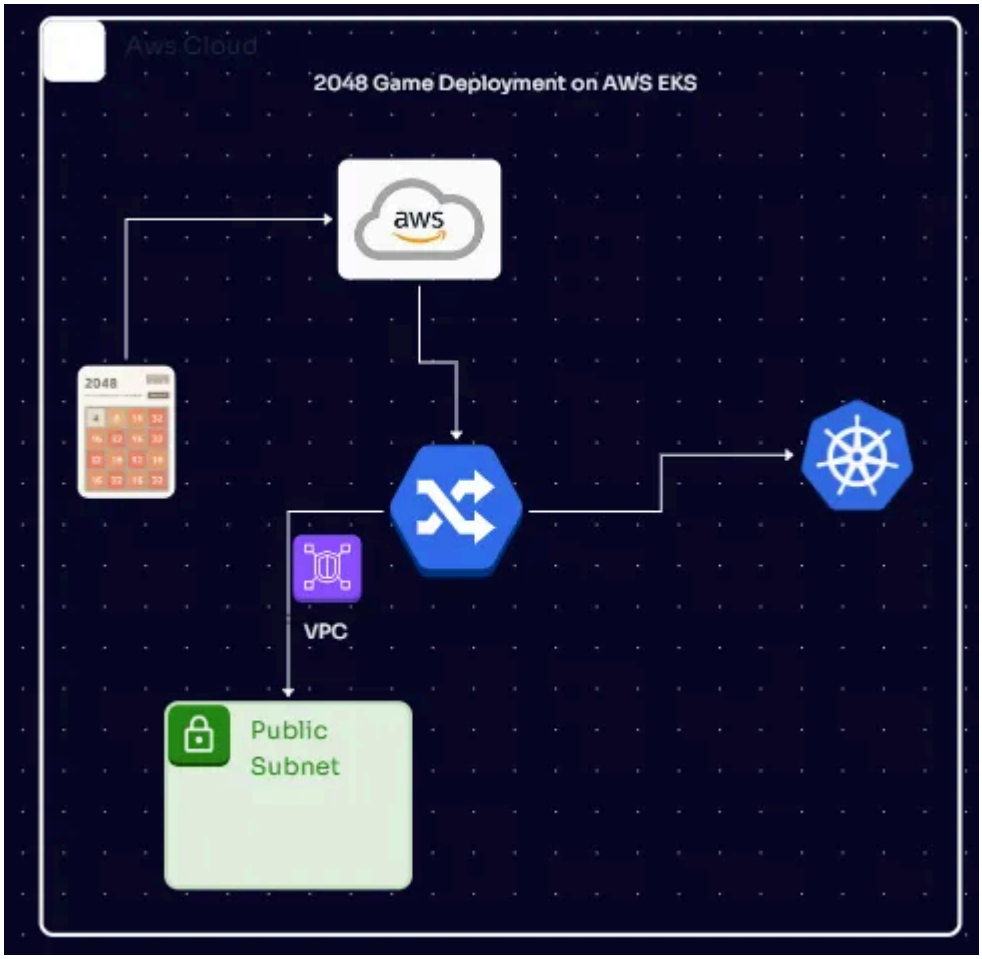
Deploying a 2048 Game on AWS EKS Fargate with Load Balancing

5 min read · Aug 25, 2024

Samsor Rahman Follow



Kubernetes (K8s) and Amazon Elastic Kubernetes Service (EKS) empower developers to build scalable, reliable, and containerized applications. In this guide, we'll demonstrate how to deploy the classic 2048 game on EKS, using AWS Fargate for a serverless experience and the AWS Load Balancer Controller for seamless traffic management.

Why AWS EKS Fargate & Load Balancing?

- Scalability: EKS effortlessly scales your application up or down based on demand, ensuring optimal performance during peak traffic periods.
- High Availability: AWS Load Balancer Controller automatically distributes incoming traffic across multiple pods of your application, preventing downtime even when individual components fail.
- Serverless Simplicity: Fargate eliminates the need to manage worker nodes, allowing you to focus on your application development. You pay only for the resources your application consumes, making it cost-effective.

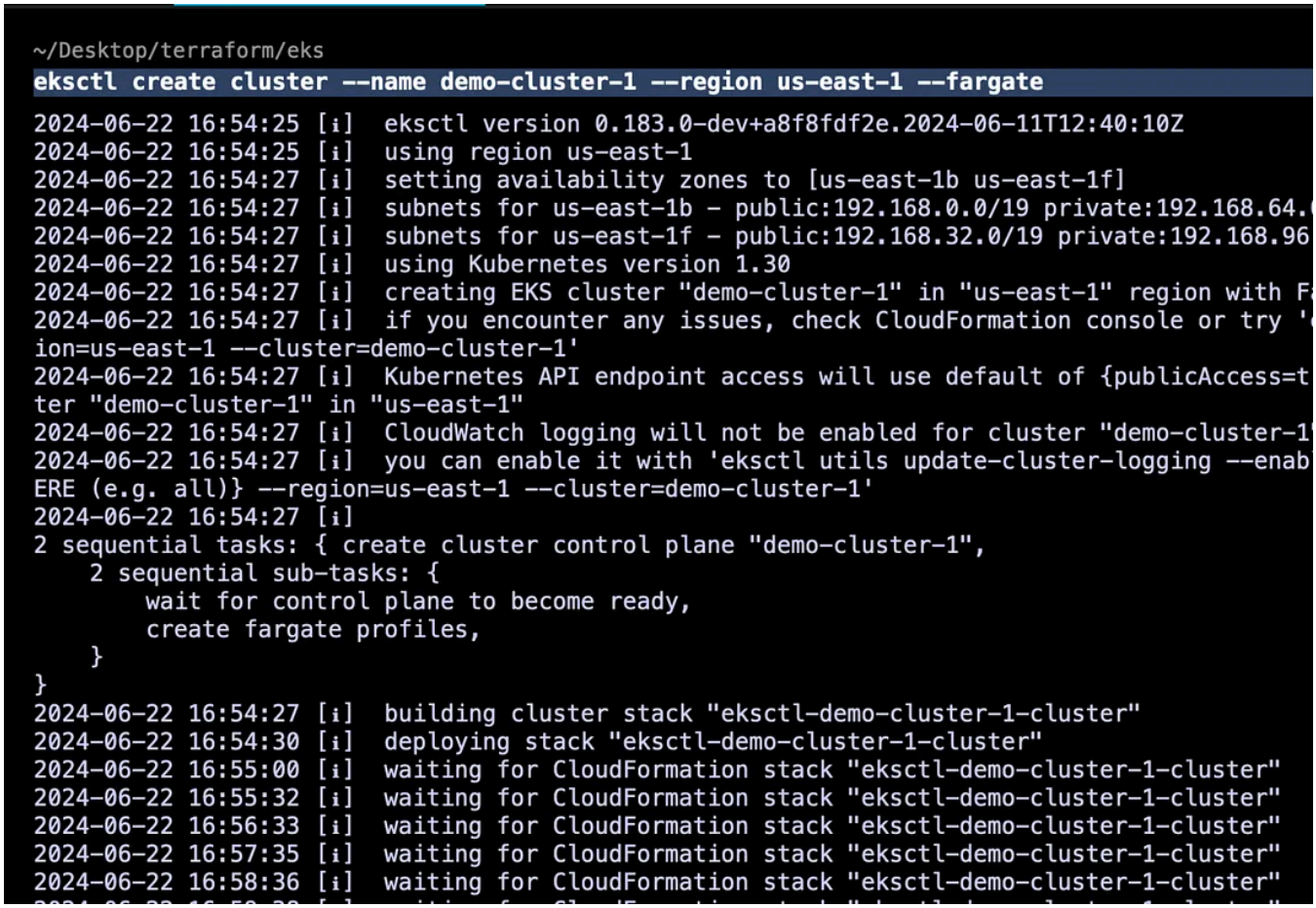
Understanding the Components

- EKS Cluster: The foundation for our application. This is our Kubernetes environment managed by AWS.
- Fargate Profile: This enables EKS to run pods directly on Fargate, AWS's serverless compute engine.
- Namespace: A virtual cluster within Kubernetes used to organize resources. Our game will live in the game-2048 namespace.
- Deployment: Defines how the 2048 game application will be deployed (how many replicas/instances, which container image to use, etc.).
- Service: Exposes the 2048 game to other components within the Kubernetes cluster.
- Ingress: Acts as the entry point to our application from the outside world. The AWS Load Balancer Controller will create a load balancer based on this Ingress configuration.
- AWS Load Balancer Controller: This is the Kubernetes controller that integrates with AWS Application Load Balancers (ALBs). It watches for Ingress resources and provisions ALBs accordingly.
- IAM OIDC Provider: This allows EKS to authenticate with AWS IAM, enabling us to assign fine-grained permissions to our Kubernetes service accounts.

Steps for deployment

1. Create EKS Cluster

```
eksctl create cluster --name demo-cluster-1 --region us-east-1 --fargate
```



This command creates an EKS cluster named “demo-cluster-1” in the us-east-1 region using Fargate.
Now Update your local kubeconfig file to connect with kubectl .

```
aws eks --region us-east-1 update-kubeconfig --name demo-cluster-1
```

This will connect your cluster with kubectl command line you will avail to control your cluster using kubectl command.

2. Create Fargate Profile and Namespace:

```
eksctl create fargateprofile \
    --cluster demo-cluster-1 \
```

```
    --region us-east-1 \
    --name alb-sample-app \
    --namespace game-2048
```

This command prepares your cluster to run the application in a serverless Fargate environment and creates a dedicated namespace (game-2048) for it.

3. Deploy the 2048 Game Application:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sigs/aws-load-bal
```

In this URL are the Deployment, Service, and Ingress resources for the 2048 game application that was already created previously.

Here is the complete breakdown of the YAML file below.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: game-2048
---
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: game-2048
  name: deployment-2048
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: app-2048
  replicas: 5
  template:
    metadata:
      labels:
        app.kubernetes.io/name: app-2048
    spec:
      containers:
      - image: public.ecr.aws/l6m2t8p7/docker-2048:latest
        imagePullPolicy: Always
        name: app-2048
        ports:
```

```
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  namespace: game-2048
  name: service-2048
spec:
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  type: NodePort
  selector:
    app.kubernetes.io/name: app-2048
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: game-2048
  name: ingress-2048
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
spec:
  ingressClassName: alb
  rules:
    - http:
        paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: service-2048
              port:
                number: 80
```

4. Configure IAM OIDC Provider and Policy:

Now Associate an IAM OIDC provider with your cluster:

```
eksctl utils associate-iam-oidc-provider --cluster demo-cluster-1 --approve
```

Then Create an IAM policy for the AWS Load Balancer Controller:

```
curl -O https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-cor
aws iam create-policy \
    --policy-name AWSLoadBalancerControllerIAMPolicy \
    --policy-document file://iam_policy.json
```

Now iam policy is created .

```
$ aws iam create-policy \
    --policy-name AWSLoadBalancerControllerIAMPolicy \
    --policy-document file://iam_policy.json
{
    "Policy": {
        "PolicyName": "AWSLoadBalancerControllerIAMPolicy",
        "PolicyId": "ANPARHQPWKDGQN53NP7BI",
        "Arn": "arn:aws:iam::084858523853:policy/AWSLoadBalancerControllerIAMPc
        "Path": "/",
        "DefaultVersionId": "v1",
        "AttachmentCount": 0,
        "PermissionsBoundaryUsageCount": 0,
        "IsAttachable": true,
        "CreateDate": "2024-04-25T20:02:49+00:00",
        "UpdateDate": "2024-04-25T20:02:49+00:00"
```

Now we create the role .

```
eksctl create iamserviceaccount \
  --cluster=<your-cluster-name> \
  --namespace=kube-system \
  --name=aws-load-balancer-controller \
  --role-name AmazonEKSLoadBalancerControllerRole \
  --attach-policy-arn=arn:aws:iam::<your-aws-account-id>:policy/AWSLoadBalancer
  --approve
```

Now, I'm attaching the role to the ServiceAccount of the pod. This ensures that the pod has the necessary credentials (through the ServiceAccount) to integrate with other AWS resources.

Let's proceed with creating the Application Load Balancer (ALB) Ingress controller we discussed earlier. We'll use a Helm chart for this, which will automatically create the controller pod and the ServiceAccount associated with it.

```
helm repo add eks https://aws.github.io/eks-charts
```

```
helm repo update eks
```

Now lets install the helm charts so again while installing the helm chart you have to modify couple of thisg one is vpc id

```
helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=<your-cluster-name> \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller \
  --set region=<region> \
  --set vpcId=<your-vpc-id>
```

Here i am using the helm chart for installing the load balancer controller

Now final check you have to do is you have to verify this load balancer is created and there are least two replicas of it .

```
$ kubectl get deployment -n kube-system aws-load-balancer-controller
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
aws-load-balancer-controller   2/2     2            2           2m25s
```

This confirms that your ALB Ingress controller is functioning correctly. The controller deployment creates two replicas, ensuring one runs in each Availability

Zone for redundancy. It continuously monitors for Ingress resources and ensures the ALB resources are also spread across two Availability Zones for high availability.

```
$ kubectl get ingress -n game-2048
NAME           CLASS   HOSTS   ADDRESS
ingress-2048   alb     *       k8s-game2048-ingress2-7b92b43b8b-788045392.us-ea
```

Done !!!!!!!!!!

Access the URL and play with your app

DevOps | Cloud Computing | Cloud | Kubernetes | AWS

Follow

Written by Samsor Rahman

1.1K followers · 32 following

Building infrastructure that scales :)

No responses yet

Write a response

What are your thoughts?

More from Samsor Rahman

Samsor Rahman

How to Run a Makefile in Windows

Makefile is a special file containing information related to the program's flow, targets, and libraries. It is also called a description...

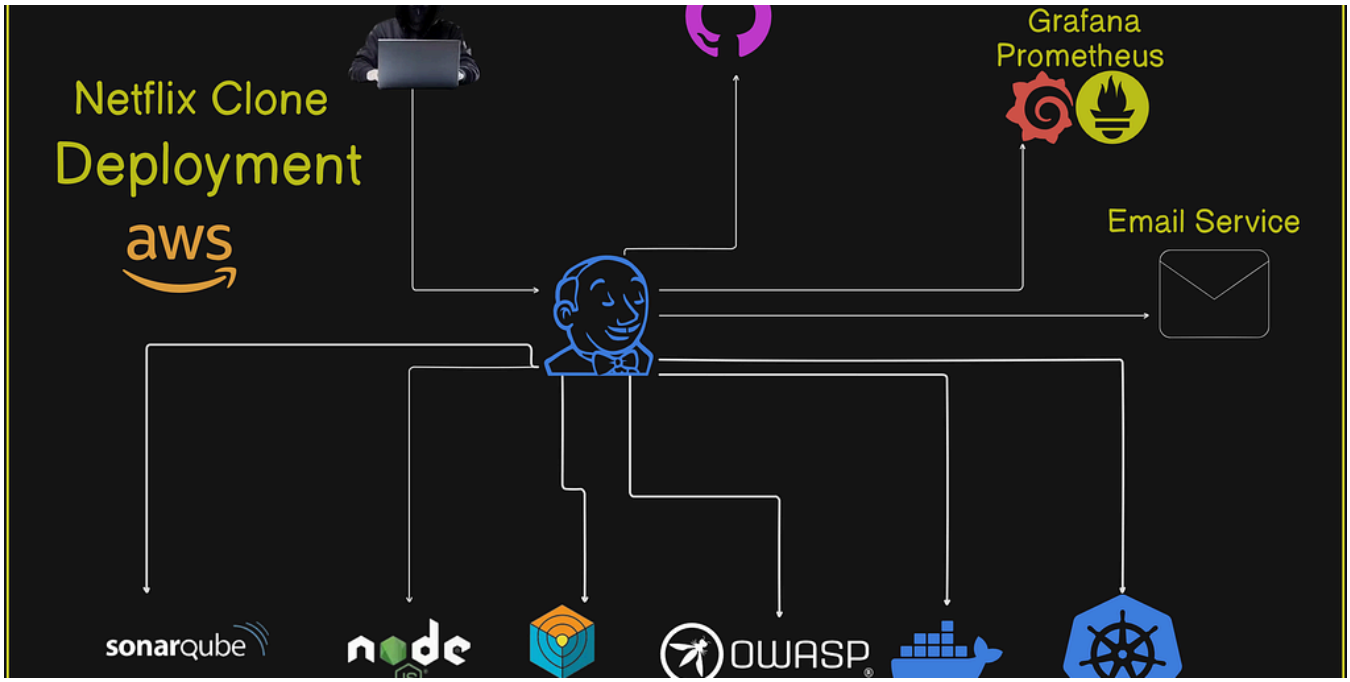
Sep 22, 2023 237 1

Samsor Rahman

How to use Celery with Django

Celery is a distributed task queue system that can be used with Django to perform asynchronous tasks such as sending emails, processing...

Sep 28, 2023 458 1

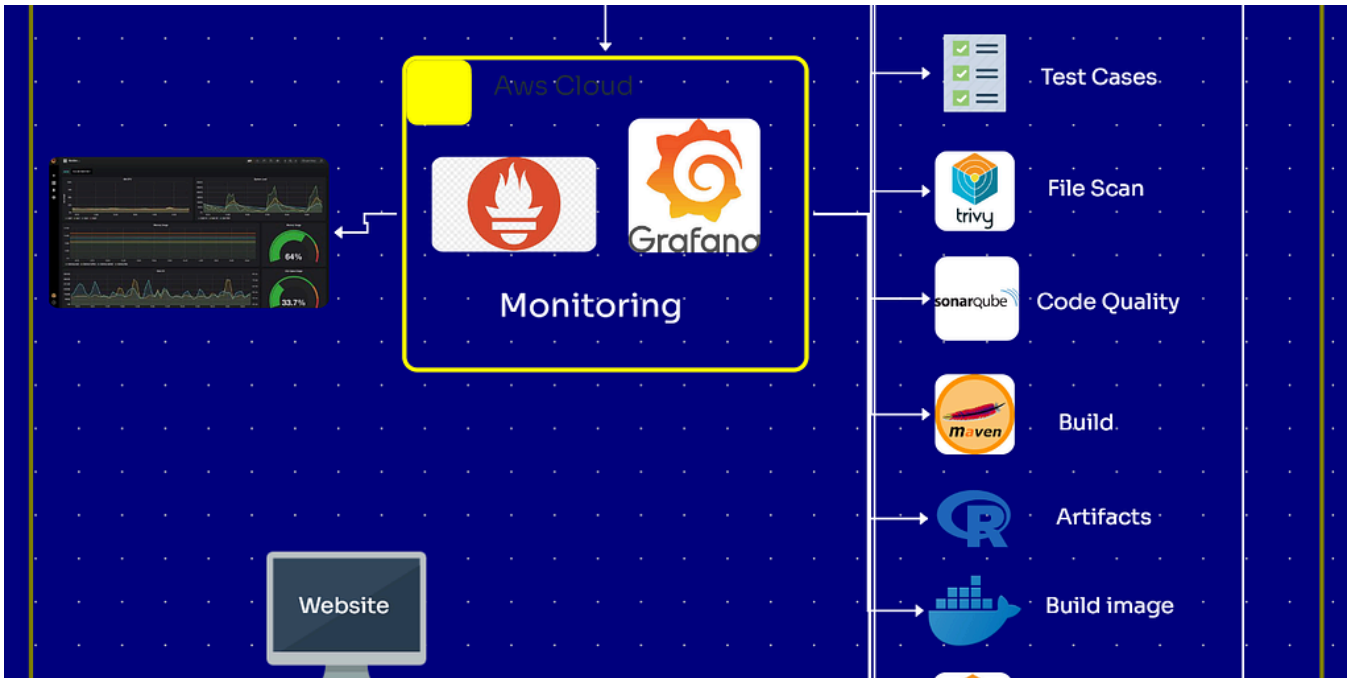


In Towards AWS by Samsor Rahman

Day 89 #90 DaysOfDevOps End-to-End DevSecOps Kubernetes Project using AWS, Jenkins CICD Pipeline

Hello friends, today we will be deploying a Netflix clone! We will be using a variety of tools to automate the build, test, and deployment...

Feb 22, 2024 403 4



Samsor Rahman

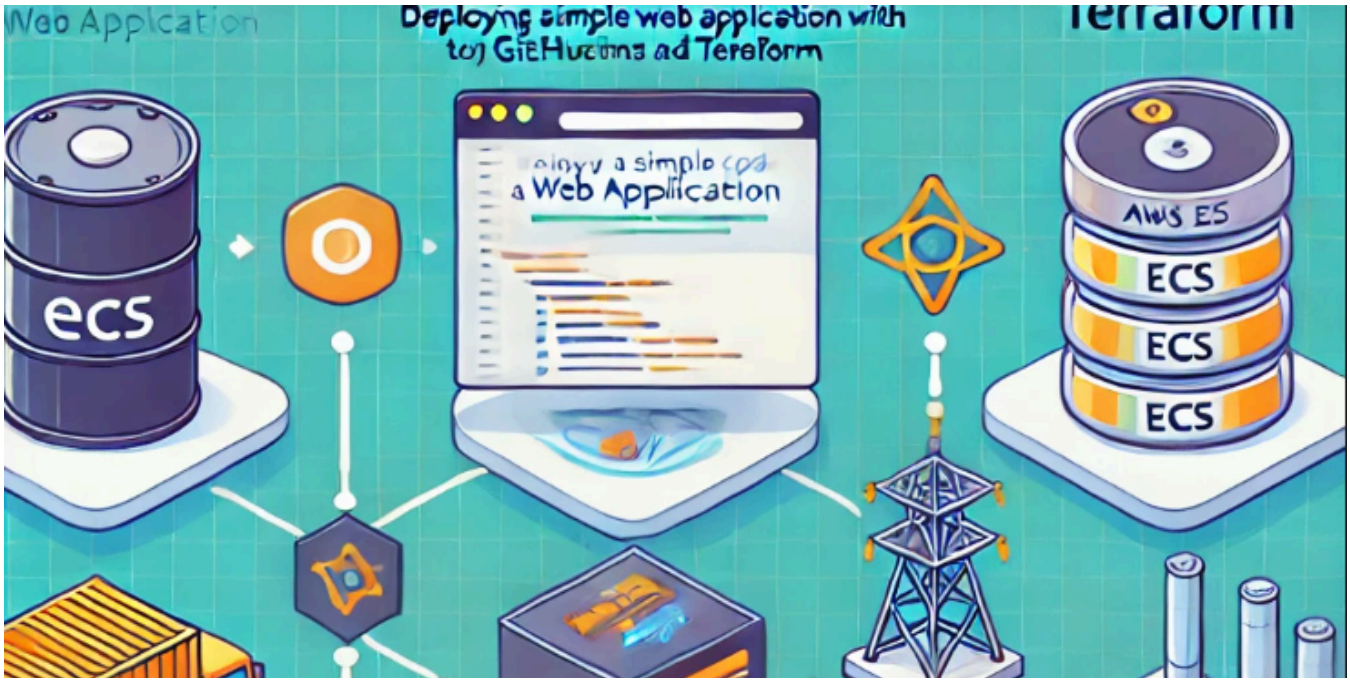
Production Level CICD Project, Terraform + EKS

Introduction

Aug 24, 2024 144

See all from Samsor Rahman

Recommended from Medium

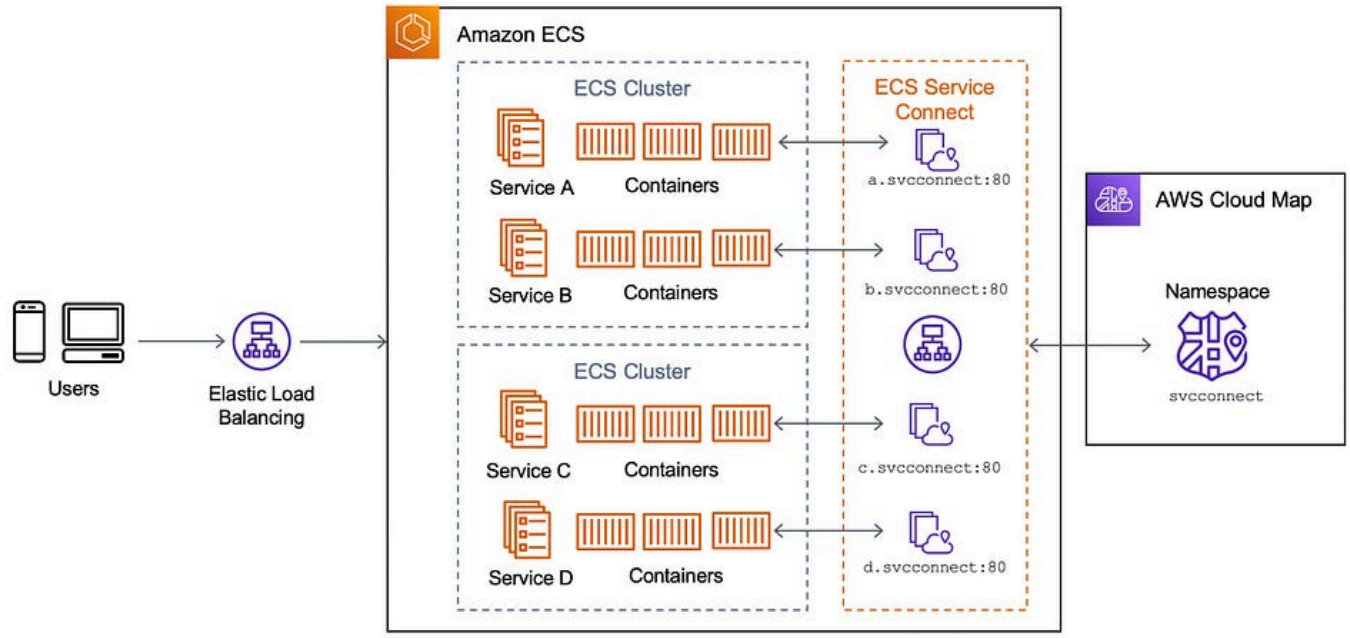


Shrihari Haridas

Deploying a Simple Web Application to AWS ECS with GitHub Actions and Terraform

Introduction: In this blog, we will walk through the process of deploying a simple “Hello World” web application to AWS ECS (Elastic...

Feb 8 29 6

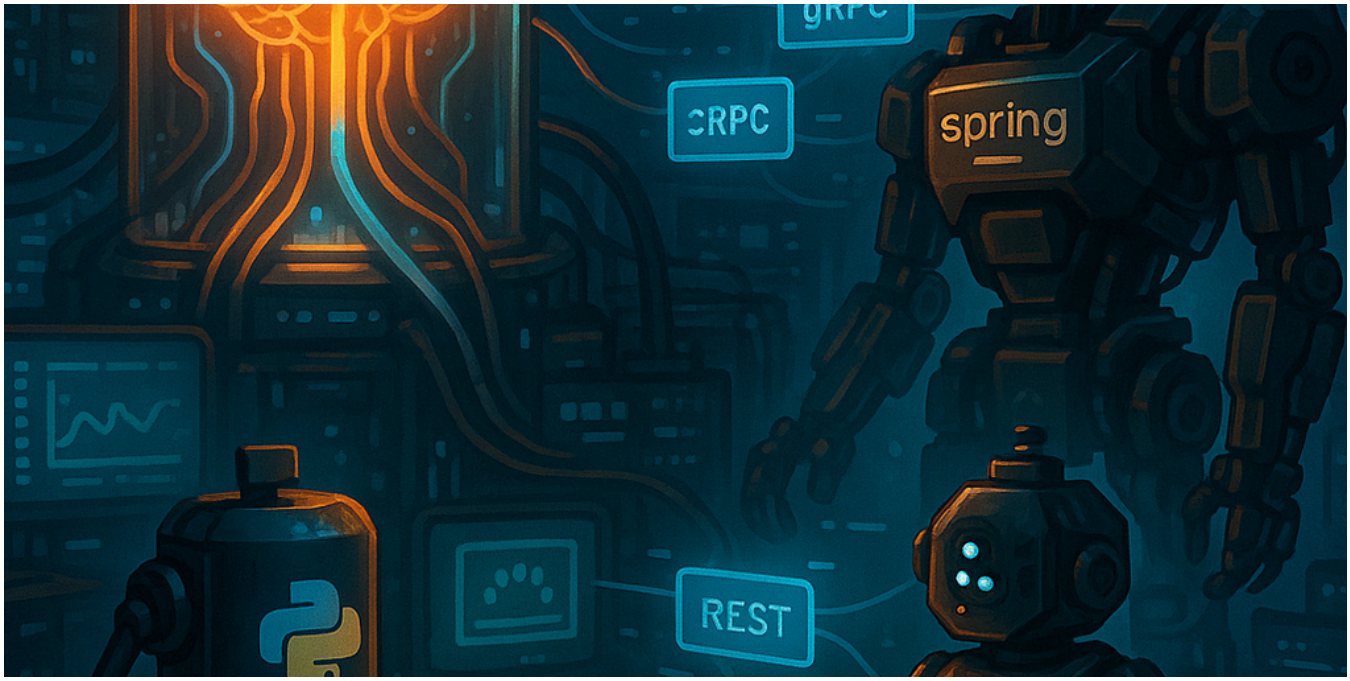


Bhanu Reddy

ECS Best Practices— Cluster Design

1) Introduction :

Apr 2 10

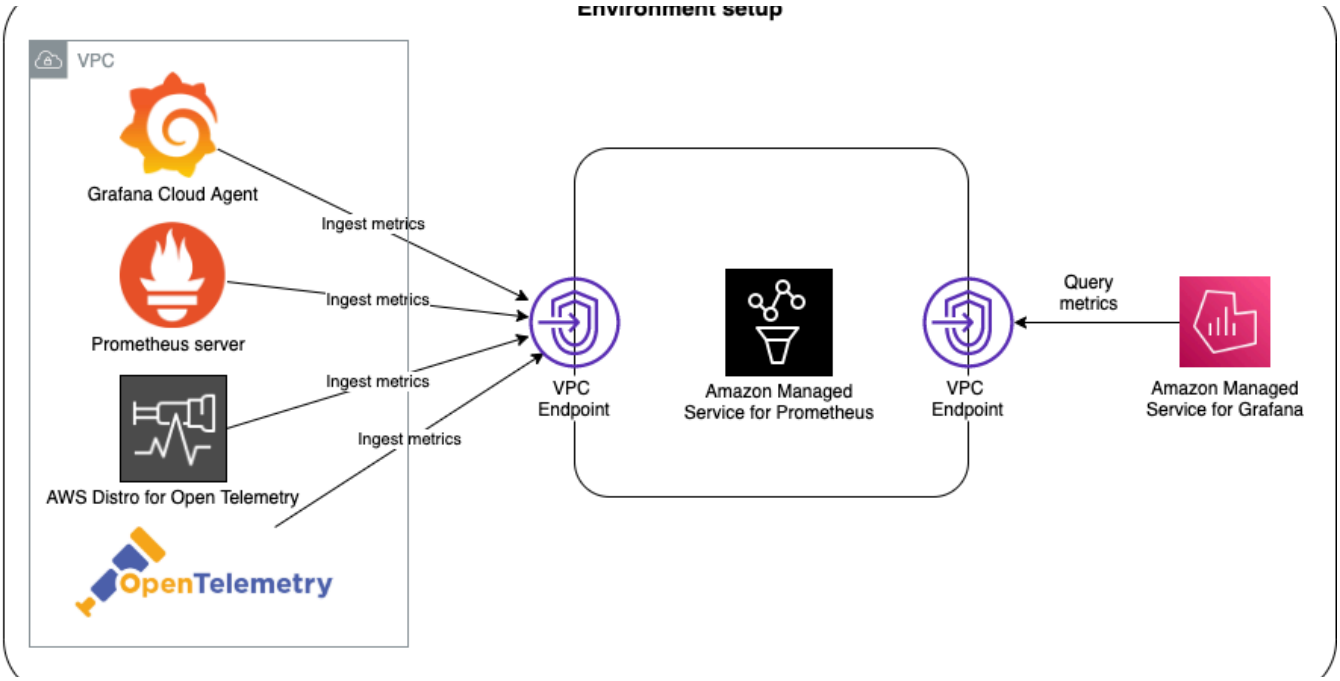


Mr_Jeet_24

Machine Learning in Python, Deployed with Java: Architecting Hybrid AI Systems

In today's digital enterprise, it's not enough to build machine learning models—you must deploy them at scale, securely, and with high...

Jul 15 24



Nagarjun (Arjun) Nagesh

Monitoring and Logging in AWS: CloudWatch vs. Prometheus vs. ELK

Monitoring and logging are critical for ensuring application performance, security, and reliability in cloud environments. AWS offers...

Mar 11

Adekola Olawale

Part 16: Monitoring with Prometheus and Grafana

A Guide for DevOps Engineers

Apr 19 1

In Towards AWS by Mohamed ElEmam

AI-Powered Observability: Secrets to Catching Production Bugs Before They Bite

How Datadog's Bits AI, Watchdog & LLM Observability Slash MTTR for Senior DevOps Engineers in 2025

Jul 24 16