Report

This report contains explanation about the approach and choices made during the implementation of assessment and also challenges faced while implementation.

1. Web Application

As instructed, I have created a simple Java web app using start.spring.io. It contains the Spring Web module. I chose the Spring Web module because it includes an embedded Tomcat server, which allows the app to be easily packaged into an image and serve your Java web app without much extra effort.

2. Dockerfile

Furthermore, I have packaged the Java web app into a Docker image by writing a Dockerfile and building the image from it.

I used the official JDK 21 image as the base to ensure security, stability, and the absence of vulnerabilities, as the JDK 21 image is officially managed by Java.

Additionally, for the reusability of the Dockerfile, I have added a variable that takes the JAR file path to copy the app package into the image. By using this variable, we can pass different builds' JAR file paths to create different versions of the same Java app in the form of a Docker image.

3. Helm Chart

I have created helm chat for java web app and it contains YAML files for following K8S objects.

1. Namespace
   This namespace will be created to contain all the Java web app-related K8S objects. This ensures that our deployment is isolated from other workloads running on the cluster. And by using this we can also deploy the same instance of application in same cluster by just changing namespace vale in helm.
2. Deployment
   Since our Java web app is a simple web service that serves just a single API, using a Deployment object is the best choice to host this kind of workload.
3. Service (ClusterIp)
   We don’t want to expose our Java web app directly to the internet, so I have created a ClusterIP type service. This service allows multiple pods to communicate internally within the cluster. Traffic from the ingress is directed to this service, which then distributes the traffic among the pods with matching selectors.
4. Ingress
   This is an Ingress resource that receives traffic from the load balancer and routes it internally within the cluster to the service according to the rules defined in it. Additionally, this Ingress resource will adjust its annotations to work with either the AWS LoadBalancer Controller or the Nginx Controller. The adjustment will be based on variable values passed to it via Helm.
5. Horizontal Pod Autoscaler
   I have also created an HPA to automatically scale up or down the pods based on CPU utilization. The minimum and maximum replica counts are also adjustable via variables provided by Helm to the HPA.
6. Pod Disruption Budget
   I have created a PDB to ensure that certain pods stay up at all times. The minimum available pods value is also adjustable via variables provided by Helm to the PDB.

I have set many variables in the values.yaml file to increase the flexibility and reusability of the Helm chart. I have also utilized helper templates to reduce code repetition.

4. Bash Script

I have also created a bash script that can be used to install, upgrade, or uninstall the Helm chart. Its functions are straightforward: one command with two parameters for installing and upgrading the Helm chart, and one command with one parameter for uninstalling it. It's a simple yet effective bash script.

5. Terraform Code

I have organized Terraform code in different files for better maintainability and understanding. I haven't used Terraform modules from registry and have written whole code from scratch for better customization.

Whole Terraform code is also customizable via variables and I have also set default values to those variables to make sure smooth infrastructure provisioning.

I have also utilized helm provider in Terraform code to setup AWS LoadBalancer Controller and Metrics server since both of these are required by java web app helm chart to function properly.

All required roles and policies are also being created by Terraform code so that our provisioned infrastructure can have all necessary permissions to work properly.

6. GitHub Actions

To enhance security, I have used AWS OIDC to establish a connection between GitHub and AWS. By using OIDC, we avoid the need to use AWS access key ID and secret access key anywhere in the automation process. Additionally, we can easily control which GitHub repository and branch can connect to our account and manage the permission levels, all from within our AWS account.

I have used ECR to push Docker images from the workflow. Pods in the EKS cluster retrieve the image from ECR, keeping the data within the AWS network, which enhances the security of the application.

For better stability of the CI/CD process, I have added a workflow that first builds and validates the Java web app. There is another workflow that validates and plans the Terraform code before applying it. This setup ensures that if there is any issue, the workflow will fail during validation and testing, preventing faulty code from being deployed.

I have also used composite actions to increase code reusability and reduce code repetition.

To make the code more modular and reusable, I have used workflow calls to build two different workflows: one for deployment and one for destruction. Both workflows share some common steps to perform the desired tasks. For example, the Terraform Test workflow is used by both workflows (deploy and destroy) to ensure the code is not faulty before applying terraform apply or terraform destroy.

Challenges:

1. CoreDns Pods Stuck

Problem

If you want to run EKS only with Fargate, you might encounter a well-known issue. When deploying an EKS cluster, CoreDNS pods will spin up and get stuck in a pending state with an event message indicating that there are no available nodes to schedule. This happens because the EKS cluster starts the CoreDNS pods immediately after the cluster creation is completed. However, the Fargate profile is created after the CoreDNS pods are scheduled, so the Fargate annotation is not applied to the CoreDNS pods. As a result, the pods remain in a pending state with an error message stating that there are no available nodes to schedule.

Solution

To address this situation, I have explicitly defined a resource for the CoreDNS add-on in the Terraform code and made it dependent on the Fargate profile. This ensures that the Fargate profile is created first, and the CoreDNS add-on is added to the EKS cluster only after the profile is created. This way, the Fargate profile will be available to host CoreDNS pods before they are scheduled.

2. EKS Access issue

Problem

When GitHub Actions provision infrastructure in your AWS account, it will use the role you created for it. By default, when an EKS cluster is created, it grants access permissions in the aws-auth ConfigMap only for the user who created the cluster. As a result, when you try to browse resources from the AWS console in the EKS cluster, you might see an error saying, “Your current IAM principal doesn’t have access to Kubernetes objects on this cluster”.

Solution

To resolve this issue, I have added an entry for my account in the trust policy of the role created for GitHub Actions. By doing this, my IAM user can assume and switch to that specific role, allowing me to access the cluster via the Console or CLI.

3. Passing tfvars file to GitHub Action

Problem

It is not recommended and is against security best practices to store our .tfvars file in the repository along with the Terraform code. So, how can we securely pass the .tfvars file to our GitHub Actions?

Solution

To pass the .tfvars file to GitHub Actions, I have used GitHub Actions Secrets. I stored the entire .tfvars file as a secret and then included a step in the workflow that retrieves the .tfvars file from the secret and writes it to a terraform.tfvars file on the fly. This approach allows me to securely pass the .tfvars file to my workflows, ensures it is not visible in logs, and deletes it once the job is completed.