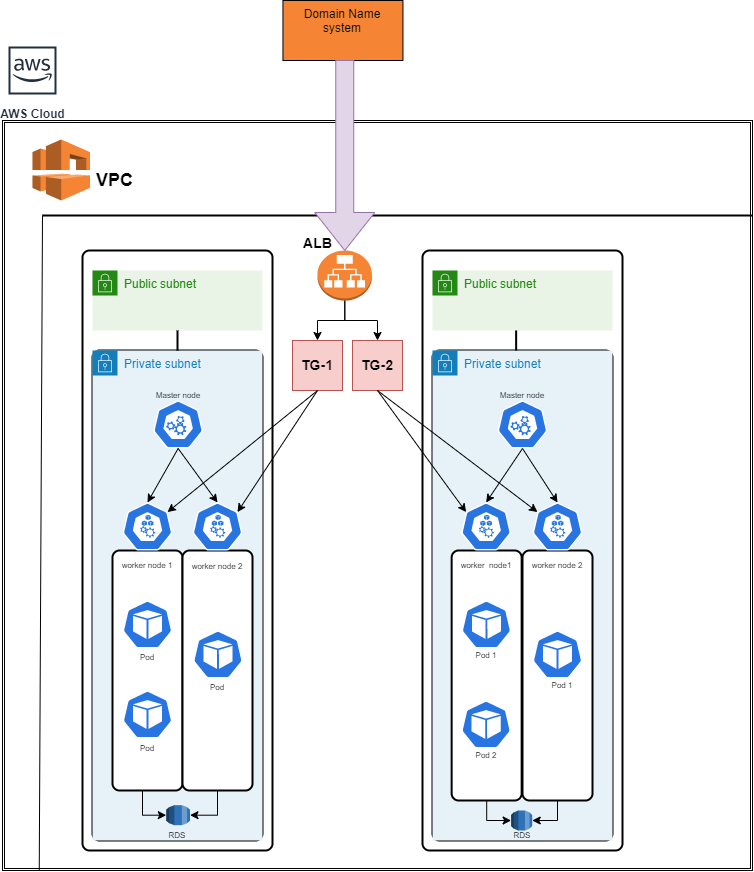
Github CI-CD FLOW

## **cicd flow github**

**Kubernetes Architecture**

****

**Create Kubernetes Cluster In Aws Use Kops Method.**

**PREREQUIREMENTS**

1. linux machine (ubuntu)

2. AWS account

3. kops binary (kubernetes cluster initiate)

4. kubectl binary (kubernetes deployments)

**KOPS BINARY SETUP**

# curl -Lo kops https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag\_name | cut -d '"' -f 4)/kops-linux-amd64

# chmod +x ./kops

# sudo mv ./kops /usr/local/bin/

**KUBECTL BINARY SETUP**

# curl -Lo kubectl https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl

# chmod +x ./kubectl

# sudo mv ./kubectl /usr/local/bin/kubectl

**SETUP IAM USER** (kops access aws resources)

kindly configure aws-cli packages in your linux machines.

In order to build clusters within AWS we'll create a dedicated IAM user for kops. This user requires API credentials in order to use kops. Create the user, and credentials, using the AWS console.

require the following IAM permissions to function properly:

1. AmazonEC2FullAccess

2. AmazonRoute53FullAccess

3. AmazonS3FullAccess

4. IAMFullAccess

5. AmazonVPCFullAccess

**cluster type:**

1. single node cluster

2. Multinode cluster

**DNS kubernetes type:**

1. single node cluter means not need any dns entry its take default machine dns record. The only requirement to trigger this is to have the cluster name end with .k8s.local.
2. Multi node cluster means it act loadbalancer type. Loadbalancer means it's need single entry point to rechaed this cluster master nodes.we need to prepare somewhere to build the required DNS records. (use route 53 DNS record)

**Cluster State storage**

In order to store the state of your cluster, and the representation of your cluster, we need to create a dedicated S3 bucket for kops to use. This bucket will become the source of truth for our cluster configuration.

# aws s3api create-bucket --bucket awsdamo --region us-east-1

# aws s3api put-bucket-versioning --bucket awsdamo --versioning-configuration Status=Enabled

**nodes authentication methods**

create sshkey for machines

# ssh-keygen

**Node.js application**

**Create the Node.js app**

First, create a new directory where all the files would live. In this directory create a package.json file that describes your app and its dependencies:

{

"name": "docker\_web\_app",

"version": "1.0.0",

"description": "Node.js on Docker",

"author": "First Last <first.last@example.com>",

"main": "server.js",

"scripts": {

"start": "node server.js"

},

"dependencies": {

"express": "^4.16.1"

}

}

Package.json

Create server.js

'use strict';

const express = require('express');

// Constants

const PORT = 8080;

const HOST = '0.0.0.0';

// App

const app = express();

app.get('/', (req, res) => {

res.send('Hello World');

});

app.listen(PORT, HOST);

console.log(`Running on [http://${HOST}:${PORT](about:blank)}`);

Server.js

**STEPS TO RUN THE NODE JS APPLICATION**

STEP 1: npm install

STEP 2 : node server.js

Creating a Dockerfile

FROM node:16

# Create app directory

WORKDIR /usr/src/app

# Install app dependencies

# A wildcard is used to ensure both package.json AND package-lock.json are copied

# where available (npm@5+)

COPY package\*.json ./

RUN npm install

# If you are building your code for production

# RUN npm ci --only=production

# Bundle app source

COPY . .

EXPOSE 8080

CMD [ "node", "server.js" ]

**Save the file like Dockerfile**

BUILD THE DOCKER IMAGE

docker build -t nodejs/app:demo .

docker push nodejs/app:demo

* Step 1: Setup AWS user, initial CLI tools, ECR

$ ekctl create cluster

$ eksctl get clusters

$ aws eks list-clusters

* Step 2: Setup our GitHub Actions Workflow repo

Line 16 AWS\_DEFAULT\_REGION: eu-west-1

Line 17 AWS\_DEFAULT\_OUTPUT: json

Line 18 AWS\_ACCOUNT\_ID: ${{ secrets.AWS\_ACCOUNT\_ID }}

Line 19 AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

Line 20 AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

* Step 3: Configure secrets

Go to git hub setting setting up the environment variables

* Step 4: App development workflow

Developers push changes to the app directory.

GitHub Actions then builds and pushes container images t[o Elastic Container Repository (ECR)](https://aws.amazon.com/ecr/), as specified by a workflow.

Flux notices a new image in ECR and deploys it to EKS by modifying the deployment manifest.

on:

push:

branches: [ main ]

pull\_request:

branches: [ main ]

name: AWS ECR push

jobs:

Deploy:

name: Deploy

runs-on: ubuntu-latest

steps:

- name: Install Octopus CLI

uses: OctopusDeploy/install-octopus-cli-action@v1.1.1

with:

version: latest

- name: Checkout

uses: actions/checkout@v2

- name: Configure AWS credentials

uses: aws-actions/configure-aws-credentials@v1

with:

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-region: us-east-2

- name: Login to Amazon ECR

id: login-ecr

uses: aws-actions/amazon-ecr-login@v1

- name: Build, tag, and push the image to Amazon ECR

id: build-image

env:

ECR\_REGISTRY: ${{ steps.login-ecr.outputs.registry }}

ECR\_REPOSITORY: ${{ secrets.REPO\_NAME }}

IMAGE\_TAG: "latest"

run: |

# Build a docker container and push it to ECR

docker build -t $ECR\_REGISTRY/$ECR\_REPOSITORY:$IMAGE\_TAG .

echo "Pushing image to ECR..."

docker push $ECR\_REGISTRY/$ECR\_REPOSITORY:$IMAGE\_TAG

echo "::set-output name=image::$ECR\_REGISTRY/$ECR\_REPOSITORY:$IMAGE\_TAG"