Deploying a Containerized Backend App on AWS EKS with Kubernetes

Introduction

Kubernetes has become the industry standard for container orchestration, and AWS Elastic Kubernetes Service (EKS) offers a managed Kubernetes solution that reduces the operational overhead of running Kubernetes clusters. This comprehensive guide walks you through deploying a containerized Flask backend application on AWS EKS, from setting up your environment to exposing your application with a load balancer.

Why EKS over other options?

While there are several ways to run containerized applications on AWS (like ECS, Fargate, or managing your own Kubernetes on EC2), EKS offers several advantages:

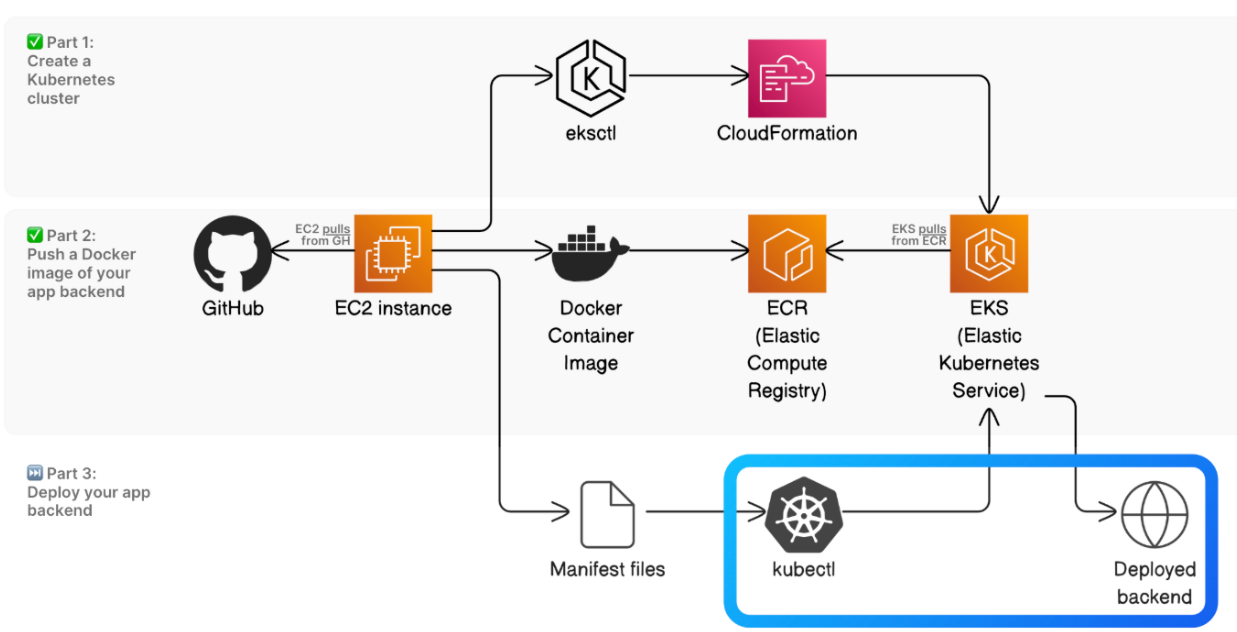
- **Managed Kubernetes control plane**: AWS manages the Kubernetes control plane, ensuring high availability and security patches.
- **Native Kubernetes compatibility**: Use standard Kubernetes tools and APIs without vendor lock-in.
- **Integration with AWS services**: Seamless integration with IAM, VPC, ELB, and other AWS services.
- **Simplified cluster management**: Automated upgrades and scaling of the control plane.

Prerequisites

Before starting this tutorial, you should have:

- An AWS account with appropriate permissions
- Basic knowledge of Docker and containerization concepts
- Familiarity with basic Kubernetes concepts (pods, deployments, services)
- Basic understanding of networking concepts

What we'll build



By the end of this guide, you'll have:

- A fully functioning EKS cluster
- A containerized Flask backend running in Kubernetes pods
- An AWS load balancer exposing your application to the internet
- Knowledge of how to troubleshoot common issues

Let's get started!

Section 1: Setting Up the Kubernetes Environment

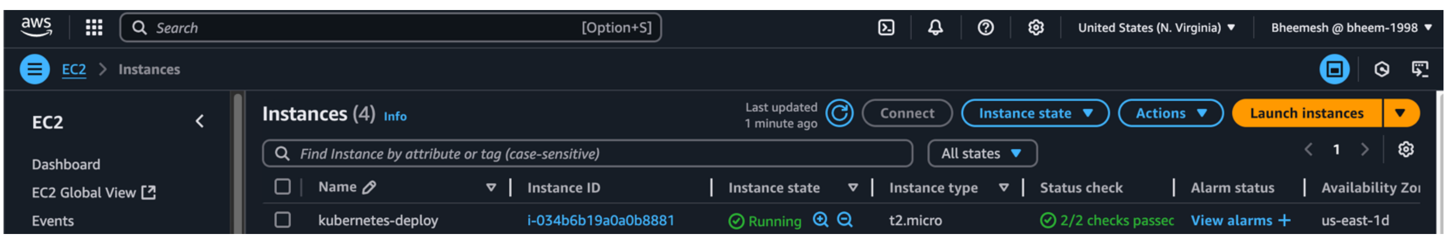
Step 1: Launch and Connect to an EC2 Instance

While you can configure AWS resources from your local machine, using an EC2 instance as a management server offers several advantages:

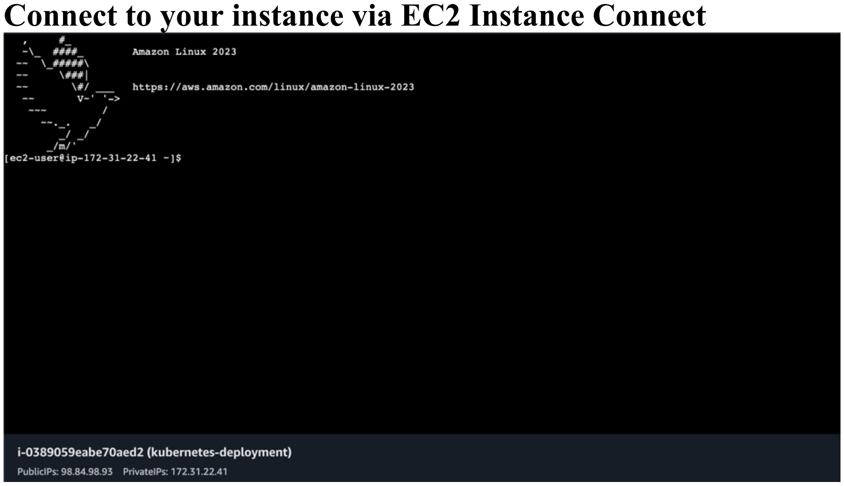
- Consistent environment configuration
- Better network connectivity to AWS services
- Avoid local machine configuration issues

To launch an EC2 instance:

1. Navigate to the EC2 console in AWS
2. Click "Launch Instance"
3. Select Amazon Linux 2 as the AMI
4. Choose t2.micro for the instance type (free tier eligible)
5. Configure security group to allow SSH access (port 22)
6. Launch the instance with your key pair



Connect to your instance via EC2 Instance Connect



Step 2: Install Required Dependencies

We'll need to install several CLI tools to interact with AWS EKS and Kubernetes:

1. Install eksctl

The `eksctl` tool simplifies EKS cluster creation and management:

```
curl --silent --location
"https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -
s)_amd64.tar.gz" | tar xz -C /tmp
sudo mv /tmp/eksctl /usr/local/bin
```

Verify installation:

```
eksctl version
```

2. Install kubectl

The `kubectl` command-line tool lets you control Kubernetes clusters:

```
curl -o kubectl https://amazon-eks.s3.us-west-
2.amazonaws.com/latest/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin
```

Verify installation:

```
kubectl version --client
```

3. Install Docker

Since we'll be building Docker images:

```
sudo yum update -y
sudo amazon-linux-extras install docker -y
sudo service docker start
sudo usermod -a -G docker ec2-user
```

Log out and log back in for group changes to take effect, or run:

```
newgrp docker
```

Verify installation:

```
docker --version
```

Step 3: Create an EKS Cluster

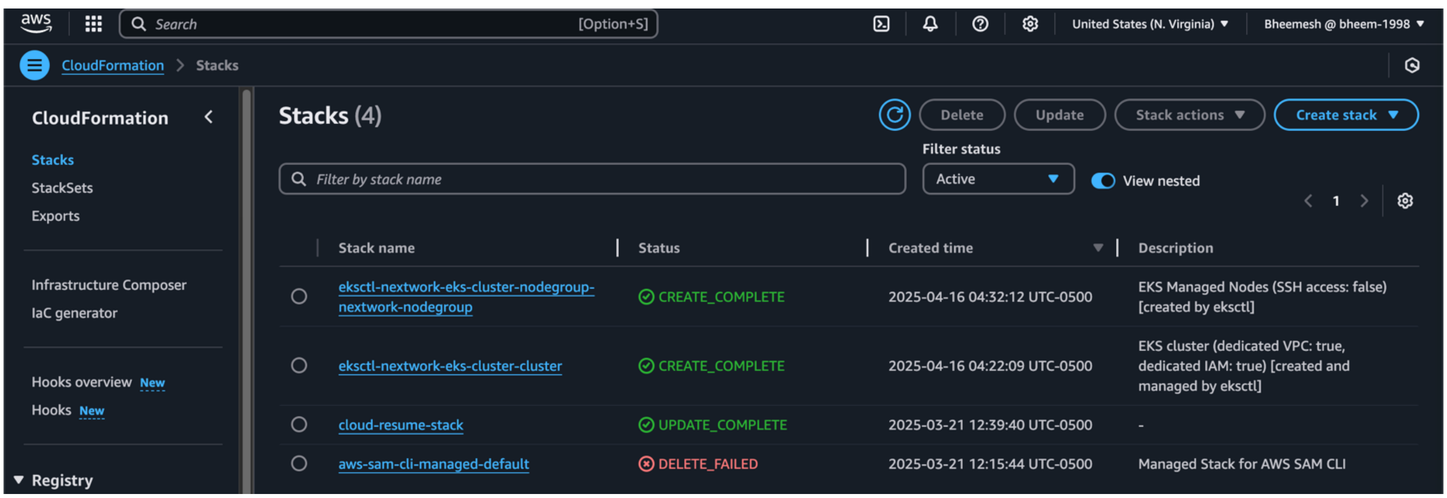
Now, let's create an EKS cluster using `eksctl`:

```
eksctl create cluster \
  --name backend-cluster \
  --region us-east-1 \
  --nodegroup-name linux-nodes \
  --node-type t2.micro \
  --nodes 2 \
  --nodes-min 1 \
  --nodes-max 3 \
  --managed
```

This command:

- Creates a cluster named "backend-cluster" in the us-east-1 region
- Creates a node group named "linux-nodes" with t2.micro instances
- Starts with 2 nodes but can scale between 1 and 3 nodes
- Uses managed node groups (AWS manages the EC2 instances)

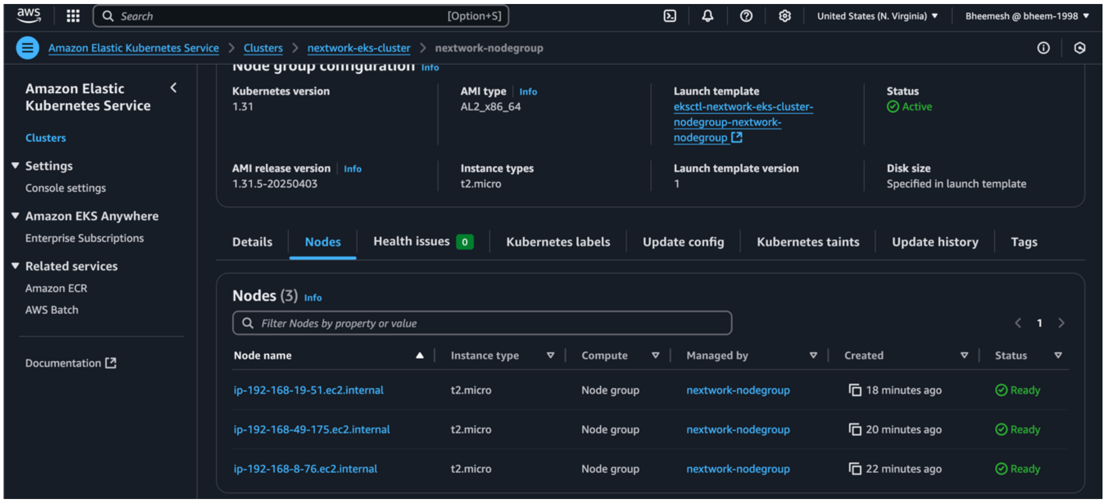
Note: This operation takes 15-20 minutes to complete.



When the cluster creation is complete, `eksctl` automatically configures `kubectl` to connect to your cluster.

Verify cluster creation:

```
kubectl get nodes
```

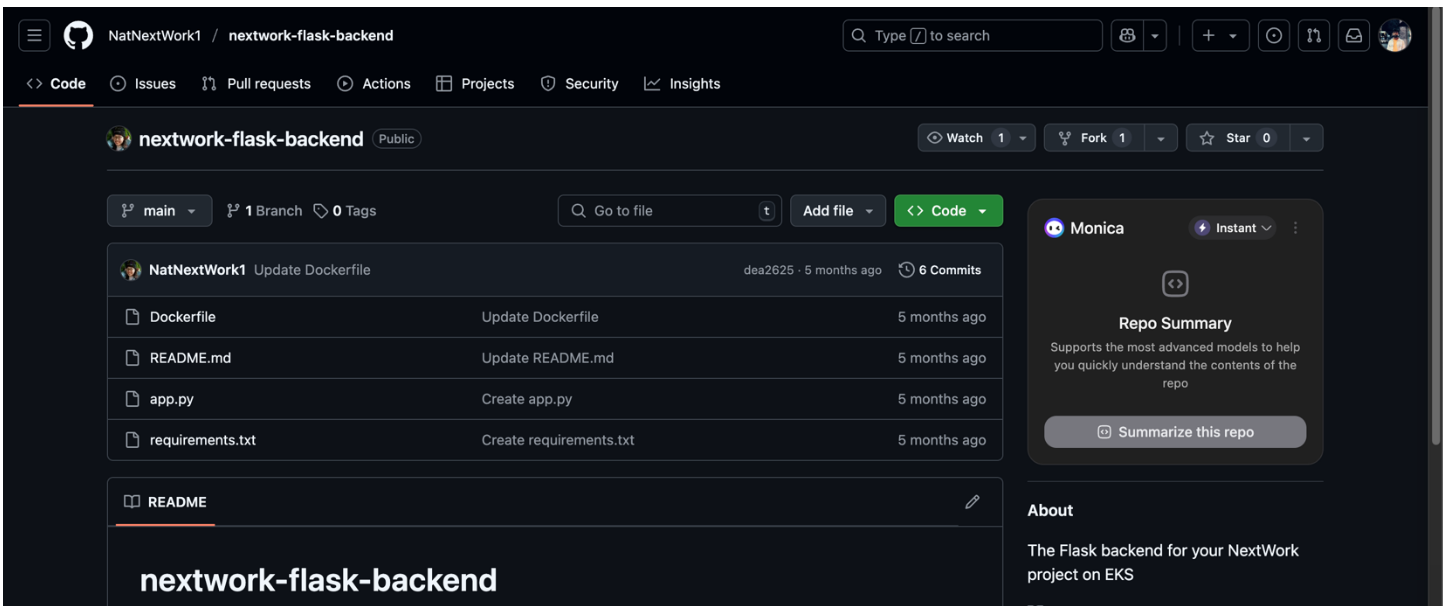


🐳 Section 2: Backend Setup and Containerization

Step 1: Clone Backend Code

First, let's clone the Flask backend application:

```
git clone https://github.com/NatNextWork1/nextwork-flask-backend.git
cd nextwork-flask-backend
```



Let's briefly examine the structure of this Flask application:

```
ls -la
```

```
[ec2-user@ip-172-31-80-247 ~]$ git config --global user.name "Bheemender"
[ec2-user@ip-172-31-80-247 ~]$ git config --global user.email "gurram.bheemender@gmail.com"
[ec2-user@ip-172-31-80-247 ~]$ git clone https://github.com/NatNextWork1/nextwork-flask-backend.git
Cloning into 'nextwork-flask-backend'...
remote: Enumerating objects: 18, done.
remote: Counting objects: 100% (18/18), done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 18 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (18/18), 6.14 KiB | 3.07 MiB/s, done.
Resolving deltas: 100% (4/4), done.
[ec2-user@ip-172-31-80-247 ~]$ ls
nextwork-flask-backend
[ec2-user@ip-172-31-80-247 ~]$
```

Key files include:

- `app.py`: The main Flask application
- `requirements.txt`: Python dependencies
- `/api`: Directory containing API routes

Step 2: Create a Dockerfile

Now, let's create a `Dockerfile` to containerize our Flask application:

```
FROM python:3.8-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV FLASK_RUN_PORT=5000

EXPOSE 5000

CMD ["flask", "run"]
```

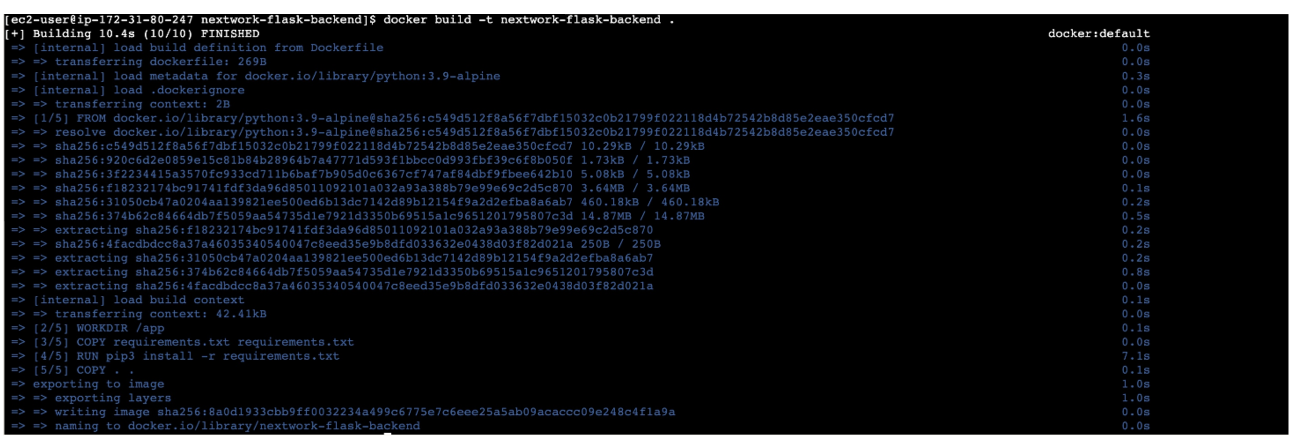
This Dockerfile:

1. Uses Python 3.8 slim as the base image
2. Sets up a working directory
3. Copies and installs dependencies
4. Copies the application code
5. Sets environment variables
6. Exposes port 5000
7. Runs the Flask application

Step 3: Build the Docker Image

Build the Docker image:

```
docker build -t flask-backend:latest .
```



If you encounter permission issues with Docker:

```
sudo usermod -aG docker $USER
newgrp docker
```

Test the container locally:

```
docker run -p 5000:5000 flask-backend:latest
```

Access the application at http://localhost:5000 if you're on a local machine, or use `curl` on the EC2 instance:

```
curl http://localhost:5000/api/health
```

You should see a response like `{"status":"healthy"}`.

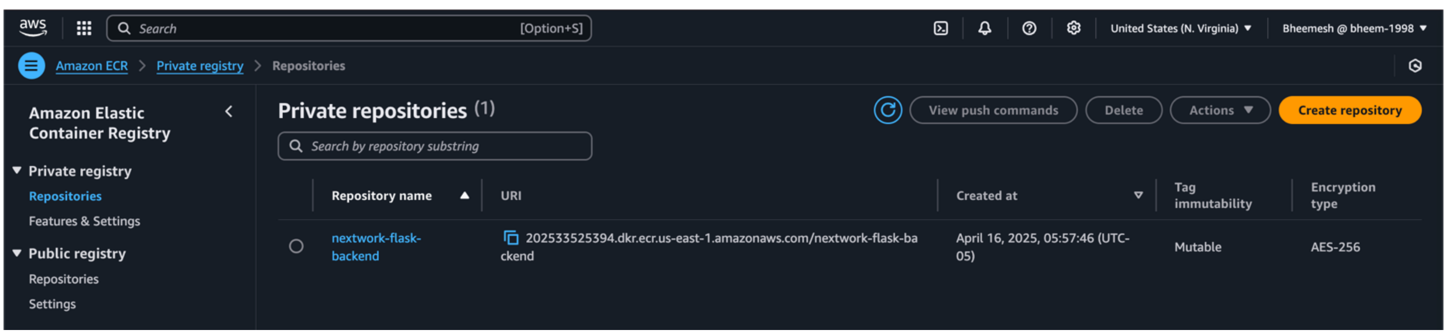
Step 4: Push to Amazon ECR

Amazon Elastic Container Registry (ECR) is a fully managed Docker container registry that makes it easy to store, manage, and deploy container images.

Create an ECR Repository

```
aws ecr create-repository --repository-name flask-backend
```

Note the repository URI in the output:



Authenticate to ECR

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin $(aws sts get-caller-identity --query Account --output text).dkr.ecr.us-east-1.amazonaws.com
```

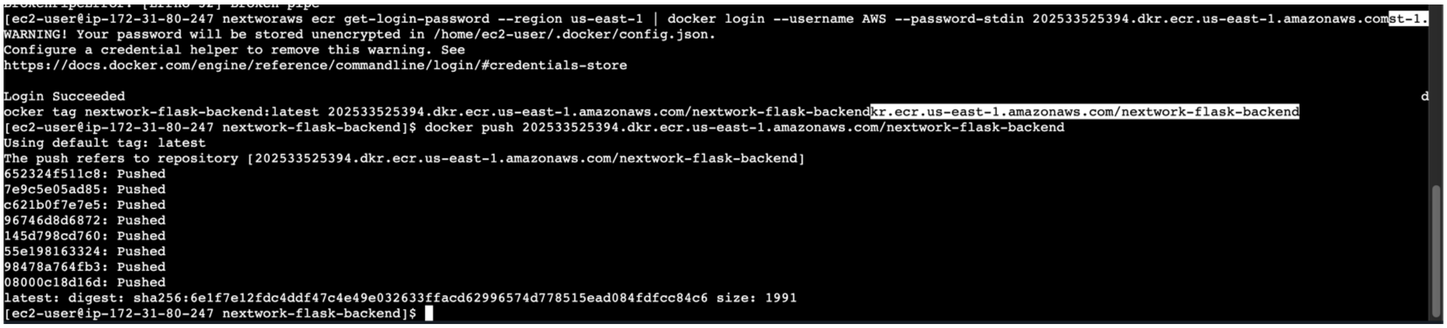
Tag and Push the Image

```
# Get your AWS account ID
AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)

# Tag the image
docker tag flask-backend:latest ${AWS_ACCOUNT_ID}.dkr.ecr.us-east-1.amazonaws.com/flask-backend:latest

# Push the image
docker push ${AWS_ACCOUNT_ID}.dkr.ecr.us-east-1.amazonaws.com/flask-backend:latest
```

Verify the push in the ECR console:



Section 3: Kubernetes Deployment and Service

Now that our container image is available in ECR, let's create Kubernetes resources to deploy it.

Step 1: Create a Kubernetes Namespace

Namespaces help organize resources within a cluster:

```
kubectl create namespace backend
```

Step 2: Create Deployment Manifest

Create a file named `deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-backend
  namespace: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flask-backend
  template:
    metadata:
      labels:
        app: flask-backend
    spec:
      containers:
      - name: flask-backend
        image: ${AWS_ACCOUNT_ID}.dkr.ecr.us-east-1.amazonaws.com/flask-backend:latest
        ports:
        - containerPort: 5000
        resources:
          requests:
            memory: "128Mi"
            cpu: "100m"
          limits:
```

```
            memory: "256Mi"
            cpu: "200m"
        readinessProbe:
          httpGet:
            path: /api/health
            port: 5000
          initialDelaySeconds: 5
          periodSeconds: 10
        livenessProbe:
          httpGet:
            path: /api/health
            port: 5000
          initialDelaySeconds: 15
          periodSeconds: 20
```

This deployment:

- Creates 2 replicas of our application
- Uses our ECR image
- Sets resource requests and limits
- Configures readiness and liveness probes for health checking

Don't forget to replace `${AWS_ACCOUNT_ID}` with your actual AWS account ID:

```
sed -i "s/\${AWS_ACCOUNT_ID}/$(aws sts get-caller-identity --query Account --output text)/" deployment.yaml
```

```
[ec2-user@ip-172-31-80-247 nextwork-flask-backend]$ cat << EOF > flask-deployment.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nextwork-flask-backend
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nextwork-flask-backend
  template:
    metadata:
      labels:
        app: nextwork-flask-backend
    spec:
      containers:
        - name: nextwork-flask-backend
          image: YOUR-ECR-IMAGE-URI-HERE
          ports:
            - containerPort: 8080
EOF
```

Step 3: Create Service Manifest

Create a file named `service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: flask-backend
  namespace: backend
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 5000
```

```
    protocol: TCP
  selector:
    app: flask-backend
```

This service:

- Creates an AWS Load Balancer
- Maps port 80 on the load balancer to port 5000 on our containers
- Uses the selector to find pods with the label `app: flask-backend`

Step 4: Apply the Manifests

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

Step 5: Verify Deployment

Check the status of the deployment:

```
kubectl get deployments -n backend
```

Check the pods:

```
kubectl get pods -n backend
```

Check the service and obtain the load balancer URL:

```
kubectl get svc -n backend
```

The load balancer's external IP will be your endpoint. It may take a few minutes for the load balancer to provision.

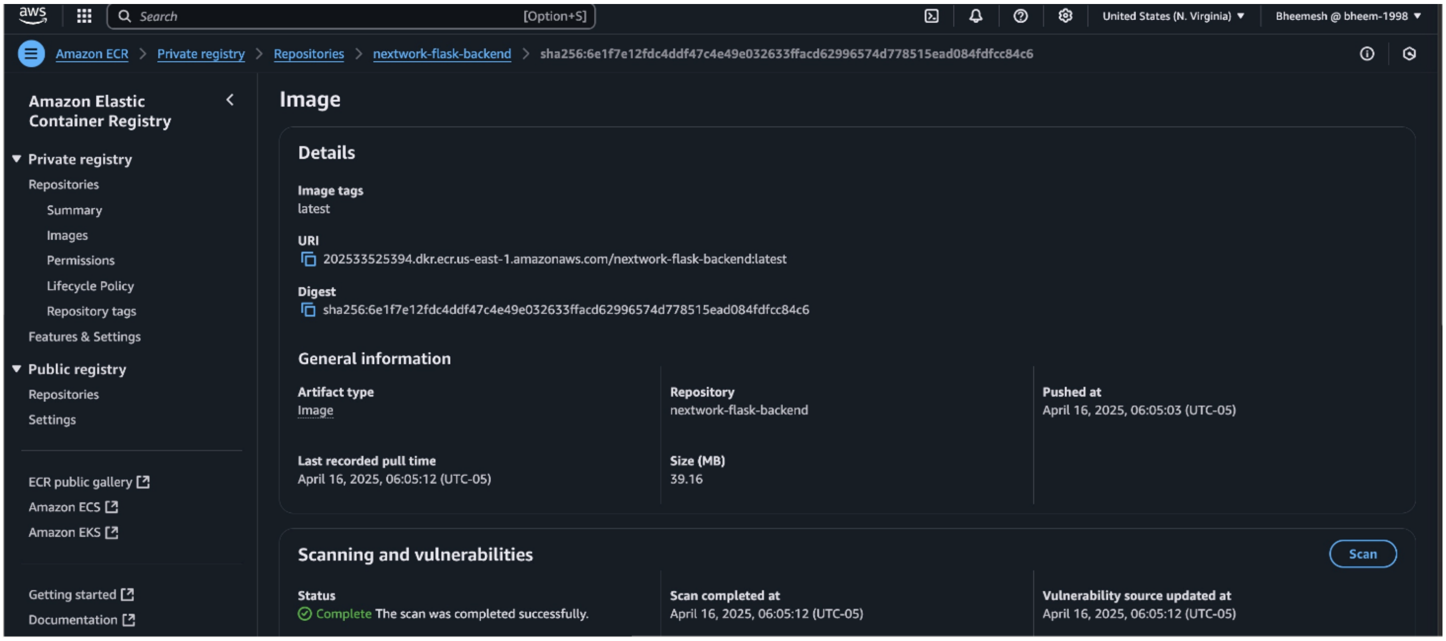
Test the deployed application:

```
curl http://your-load-balancer-url/api/health
```

Section 4: Deployment Validation and Monitoring

Step 1: Validate Deployment in AWS Console

Navigate to the EKS console to see your cluster, node groups, and workloads:



Check the load balancer in the EC2 Console:

Step 2: View Pod Logs

Logs are crucial for troubleshooting. Get the name of a pod:

```
kubectl get pods -n backend
```

View the logs:

```
kubectl logs -n backend pod-name
```

Step 3: Set Up Basic Monitoring

Let's set up basic monitoring for our cluster:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-
server/releases/latest/download/components.yaml
```

After a few minutes, you can view node metrics:

```
kubectl top nodes
```

And pod metrics:

```
kubectl top pods -n backend
```

Section 5: Security Best Practices

While our deployment is functional, let's review some security best practices:

1. IAM Roles

Use IAM roles for service accounts to grant fine-grained permissions:

```
eksctl create iamserviceaccount \
  --name flask-backend-sa \
  --namespace backend \
  --cluster backend-cluster \
  --attach-policy-arn arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess \
  --approve
```

Update your deployment to use this service account:

```
spec:
  template:
    spec:
      serviceAccountName: flask-backend-sa
```

2. Network Policies

Restrict network traffic with network policies. Create a file named `network-policy.yaml`:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: flask-backend-policy
  namespace: backend
spec:
  podSelector:
    matchLabels:
      app: flask-backend
  policyTypes:
  - Ingress
  ingress:
  - from: []
    ports:
    - protocol: TCP
      port: 5000
```

Apply it:

```
kubectl apply -f network-policy.yaml
```

3. Secret Management

For real-world applications, you should use Kubernetes Secrets or AWS Parameter Store to manage sensitive information.

Example using Kubernetes Secrets:

```
kubectl create secret generic app-secrets \
  --namespace backend \
  --from-literal=DATABASE_PASSWORD=your-secure-password
```

Reference in your deployment:

```
spec:
  containers:
  - name: flask-backend
    env:
    - name: DATABASE_PASSWORD
      valueFrom:
```

```
      secretKeyRef:
        name: app-secrets
        key: DATABASE_PASSWORD
```

Section 6: Common Issues and Troubleshooting

Issue 1: ImagePullBackOff Error

This error occurs when Kubernetes can't pull the container image.

Potential causes:

- Incorrect image name or tag
- Missing ECR permissions
- Network connectivity issues

Solutions:

1. Verify the image URL in your deployment:

   ```
   kubectl describe deployment flask-backend -n backend
   ```

2. Check ECR permissions:

   ```
   aws ecr get-repository-policy --repository-name flask-backend
   ```

3. Consider creating an ECR pull secret:

   ```
   kubectl create secret docker-registry ecr-secret \  --docker-
   server=${AWS_ACCOUNT_ID}.dkr.ecr.us-east-1.amazonaws.com \  --docker-username=AWS
   \  --docker-password=$(aws ecr get-login-password) \  --namespace=backend
   ```

Issue 2: CrashLoopBackOff Error

This occurs when your container starts but then crashes repeatedly.

Potential causes:

- Application errors
- Resource limits too low
- Configuration issues

Solutions:

1. Check pod logs:

   ```
   kubectl logs -n backend pod-name
   ```

2. Check pod events:

```
kubectl describe pod -n backend pod-name
```

3. Test the container locally:

```
docker run -p 5000:5000 ${AWS_ACCOUNT_ID}.dkr.ecr.us-east-1.amazonaws.com/flask-backend:latest
```

Issue 3: Service Not Accessible

If you can't access your service through the load balancer:

Potential causes:

- Load balancer still provisioning
- Security group rules
- Service selector mismatch

Solutions:

1. Check service status:

```
kubectl describe service flask-backend -n backend
```

2. Verify endpoints:

```
kubectl get endpoints -n backend
```

3. Check load balancer security groups in the AWS console

Section 7: Cost Considerations

Running an EKS cluster incurs several costs:

1. **EKS Cluster**: $0.10 per hour (~$73 per month)
2. **EC2 Instances**: Varies by instance type (t2.micro is ~$8.40 per month per instance)
3. **Load Balancer**: Classic Load Balancer is ~$18 per month
4. **ECR Storage**: $0.10 per GB-month
5. **Data Transfer**: Varies based on usage

Cost Optimization Tips:

- Use Spot Instances for non-critical workloads
- Implement cluster autoscaling to scale down during off-hours
- Use AWS Cost Explorer to monitor expenses
- Consider Fargate for smaller workloads

Section 8: Next Steps and Advanced Topics

Now that you have a basic deployment, consider these next steps:

1. CI/CD Pipeline

Set up a CI/CD pipeline using AWS CodePipeline or GitHub Actions to automate deployments.

2. Horizontal Pod Autoscaling

Scale your application based on CPU or custom metrics:

```
kubectl autoscale deployment flask-backend -n backend --cpu-percent=70 --min=2 --max=10
```

3. Cluster Autoscaling

Enable cluster autoscaling to automatically adjust the number of nodes:

```
eksctl scale nodegroup --cluster=backend-cluster --nodes-min=1 --nodes-max=5 --
name=linux-nodes
```

4. Ingress Controller

Deploy an Ingress Controller for advanced routing:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-
v1.1.0/deploy/static/provider/aws/deploy.yaml
```

5. Service Mesh

Consider implementing a service mesh like AWS App Mesh or Istio for advanced networking, security, and observability.

Learnings and Reflections

Throughout this journey, I gained valuable insights:

1. **Container Build Optimization**
   - Multi-stage builds can significantly reduce image size
   - Proper layer caching improves build times
2. **Kubernetes Deployment Strategies**
   - Rolling updates minimize downtime
   - Resource limits prevent node resource starvation
3. **Monitoring and Observability**
   - Logs are just the beginning
   - Metrics and distributed tracing provide deeper insights
4. **Security Considerations**
   - Principle of least privilege with IAM roles
   - Network policies as virtual firewalls

5. **Cost Management**
    - Proper instance sizing is crucial
    - Understanding the pricing model helps optimize costs

Conclusion

In this guide, we've walked through deploying a containerized Flask backend application on AWS EKS. We started with setting up the necessary tools, created and pushed a Docker image to ECR, and deployed it on Kubernetes with a load balancer.

While this setup is suitable for development and testing, production deployments would require additional considerations such as:

- SSL/TLS encryption
- Database integration
- Secret management
- Comprehensive monitoring
- High availability across multiple availability zones

By following this guide, you've established a solid foundation for containerized application deployment on AWS EKS. The skills and knowledge gained here can be applied to more complex applications and architectures.

Remember, Kubernetes is a powerful platform with many features and configurations. Keep exploring and learning to make the most of it!

Author

Bheemender Gurram

GitHub: https://github.com/Bheemender1998/deploy-backend-on-EKS-kubernetes
Linkedin: https://www.linkedin.com/in/bheemendergurram/
Medium: https://medium.com/@bheemender.gurram08
