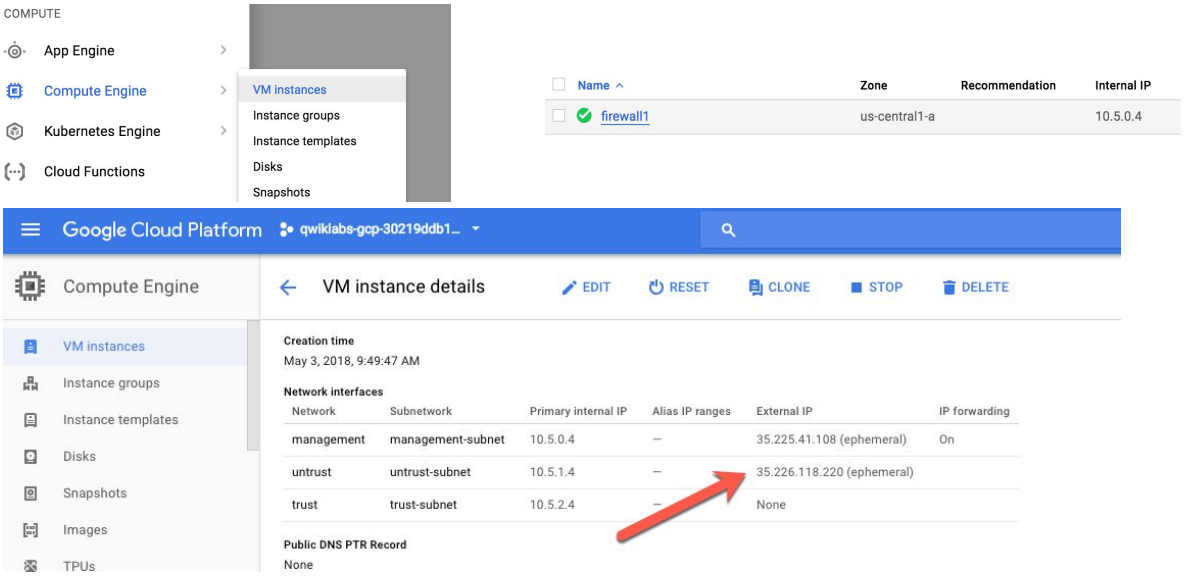
➔ When the next dialog box opens, Click the **Commit** button



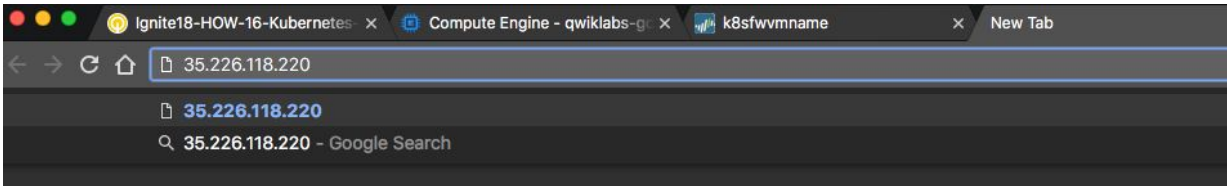
Task 3 – Connect to the Guestbook Frontend

The VM-Series is now protecting your Kubernetes workload. In order to connect to the guestbook's frontend service, you will connect to the untrust ip-address of the firewall.

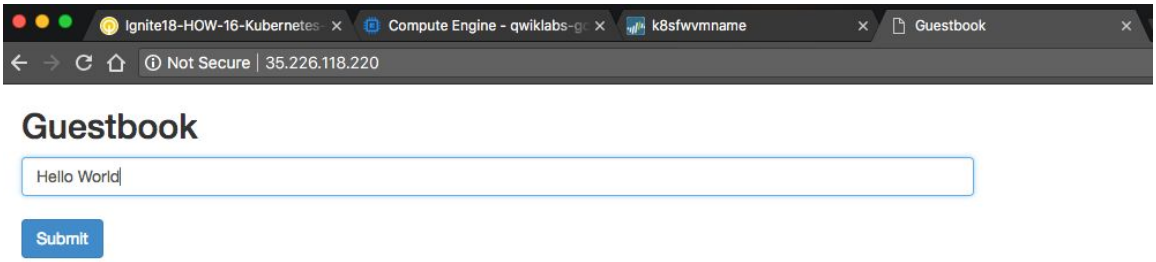
➔ Navigate to the **Compute Engine> VM Instances** screen. Then click on the firewall and **copy** the **External IP address** of the **untrust** interface:



➔ In your browser navigate to the untrust-ip address of the firewall: http://<firewall untrust External IP>

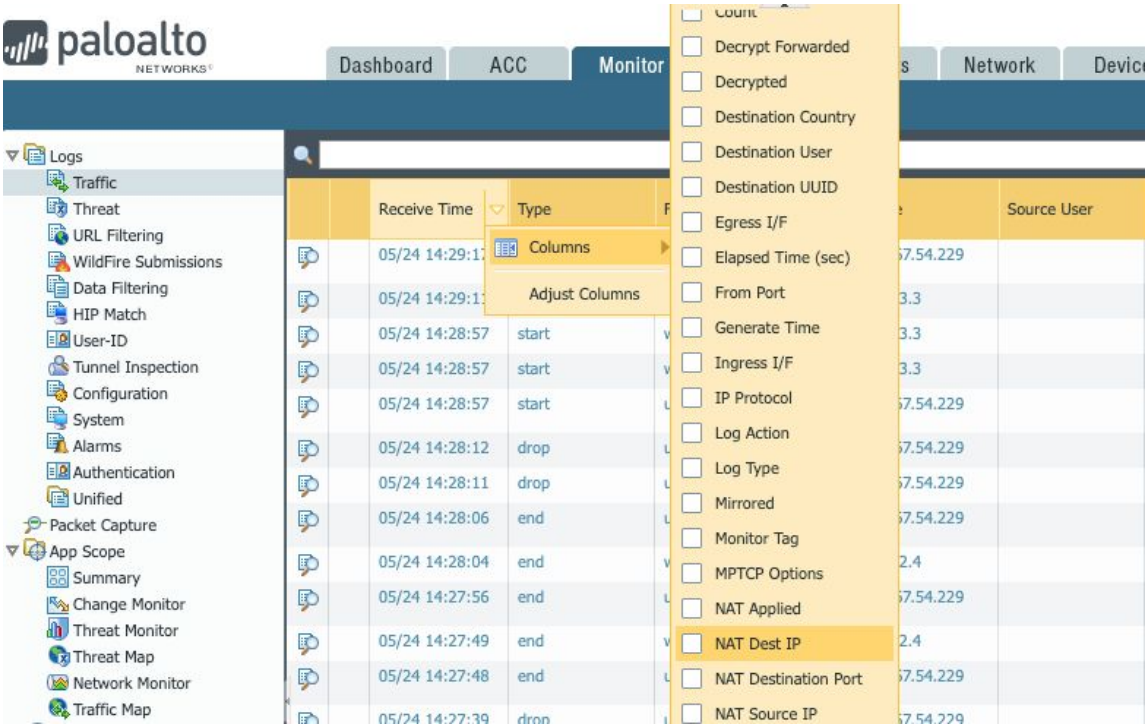


And you should be greeted with the frontend of your guestbook application:

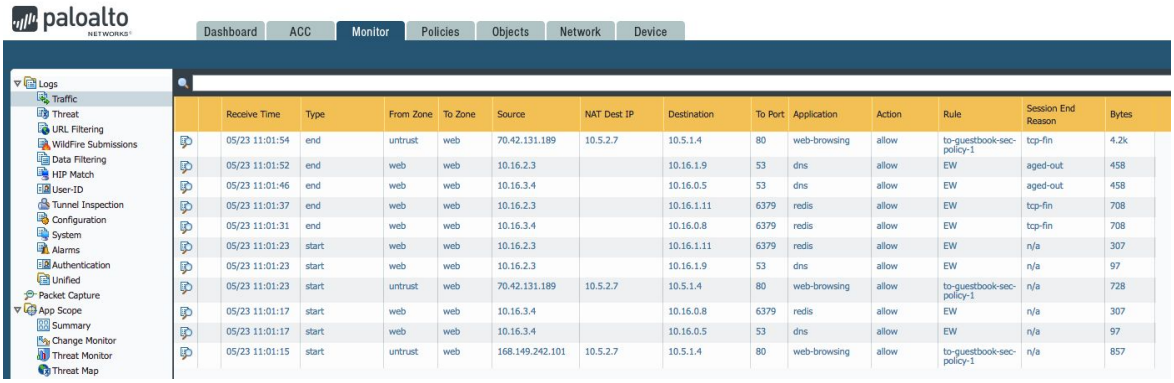


➔ Type a message in the dialog box and click **Submit**. The message should be echoed on the website.

➔ Open the firewall interface and navigate to the **Monitor** tab. Add the **NAT Dest IP** column.



➔ With the **NAT Dest IP** Column visible, it is possible to see the address translation to the internal load balancer IP address. Also notice that there is some E/W traffic between the Frontend and the DB Backend showing up in the logs. More on this in Activity 8.



Receive Time	Type	From Zone	To Zone	Source	NAT Dest IP	Destination	To Port	Application	Action	Rule	Session End Reason	Bytes
05/23 11:01:54	end	untrust	web	70.42.131.189	10.5.2.7	10.5.1.4	80	web-browsing	allow	to-guestbook-sec-policy-1	tcp-fin	4.2k
05/23 11:01:52	end	web	web	10.16.2.3		10.16.1.9	53	dns	allow	EW	aged-out	458
05/23 11:01:46	end	web	web	10.16.3.4		10.16.0.5	53	dns	allow	EW	aged-out	458
05/23 11:01:37	end	web	web	10.16.2.3		10.16.1.11	6379	redis	allow	EW	tcp-fin	708
05/23 11:01:31	end	web	web	10.16.3.4		10.16.0.8	6379	redis	allow	EW	tcp-fin	708
05/23 11:01:23	start	web	web	10.16.2.3		10.16.1.11	6379	redis	allow	EW	n/a	307
05/23 11:01:23	start	web	web	10.16.2.3		10.16.1.9	53	dns	allow	EW	n/a	97
05/23 11:01:23	start	untrust	web	70.42.131.189	10.5.2.7	10.5.1.4	80	web-browsing	allow	to-guestbook-sec-policy-1	n/a	728
05/23 11:01:17	start	web	web	10.16.3.4		10.16.0.8	6379	redis	allow	EW	n/a	307
05/23 11:01:17	start	web	web	10.16.3.4		10.16.0.5	53	dns	allow	EW	n/a	97
05/23 11:01:15	start	untrust	web	168.149.242.101	10.5.2.7	10.5.1.4	80	web-browsing	allow	to-guestbook-sec-policy-1	n/a	857

End of Activity 8

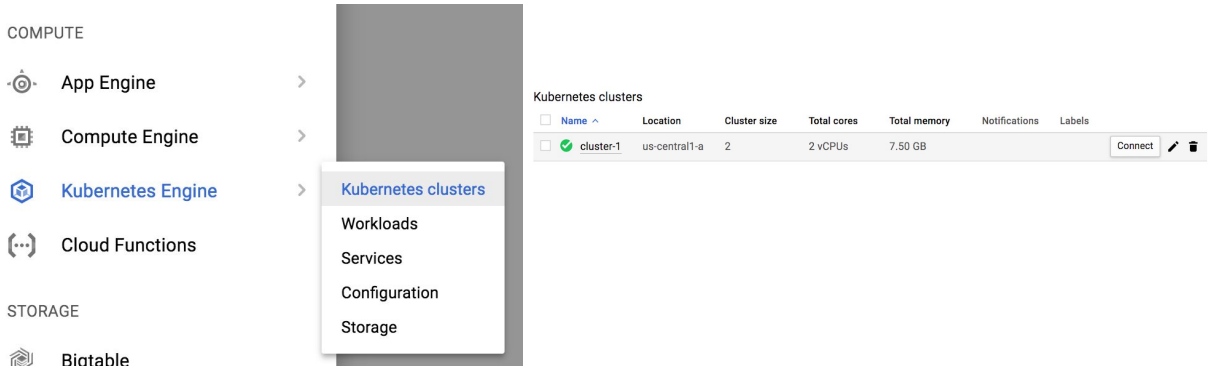
Activity 9 – Securing Outbound Traffic

In this activity, you will:

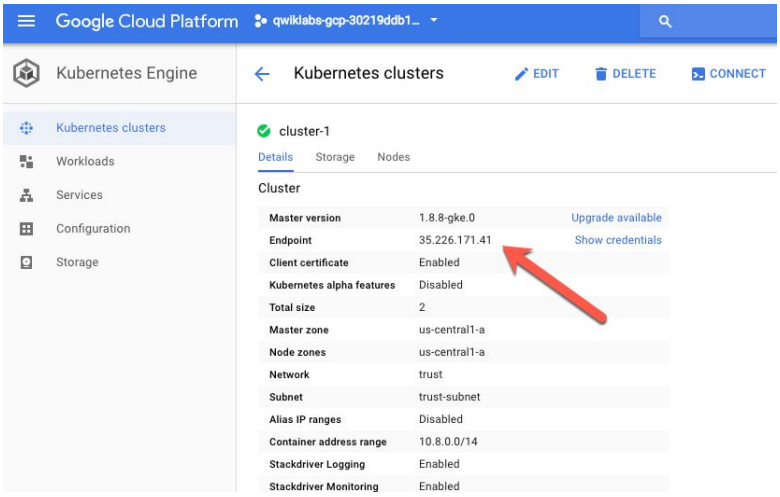
- ***Secure outbound traffic from the cluster***
- ***Validate traffic is in the Firewall logs***

Task 1 – Add Kube-API server route

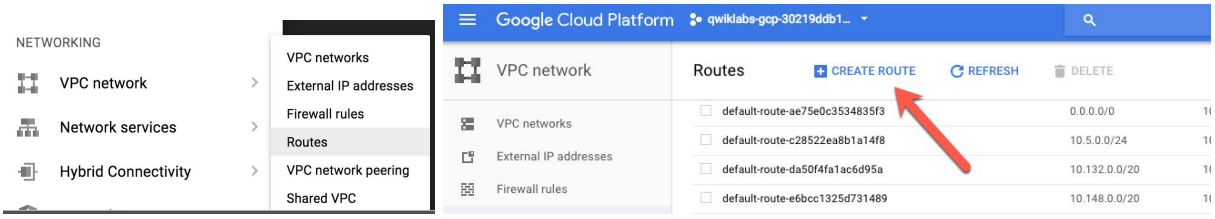
➔ Navigate to **Kubernetes Engine > Kubernetes clusters** and click on the cluster.



➔ When the cluster **Details** page opens, identify the kube-apiserver ip address. Labeled as **Endpoint** in the page. Copy this address:

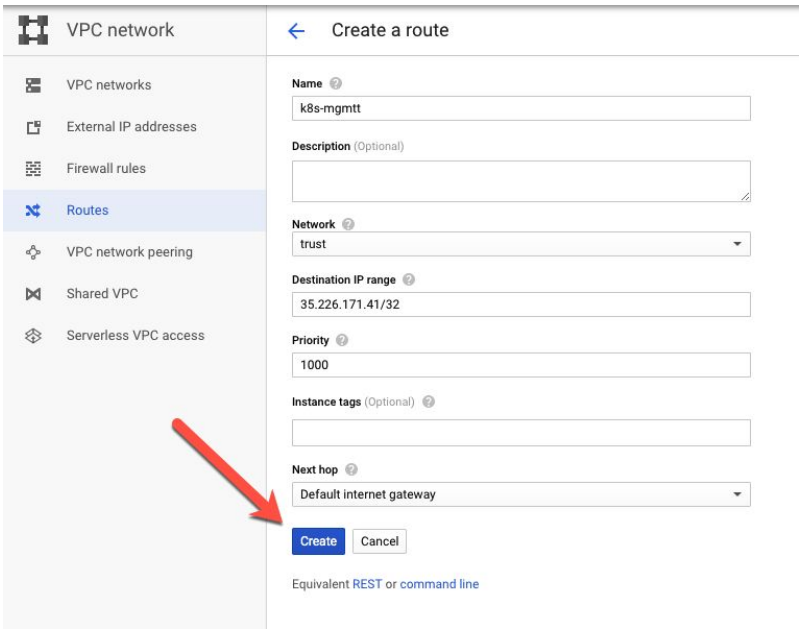


➔ Navigate back to **VPC Networks > Routes** and create another route that will allow the kube-apiserver to perform health checks without the firewall getting in the way:



Create a route with the following parameters:
Name: k8s-mgmt
Network: trust
Destination IP Range: [Insert kube-apiserver IP] /32
Priority: 10
Next hop: Default internet gateway

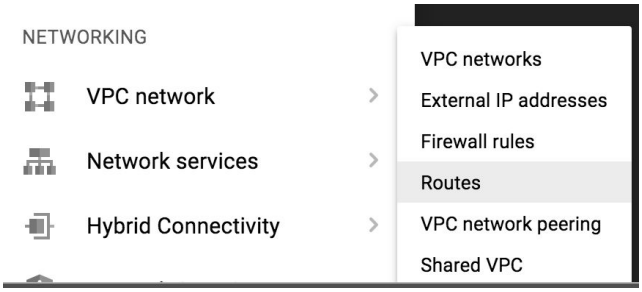
➔ And then click **Create**.



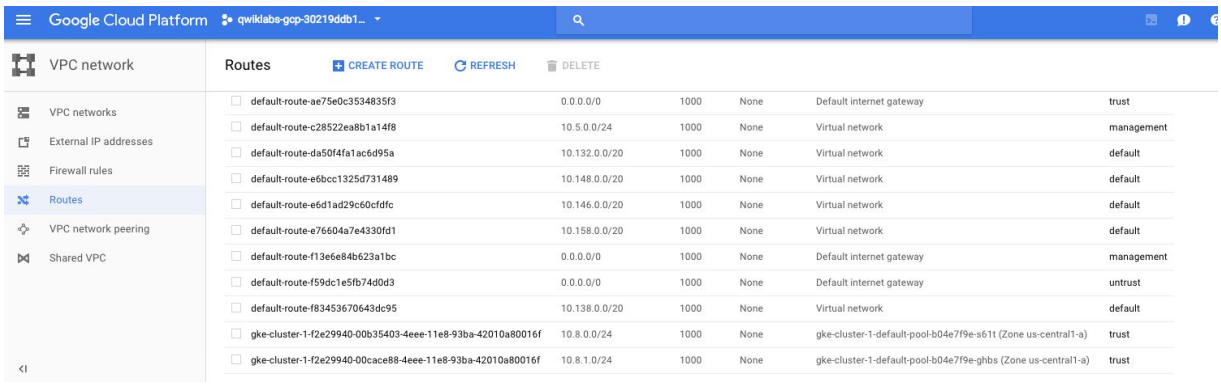
Task 2 – Add Outbound Route

To secure any traffic that is originating from within the cluster we need to add a couple of routes in the GCP console.

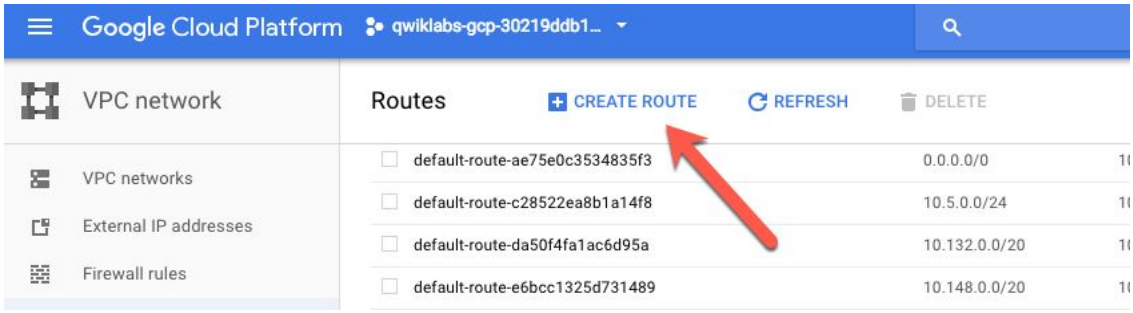
➔ Go to **VPC Network > Routes**



You will see a few default routes added by Kubernetes during the initial template launch:

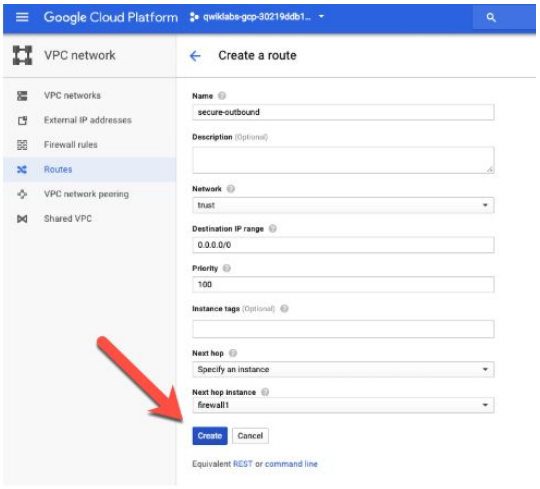


➔ Click on **Create Route**:



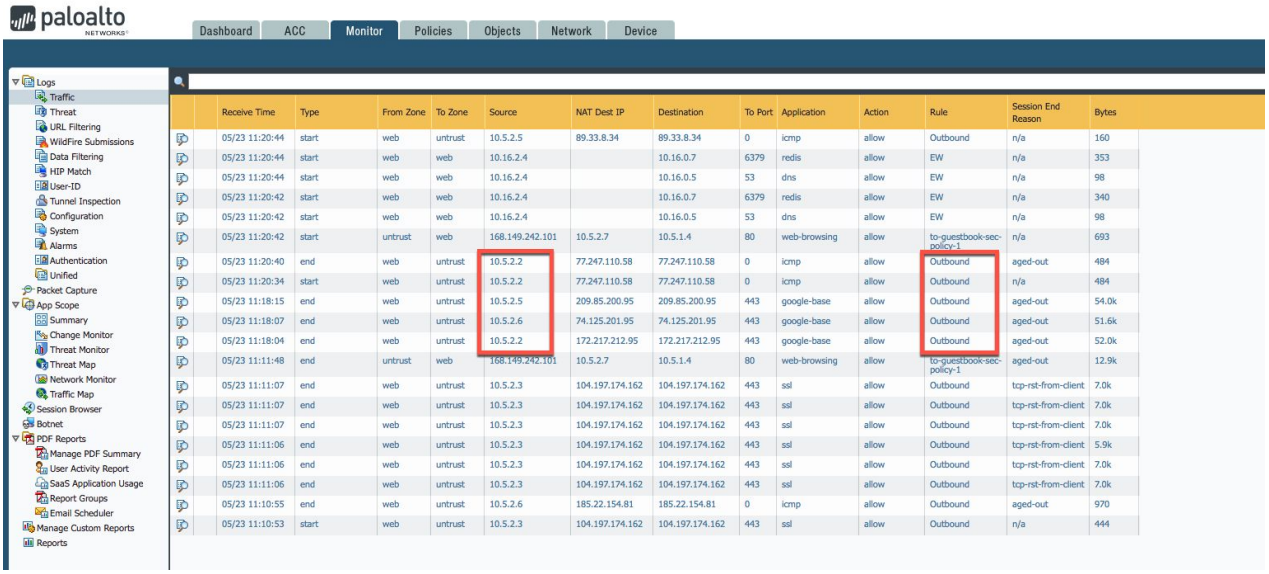
Create a route with the following Parameters:
Name: secure-outbound
Network: trust
Destination IP Range: 0.0.0.0/0
Priority: 10
Next hop: Specify an instance
Next Hop Instance: firewall1

➔ And then click **Create**

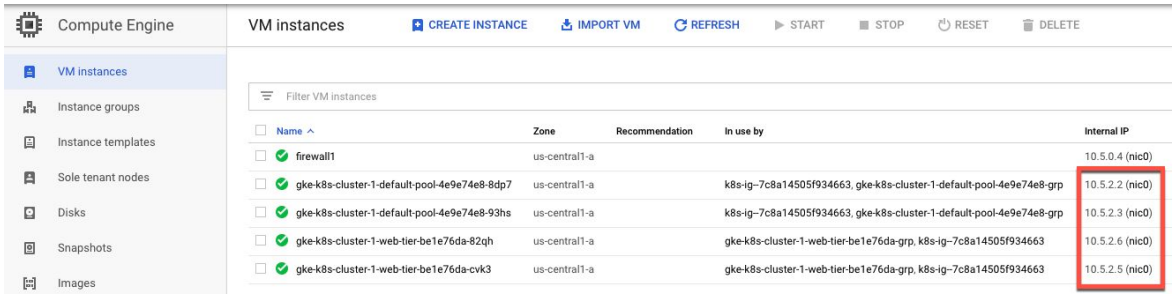


➔ Navigate back to the firewall monitor tab and you should now see outbound traffic as well from the cluster node IPs.

Note: You might need to refresh the Monitor page by clicking on the refresh button in the top right corner.



The source addresses for the outbound traffic are the instance addresses of the Kubernetes cluster nodes.

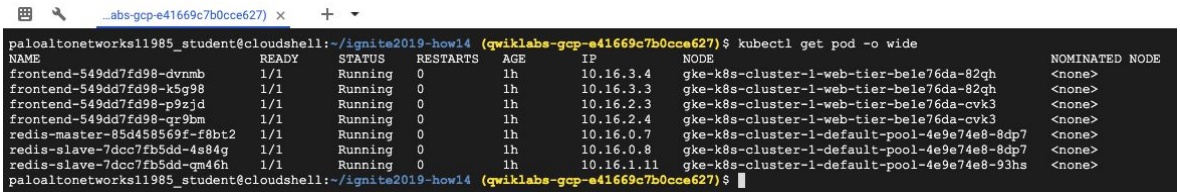


Task 3 – Test Outbound Pod Traffic

After setting up the outbound routes and looking at the logs in the VM-Series firewall there where most likely traffic showing up that was being originated from the K8s nodes. In this section, you will connect to a Pod and generate outbound traffic to validate visibility.

➔ Navigate to the GCP Shell and execute the following command to show the current pods:

kubectl get pod -o wide



➔ Copy the first pod name and type in the following command:

kubectl exec -it <pod name> /bin/bash

In my example the final command would look like this:

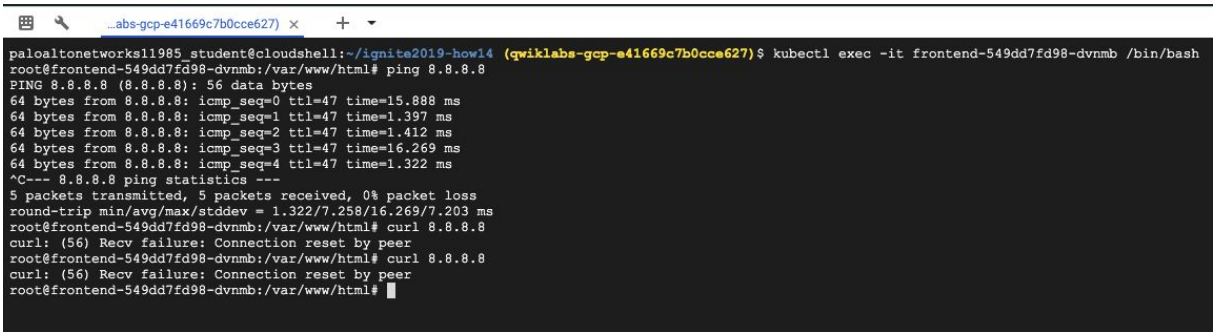
kubectl exec -it frontend-549dd7fd98-dvnmb /bin/bash

➔ Next let's attempt to generate outbound internet traffic by pinging a public site:

ping 8.8.8.8

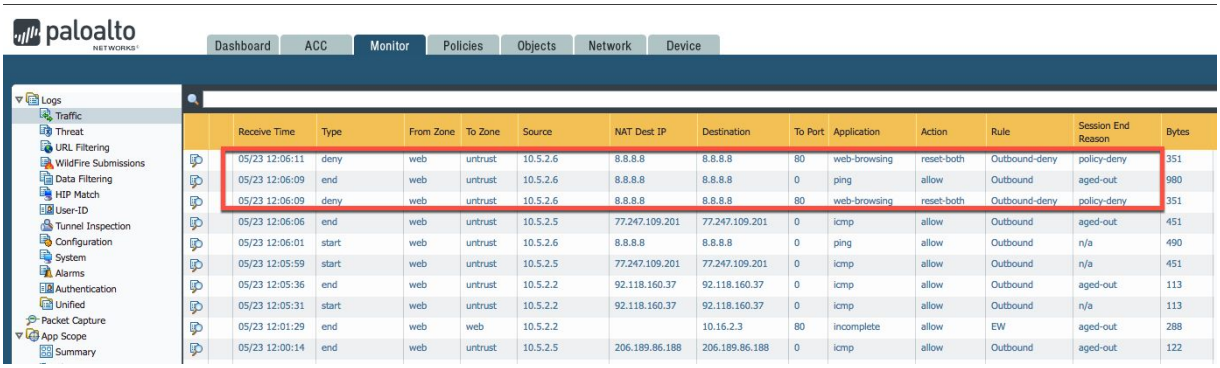
➔ Next attempt some web browsing using curl:

curl 8.8.8.8



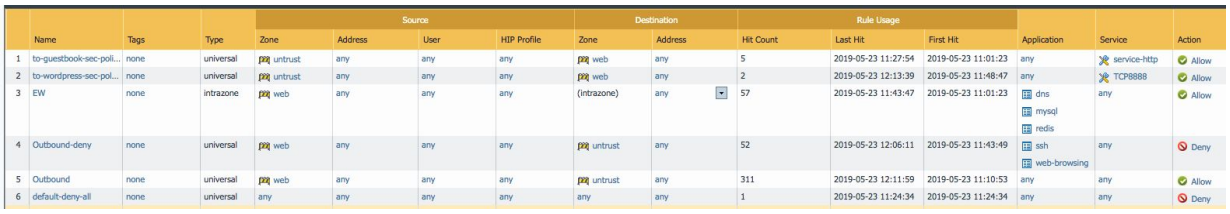
You should see that the ping traffic was successful but the web browsing failed.

➔ Navigate to the firewall and click on the Monitor tab



Notice that the web-browsing traffic is being denied but the pings are allowed via a different rule.

➔ Navigate to the Policies tab and look at the **Outbound-deny** rule **Action** column.



The **Outbound-Deny** rule is blocking select traffic and providing an extra level of visibility. Now it is clear why the web traffic failed.

End of Activity 9

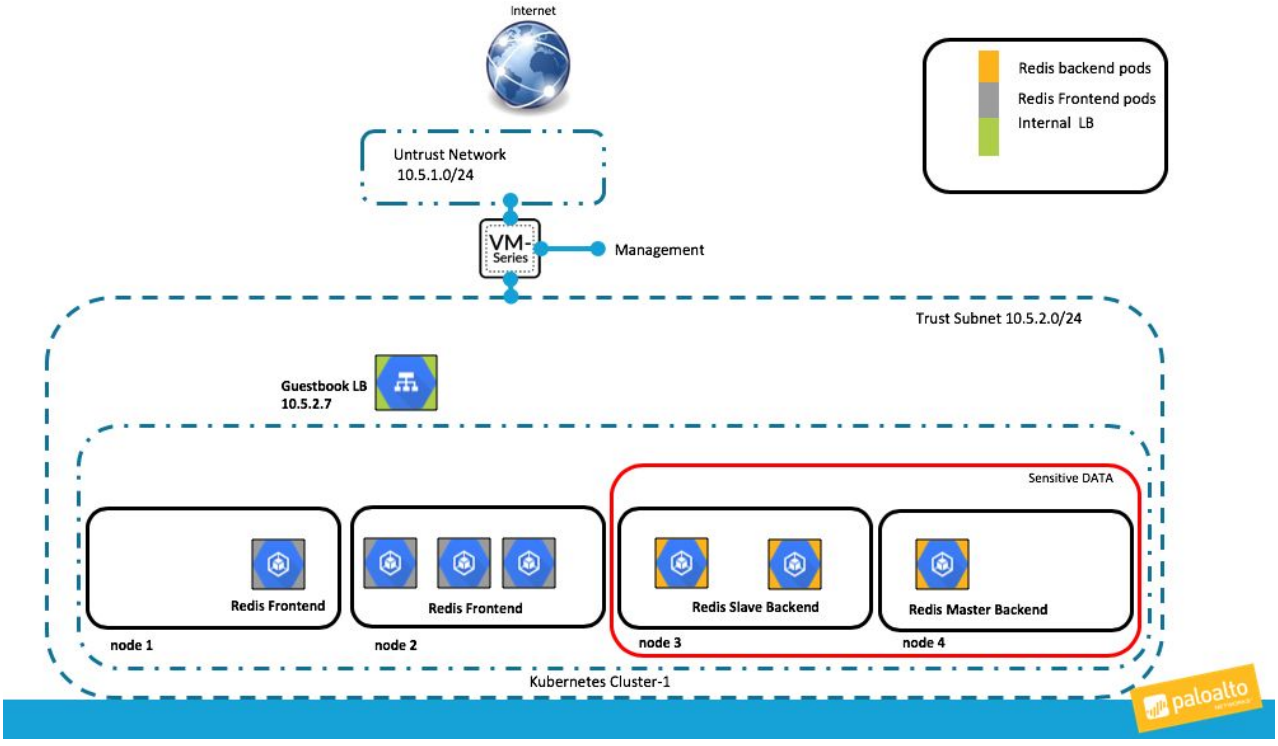
Activity 10 – Investigate Inter Node-Pool traffic

In this activity, you will:

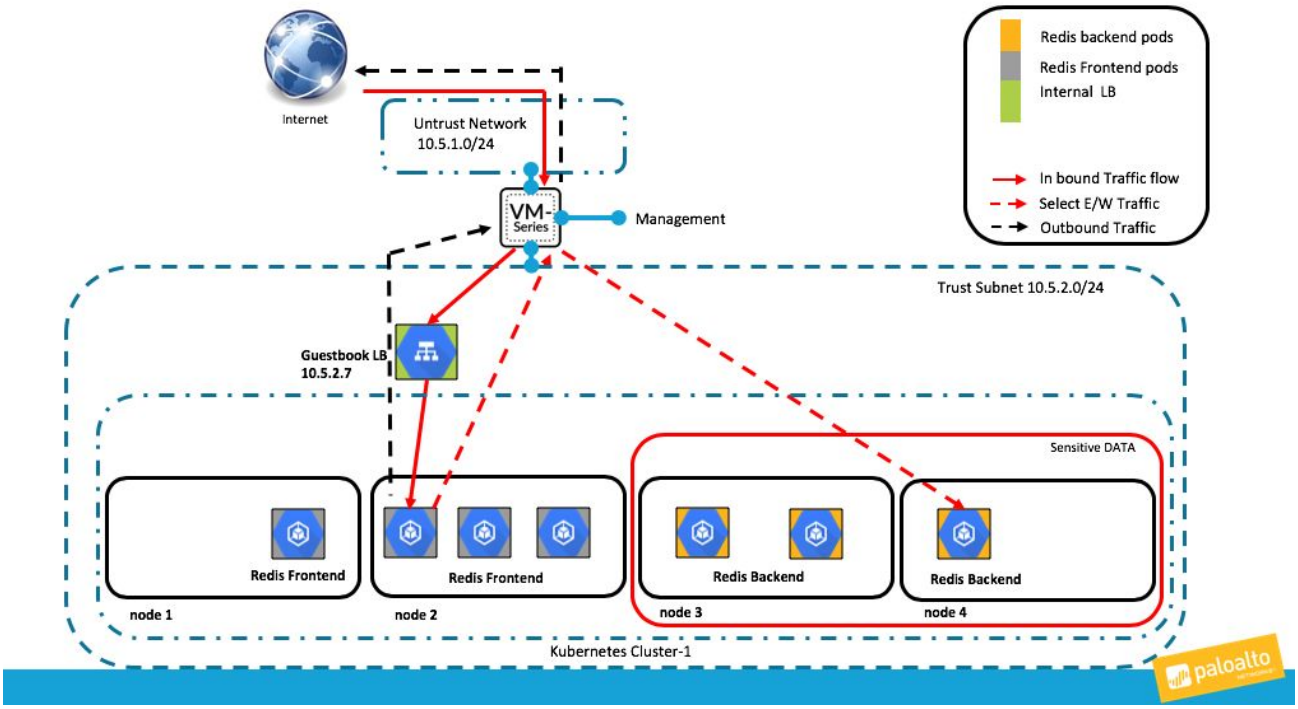
- ***Investigate how Inter Node Traffic is being routed through the firewall***
- ***Validate that traffic is visible in the Firewall logs***

One of the key challenges when deploying microservices is the lack of visibility of traffic that is flowing between services (or pods) within the cluster. When services are deployed that contain sensitive information it is important to increase the level of visibility. Being able to selective gain a higher level of security over the cloud native port/protocol approaches increases the overall security posture of applications migrated to the cloud.

In this lab, Node Pools are being used to isolate pods that have sensitive information and traffic destined for these pods are being directed to the firewall for visibility. As shown previously the current environment is setup like this:



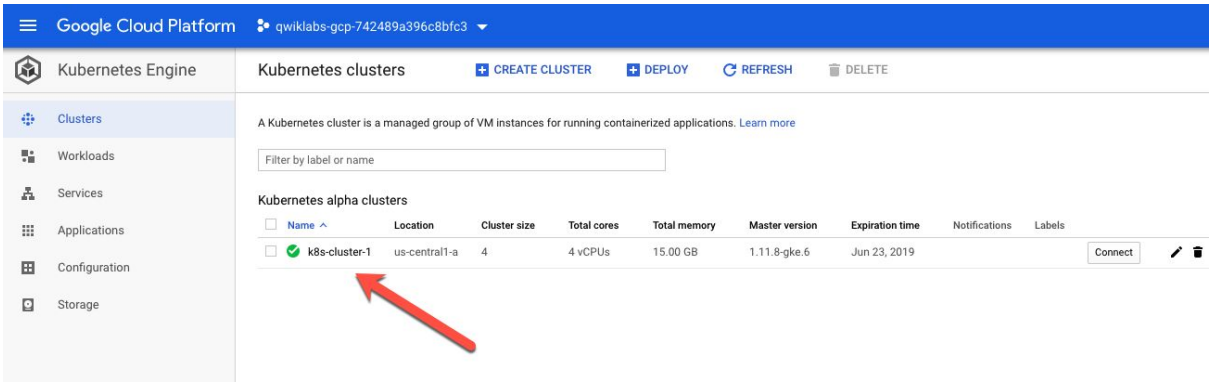
The following diagram shows how traffic is moving through the lab environment:



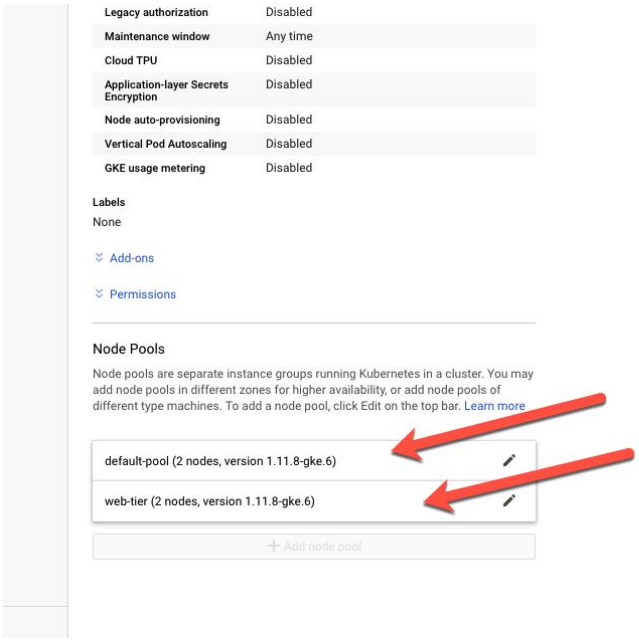
Task 1 – View Node Pool Setup

Each node has been associated with a Node-Pool during the initial setup. You can see the cluster nodes pools opening the GCP and Navigate to **Kubernetes Engine>Clusters**

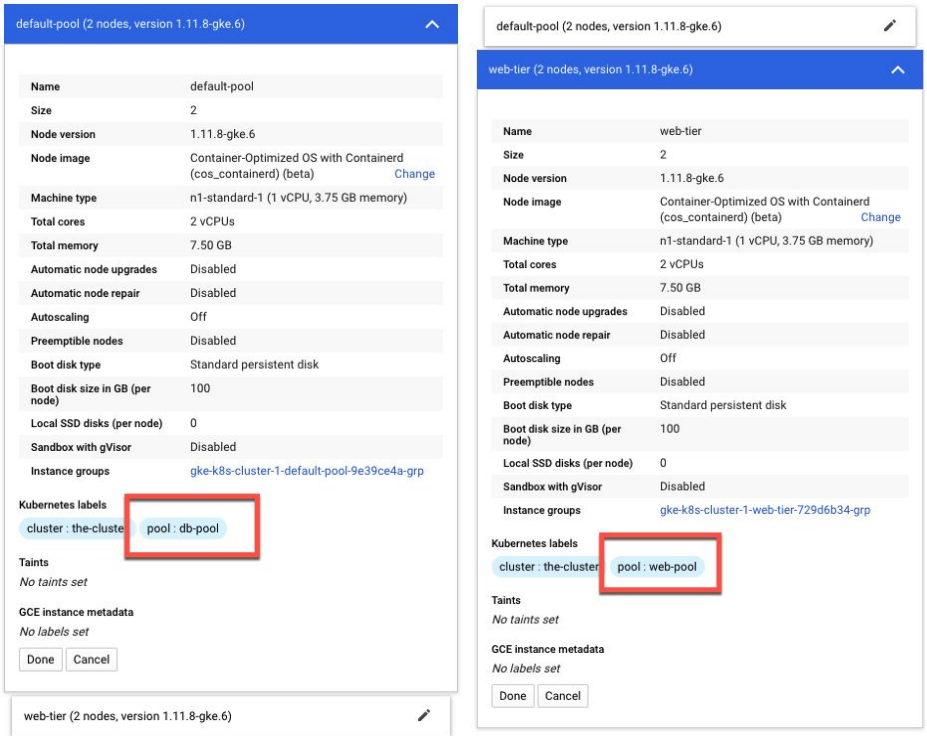
➔ Click k8s-cluster-1



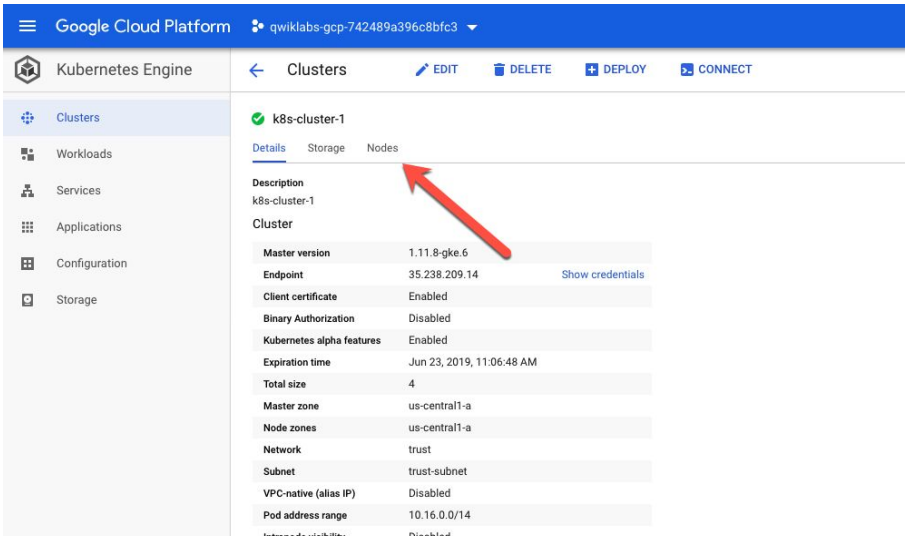
➔ Scroll down and click to expand each node pool



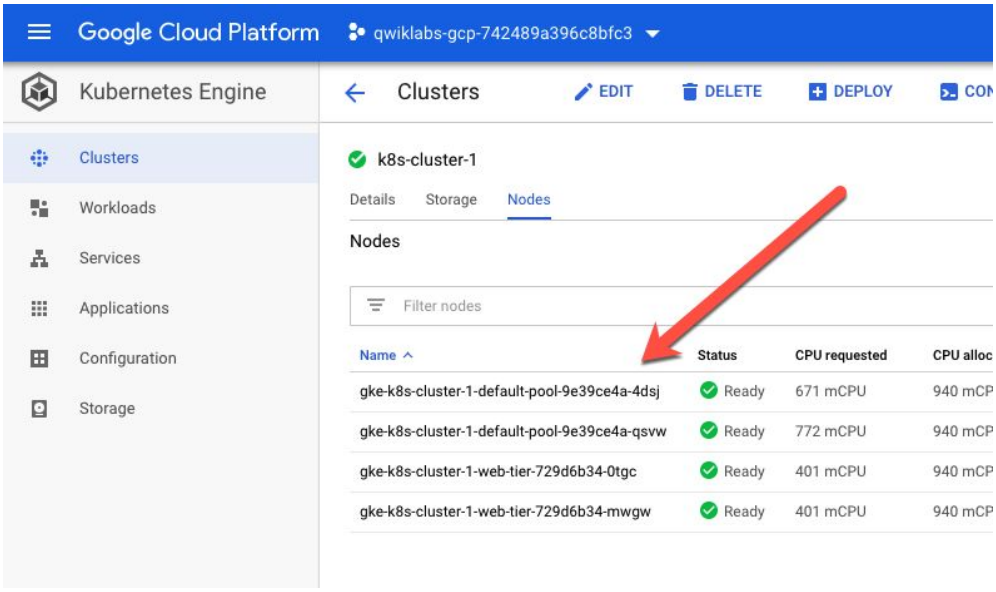
Look at the **Kubernetes Labels** section. Each node during deployment was given a label that automatically put it into the appropriate pool.



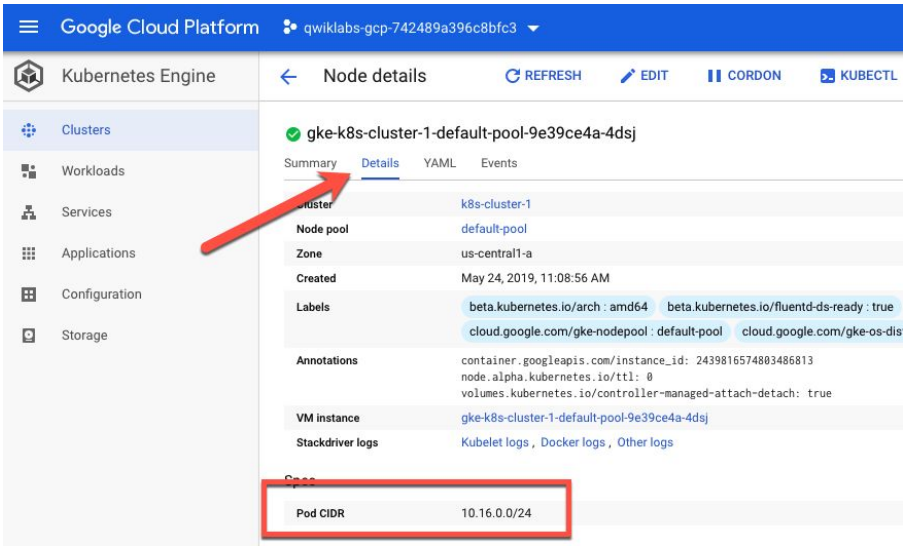
➔ Next, click on the **Nodes** tab:



➔ Click on one of the nodes:



➔ Click on the **Details Tab** and view the **Pod CIDR** Range for this node.

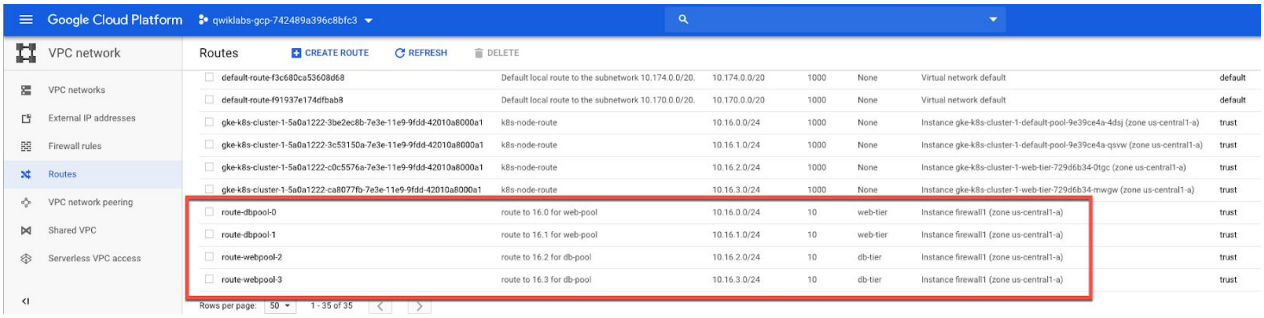


You can repeat this for the other nodes and see that the nodes in each Node has a defined range for pod deployments.

Task 2 – View GCP Routing Rules

Now lets see how this is used within GCP to redirect traffic to the firewall.

➔ Within GCP, Navigate to **VPC Network > Routes**

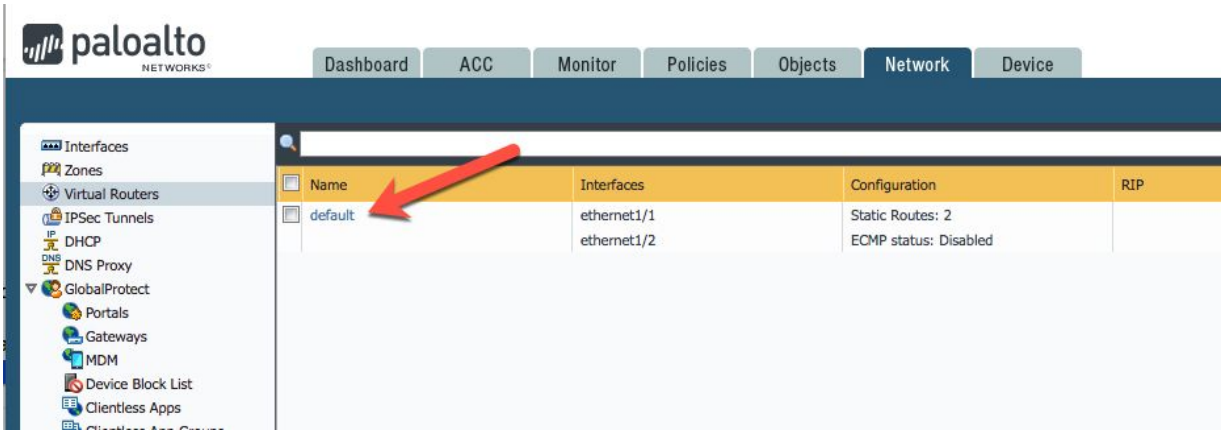


Observe that we have used route tagging during the deployment to also get routes associated with Nodes so traffic that is being sent between the Node-Pools is getting redirected to the VM-Series Firewall.

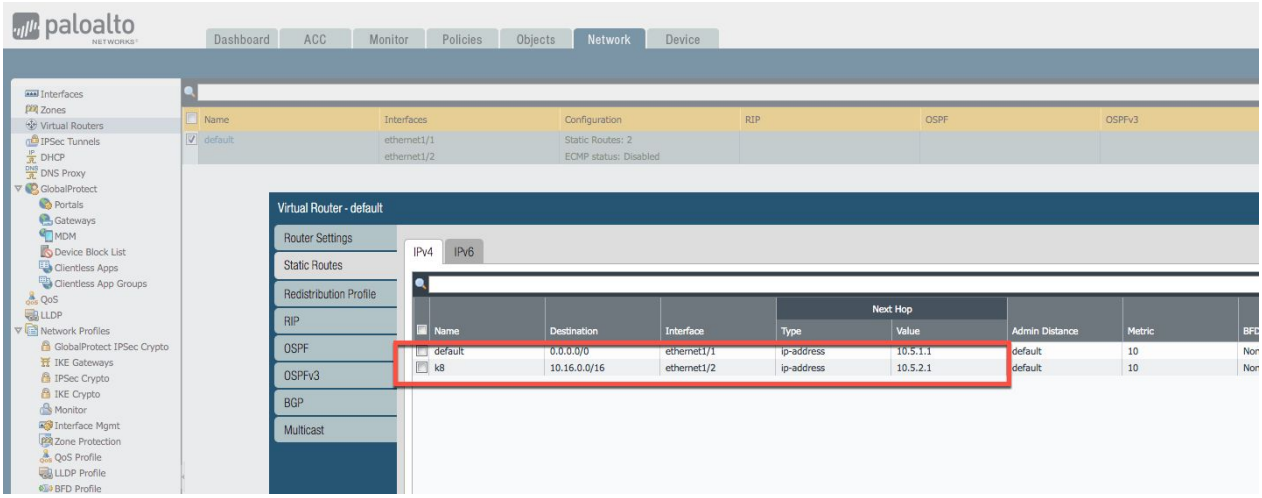
Task 3 – View the VM-Series Firewall Routing

As a result of the GCP setup, the routing is setup on the VM-Series Firewall is very straight forward. Let's take a look.

➔ On the VM-Series Firewall, Navigate to **Network > Virtual Routers** and click on **default.**



➔ Click on **Static Routes**

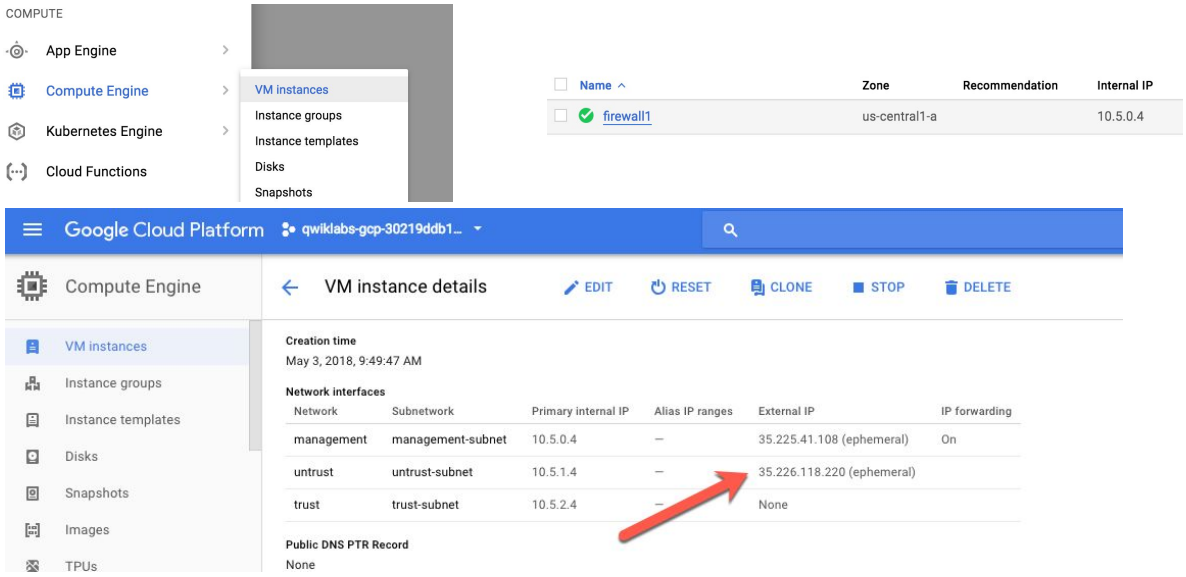


Notice that there is a **10.16.0.0/16** route that will cover all the Pod CIDRs on the Nodes. This allows the firewall to send all inter pod traffic back to the GCP fabric without the need for SNAT.

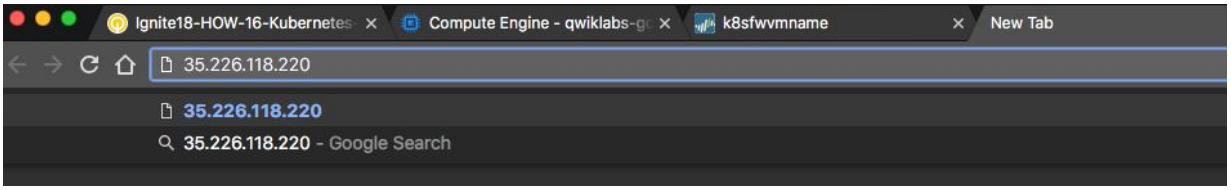
Task 4 – Validate North/South and East/West traffic in the firewall logs

First, repeat the steps done in Activity 6, Task 3.

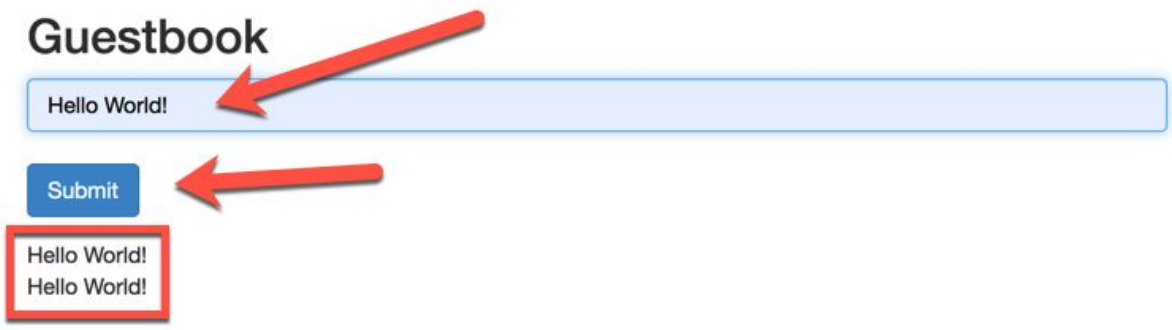
➔ Navigate to the **Compute Engine> VM Instances** screen. Then click on the firewall and **copy** the **External IP address** of the **untrust** interface:



➔ In your browser navigate to the untrust-ip address of the firewall: http://<firewall untrust External IP>

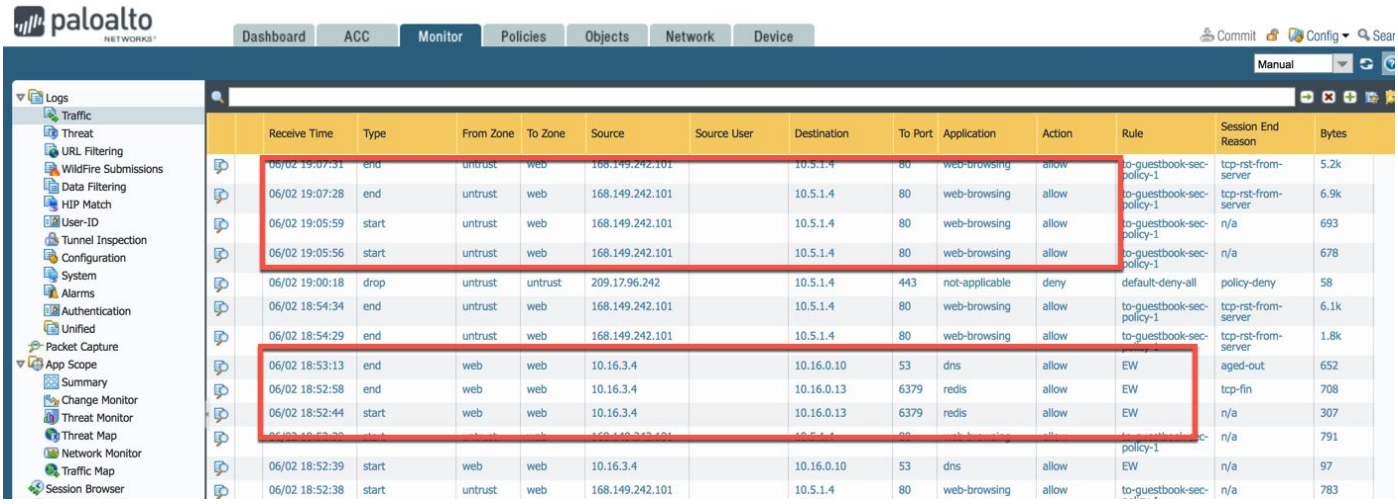


➔ This time when the Guestbook application comes up, enter something in the text field and click **Submit.**



If the text is echoed below the Submit button, that is showing that the application successfully wrote the entry to the redis backend.

➔ Navigate back to the Firewall and click on the Monitor Tab



Looking at the logs, it is evident that the firewall is seeing the inbound web traffic and the Redis traffic between the Frontend Pods and the Backend Redis pods. Also notice that the East/West traffic is hitting a specific rule that will give the ability to create an application based rule and also enable Threat analysis to protect the backend pods with more security capabilities than the native cloud ACL based constructs.

End of Activity 10

Conclusion

Congratulations! You have now successfully integrated the Palo Alto Networks VM-Series firewall to gain visibility into North/South traffic entering and leaving the Kubernetes cluster and E/W traffic between a Guestbook Application Frontend and Backend pods. You have also leveraged the Prisma Public Cloud free public APIs to ensure that the deployment is not introducing known CVEs from the repository via the manifest file.