SK Assignment

Coding task, container task and terraform code task are placed on public github repository:

https://github.com/Nedzadp/skf-assignment

Architecture Task

Written Explanation of the Diagram Components, Operation, and Dataflow:

We need to decide which region to use to deploy the infrastructure.

1. ECR

The first step is to create an ECR repository where we will store the Docker image of the app. I also added a lifecycle policy to retain only the last two Docker images.

2. VPC

Next, we create a new VPC with three public subnets in different Availability Zones (AZ) to ensure high availability and fault tolerance of the resources.

3. ECS

The next step is to create an ECS cluster. After the creation of the ECS cluster, we proceed with the creation of a task definition.

The task definition will specify which Docker image to use from the ECR. That's why, in the diagram, we can see an arrow from the task definition to the ECR. The task definition execution role has permission to retrieve the image from the ECR. CloudWatch logging is enabled, and a log stream is created.

Next, we create an ECS service. When defining the ECS service, we choose the launch type FARGATE.

Since we want our app running continuously, we will set the application type to be a service. In the ECS service, we will point to the already created task definition.

The ECS service should select the VPC and subnets created in step 2. During the ECS service creation step, we also define a load balancer. A new application load balancer is created, and a new target group is set up.

In the ECS service, we can also define the number of tasks to run. Each task represents containers defined in the task definition. The ECS service creation can also include scaling policies, but for the purpose of this assignment, this is skipped.

4. Security Groups

Create a security group for the ALB and allow access from any IP.

We define a new security group for the ECS service to only allow access from the security group of the ALB.

API Access

To access our service, the application load balancer is exposed to the public. The application load balancer, after receiving a request, will route the request to the appropriate service task in the ECS cluster.

First Run

On the first run, all AWS components should be created. However, since we haven't pushed any Docker image to the ECR, the ECS service deployment will fail.

Pushing the Docker image to the ECR should be done through a CI/CD pipeline.

CI/CD will also apply terraform code to install infrastructure and then docker image should be build and pushed to created ECR.

Also, in github repository https://github.com/Nedzadp/skf-assignment/blob/main/3_architecture_task/skf-assignment.drawio.png diagram is available.

CI/CD Task

The Azure DevOps (ADO) pipeline runs by default on a Microsoft-hosted agent. Alternatively, we can configure a self-hosted agent, but this requires managing the infrastructure. To specify a particular agent, use the following syntax in the azure-pipeline.yml file:

```
pool:
  name: 'self-hosted-agent-pool'
```

To automate the entire process from creating the infrastructure to releasing new code changes, I would follow these steps:

1. As mentioned in the description, we need to define multiple environments, which means multiple infrastructures.
2. To achieve this, I would store the Terraform state file on a remote backend like S3 and use a DynamoDB table to prevent locking.
3. Each environment—dev, QA, staging, and production—should have a defined remote backend configuration and use different S3 buckets and different DynamoDB tables.
4. In the Terraform code, we can include a backend.config file for each environment, containing the configuration of the S3 remote backend. Then, during the initialization, provide this file using the -backend-config flag.

The CI/CD pipeline should trigger on each commit to the main branch. Commits to branches should only occur after a pull request (PR) is reviewed and approved for merging into the main branch.

After a merge to the main branch, the CI/CD pipeline should automatically trigger.

Each environment deployment will represent a separate stage in the pipeline.

We also need to define pipeline environment variables to access AWS.

Each stage will have a defined job to run the following steps:

1. Run terraform init & terraform apply for the specific environment.
2. If the application requires environment variables, I would store them in the Azure DevOps secure files library and then use them in this step to override the existing .env file in the project.
3. Build the Docker image.
4. Push the Docker image to AWS ECR.

The production environment is unique. Due to its sensitivity, in Continuous Delivery, the software is always release-ready, but with manual approval.

Azure DevOps enables the definition of approval gates. For the production stage, we can set reviewers responsible for approving the stage's execution. To achieve this, we need to define approvers (a group or user ID that can approve the deployment). In the azure-pipeline.yml file, we need to define the - approvals directive and specify a condition for the production stage.

I also included sample azure-pipeline.yml https://github.com/Nedzadp/skf-assignment/blob/main/5_cicd/azure-pipeline.yml