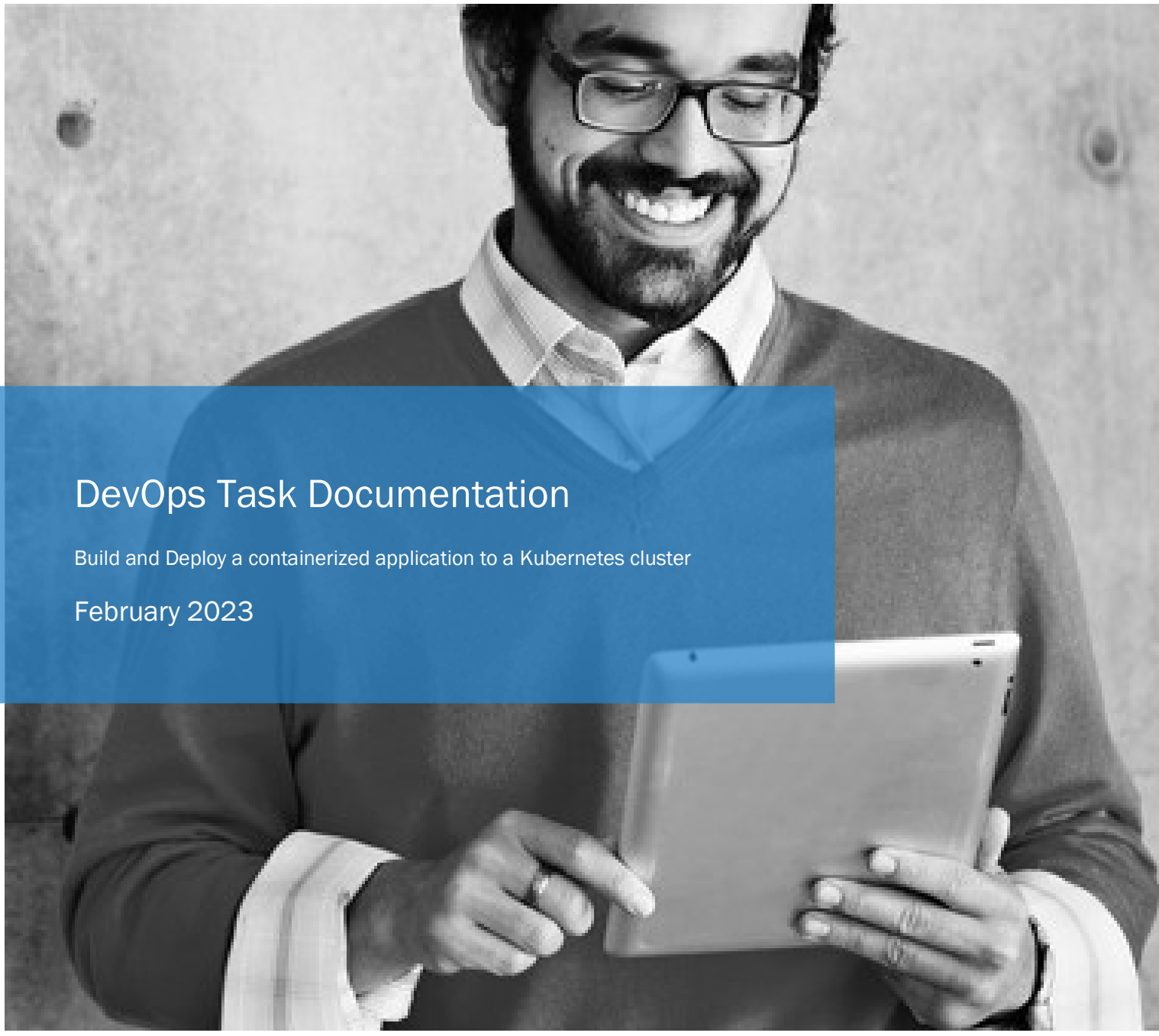


DevOps Task Documentation

Build and Deploy a containerized application to a Kubernetes cluster

February 2023

Table of Contents

Problem Statement

Task 1 - Dockerize the Application

The first task is to dockerise this application - as part of this task you will have to get the application to work with Docker and Docker Compose. You can expsoe the frontend using NGINX or HaProxy The React container should also perform npm build every time it is built. Hint/Optional - Create 3 separate containers. 1 for the backend, 2nd for the proxy and 3rd for the react frontend. The only strict requirement is that the application should spin up with docker-compose up --build command

Task 2 - Deploy on Cloud

Next step is to deploy this application on AWS At this point the application is already containerized, so you could deploy it to services which take an advantage of that fact, example ECS Fargate/EKS. The deliverable of this task is infrastructure as a code so we are interested in seeing the terraform/ansible files. We don't need to see the environment running but we will use your deliverable to try it out, so make sure it is easy for us to configure credentials etc.

Task 3 - Get it to work with Kubernetes

Next step is completely separate from step 2. Go back to the application you built in Task 1 and get it to work with Kubernetes. Separate out the two containers into separate pods, communicate between the two containers, add a load balancer (or equivalent), expose the final App over port 80 to the final user (and any other tertiary tasks you might need to do) Add all the deployments, services and volume (if any) yaml files in the repo. The only hard-requirement is to get the app to work with minikube.

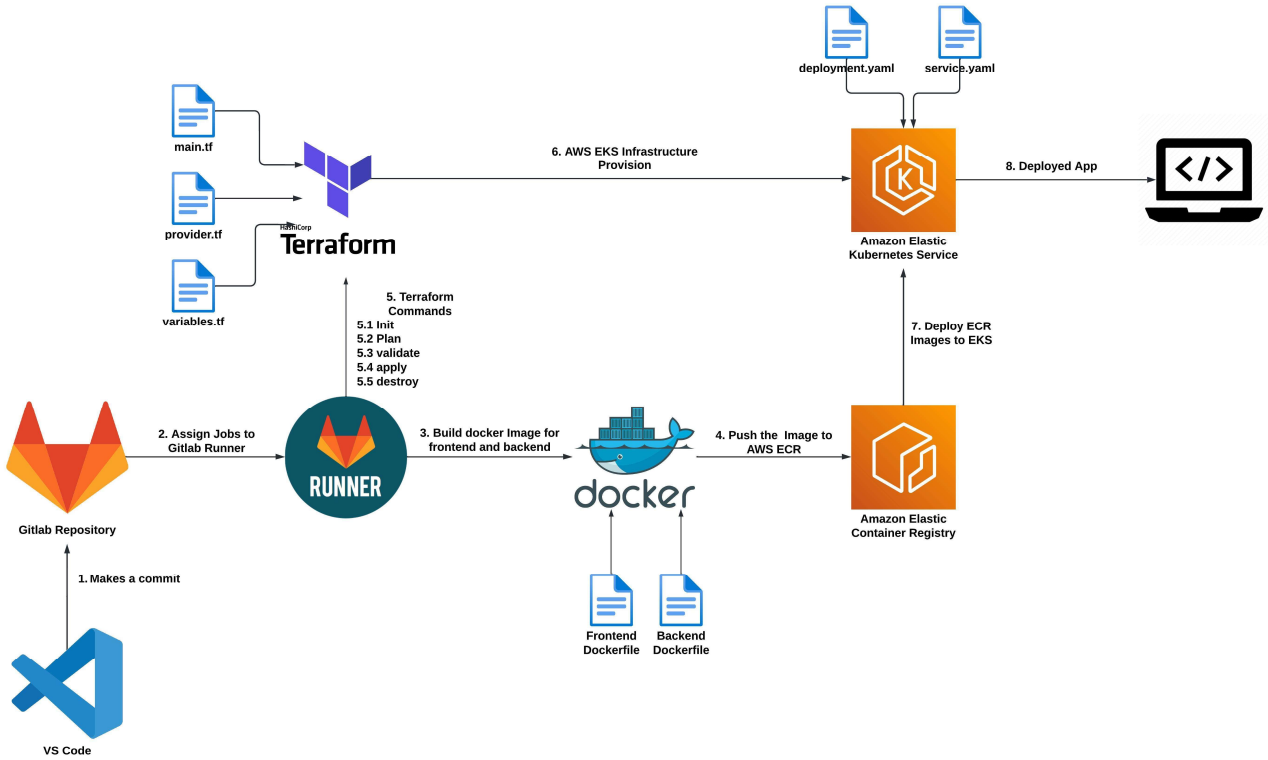
Tools and Technologies

1. VS Code: Visual Studio Code is a code editor redefined and optimized for building and debugging modern web and cloud applications.
2. GitLab CI/CD: GitLab CI/CD is the part of GitLab that we use for all of the continuous methods (Continuous Integration, Delivery, and Deployment). With GitLab CI/CD, we can test, build, and publish your software with no third-party application or integration needed. Initially we create a new project where we can store our codes and develop our pipeline. All the codes provided are initially uploaded to Gitlab Repository and then we create a .gitlab-ci.yml file which is used for creating CI/CD pipeline.
3. Terraform: Terraform is a popular infrastructure-as-code tool that allows us to automate the provisioning and management of infrastructure resources. Here we use terraform to provision AWS EKS and its dependencies.
4. AWS Cloud: Amazon Web Services (AWS) is a cloud solution provider with an on-demand delivery for cloud computing resources on the internet. Here we are going to use different services of aws such as
    1. Amazon Elastic Container Registry
    2. Elastic Kubernetes Service

3. Virtual private cloud
4. Identity and Access Management
5. AWS EC2

5. Docker: Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

Architecture



CI/CD Pipeline

For this project the CI/CD pipeline was build using GitLab. This GitLab CI/CD pipeline is designed to build and deploy a containerized application to a Kubernetes cluster. It consists of four stages, each with its own specific function:

Stage1: docker_build

The **docker_build** stage is responsible for building the Docker images for the frontend and backend services and pushing them to an **Amazon Elastic Container Registry (ECR).** The stage

consists of two jobs, **build_frontend** and **build_backend**, each of which builds the Docker image for the corresponding service, tags it with the ECR registry and image name, and then pushes it to the ECR repository.

The build_frontend and build_backend jobs use the **amazon/aws-cli** Docker image as the base image and have the **docker:dind** service running to provide access to the Docker daemon during the build process. Before the build process begins, the necessary dependencies such as the Docker and AWS CLI are installed, and the credentials are retrieved using the aws ecr get-login-password command.

Finally, the built Docker images are pushed to the ECR registry using the docker push command, with the ECR registry and image name provided through the environment variables $DOCKER_REGISTRY and $REGISTRY_NAME_FRONTEND or $REGISTRY_NAME_BACKEND.

Stage2: docker_compose

The docker_compose stage in this pipeline is responsible for creating a Docker Compose environment for the project. This environment is built using the images created in the docker_build stage, which creates two Docker images for the project's frontend and backend services, respectively.

The stage is defined to run after the docker_build stage, as it depends on the images created by that stage. The stage uses the amazon/aws-cli Docker image as the base image and uses docker:dind as a service for running Docker inside Docker.

Stage3: iac_terraform

The iac_terraform stage in this pipeline usesTerraform to manage the infrastructure as code (IAC) for the Kubernetes cluster.

The stage uses a lightweight Terraform Docker image to perform the following tasks:

- It installs the Terraform binary and verifies its version using the terraform --version command.
- It sets up the backend configuration for Terraform, which is configured to store the state file remotely using GitLab's API (using the export GITLAB_ACCESS_TOKEN command).
- It initializes Terraform by running terraform init with the necessary backend configuration options.
- It validates the Terraform configuration files using the terraform validate command.
- It generates an execution plan for the Terraform changes using the terraform plan command. This step shows what changes will be applied to the infrastructure and allows you to review them before applying them.
- After the plan, it will run the terraform apply which will provision the infrastructure in aws.

- Finally, it caches the Terraform initialization state files in GitLab's cache, so that subsequent pipeline runs can reuse the previously downloaded dependencies and speed up the pipeline.

By using Terraform, this pipeline automates the creation and management of the infrastructure for the Kubernetes cluster, making it more reliable and easier to scale.

Stage4: deploy_k8s

The deploy_k8s stage is responsible for deploying the application to a Kubernetes cluster. The stage includes the following tasks:

- It defines an image with the necessary tools, including the awscli and kubectl commands, and also sets up the Docker-in-Docker (DinD) service.
- Before running the actual deployment, the script performs some preparations, such as installing the awscli and configuring the Kubernetes cluster credentials using the aws eks update-kubeconfig and kubectl config use-context commands.
- The script then applies the Kubernetes deployment and service files for both the frontend and backend components using the kubectl apply command.
- Finally, the script verifies that the services are running using the kubectl get services command.

Also, the when: manual option means that this stage needs to be triggered manually and will not run automatically. This is to ensure that the deployment only occurs when intended and to prevent accidental deployments.

End-to-End Steps:

Configure GitLab:

Login in to GitLab. Create a new project in the GitLab. Create a new repository in the project. Clone the repository to VSCode. Clone and push the initial files provided in the repo (https://gitlab.com/redacre/test-project) to our repository.

Navigate to CI/CD and configure a shared runner for executing pipeline.

Task 1: Dockerize the application

I have created 2 Dockerfile, one for the react frontend and one for the python backend and also a nginx configuration file.

Dockerfile – frontend

This Dockerfile defines a multi-stage build for the web application.

- The first stage begins with a *node:14.16.0-alpine* base image and creates a new build stage named build.
- The **WORKDIR** command sets the working directory for subsequent commands to /app.
- The **COPY** command copies package.json to the working directory.
- The **RUN** command installs the dependencies defined in package.json.
- The **COPY . .** command copies the remaining application files to the working directory.
- Finally, the **RUN** command runs the build script defined in the package.json file.

- The second stage begins with an **nginx:alpine** base image and creates a new build stage without a name.
- The COPY command copies the nginx.conf file to the /etc/nginx/ directory.
- The COPY --from=build /app/build /usr/share/nginx/html command copies the built application files from the build stage to the default document root directory for Nginx.
- The EXPOSE command indicates that the container will listen on port 80.
- The CMD command specifies the command to run when the container is started, in this case, starting Nginx with the "nginx", "-g", "daemon off;" arguments.

Dockerfile – backend

This Dockerfile defines a Docker image for Python backend.

- The Dockerfile begins by using the python:3.8-slim-buster base image.
- The WORKDIR command sets the working directory for subsequent commands to /app.
- The COPY command copies the requirements.txt file to the working directory.
- The RUN command installs the Python packages specified in the requirements.txt file using pip.
- The next COPY command copies the application code to the working directory. This assumes that the application code is in the same directory as the Dockerfile.
- Finally, the CMD command specifies the command to run when the container is started, in this case, running the app.py script using the python command.

Nginx configuration file

This configuration sets up a basic web server that listens on port 80 and serves files from the /usr/share/nginx/html directory, with a default index file of index.html.

Docker-compose file

This configuration file for Docker Compose sets up three services: backend, frontend, and nginx.

- The backend service runs a program located in the ./api directory and listens on port 3000, which is also exposed to the host.
- The frontend service runs a program located in the ./sys-stats directory and exposes port 80 to other containers.
- The nginx service uses the nginx:alpine Docker image and runs an instance of the Nginx web server that listens on port 80, which is also exposed to the host.

The nginx service mounts a configuration file named nginx.conf located in the ./sys-stats directory on the host to configure the Nginx server. The nginx service depends on the frontend service to be started first.

These files are pushed to the GitLab repository and GitLab pipeline is configured to build the docker images. For runner we are using the **amazon/aws-cli** container image so that we can use the amazon cli services easily.

Since we need to dockerize and build our application, we need to install the docker on the amazon/aws-cli image using **amazon-linux-extras install docker** command and set the DOCKER_HOST variable to **tcp://docker:2375** .

Now we can build the image using docker build command and the image can be uploaded to ECR. Inorder to connect to ECR, we need to establish a connection to aws account. For authenticating using our GitLab pipeline, we need to configure **AWS_ACCESS_KEY_ID** and **AWS_SECRET_ACCESS_KEY.**

For that, we need to create an IAM user and generate credentials and pass that credentials to the GitLab pipeline through variables. Navigate to aws management console. Go to IAM and create a new user. In attach existing policies, select **AmazonEC2ContainerRegistryPowerUser** policy and create the user.

Now go to the Security credentials tab and Navigate to **Access keys** and create an Access Key. This will give an Access Key ID and Access Key Secret.

Now in GitLab, find project settings and go to CI/CD. Navigate to Variables and create a new variable name us **AWS_ACCESS_KEY_ID** and **AWS_SECRET_ACCESS_KEY** and add the ID and Secret Key there. Now we can use the amazon/aws-cli to authenticate to aws and push the image to ECR.

Task 2: Provisioning EKS infrastructure using Terraform

For Infrastructure provisioning using Terraform, I have created 2 files, main.tf and provider.tf.

provider.tf file

The code includes two main sections:

Provider Block: This block specifies the cloud provider being used and the region where the resources will be created. In our case, the provider is AWS (Amazon Web Services) and the region is "us-east-1".

Terraform Block: This block specifies the version of the AWS provider that is required and the backend configuration to be used. The required version of the AWS provider is "~> 3.0", which means that any version of the provider in the 3.x range can be used. The backend configuration is set to "http", which means that the Terraform state will be stored in a remote HTTP server i.e, the GitLab remote server.

main.tf file

This Terraform code provisions an Elastic Kubernetes Service (EKS) cluster on AWS with two private and two public subnets and creates the necessary infrastructure for the cluster to function. The code creates a Virtual Private Cloud (VPC), an internet gateway, a network address translation (NAT) gateway, four subnets (two private and two public), and two route tables (one for public and one for private). The code also creates an AWS Identity and Access Management (IAM) role for the EKS cluster.

The private subnets are used for worker nodes, and the public subnets are used for the EKS control plane. The NAT gateway allows the worker nodes to access the internet, and the IAM role allows the EKS control plane to manage the worker nodes.

These files are pushed to the GitLab repository and we need to define another stage in our gitlab yaml file for the provision of EKS using terraform. The job is defined to run in the "iac_terraform" stage, using the "**hashicorp/terraform:light**" Docker image as the runtime environment.

In the before_script section, it first displays the version of Terraform being used, then it exports a GitLab access token, and then it changes the working directory to the Terraform directory specified by the variable ${TF_DIR}. Next, it initializes Terraform with some specific backend configurations, such as the backend's address, lock method, unlock method, and authentication details. The created terraform state file is stored in the GitLab Terraform Infrastructure. Inorder to store the state file in GitLab we need to provide authentication to GitLab through gitlab pipeline.

For authentication I'm using GitLab username and PAT (Personal Access Token). Inorder to create a PAT token, navigate to edit profile and go to access tokens. Create a new PAT token with necessary scope (here I select all scopes) and create a new PAT token. Add the username and PAT token in the CI/CD variables as Username and TOKEN.

Now that we have setup the authentication with GitLab for storing the state file, now we need to setup the authentication to aws for infrastructure provisioning. For that we need to update the user which we created in the previous task with necessary policies. Here I have granted the Administrator access policy for the time being (Not recommended) for easy provisioning of EKS infrastructure.

Now that we can run our pipeline to provision the EKS infrastructure, the terraform will first validate, then run the terraform plan and finally everything looks good, it will run the terraform apply and provision the infrastructure.

Task 3: Deploy the app to Kubernetes

For the deployment of application and getting the service, I have created deployment.yaml and service.yaml file.

deployment.yaml file is used to define the desired state of the application that we want to run on a Kubernetes cluster. It contains information about the containers that need to be created, how many replicas of the containers should be running, the networking configuration, and other related details.

service.yaml file is used to create a network endpoint to expose the application running in the deployment. It provides a stable IP address and DNS name that can be used to access the application, even if the underlying pods or containers are updated, replaced or moved.

frontend-deployment.yaml

This Kubernetes deployment configuration file describes how to run the frontend application. The file specifies that there should be two replicas of the application running, with the name "frontend-deployment". The deployment will manage a set of pods based on a selector that matches the labels with the value "app: frontend".

The configuration defines a single container called "frontend" with the container image from ECR. The container is also configured to expose port 3000.

Also, the container is given an environment variable, "API_URL", which points to a backend service running on port 5000, with the hostname "backend-service". This allows the frontend application to communicate with the backend service.

backend-deployment.yaml

This Kubernetes deployment configuration file describes how to run the backend application. It tells Kubernetes to create and manage two replicas of the backend application. The deployment uses a container image from ECR.

The container runs the backend application and listens on port 5000. The deployment specifies a selector that matches the label app: backend, which is used to match this deployment with a corresponding Kubernetes service.

frontend-service.yaml

This is the configuration file for Kubernetes service called "frontend-service". The service is designed to route incoming network traffic to pods that have a label of "app: frontend".

The service listens for incoming traffic on port 80 using the TCP protocol, and then forwards that traffic to the target port 3000 on the pods.

This service is configured to be of type "NodePort", which means that it exposes the service on a static port on each node in the cluster. This makes the service accessible from outside the cluster by using the IP address of any node and the configured port number. The service can also be configured using a load balance and since this app is not for a production purpose, I use a NodePort to expose a static port.

backend-service.yaml

This is configuration file for Kubernetes service called "backend-service". The service is designed to route incoming network traffic to pods that have a label of "app: backend".

The service listens for incoming traffic on port 5000 using the TCP protocol, and then forwards that traffic to the target port 5000 on the pods.

This service is configured to be of type "ClusterIP", which means that it only exposes the service on a cluster-internal IP address. This type of service can be used to allow other pods in the cluster to access the backend pods through the service's internal IP address and port. However, this type of service is not accessible from outside the cluster, so it cannot be used to allow external traffic to access the backend pods.

These files are pushed to the GitLab repository and we need to define another stage in our gitlab yaml file for deploying the images to EKS using the AWS CLI, Docker, and kubectl.

The job is defined with a before_script section that installs Docker and the AWS CLI, and then configures kubectl to use an Amazon EKS cluster by updating the kubeconfig and setting the current context to the EKS cluster.

The script section applies Kubernetes deployment and service configuration files to deploy the frontend and backend services. Finally, it verifies that the services are running by getting a list of services using kubectl.

For authentication to aws and deploy the services to EKS, we need to create a new IAM user/ we can use the existing one and attach the necessary policies. Since we are already using Administrator policy attached IAM user we can deploy the images to EKS directly.