✦ Member-only story

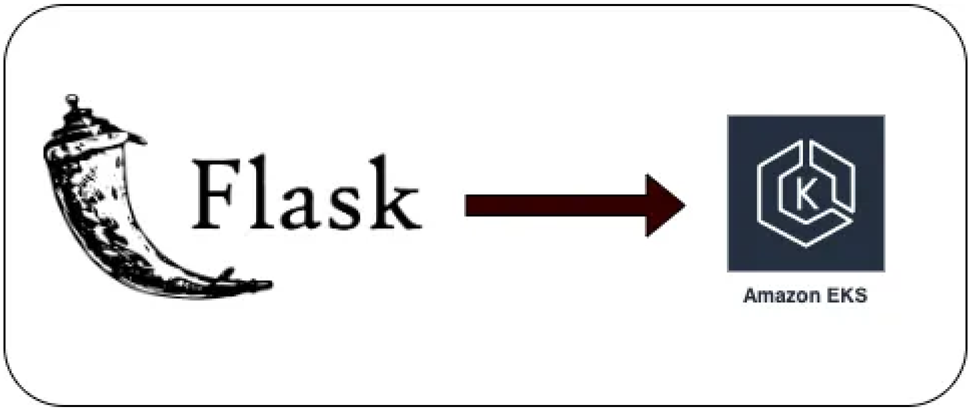
Python App on EKS — From Scratch to Hosting

Vinod Kisanagaram · Follow
Published in Towards AWS
10 min read · Feb 28

Listen Share More

In the below article, we will explore setting up Python Flask application and then deploying it to Minikube and then to EKS. We will try to achieve below

- Accessing service via localhost
- Accessing service via minikube
- Deploying to EKS
- Accessing via external IP
- Accessing via Application Load Balancer
- Finally, accessing via Route53



Python Flask on EKS

> *Disclaimer — opinions are my own and content is not vetted/reviewed/approved by my employer*

Step 1 — Creating Python Flask App

For developing our Python application, we choose Python Flask.

Python Flask is a micro web framework for building web applications with Python. It is designed to be lightweight, flexible, and easy to use, making it a popular choice for building web APIs and small to medium-sized web applications

Our app.py code (reference code here) simply tries displaying a *Hello World* when we call the domain name or root url (/). Below is the content of app.py

```
from flask import Flask, jsonify, request

app = Flask(__name__)


@app.route('/name', methods=['GET'])
def name():
    if (request.method == 'GET'):
        data = {"data": "Vinod here!!"}
        return jsonify(data)

@app.route('/', methods=['GET'])
def index():
    if (request.method == 'GET'):
        data = {"data": "Hello World!"}
        return jsonify(data)
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

In order run this code in local, you would ned python & pip installed. You can refer to below steps

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
python3 get-pip.py
pip3 --version
alias pip=pip3 >> ~/.bash_aliases
pip --version
```

For running Flask App in local

```
python3 -m venv venv
. venv/bin/activate
pip install Flask
export FLASK_APP=app.py
flask run
```

Our app is now live and you can check that with

```
curl http://127.0.0.1:5000/name
```

Step 2 — Run in minikube

You need to have docker installed and running — check steps at the Docker site

Install minikube

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-a
sudo install minikube-darwin-amd64 /usr/local/bin/minikube
```

If you are having Mac M1 chip, try below

```
curl -LO https://github.com/kubernetes/minikube/releases/download/v1.29.0/minikube
sudo install minikube-darwin-arm64 /usr/local/bin/minikube
```

Start minikube

```
minikube start
```

To work with minikube, we will push our images to a local repository. For that, let’s do below

```
docker run -d -p 5001:5000 --restart=always --name registry registry:2
```

Now our local repository is running on localhost:*5001*

Point minikube to the local registry

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /python-docker

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

and we run below to build the image

```
docker build --tag eksdemo:0.0.1 .
```

We will tag this image for local repository

```
docker tag eksdemo:0.0.1 localhost:5001/eksdemo
```

And push this image to the local repository

```
docker push localhost:5001/eksdemo
```

Check if the image is in local repository

```
docker images localhost:5001/eksdemo
```

Now, we will create a deployment in minikube

```
kubectl apply -f deployment.yaml
```

We can check pod status

```
kubectl get all
```

For checking whether our app is actually from minikube, let’s expose the port as 8083

```
kubectl port-forward svc/demo 8083:8080
```

You can see the minikube based app running now on localhost:8083

```
curl http://localhost:8083/name
```

So far, we built a simple Python Flask app and we deployed it onto minikube. This is an easy way to test k8s things in the local environment before we actually deploy in AWS

Step 3 — Become a member of Medium :)

Skip this step if you already are a Medium member.

Others — *you may want to revisit this article again* as all the steps may not be completed in your AWS account at *one-go* and there are only a *limited* number of free articles in Medium.

If *you decide* to become a member, follow this referral link to setup.

Step 4— Push the image to ECR

For demo purposes, create a user with admin access on AWS IAM having programmatic access and configure your terminal to use this user

```
aws configure
aws sts get-caller-identity #check configured user
```

Since we are working on EKS, we need to use image artifactory hosted somewhere. For simplicity, we will leverage ECR and create a repository there to host our demo application images.

To create a repo in ECR

```
aws ecr create-repository \
--repository-name demo-repo \
--image-tag-mutability IMMUTABLE \
--image-scanning-configuration scanOnPush=true
```

Now we need to push our locally built image to ECR (in an ideal world, we will be using CI/CD tools to build, scan and push images to ECR)

To do that, we need to get temp token from ECR

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --pass
```

After we get the token, next is to build an image for this ECR

```
docker tag eksdemo:0.0.1 777258879183.dkr.ecr.us-east-1.amazonaws.com/demo-repo:la
```

Push image to ECR

```
docker push 777258879183.dkr.ecr.us-east-1.amazonaws.com/demo-repo:latest
```

You can confirm that push is a success by logging into AWS Console and choose ECR service

Quick recap here — we built our app image, tested on minikube, created an ECR repo, tagged our image, and pushed it to the repo

Now it's EKS time

Step 5— Working with EKS

To work with EKS, we will use the AWS EKS CLI tool "eksctl". To install

```
brew tap weaveworks/tap
brew install weaveworks/tap/eksctl
eksctl version
```

It's time to create an EKS cluster. Hope you have the repo checked out. Edit cluster.yaml to point to your VPC availability zones and subnets

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: my-cluster
  region: us-east-1

vpc:
  subnets:
    private:
      us-east-1a: { id: subnet-d026d49a-replace-me }
      us-east-1b: { id: subnet-fedb3ffg-replace-me }
      us-east-1c: { id: subnet-b50c23eb-replace-me }
```

```
nodeGroups:
  - name: ng-1-workers
    labels: { role: workers }
    instanceType: t3.small
    desiredCapacity: 2
    privateNetworking: true
  - name: ng-2-builders
    labels: { role: builders }
    instanceType: t3.small
    desiredCapacity: 2
    privateNetworking: true
    iam:
      withAddonPolicies:
        imageBuilder: true
```

When ready with your changes, create a cluster (this will take roughly 15–20 mins) and in cloud formation console — you will see 3 stacks getting created : one for the cluster and 2 for node groups

```
eksctl create cluster -f cluster.yaml
```

To check cluster creation, you can run below

```
kubectl get svc
kubectl get pods --all-namespaces -o wide
```

You should see value for *CLUSTER-IP* and pods should be in *Running* status — if not, revisit your cluster.yaml file and see if you have configured subnet properly

After the cluster is created, it's time to deploy our image — replace with your image repo

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deployment
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: eksdemo
  template:
    metadata:
      labels:
        app: eksdemo
    spec:
      containers:
        - name: back-end
          image: 777258879183.dkr.ecr.us-east-1.amazonaws.com/demo-repo:latest
          ports:
            - containerPort: 8080
```

Apply

```
kubectl apply -f eks-deployment.yaml
```

Check

```
kubectl get deployments
```

Now we need to expose our deployments to other members via the NodePort service. If you notice comments, nodePort is used by external members (non-cluster resources), the port is used by cluster resources and targetPort is where our app (container) is currently running

```
apiVersion: v1
kind: Service
metadata:
  name: demo-service
spec:
  type: NodePort
  selector:
    app: eksdemo
  ports:
    - nodePort: 31479 #external traffic
      port: 8081 #port of this service. Cluster members talk via this port
      targetPort: 8080 #where container is actually running
```

Apply

```
kubectl apply -f eks-service.yaml
```

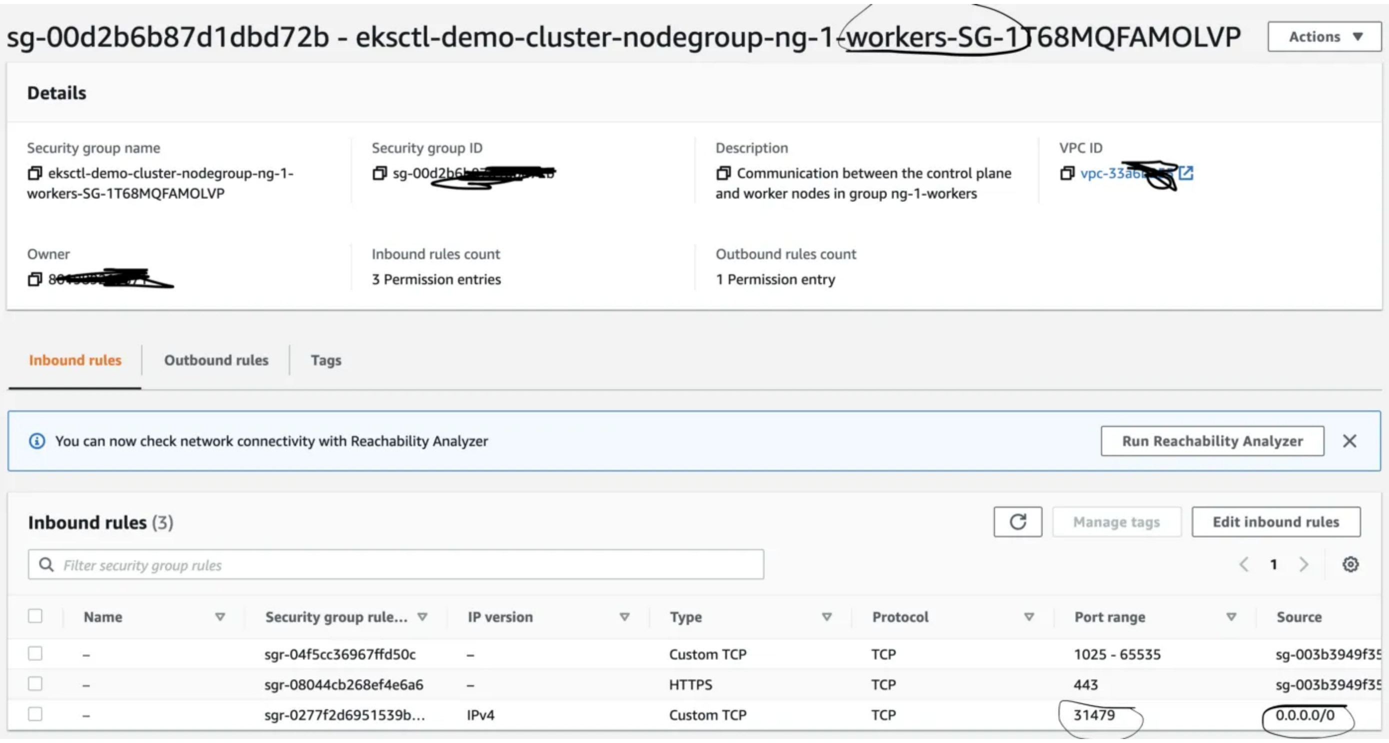
Check

```
kubectl get pods -o wide
kubectl get nodes -o wide
```

Gather the externally exposed IP address

Step 6— Accessing via external IP

We need to modify the worker node security groups to allow traffic from the outside the world on port 31479



sg-00d2b6b87d1dbd72b - eksctl-demo-cluster-nodegroup-ng-1-workers-SG-1T68MQFAMOLVP

Details

Security group name eksctl-demo-cluster-nodegroup-ng-1-workers-SG-1T68MQFAMOLVP	Security group ID sg-00d2b[redacted]	Description Communication between the control plane and worker nodes in group ng-1-workers	VPC ID vpc-33a[redacted]
Owner 80[redacted]	Inbound rules count 3 Permission entries	Outbound rules count 1 Permission entry	

You can now check network connectivity with Reachability Analyzer

Inbound rules (3)

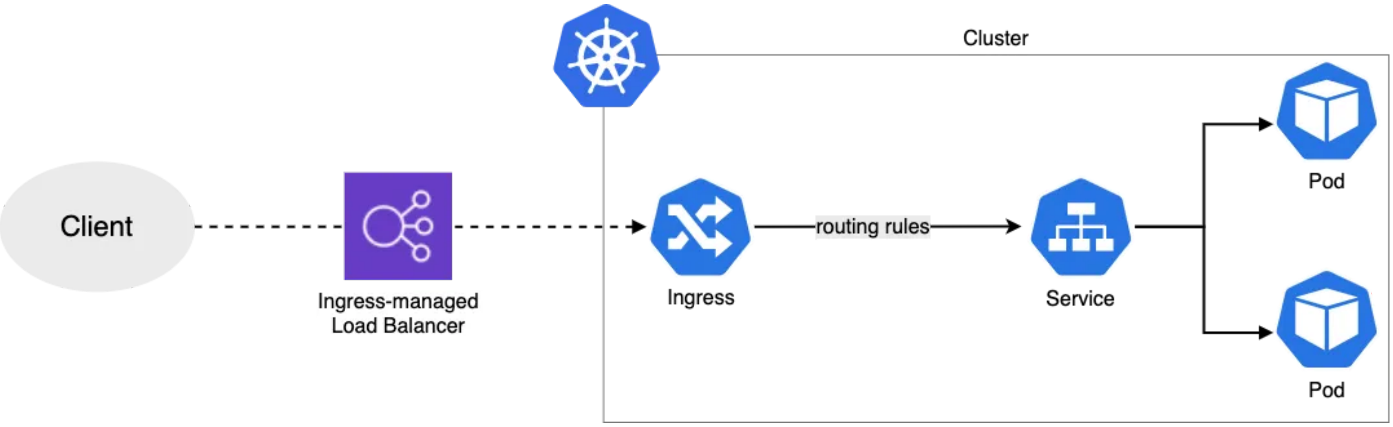
Filter security group rules

□	Name	Security group rule...	IP version	Type	Protocol	Port range	Source
□	–	sgr-04f5cc36967ffd50c	–	Custom TCP	TCP	1025 - 65535	sg-003b3949f35
□	–	sgr-08044cb268ef4e6a6	–	HTTPS	TCP	443	sg-003b3949f35
□	–	sgr-0277f2d6951539b...	IPv4	Custom TCP	TCP	31479	0.0.0.0/0

Once security group changes are done, we can now check that our app is running in EKS on

```
http://18.206.95.241:31479/name
http://3.93.200.100:31479/name
```

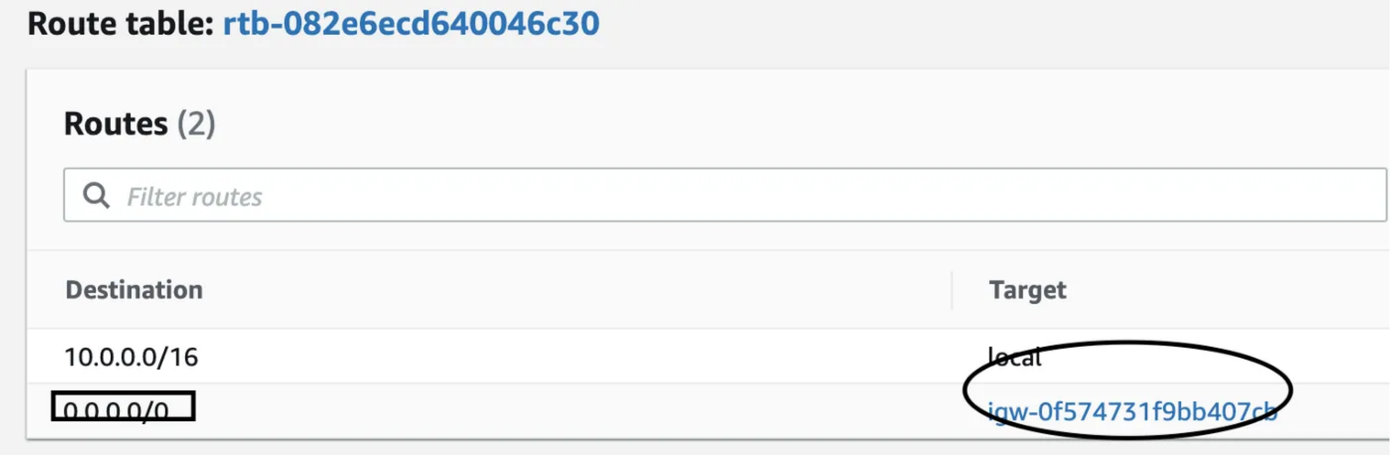
Step 7— Accessing via Application Load Balancer



In order to setup ALB, we need to tag few our VPC subnets with below so that ALB knows which subnets to choose for provisioning

```
kubernetes.io/cluster/<your cluster name here>: shared
```

Out of these subnets, choose min 2 subnets that are externally facing — those subnets that have route table mapped to the internet gateway



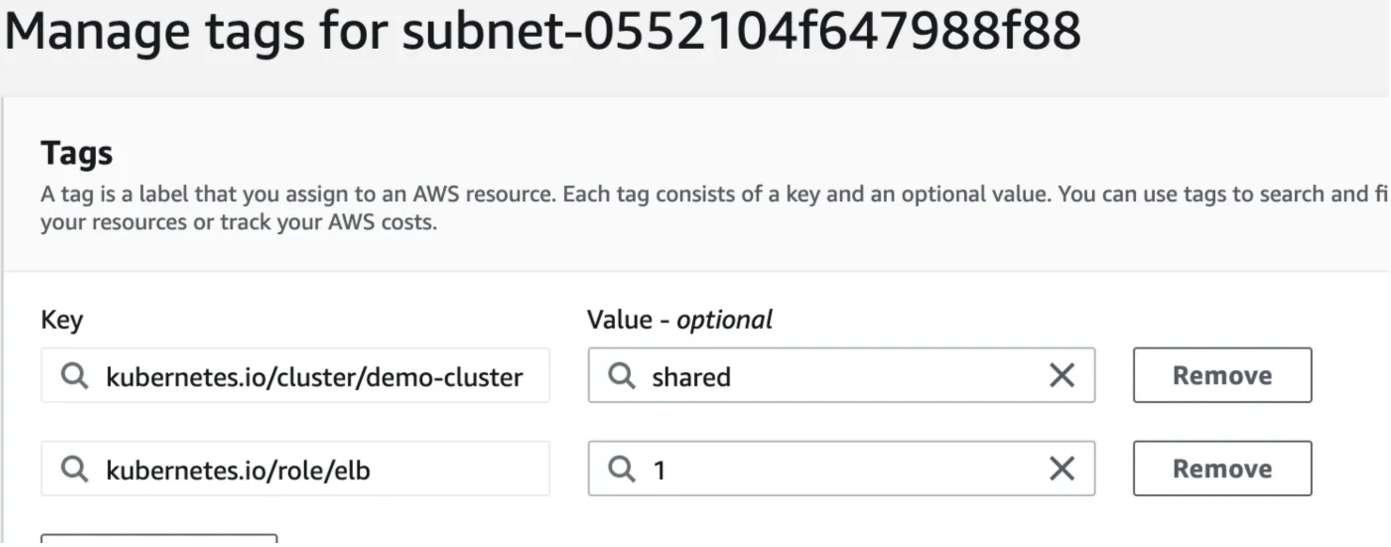
Route table: rtb-082e6ecd640046c30

Routes (2)

Destination	Target
10.0.0.0/16	local
0.0.0.0/0	igw-0f574731f9bb407cb

and tag them

```
kubernetes.io/role/elb: 1
```



Manage tags for subnet-0552104f647988f88

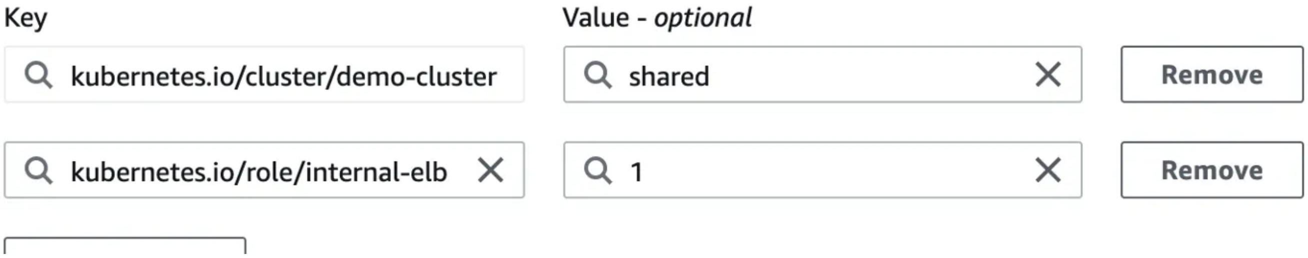
Tags

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and fi your resources or track your AWS costs.

Key	Value - *optional*
kubernetes.io/cluster/demo-cluster	shared
kubernetes.io/role/elb	1

and for the third subnet — this need not be internet-facing — tag as

```
kubernetes.io/role/internal-elb: 1
```



Key	Value - *optional*
kubernetes.io/cluster/demo-cluster	shared
kubernetes.io/role/internal-elb	1

With the above, we have let ALB pick subnets for provisioning

Create Ingress Controller IAM policy

```
aws iam create-policy \
--policy-name ALBIngressControllerIAMPolicy \
--policy-document file://iam-policy.json
```

Create Cluster Role, Service Account and bind this role to the service account

```
kubectl apply -f rbac-role-alb-ingress-controller.yaml
```

We need to associate OIDC provider to our cluster

```
eksctl utils associate-iam-oidc-provider \
--region us-east-1 \
--cluster demo-cluster \
--approve
```

Edit eks-ingress-trust-iam-policy.json with OIDC URL value — you can find that in IAM > Identity Providers

```
{
    "Version":"2012-10-17",
    "Statement":[
       {
          "Effect":"Allow",
          "Principal":{
             "Federated":"arn:aws:iam::777258879182:oidc-provider/<OIDC URL>"
          },
          "Action":"sts:AssumeRoleWithWebIdentity",
          "Condition":{
             "StringEquals":{
                "<OIDC URL>:sub":"system:serviceaccount:kube-system:alb-ingress-co
             }
          }
       }
    ]
 }
```

and attach to a new role

```
aws iam create-role --role-name eks-alb-ingress-controller --assume-role-policy-do
```

Attach Ingress Controller IAM policy

```
aws iam attach-role-policy --role-name eks-alb-ingress-controller --policy-arn=arn
```

We will now annotate service account with this role

```
kubectl annotate serviceaccount -n kube-system alb-ingress-controller \
eks.amazonaws.com/role-arn=arn:aws:iam::777258879183:role/eks-alb-ingress-controll
```

Edit eks-alb-ingress-controller.yaml with your cluster name

```
# Application Load Balancer (ALB) Ingress Controller Deployment Manifest.
# This manifest details sensible defaults for deploying an ALB Ingress Controller.
# GitHub: https://github.com/kubernetes-sigs/aws-alb-ingress-controller
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: alb-ingress-controller
  name: alb-ingress-controller
  # Namespace the ALB Ingress Controller should run in. Does not impact which
  # namespaces it's able to resolve ingress resource for. For limiting ingress
  # namespace scope, see --watch-namespace.
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: alb-ingress-controller
  template:
    metadata:
      labels:
        app.kubernetes.io/name: alb-ingress-controller
    spec:
      containers:
        - name: alb-ingress-controller
          args:
            # Limit the namespace where this ALB Ingress Controller deployment wil
            # resolve ingress resources. If left commented, all namespaces are use
            # - --watch-namespace=your-k8s-namespace

            # Setting the ingress-class flag below ensures that only ingress resou
            # annotation kubernetes.io/ingress.class: "alb" are respected by the c
            # choose any class you'd like for this controller to respect.
            - --ingress-class=alb

            # REQUIRED
            # Name of your cluster. Used when naming resources created
            # by the ALB Ingress Controller, providing distinction between
            # clusters.
            - --cluster-name=demo-cluster

            # AWS VPC ID this ingress controller will use to create AWS resources.
            # If unspecified, it will be discovered from ec2metadata.
            # - --aws-vpc-id=vpc-xxxxxx

            # AWS region this ingress controller will operate in.
            # If unspecified, it will be discovered from ec2metadata.
            # List of regions: http://docs.aws.amazon.com/general/latest/gr/rande.
            # - --aws-region=us-west-1

            # Enables logging on all outbound requests sent to the AWS API.
            # If logging is desired, set to true.
            # - --aws-api-debug

            # Maximum number of times to retry the aws calls.
            # defaults to 10.
            # - --aws-max-retries=10
          env:
            # AWS key id for authenticating with the AWS API.
            # This is only here for examples. It's recommended you instead use
            # a project like kube2iam for granting access.
            # - name: AWS_ACCESS_KEY_ID
            #   value: KEYVALUE

            # AWS key secret for authenticating with the AWS API.
            # This is only here for examples. It's recommended you instead use
            # a project like kube2iam for granting access.
            # - name: AWS_SECRET_ACCESS_KEY
            #   value: SECRETVALUE
          # Repository location of the ALB Ingress Controller.
          image: docker.io/amazon/aws-alb-ingress-controller:v1.1.8
      serviceAccountName: alb-ingress-controller
```

Apply

```
kubectl apply -f  eks-alb-ingress-controller.yaml
```

Our ingress controller is now ready.

Let’s create ingress resource for our cluster

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
  labels:
    app: demo-ingress
spec:
  rules:
    - http:
        paths:
          - path: /*
            backend:
              serviceName: demo-service
              servicePort: 8081
```

Apply

```
kubectl apply -f eks-ingress.yaml
```

We can check the load balancer details by running below

```
kubectl get ingress/demo-ingress -n default
```

Now we can check if the application is live

```
curl http://81495355-default-demoingre-d07d-220047569.us-east-1.elb.amazonaws.com/
```

Hurrah!! we are able to hit our app inside EKS from the internet via our ALB

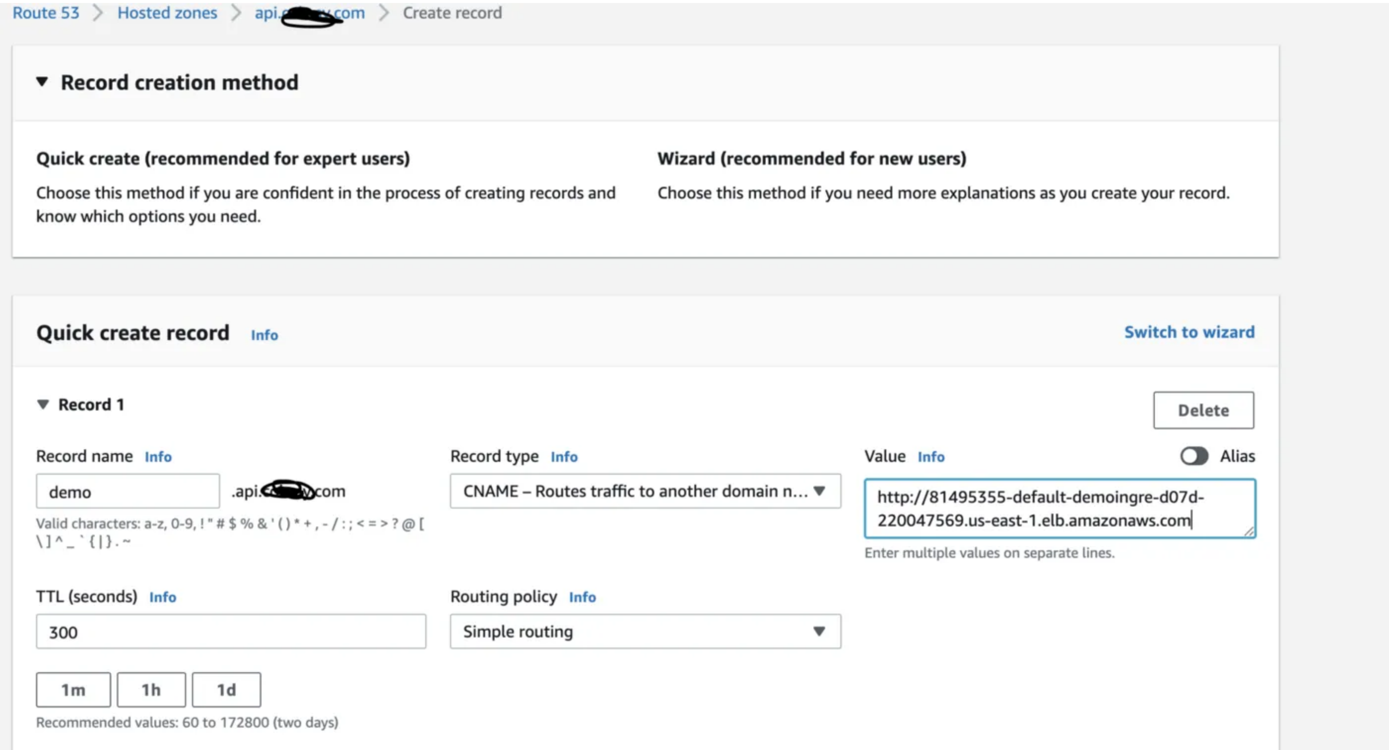
You can check ingress controller logs at

```
kubectl logs -n kube-system deployment.apps/alb-ingress-controller
```

Step 8— Accessing via Route53

Assuming you have purchased a Domain Name from Route53 or transferred the domain from another DNS provider to Route 53

we need to create a Hosted Zone CNAME record for our ALB



Route 53 > Hosted zones > api.[redacted]com > Create record

▼ Record creation method

Quick create (recommended for expert users)

Choose this method if you are confident in the process of creating records and know which options you need.

Wizard (recommended for new users)

Choose this method if you need more explanations as you create your record.

Quick create record Info — Switch to wizard

▼ Record 1 — Delete

Record name Info: demo .api.[redacted]com

Valid characters: a-z, 0-9, ! " # $ % & ' ( ) * + , - / : ; < = > ? @ [\ ] ^ _ ` { | } . ~

Record type Info: CNAME – Routes traffic to another domain n...

Value Info: http://81495355-default-demoingre-d07d-220047569.us-east-1.elb.amazonaws.com — Alias

Enter multiple values on separate lines.

TTL (seconds) Info: 300

1m 1h 1d

Recommended values: 60 to 172800 (two days)

Routing policy Info: Simple routing

You can configure more fine-grain details here — like Routing policy/TTL, etc

In the above I created a CNAME for ELB under demo.api.mydomain.com — once this creation is successful, we can check that our application is available worldwide behind our domain

```
curl http://demo.api.mydomain.com/name
```

That’s it

Full source code is available at

GitHub - kvr2277/python-flask-eks

github.com

You can enhance this with authentication/authorization, logging, monitoring, etc.

Python For Aws · Python · AWS · Kubernetes · Containers

Follow

Written by Vinod Kisanagaram

96 Followers · Writer for Towards AWS

Solutions Architect @ AWS

More from Vinod Kisanagaram and Towards AWS

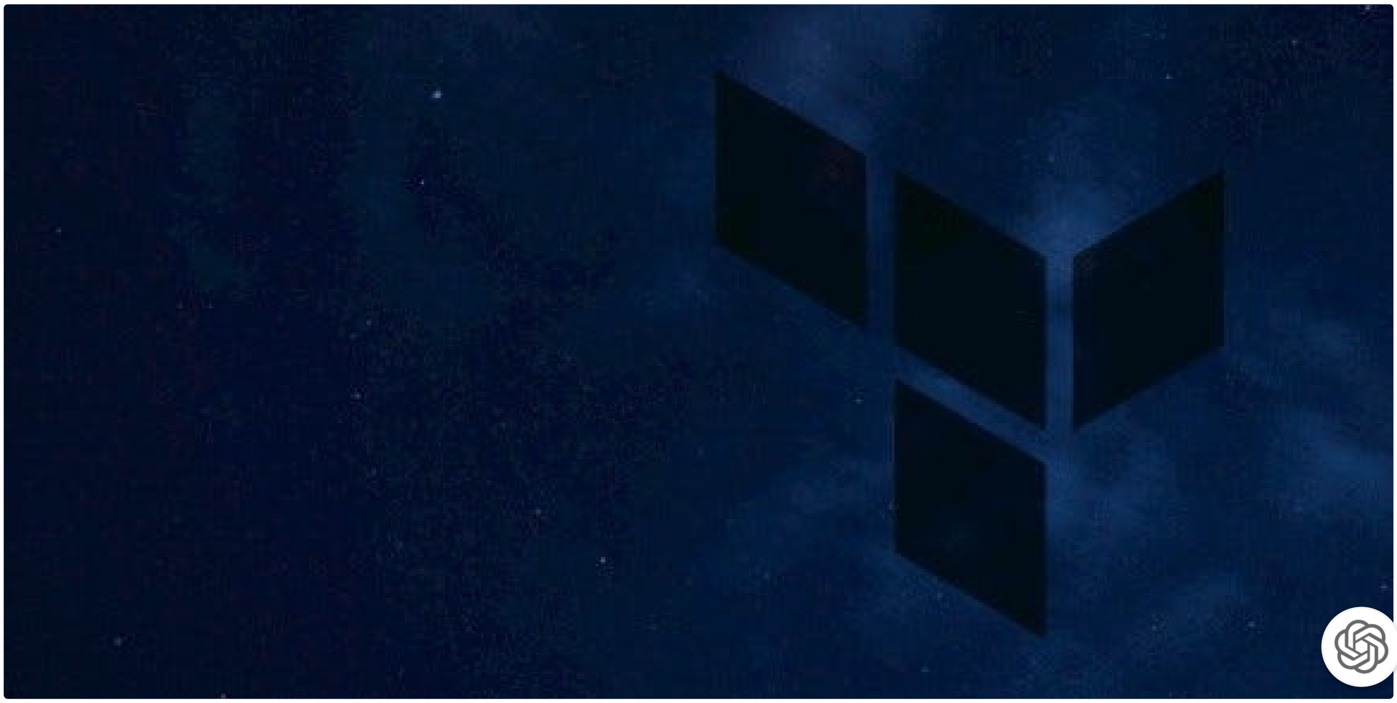
Vinod Kisanagaram

DynamoDB Batch Write—NodeJS

As per official AWS documentation, below are the limits

✦ · 1 min read · May 7, 2020

5



Melissa Gibson in Towards AWS

Building a Production Environment with Terraform and AWS ECS

The goal of this project is to provide a comprehensive example of setting up Terraform for your production environment requirements. I...

✦ · 11 min read · Jun 18

124 1

Shomarri Romell Diaz in Towards AWS

Terraform: Deploy A Two-Tier Architecture With AWS. Configuring a CI/CD Pipeline on Terraform Cloud

Configuring a CI/CD Pipeline on Terraform Cloud

✦ · 17 min read · May 11

54 1

Vinod Kisanagaram in Towards AWS

Serverless GraphQL with AppSync — Java and NodeJS Clients

Before we jump into building a GraphQL solution — here are some important notes about why use GraphQL

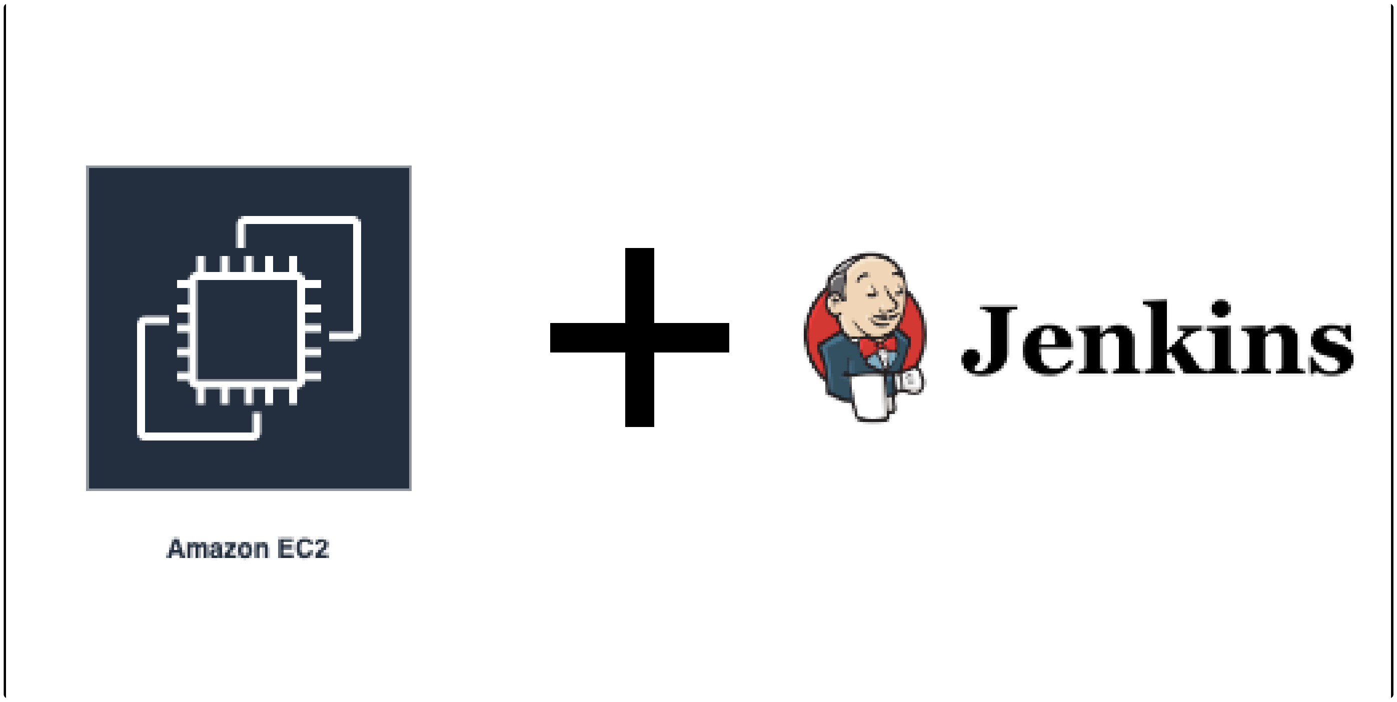
· 5 min read · Jan 1

61

See all from Vinod Kisanagaram

See all from Towards AWS

Recommended from Medium



George Baidoo Jr.

Install Jenkins on AWS EC2 Instance(Ubuntu)

Hello all and welcome to another project. In this project I will be walking through the steps on how to launch and AWS EC2 instance and…

✦ · 5 min read · Mar 24

28 1

python
Beautiful is better than ugly.
Simple is better than complex.
Flat is better than nested.
Complex is better than complicated.
Readability counts.
Although practicality beats purity.
Errors should never pass silently.
Now is better than never.
Although never is often better than *right* now.
Package AWS Lambda Layers for Python 🐍

Dmit in Python in Plain English

Package AWS Lambda Layers for Python 🐍

AWS Lambda Layer is a distribution mechanism for libraries, custom runtimes, and other function dependencies software developers or...

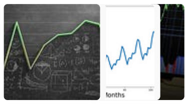
✦ · 3 min read · Mar 10

52

Lists

Coding & Development

11 stories · 55 saves

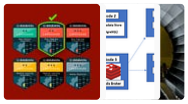


Predictive Modeling w/ Python

18 stories · 135 saves

Practical Guides to Machine Learning

10 stories · 148 saves



New_Reading_List

174 stories · 27 saves

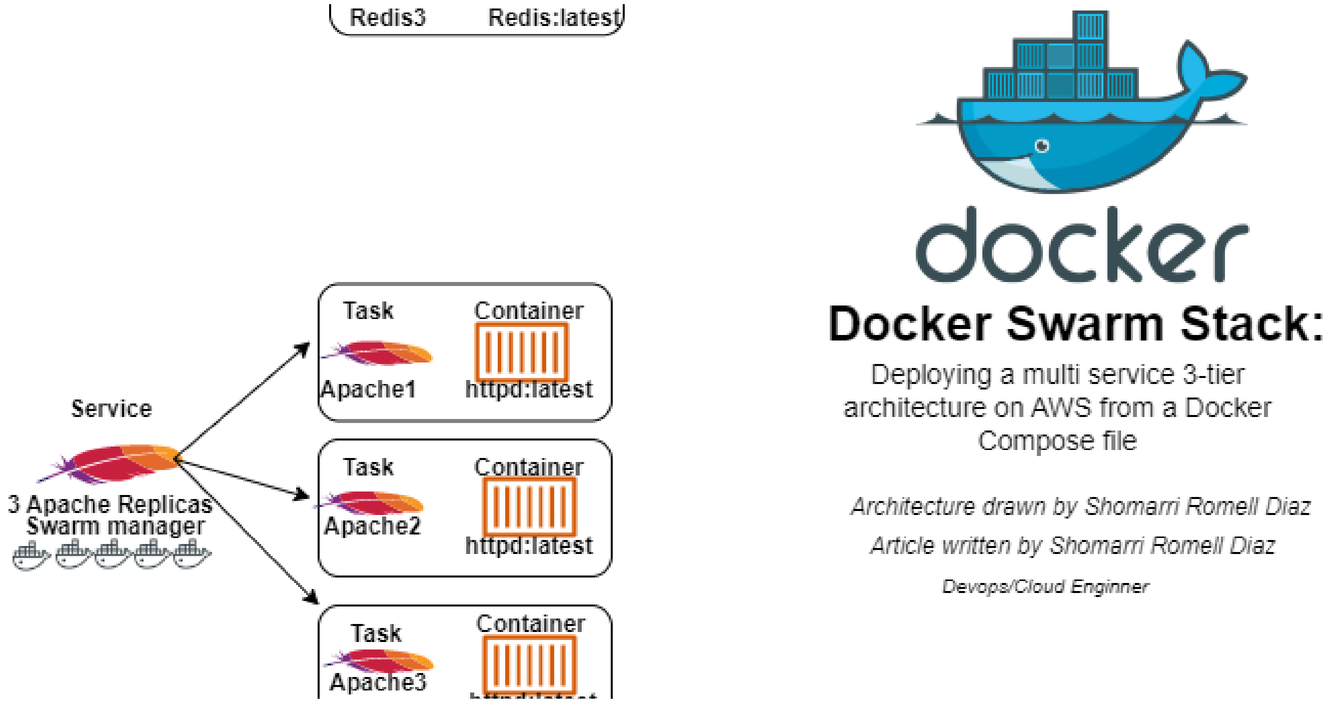
headintheclouds in AWS Tip

Full Stack DevOps Environment Setup Bash Script for Ubuntu, CentOS and MacOS

Introduction

✦ · 8 min read · Jan 24

159 7



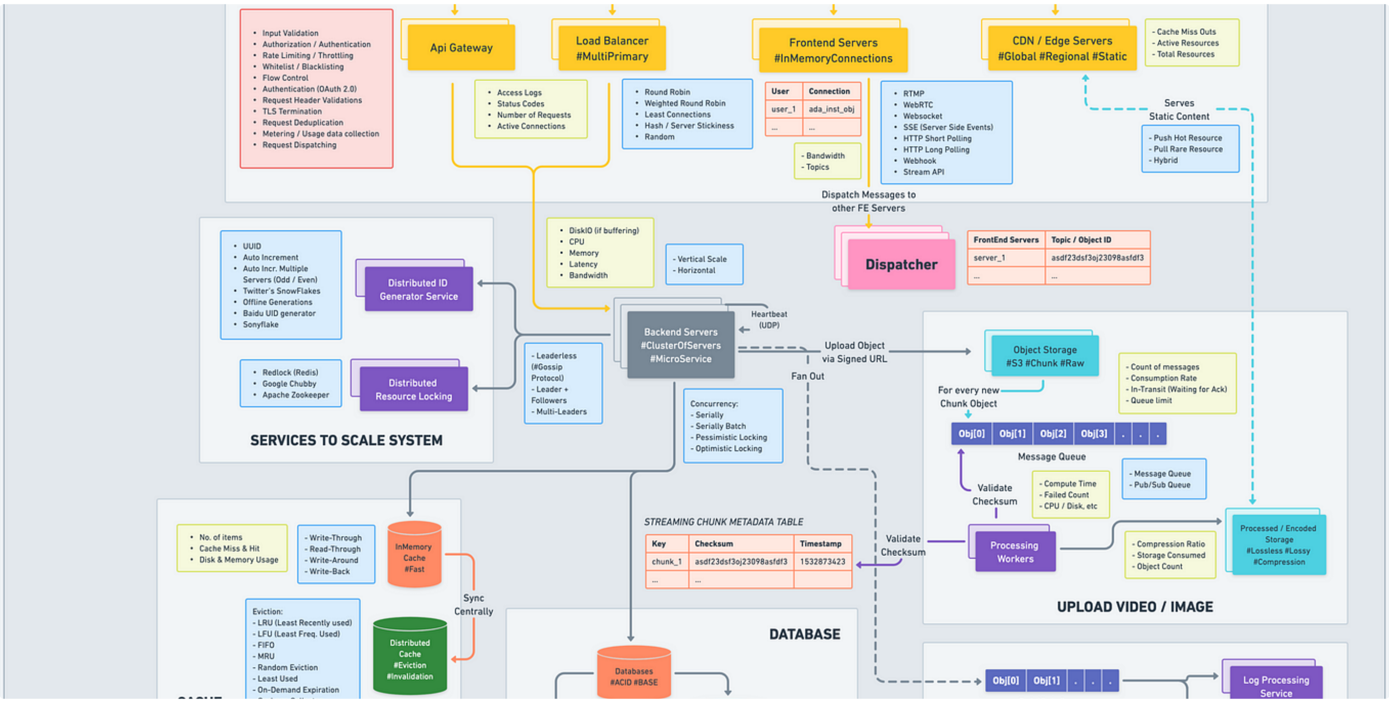
Shomarri Romell Diaz in FAUN—Developer Community 🐾

Docker Swarm Stack: Creating A Multi-Service Architecture From a Docker-Compose File

Deploying a multi service docker swarm stack on AWS from a Docker compose-file.

✦ · 8 min read · Mar 10

33



User	Connection
user_1	ada_inst_obj
...	...

FrontEnd Servers	Topic / Object ID
server_1	asdf23dsf3oj23098asfdf3
...	...

Key	Checksum	Timestamp
chunk_1	asdf23dsf3oj23098asfdf3	1532873423
...	...	

Love Sharma in ByteByteGo System Design Alliance

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

✦ · 9 min read · Apr 20

6.3K 53

headintheclouds in Dev Genius

2 Methods: Deploy an App with a Basic Ingress Service on AWS EKS Using Kubernetes and Terraform…

✦ · 9 min read · Mar 5

8

See more recommendations