Lab1: Deploying GKE resources using Terraform

Contents

[Lab Overview 2](#_Toc189558899)

[Components to be Created 2](#_Toc189558900)

[Files 2](#_Toc189558901)

[Phase1:Run1 Deployment without pod resource controls 2](#_Toc189558902)

[Introduction 2](#_Toc189558903)

[Steps 3](#_Toc189558904)

[Analysis 7](#_Toc189558905)

[Phase1:Run2 Deployment with pod resource controls 7](#_Toc189558906)

[Introduction 7](#_Toc189558907)

[Steps 7](#_Toc189558908)

[Analysis 9](#_Toc189558909)

[Phase1:Run3 Deployment with horizontal pod autoscaler 9](#_Toc189558910)

[Introduction 9](#_Toc189558911)

[Steps 10](#_Toc189558912)

[Analysis 12](#_Toc189558913)

[Phase2 Deployment using Helm chart 12](#_Toc189558914)

[Introduction to Helm charts 12](#_Toc189558915)

[Steps 13](#_Toc189558916)

[Lab Clean-Up 15](#_Toc189558917)

## Lab Overview

The objective of this lab is to provision a Google Kubernetes Engine (GKE) cluster and deploy a sample NGINX application into it using Terraform. This lab focuses on understanding the core components required to set up Kubernetes infrastructure in Google Cloud Platform (GCP) and manage workloads using Infrastructure as Code (IaC) principles.

**Lab Phases:**

**Phase 1:** Deploy applications using Terraform with three modifications:

* **Run 1:** Deployment without pod resource controls or deployment autoscaling.
* **Run 2:** Deployment with pod resource controls but no deployment autoscaling
* **Run 3:** Deployment with pod resource controls and deployment autoscaling.

**Phase 2:** Deploy applications using Helm charts, managed by Terraform.

## Components to be Created

**GCP Project (existing):** Where resources will be deployed.

**GKE Cluster:** A managed Kubernetes cluster hosted on GCP.

**Kubernetes Deployment:** To manage the desired state of the NGINX application.

**Kubernetes Service:** A LoadBalancer service to expose the NGINX application externally.

**Terraform Outputs:** To retrieve service IP.

**Helm Chart:** A single resource for all NGINX requirements.

## Files

Lab1/

├── main.tf Defines the GKE cluster resource

├── providers.tf Defines providers: google, kubernetes, and helm

├── outputs.tf Outputs nginx service IP

├── helm.tf1 Helm chart (initially inactive)

└── deployments.tf Defines kubernetes nginx deployment, service and autoscaler

## Phase1:Run1 Deployment without pod resource controls

### Introduction

1. The current terraform files will deploy a 2-node zonal GKE cluster in Europe-West2 (main.tf). Into this cluster, it will then deploy a kubernetes deployment comprising of 2 replicas of a nginx pod which will sit behind a Loadbalancer service, front-faced with a public IP address (see deployments.tf).

### Steps

1. In your IDE terminal, ensure you have navigated to C:\google-tf-int\lab1 and then initialize terraform

**terraform init**

1. Once initialized run the following commands to view, and then apply the deployment

**terraform plan**

**terraform apply** followed by **yes**

**A screen shot of a computer program

Description automatically generated**

1. The deployment may take up to 15 minutes to complete so grab a break. Navigate to the Google Cloud Console and verify the creation of 2 resource groups. The main resource group is defined in our code (line 7 of provider.tf), the 2nd one is the node resource group and is created automatically. It is auto-named by default, but its name can be defined (line 6 of main.tf)…

##### Review the deployed kubernetes resources; 1 deployment, 2 pods, 1 Replica set and 1 Service …

1. Aside from the GUI, **kubectl** can be used to interact with the cluster API service for cluster administration tasks. Running **kubectl** commands against the cluster requires authentication and cluster identification. These details are stored in a config file in the .kube folder of your home directory by default. The following command will get your cluster details and credential and will append them to you current .kube\config file…

**gcloud container clusters get-credentials <cluster name> --zone <cluster zone> --project <project\_id>**

1. To view the results of the Kubernetes deployment using terraform, run the following commands…

**kubectl get services**

**A screen shot of a computer

Description automatically generated**

This command reveals the public IP address of the LoadBalancer service that is sitting in front of the nginx pods. (This is also displayed as an output as specified in lines 6-9 in **outputs.tf**)

Browse to this address using http://<LoadBalancer External IP> ….

**A screenshot of a computer

Description automatically generated**

**kubectl get deployments**

**A black background with white text and letters

Description automatically generated**

This command shows that the nginx deployment is for 2 replica pods and that both are up to date and available.

**kubectl get nodes**

**A screenshot of a computer screen

Description automatically generated**

This command shows the 2 nodes that make up the cluster.

**kubectl get pods**

**A screen shot of a computer

Description automatically generated**

This command identifies the name of the current pods that make up this 2 replica deployment

**kubectl top nodes** and **kubectl top pods**

**A screen shot of a computer program

Description automatically generated**

These commands show the current cpu and memory utilization levels being supported by the nodes as generated by the pods

1. We will now access a pod and run a cpu intensive process to view and understand how Kubernetes deals with this. Note that the pod specification (lines 28 to 36 of **deployments.tf)** currently has no resource limits applied to the container that runs inside the pods.
2. Open a new terminal session, identify one of the pods in the deployment, connect to it and initialize a cpu intensive process, “stress”.…

A screenshot of a computer

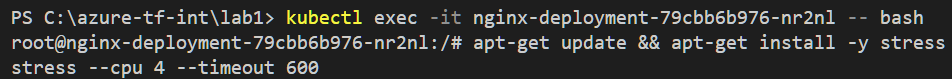
Description automatically generated

**kubectl get pods**

**kubectl exec -it <one-of-your-nginx-pods-name> -- bash**

**apt-get update && apt-get install -y stress**

**stress --cpu 4 --timeout 600**

****

1. Switch to your initial terminal

**kubectl top pods** and **kubectl top nodes**

**A screenshot of a computer program

Description automatically generated**

You may need to re-run these command a few times to get updated values. Notice how the one pod is comsuming all of the cpu on one of the cluster nodes.

1. Switch to second terminal and kill the “stress” process

**Ctrl+c**

**exit**

1. Switch back to initial terminal

**kubectl top pods** and **kubectl get nodes**

Periodically re-run these commands….

**A screen shot of a computer program

Description automatically generated**

### Analysis

Without constraints, a pod can demand excessive resources from the node on which it is running. This will affect the overall capacity of your cluster and is not best practice.

## Phase1:Run2 Deployment with pod resource controls

### Introduction

1. In this run you will impose resource constraints on the pods that make up the deployment and test the impact.

### Steps

1. Un-comment lines 29-36 in **deployments.tf** to set resource limits on your pods…

A screen shot of a computer program

Description automatically generated

1. Apply these changes..

**terraform apply** followed by **yes**

**A computer screen with white text

Description automatically generatedA screen shot of a computer

Description automatically generated**

1. Switch to your second terminal session, identify one of the pods in the deployment (pods are immutable so the changes to the deployment will have destroyed the original pods and recreated new ones), connect to it and initialize the cpu intensive process, “stress”.…,

**kubectl get pods**

**kubectl exec -it <pod name> -- bash**

**apt-get update && apt-get install -y stress**

**stress --cpu 4 --timeout 600**

A screen shot of a computer

Description automatically generated

1. Switch back to your initial terminal session..

**kubectl** **top pods/nodes**

A screen shot of a computer program

Description automatically generated

Repeat these commands several time and observe the increase in pod cpu utilization and corresponding increase in node cpu utilization. The pod will not be granted more than 200m of CPU, as defined in the pod specification…

1. Switch to your second terminal and kill the stress processes

**Ctrl+c**

**exit**

1. Kill this terminal (right click over highlighted terminal and select Delete), leaving 1 terminal session..

A blue and white squares with white symbols

Description automatically generated

1. Monitor the load decrease…

**kubectl top pods** and **kubectl top nodes**

Repeat these commands several time and observe the decrease in pod cpu demand and corresponding decrease in node cpu utilization.

A screen shot of a computer program

Description automatically generated

### Analysis

With constraints, a pod cannot demand excessive resources from the node on which it is running. This will ensure the nodes are not overutilized, but the application itself may perform poorly under heavy demand situations.

## Phase1:Run3 Deployment with horizontal pod autoscaler

### Introduction

1. In this run you will leave resource constraints on the pods in place but allow more capacity as necessary by utilizing a horizontal pod autoscaler that allows more replicas of the pod to be initialized based on pod resource utilization levels.

**Note.** In this lab, the target cpu utilization is set to 5%, indicating that a pod generating more that 5% cpu utilization be considered overworked. This threshold value would be much higher in production environments.

### Steps

1. Un-comment lines 65-82 in **deployments.tf** to configure the horizontal pod autoscaler. Note that it specifies a minimum of **2** and a maximum of **4** replicas. It also sets the target cpu utilization at **5%**

A screen shot of a computer program

Description automatically generated

1. Plan and apply these changes..

**terraform plan**

**terraform apply** followed by **yes**

**A black background with white text

Description automatically generated**

**A black screen with white text

Description automatically generated**

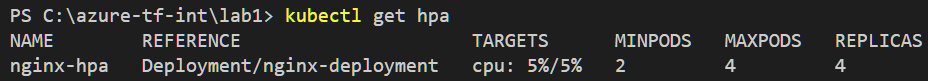
1. Generate web traffic against the load balancer by running 10 parallel web requests against it, replacing **<nginx\_service\_ip>** with the IP displayed as output in your terminal…

**for ($i = 0; $i -lt 10; $i++) { Start-Job -ScriptBlock { while ($true) { Invoke-WebRequest -Uri http://<nginx\_service\_ip> -UseBasicParsing | Out-Null } } }**

A screen shot of a computer

Description automatically generated

1. Periodically run **kubectl get hpa** to view the horizontal pod autoscaler increase the number of replicas from 2 to 4 as the cpu utilization of the initial pods exceeds 5%…



1. List the currently running pods and their cpu load using **kubectl top pods**

A screenshot of a computer code

Description automatically generated

**The issue**

1. Kubernetes has increased the pod count to 4 but the terraform files state there should be 2 replicas and plan/apply operations would propose destroying 2 pods, contradicting what the kubernetes control pane thinks the kubernetes deployment state should be.

**terraform plan**

A black background with white text

Description automatically generated

1. Stop the running jobs that are generating web-traffic..

**Get-Job | Where-Object { $\_.Command -like "\*Invoke-\*" } | Stop-Job**

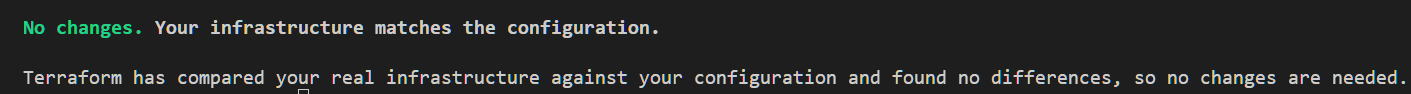
**Get-Job | Where-Object { $\_.State -eq 'Stopped' } | Remove-Job**

1. Wait till the autoscaler scales down the deployment, this can take 5 minutes so grab a break, then run **kubectl get hpa** and **kubectl top pods** periodically until replica count drops back to 2…

A screen shot of a computer

Description automatically generated

1. Re-run **terraform plan**…



1. Note that now the kubernetes state matches the terraform state. This behaviour can cause state inconsistencies

### Analysis

1. While Terraform provides robust infrastructure management capabilities, managing Kubernetes deployments directly with Terraform introduces certain challenges. Terraform maintains a state file to track the resources it manages. If changes are made outside of Terraform (e.g., scaling deployments with kubectl scale), the actual cluster state can drift from Terraform's known state.
2. Kubernetes resources, especially pods, are dynamic by nature. Terraform, designed for more static infrastructure, may struggle to track transient changes like pod restarts, rescheduling, or autoscaling activities. As deployments grow in complexity, managing all configurations (like ConfigMaps, Secrets, RBAC policies) within Terraform can become cumbersome compared to Kubernetes-native tools. Some Kubernetes resource configurations and advanced features may not be fully supported or exposed through the Terraform Kubernetes provider, requiring workarounds or custom solutions.

## Phase2 Deployment using Helm chart

### Introduction to Helm charts

Helm is a powerful package manager for Kubernetes that simplifies the deployment, configuration, and management of applications within a Kubernetes cluster. Much like how apt or yum manage packages on Linux systems, Helm manages Kubernetes applications through reusable templates known as **Helm charts**. These charts bundle Kubernetes manifests (like deployments, services, and config maps) with default configuration values, allowing users to deploy complex applications with just a few commands or configuration adjustments.

When managing Kubernetes with Terraform, Helm plays a crucial role in reducing complexity. While Terraform excels at infrastructure provisioning—such as setting up clusters, networking, and cloud resources—managing dynamic, application-level resources like deployments, services, and autoscalers can become cumbersome. Defining these resources directly with Terraform requires extensive boilerplate code, and handling updates, rollbacks, or versioning manually increases operational overhead.

Integrating Helm with Terraform bridges this gap. Terraform's **Helm provider** allows you to deploy and manage Helm charts declaratively, just like other infrastructure components. Instead of writing verbose Kubernetes resource definitions, you simply reference a Helm chart and specify key configuration values. This approach reduces code duplication, simplifies resource management, and leverages the Kubernetes community’s best practices baked into popular charts. Additionally, Helm’s versioning and rollback features provide greater flexibility in managing application lifecycle changes, making deployments more resilient and adaptable in dynamic environments.

By combining Terraform’s infrastructure-as-code capabilities with Helm’s application management strengths, teams can achieve a more streamlined, efficient, and scalable approach to Kubernetes operations.

### Steps

1. Run **terraform state list** to review the currently deployed resources…

A screen shot of a computer program

Description automatically generated

1. Rename **deployments.tf** to **deployments.tf1** (right-click over file and select rename), thus removing it from consideration by terraform.
2. Comment out lines 1-4 in **outputs.tf**
3. Run **terraform apply** followed by **yes** to delete the resources previously specified in the deployment file..

A black background with white text

Description automatically generated

1. Run **terraform state list** again to review the remaining deployed resources…

A screen shot of a computer code

Description automatically generated

1. Rename **helm.tf1** to **helm.tf**, thus bringing it into consideration by terraform
2. Uncomment line 18-25 in **provider.tf**
3. Re-initialize terraform to register the helm provider using **terraform init**
4. Apply these changes with **terraform plan**

A screen shot of a computer program

Description automatically generated

Notice that terraform will now create 1 resource as opposed to three previously. This is a helm chart for nginx and allows multiple values to be set. Scroll up to review.

1. Deploy the chart using **terraform apply**, followed by **yes**
2. Once deployed, review the kubernetes resources that have been created by running the following commands:

**kubectl get services**

**kubectl get hpa**

**kubectl get pods**

**kubectl get deployments**

A screen shot of a computer

Description automatically generated

1. Run **terraform state list**

A screen shot of a computer code

Description automatically generated

1. The single helm chart deploys the same resources previously configured across several separate resource blocks, greatly simplifying administration.
2. The nginx chart offers many configuration values that can be set, see: <https://github.com/bitnami/charts/tree/main/bitnami/nginx/#installing-the-chart>

## Lab Clean-Up

1. Destroy all deployed resources with **terraform destroy** followed by **yes**

A screenshot of a computer program

Description automatically generated

1. Switch to the console and verify the deletion of the 2 resource groups associated with this lab.

**### Congratulations, you have completed this lab ###**