

# Configuration Manual

MSc Research Project  
Cloud Computing

**Shashank Arvind Gokhale**  
Student ID: x21226661

School of Computing  
National College of Ireland

Supervisor: Dr. Rashid Mijumbi

**National College of Ireland**  
**MSc Project Submission Sheet**



**School of Computing**

Shashank Arvind Gokhale

**Student Name:** .....

**Student ID:** X21226661 .....

**Programme:** Cloud Computing ..... **Year:** 2023 .....

**Module:** MSc Research Project .....

**Lecturer:** Dr. Rashid Mijumbi .....

**Submission Due Date:** 14/08/2023 .....

**Project Title:** Designing a Robust Multi-Cloud DevOps Strategy for Kubernetes  
 Disaster Recovery: An Open Source Approach .....

..... 1482 ..... 10 .....

**Word Count:** ..... **Page Count:** .....

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

**Signature:** Shashank Arvind Gokhale .....

**Date:** 14/08/2023 .....

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST**

Attach a completed copy of this sheet to each project (including multiple copies)	<input type="checkbox"/>
<b>Attach a Moodle submission receipt of the online project submission,</b> to each project (including multiple copies).	<input type="checkbox"/>
<b>You must ensure that you retain a HARD COPY of the project,</b> both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

<b>Office Use Only</b>	
Signature:	
Date:	
Penalty Applied (if applicable):	

# Configuration Manual

Shashank Arvind Gokhale  
Student ID: x21226661

## 1 Prerequisites

This project aimed to create a robust multi-cloud DevOps strategy for Kubernetes disaster recovery utilizing open-source tooling. The main objective of this study is to create a CI/CD pipeline for Kubernetes disaster recovery and compare the recovery time objective (RTO) and recovery point objective (RPO) variances between two distinct cloud providers. In this study, we considered Google Cloud Platform (GCP) and Microsoft Azure as our two different cloud providers.

### 1.1 Cloud Stack

Create free tier accounts on both GCP and Azure before starting with the implementation of this project. Once the cloud accounts are created create the service principle accounts for both GCP and Azure to provide programmatic access to perform the required changes during pipeline execution. Make sure to generate the secret keys and provide access to these service principles to have contributor-level access as these accounts can assume roles and perform automated tasks on behalf of you.

<input type="checkbox"/>	Email	Status	Name ↑	Description
<input type="checkbox"/>	<a href="mailto:860974856875-compute@developer.gserviceaccount.com">860974856875-compute@developer.gserviceaccount.com</a>	Enabled	Compute Engine default service account	
<input type="checkbox"/>	<a href="mailto:terraform-githubactions-ci-cd@eco-tenure-393916.iam.gserviceaccount.com">terraform-githubactions-ci-cd@eco-tenure-393916.iam.gserviceaccount.com</a>	Enabled	terraform-githubactions-ci-cd	terraform github actions

#### (a) Service account for GCP

Display name ↑	Application (client) ID	Created on ↑ ↓	Certificates & secrets
sp-velero-aks1	77766d6b-26a2-4087-ada4-8a54a08e93d6	29/07/2023	Current
terraform-ci-cd-shagok	c212a1d5-fc5b-4af4-9c2b-963352a403b3	28/07/2023	Current
terraform-github-actions	84d6344b-5c7a-4542-997c-aa3122709eb5	28/07/2023	Current

#### (b) Service account for Azure

Figure 1: Service Accounts Created for our Account.

## 1.2 GitHub

We have used GitHub action as our CI/CD pipeline enabling open-source tool so we need to create a GitHub repository to store all the configuration files written for infrastructure as code (IaC) to be successfully executed by pipeline when initiated. Additionally, this distributed version control git ensures that there is only one source of truth to the pipeline and helps mitigate configuration drift.



Figure 2: GitHub repository created to host configuration files.

Name	Last commit message	Last commit date
..		
Deploy-velero.yml	Update Deploy-velero.yml	2 weeks ago
GCP_Azure_failover.yml	Update GCP_Azure_failover.yml	2 weeks ago
application_restore_azure.yml	Update application_restore_azure.yml	5 days ago
secondary-region.yml	Update secondary-region.yml	3 days ago
terraform-destroy.yml	Update terraform-destroy.yml	2 weeks ago
terraform.yml	Update terraform.yml	2 weeks ago

Figure 3: GitHub workflow files created for Azure

## 2 Terraform Setup

We have utilized terraform as an infrastructure orchestrator to create necessary resources which are the google kubernetes engine (GKE) and azure kubernetes service (AKS).

### 2.1 Terraform Prerequisites

We are utilizing Terraform backend to store the state files generated by Terraform. So as to achieve remote backend capability we need to create object stores on both GCP and Azure and configure terraform to use them. In our case, we have a bucket named "terraformstate" for Azure and "terraform-state-github-ci-cd" for GCP as shown in Figure 4. Terraform state file is locked if a job is initialized to make resource changes having this on the object store helps achieve easy collaboration with multiple developers.

```

Code Blame 45 lines (36 loc) · 1.1 KB
1 terraform {
2   backend "azurerm" {
3     resource_group_name = "my-aks-cluster-rg"
4     storage_account_name = "terraformstateshagok"
5     container_name      = "terraformstate"
6     key                  = "secondary.terraform.tfstate"
7   }
8 }
9
10 provider "azurerm" {
11   features {}
12 }
13

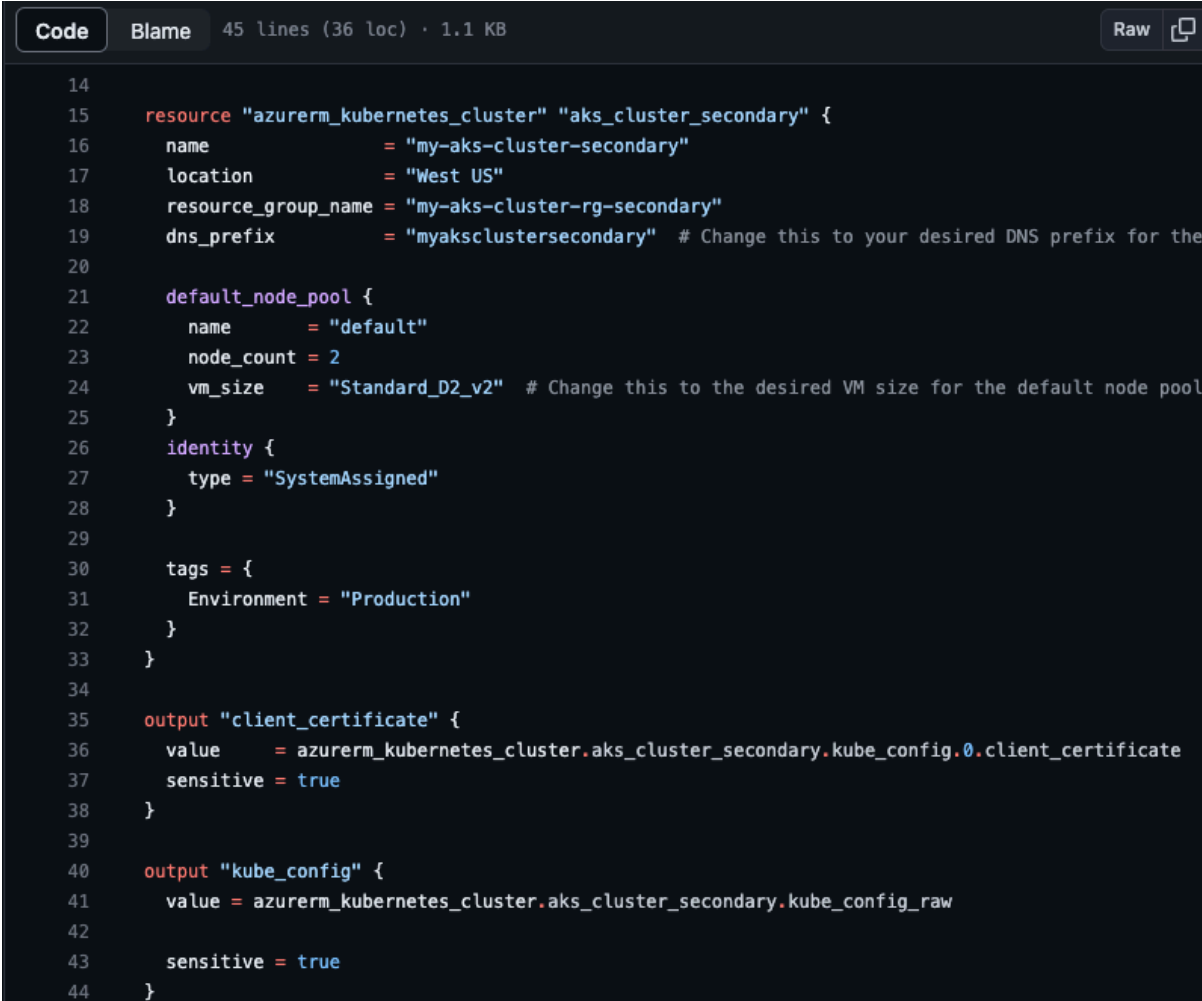
Code Blame 17 lines (16 loc) · 308 Bytes
1 terraform {
2   required_providers {
3     google = {
4       source = "hashicorp/google"
5       version = "4.75.0"
6     }
7     kubectl = {
8       source = "gavinbunney/kubectl"
9       version = ">= 1.7.0"
10    }
11  }
12  backend "gcs" {
13    bucket = "terraform-state-github-ci-cd"
14    prefix = "prodsecondary"
15  }
16 }

```

Figure 4: Remote backend for GCP and Azure

## 2.2 Terraform Code Developed

Figure 5 shows the terraform code snippet developed to create an AKS cluster on Azure. VM instance used for AKS is "Standard-D2-v2" and 2 nodes of this have been deployed to ensure resilience and the location is "West US". These parameters can be changed as per your requirement. similarly terraform code is generated for GCP.



```
Code Blame 45 lines (36 loc) · 1.1 KB Raw
14
15 resource "azurerm_kubernetes_cluster" "aks_cluster_secondary" {
16     name             = "my-aks-cluster-secondary"
17     location         = "West US"
18     resource_group_name = "my-aks-cluster-rg-secondary"
19     dns_prefix       = "myaksclustersecondary" # Change this to your desired DNS prefix for the
20
21     default_node_pool {
22         name         = "default"
23         node_count  = 2
24         vm_size     = "Standard_D2_v2" # Change this to the desired VM size for the default node pool
25     }
26     identity {
27         type = "SystemAssigned"
28     }
29
30     tags = {
31         Environment = "Production"
32     }
33 }
34
35 output "client_certificate" {
36     value     = azurerm_kubernetes_cluster.aks_cluster_secondary.kube_config.0.client_certificate
37     sensitive = true
38 }
39
40 output "kube_config" {
41     value = azurerm_kubernetes_cluster.aks_cluster_secondary.kube_config_raw
42
43     sensitive = true
44 }
```

Figure 5: AKS creation terraform code snippet

## 2.3 Terraform Critical Commands

Some critical commands in Terraform are mentioned below:-

- 'terraform init':- Initializes the terraform configuration, and downloads required plugins declared in terraform configuration files.
- 'terraform plan':- Generates an execution plan by comparing it with the current state which is stored in terraform.tfstate with the desired state defined in terraform configuration files. This command will not alter the provisioned resources of the infrastructure but shows the drift in configuration if any.

- ‘terraform apply’:- Applies the changes to the infrastructure resources defined in terraform configuration files.
- ‘terraform destroy’:- Deletes all the infrastructure resources which were created in apply phase that is present in terraform.tfstate file.

## 2.4 Terraform Initialization in Pipeline

The below code shows how terraform is installed and initialized in the pipeline for the azure cloud:-

```

Azure-DR-pipelines / .github / workflows / terraform.yml
Code Blame 52 lines (45 loc) · 1.75 KB
11     id-token: 'write'
12     runs-on: ubuntu-latest
13     defaults:
14       run:
15         shell: bash
16     steps:
17     - name: Checkout
18       uses: actions/checkout@v2
19     # Install the latest version of Terraform CLI and configure the Terraform CLI configuration
20     - name: Setup Terraform
21       uses: hashicorp/setup-terraform@v1
22
23
24     # Initialize a new or existing Terraform working directory by creating initial files, loadi
25     - name: Terraform Init
26       env:
27         ARM_CLIENT_ID : ${ secrets.ARM_CLIENT_ID }
28         ARM_TENANT_ID : ${ secrets.ARM_TENANT_ID }
29         ARM_SUBSCRIPTION_ID : ${ secrets.ARM_SUBSCRIPTION_ID }
30         ARM_CLIENT_SECRET : ${ secrets.ARM_CLIENT_SECRET }
31       run: terraform init
32
33     # Checks that all Terraform configuration files adhere to a canonical format
34     - name: Terraform Format
35       run: terraform fmt
36
37     # Generates an execution plan for Terraform
38     - name: Terraform Plan
39       env:
40         ARM_CLIENT_ID : ${ secrets.ARM_CLIENT_ID }
41         ARM_TENANT_ID : ${ secrets.ARM_TENANT_ID }
42         ARM_SUBSCRIPTION_ID : ${ secrets.ARM_SUBSCRIPTION_ID }
43         ARM_CLIENT_SECRET : ${ secrets.ARM_CLIENT_SECRET }
44       run: terraform plan
45
46     - name: Terraform Apply
47       env:
48         ARM_CLIENT_ID : ${ secrets.ARM_CLIENT_ID }
49         ARM_TENANT_ID : ${ secrets.ARM_TENANT_ID }
50         ARM_SUBSCRIPTION_ID : ${ secrets.ARM_SUBSCRIPTION_ID }
51         ARM_CLIENT_SECRET : ${ secrets.ARM_CLIENT_SECRET }
52       run: terraform apply -auto-approve

```

Figure 6: Terraform initialization on pipeline

The GitHub runner which we are making use of is running the ubuntu operating system and the default shell configured is bash. We are directing GitHub actions to install Terraform as specified in line 20. lines 31, 35, 44, & 52 perform terraform init, format, plan, & apply respectively.

### 3 Velero Setup

This project utilizes Velero as a backup and restore utility for containerized applications deployed on Kubernetes clusters. velero requires an object store to keep the backed-up content using restic plugin. Figure 7 shows velero architecture.

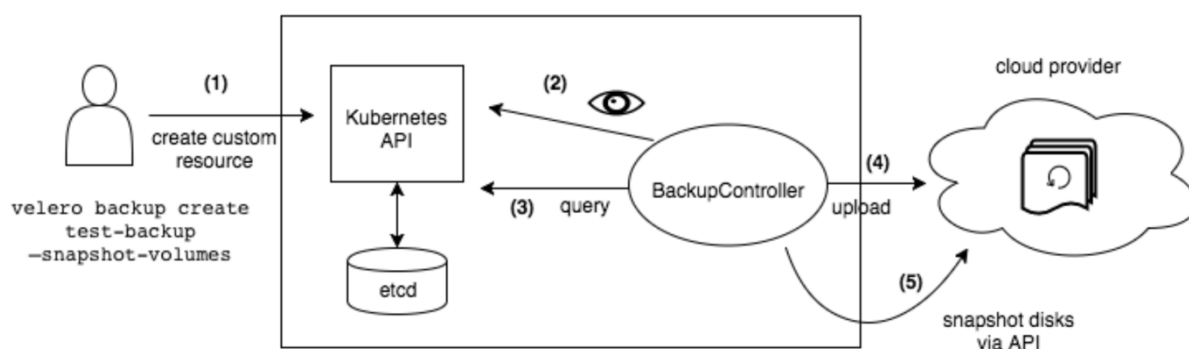


Figure 7: Velero Architecture

The pipeline workflow file dictates downloading velero CLI and making it executable as shown in line 5 of listing 2. Once CLI is downloaded we start installing the velero on kubernetes cluster line 48 shows which provider we are using in this case it is GCP this needs to be changed to azure if we want to deploy it in Azure. Line 51 shows the bucket name which will be utilized to store the backups on google cloud.

```

Code Blame 105 lines (95 loc) · 3.73 KB
32 - name: 'create velero service account'
33   run: |
34     echo "-----Download and Install velero CLI if needed"
35     if [ ! -f ~/gke-casa/velero ]; then
36       wget https://github.com/vmware-tanzu/velero/releases/download/v1.10.2/velero-v1.10.2-linux-amd64.tar.gz
37       tar -zxvf velero-v1.10.2-linux-amd64.tar.gz
38     fi
39 - name: 'Add velero to PATH'
40   run: |
41     mkdir -p ~/bin && cp velero-v1.10.2-linux-amd64/velero ~/bin && echo 'export PATH="$PATH:$HOME/bin"' >> ~/.bashrc && export PATH="$PATH:$HOME/bin"
42     echo '${secrets.GCP_CREDENTIALS}' > credentials-velero
43     ls
44     gcloud container clusters get-credentials k8s-cluster --zone us-central1-a --project eco-tenure-393916
45     velero install \
46       --uploader-type restic \
47       --use-volume-snapshots=false \
48       --provider gcp \
49       --plugins velero/velero-plugin-for-gcp:v1.6.0,velero/velero-plugin-for-csi:v0.4.2,velero/velero-plugin-for-microsoft-azure:v1.6.0 \
50       --use-node-agent --features=enableCSI \
51       --bucket 'shagok-velero-backup' \
52       --secret-file ./credentials-velero

```

Figure 8: velero installation on cluster.

Line 52 passes the credential file to Velero installation as Velero needs to assume roles on the managed k8s cluster on GCP to create backups and restores. this credential file is passed as an environment variable to the GitHub runner ensuring security. Line 46 deploys a demon set on Kubernetes cluster equipped with restic plugin to act as a node agent to perform file-system

backups and store them as repositories in the object store making it easy to migrate to another cloud in the event of disruptions.

### 3.1 Velero Backup Command

Velero can be used to backup different tiers of the k8s cluster such as it can backup only certain pods, namespaces, entire clusters and more. the below command can be used to backup a namespace.

```
velero backup create --include-namespaces "namespace-name" --default-volumes-to-fs backup=true "backup-name"
```

### 3.2 Velero Restore Command

The below-mentioned command can be used to restore the backed-up content on the k8s cluster in the same region, a different region or another cloud provider.

```
velero restore create --include-namespaces "namespace-name" --from-backup "backup-name"
```

## 4 Application Install on Kubernetes cluster

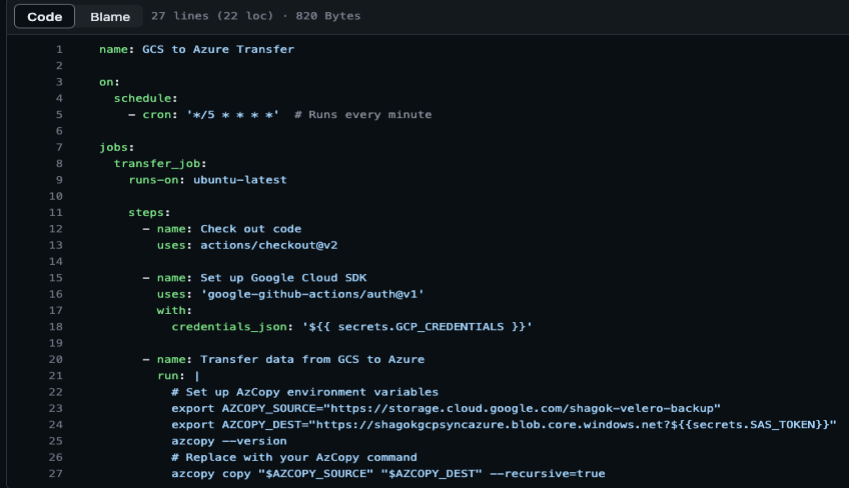
Any application of your choice can be installed to test this pipeline but make sure to change the pipeline code to include the namespace of the application which you have installed and want to perform a restore using velero.

In our case, we made use of a simple wordpress blog as an application to be tested. which uses mariaDB as the database and uses persistent volume claims. we made use of helm package manager to install this application. Once the application is installed login to the WordPress admin page and post a new blog to test whether Velero was successful in restoring the application along with persistent volume or not.

```
helm install my-release oci://registry-1.docker.io/bitnamicharts/wordpress --namespace=wordpress
```

## 5 Azcopy Setup

we have made use of Azcopy to migrate the backups from one object store to another object store hosted on different cloud providers i.e., GCP storage bucket content is migrated to Azure storage bucket and vice versa.



```
Code Blame 27 lines (22 loc) · 820 Bytes
1 name: GCS to Azure Transfer
2
3 on:
4   schedule:
5     - cron: '*/* * * * *' # Runs every minute
6
7 jobs:
8   transfer_job:
9     runs-on: ubuntu-latest
10
11   steps:
12     - name: Check out code
13       uses: actions/checkout@v2
14
15     - name: Set up Google Cloud SDK
16       uses: 'google-github-actions/auth@v1'
17       with:
18         credentials_json: '${{ secrets.GCP_CREDENTIALS }}'
19
20     - name: Transfer data from GCS to Azure
21       run: |
22         # Set up AzCopy environment variables
23         export AZCOPY_SOURCE="https://storage.cloud.google.com/shagok-velero-backup"
24         export AZCOPY_DEST="https://shagokgcpyncazure.blob.core.windows.net/${{secrets.SAS_TOKEN}}"
25         azcopy --version
26         # Replace with your AzCopy command
27         azcopy copy "$AZCOPY_SOURCE" "$AZCOPY_DEST" --recursive=true
```

Figure 9: Shows the code for GCP to Azure data transfer workflow.



## 6 Scenario Implementation

We have selected namely two scenarios which are as follows:

1. Software Upgrade Failure
2. Area Wide Power Outage/ Datacenter Outage in a Region.

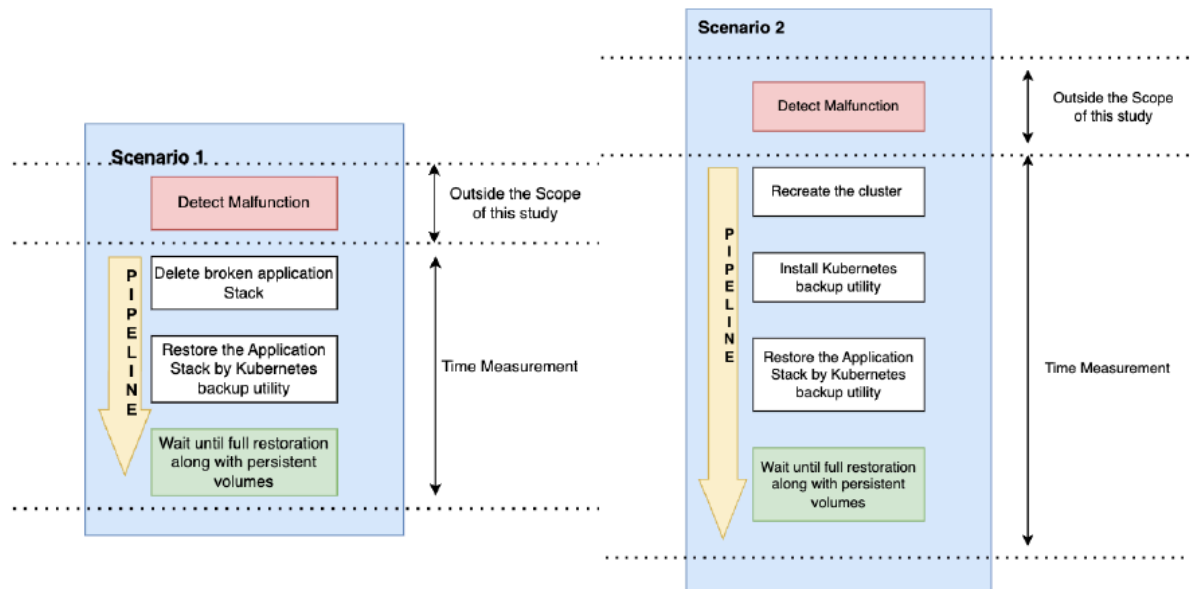


Figure 10: Shows how the implementation should be for the said scenarios.

### 6.1 Scenario1

Below is the code snippet to implement the pipeline for scenario 1 for GCP.

```

78     # Loop 20 times
79     for ((i = 1; i <= 20; i++)); do
80         echo "Running loop $i"
81         # Start timing
82         start_time=$(date +%s)
83         velero get backup-locations
84         velero get backups
85         velero restore create --include-namespaces wordpress --from-backup azure-restic-test
86         sleep 10
87         # Wait for all resources in the namespace to be ready
88         time kubectl wait pods --all --namespace=wordpress --for=condition=Ready --timeout=300s 2>&1 |
89         # Calculate duration
90         end_time=$(date +%s)
91         duration=$((end_time - start_time))
92         #clear the cluster for next iteration
93         kubectl delete ns wordpress
94         kubectl wait namespace wordpress --for=delete
95         # Print total duration with iteration number
96         print_timing "$duration"
97         echo "Loop $i completed."
98     done
99

```

Figure 11: Scenario 1 implementation in pipeline workflow.

Line 85 in the above code restores the WordPress application, and line 88 waits till all the restored pods attain a ready state and appends the time taken to restore in the timing-output.txt

file. Line 93 deletes the restored namespace for the next iteration. We can see that this program iterates 20 times as shown in line 80.

## 6.2 Scenario 2

### 6.2.1 Cloud Failover

The below code snippet shows the pipeline for Azure to GCP failover.

```
Gcloud-DR-pipelines / .github / workflows / Azure_to_GCP_failover.yml
Code Blame 159 lines (141 loc) · 6.14 KB
76 cat << EOF >> /credentials-velero
77 AZURE_SUBSCRIPTION_ID=${{secrets.AZURE_SUBSCRIPTION_ID}}
78 AZURE_TENANT_ID=${{secrets.AZURE_TENANT_ID}}
79 AZURE_CLIENT_ID=${{secrets.AZURE_CLIENT_ID}}
80 AZURE_CLIENT_SECRET=${{secrets.AZURE_CLIENT_SECRET}}
81 AZURE_RESOURCE_GROUP=my-aks-cluster-rg
82 AZURE_CLOUD_NAME=AzurePublicCloud
83 EOF
84 velero install \
85   --uploader-type restic \
86   --use-volume-snapshots=false \
87   --provider azure \
88   --plugins velero/velero-plugin-for-gcp:v1.6.0,velero/velero-plugin-for-csi:v0.4.2,velero/velero-
89   --use-node-agent --features=enableCSI \
90   --bucket 'velerobackup' \
91   --backup-location-config resourceGroup=my-aks-cluster-rg,storageAccount=velerobackupshagok \
92   --secret-file ./credentials-velero
93 cat << EOF > storage-class-Azure.yml

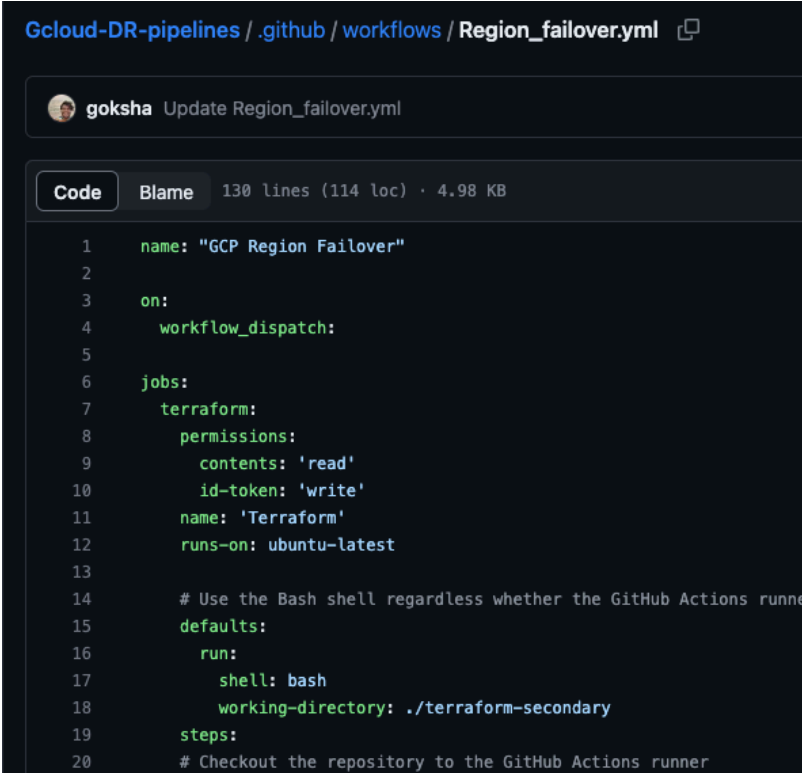
Gcloud-DR-pipelines / .github / workflows / Azure_to_GCP_failover.yml
Code Blame 159 lines (141 loc) · 6.14 KB
119 for ((i = 1; i <= 1; i++)); do
120   echo "Running loop $i"
121   # Start timing
122   start_time=$(date +%s)
123   velero get backup-locations
124   velero restore create --include-namespaces wordpress --from-backup azure-backup
125   sleep 6
126   # Wait for all resources in the namespace to be ready
127   time kubectl wait pods --all --namespace=wordpress --for=condition=Ready --timeout=300s 2>&1 |
128   # Calculate duration
129   end_time=$(date +%s)
130   duration=$((end_time - start_time))
131   #clear the cluster for next iteration
132   #kubectl delete ns wordpress
133   #kubectl wait namespace wordpress --for=delete
134   # Print total duration with iteration number
135   print_timing "$duration"
136   #echo "Loop $i completed."
137   done
138
139 - name: Terraform Destroy
140   run: terraform destroy -auto-approve
141   env:
142     GOOGLE_CREDENTIALS: ${ secrets.GCP_CREDENTIALS }
143
```

Figure 12: Code snippet showing Azure to GCP failover.

The first half of the pipeline till line number 55 is responsible for creating the infrastructure resources and configuring gcloud CLI to establish communication with the GCP programmatically as shown in Figure 6. Lines 84 to 137 are responsible to install the velero and perform the restoration WordPress application. line 139 calls for terraform destroy which destroys all infrastructure resources for the next iteration. In a similar manner, GCP to Azure failover has been coded.

### 6.2.2 Region Failover

Region failover is similar to cloud failover, but the only change is the terraform part of the code will be changed and the working directory will be changed in the pipeline as shown in the below code.



```
1 name: "GCP Region Failover"
2
3 on:
4   workflow_dispatch:
5
6 jobs:
7   terraform:
8     permissions:
9       contents: 'read'
10      id-token: 'write'
11     name: 'Terraform'
12     runs-on: ubuntu-latest
13
14     # Use the Bash shell regardless whether the GitHub Actions runner
15     defaults:
16       run:
17         shell: bash
18         working-directory: ./terraform-secondary
19     steps:
20     # Checkout the repository to the GitHub Actions runner
```

Figure 13: Shows the code snippet of application restore in different region.

Line 18 shows the working directory being changed to terraform-secondary where terraform configuration files are kept for that region.

## 7 Analysis & Plot for Timing Data Collected

Once both scenarios are iterated for 20 cycles we get the timing data to calculate the mean time to restore. we have made use of Python to plot the density plot for the scenarios below is a Python program used to plot a comparative chart for scenario 1. Similarly, it is generated for scenario 2.

```

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from scipy.stats import norm

# Sample data in seconds (replace with your actual data)
gcp_durations = [71, 80, 73, 71, 105, 67, 71, 67, 84, 87, 66, 128, 75, 67, 69, 74, 76, 76, 67, 67]
azure_durations = [122, 98, 112, 111, 234, 101, 110, 137, 104, 120, 89, 109, 105, 93, 70, 113, 98, 135, 101, 91]

# Combine data and labels
data = {'Cloud Provider': ['GCP'] * len(gcp_durations) + ['Azure'] * len(azure_durations),
        'Loop Duration': gcp_durations + azure_durations}

# Create a density plot
plt.figure(figsize=(5, 3))
sns.kdeplot(data=data, x='Loop Duration', hue='Cloud Provider', fill=True)

# Calculate the means and standard deviations for each group
gcp_mean = np.mean(gcp_durations)
gcp_std = np.std(gcp_durations, ddof=1) # Using Bessel's correction for sample standard deviation
azure_mean = np.mean(azure_durations)
azure_std = np.std(azure_durations, ddof=1)

# Calculate the confidence intervals for the means
gcp_margin_error = 1.96 * (gcp_std / np.sqrt(len(gcp_durations))) # 95% Confidence Interval
azure_margin_error = 1.96 * (azure_std / np.sqrt(len(azure_durations))) # 95% Confidence Interval

# Plot the confidence intervals as shaded areas
plt.fill_betweenx([0, 0.06], gcp_mean - gcp_margin_error, gcp_mean + gcp_margin_error, color='blue', alpha=0.3)
plt.fill_betweenx([0, 0.06], azure_mean - azure_margin_error, azure_mean + azure_margin_error, color='orange', alpha=0.3)

# Add labels and title
plt.xlabel('Loop Duration (seconds)')
plt.ylabel('Density')
plt.title('Density Plot of Loop Durations for GCP and Azure \n with 95% Confidence Intervals')

# Show the plot
plt.show()

```

Figure 14: code snippet Python Code for Density Plot and 95% Confidence Intervals.

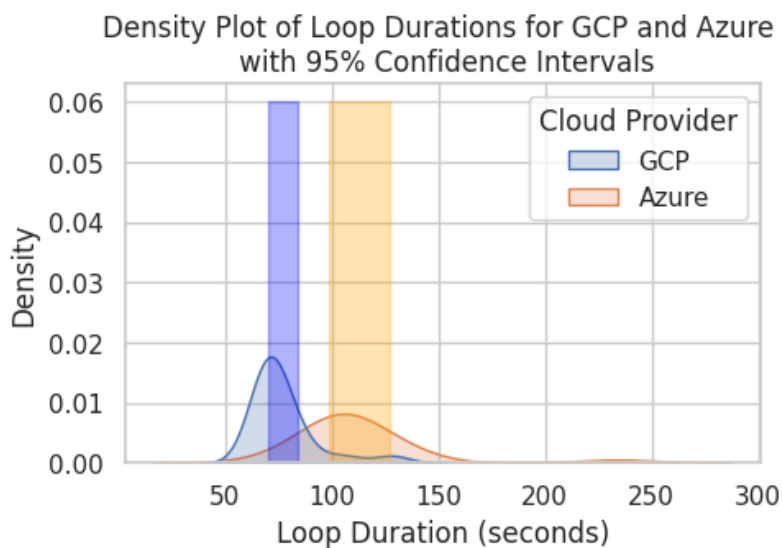


Figure 15: This will be the generated plot from the running code provided in Figure 14.

