EKS Infrastructure and Deployment Spike

Objective: To investigate and document a standardized, repeatable, and cost-effective process for provisioning an EKS cluster and deploying containerized applications to it. This document is intended as an overview for the team. Though by no means is it fully comprehensive.

Author: Vitaliy Sviridyuk

Date: September 26, 2025

Executive Summary & Recommendations

This document outlines a recommended approach for managing our EKS infrastructure and application deployment lifecycle. The core recommendations are:

1. **Infrastructure as Code (IaC):** Use eksctl with a YAML configuration file as the single source of truth for our cluster's infrastructure. This ensures consistency, repeatability, and version control.
2. **Phased Deployment Strategy:** Begin with a manual deployment process to ensure a clear understanding of the core mechanics, with a plan to implement a fully automated CI/CD pipeline using GitHub Actions in the future.
3. **Secure and Accessible Networking:** Utilize private networking for cluster nodes as a security best practice, and use standard Kubernetes tools like kubectl port-forward for direct debugging and testing access during development.

1. Cluster Infrastructure as Code with eksctl

Managing infrastructure manually through a UI is prone to error and is not repeatable. By defining our entire cluster configuration in a single YAML file, we treat our infrastructure the same way we treat our application code. eksctl is the official CLI tool for EKS and is the best tool for this job.

The Process

The process of creating or updating a cluster is reduced to a single command: eksctl apply cluster -f <cluster-config.yaml>.

Sample Configuration (cluster.yaml)

This configuration file creates a cost-effective cluster suitable for development and test environments. It uses a small, on-demand EC2 instance to ensure reliability while keeping costs low.

```yaml
# cluster.yaml
#
# Defines a cost-effective EKS cluster.
# To create or update, run: eksctl apply cluster -f cluster.yaml

apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: primary-app-cluster
  region: us-east-2
  version: "1.33"

# Disable the NAT Gateway to further reduce costs.
# Note: This means pods in private subnets won't have outbound internet
access
# by default, which is needed to pull images from public registries like
Docker Hub.
# Access to Amazon ECR will still work via VPC Endpoints.
vpc:
  nat:
    gateway: Disable

managedNodeGroups:
  - name: small-general-nodes
    # A t3.small provides a good balance of cost and performance (2 vCPU, 2
GiB RAM)
    instanceType: t3.small
    minSize: 1
    maxSize: 3 # Allows for scaling if needed
    desiredCapacity: 1
    volumeSize: 20 # GiB
    privateNetworking: true # Security best practice
```

2. Application Deployment Workflow (Manual Stop-Gap)

While the long-term goal is a fully automated CI/CD pipeline, the initial process will be manual. This approach ensures developers understand each step involved in getting an application from source code to a running state in Kubernetes.

Note: *This manual process is a temporary measure. This document will be updated in a future story to detail the fully automated CI/CD workflow using GitHub Actions.*

The Manual Workflow Steps

1. **Build the Docker Image:** From your application's directory (where the Dockerfile is), build the image.

```
docker build -t my-app:latest .
```

2. **Tag the Image for Amazon ECR:** Retag the image to match the format required by the Amazon Elastic Container Registry (ECR).

```
docker tag my-app:latest
<aws_account_id>.dkr.ecr.<region>[.amazonaws.com/my-app:v1.0.1](https://.am
azonaws.com/my-app:v1.0.1)
```

3. **Authenticate and Push to ECR:** Get login credentials for Docker and push the image to the repository.

```
aws ecr get-login-password --region <region> | docker login --username AWS
--password-stdin <aws_account_id>.dkr.ecr.<region>.amazonaws.com
docker push
<aws_account_id>.dkr.ecr.<region>[.amazonaws.com/my-app:v1.0.1](https://.am
azonaws.com/my-app:v1.0.1)
```

4. **Update Kubernetes Manifest:** Open your application's deployment.yaml file and update the image field to point to the new image tag you just pushed.

```
# deployment.yaml
...
spec:
   containers:
   - name: my-app-container
     image:
<aws_account_id>.dkr.ecr.<region>[.amazonaws.com/my-app:v1.0.1](https://.am
azonaws.com/my-app:v1.0.1) # <-- Update this line
...
```

5. **Apply the Change:** Use kubectl to apply the updated manifest to the cluster. Kubernetes will perform a rolling update to the new version.
   kubectl apply -f deployment.yaml

3. Debugging and Testing with Port-Forwarding

By default, the pods you deploy run in a private, isolated network inside the cluster. They are not directly accessible from your local machine. While a LoadBalancer or Ingress can expose them to the internet, this is often too slow and cumbersome for quick debugging or testing.

kubectl port-forward is a utility that creates a direct, secure tunnel from your local machine to a specific pod in the cluster.

Usecase

Use port-forwarding when you need to quickly test an application's endpoint as if you were inside the cluster, without creating any permanent public routes.

Walkthrough

1. **Find the Pod Name:** First, list the running pods to get the exact name of the one you want to connect to.

```
kubectl get pods
```

You'll get a list like my-app-pod-55649fd788-abcde.

2. **Run the Port-Forward Command:** Use the following command to start the tunnel. This example forwards traffic from port 8080 on your local machine to port 80 on the pod.

```
# Syntax: kubectl port-forward pod/<pod-name> <local-port>:<pod-port>
kubectl port-forward pod/my-app-pod-55649fd788-abcde 8080:80
```

Your terminal will show Forwarding from 127.0.0.1:8080 -> 80 and will hang. **This command must be left running** for the tunnel to remain active.

3. **Access the Pod:** Open a web browser on your local machine and navigate to http://localhost:8080. Your request will be securely tunneled directly to the pod.