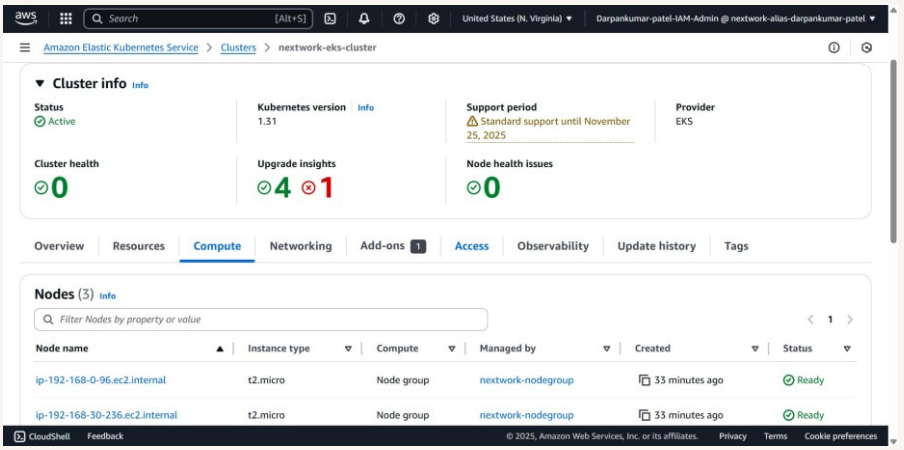
Darpan Patel

Launch a Kubernetes Cluster



Darpan Patel

Introducing Today's Project!

In this project, I will launch and configure a Kubernetes cluster using Amazon EKS because it’s a powerful way to orchestrate containerized applications in the cloud. By completing this project, I’ll gain hands-on experience with key AWS services like EC2, IAM, and CloudFormation, while learning how Kubernetes automates deployment, scaling, and management of containerized applications. This foundational step sets the stage for more advanced projects in container orchestration and cloud-native infrastructure.

What is Amazon EKS?

Amazon EKS (Elastic Kubernetes Service) is a fully managed Kubernetes service that simplifies running Kubernetes workloads on AWS. In today's project, I used EKS to deploy a Kubernetes cluster using `eksctl`, which automatically created the required infrastructure through CloudFormation, including VPCs and subnets. I then set up a managed node group with EC2 instances to run containerized applications. To gain
Kubernetes access, I created an IAM access entry, linking my AWS IAM user to Kubernetes RBAC. I also tested Kubernetes' self-healing by terminating EC2 instances, and EKS automatically launched new ones to maintain the desired node count. This project highlighted EKS's power in handling infrastructure, scaling, and recovery, while giving me hands-on experience managing a production-grade Kubernetes environment on AWS.

One thing I didn't expect

One thing I didn’t expect in this project was that having AdministratorAccess in AWS wasn’t enough to access my EKS cluster.

I assumed full AWS permissions would automatically grant me full control, but I learned that Kubernetes has its own access control system** (RBAC), and I needed to manually create an IAM access entry to bridge my AWS IAM identity with Kubernetes permissions. It was a good reminder that managing Kubernetes on AWS requires understanding both AWS and Kubernetes security models.

Darpan Patel

This project took me...

This AWS project took me approximately 1 hour to complete. The part that took the longest was waiting for the CloudFormation stacks to finish creating the EKS cluster and the node group, which took around 10–15 minutes each. While waiting, I explored the AWS Management Console and reviewed documentation, but the creation times still added up. Additionally, setting up the IAM access entry and understanding Kubernetes RBAC was a bit time-consuming, since it required a shift in thinking from just using AWS IAM alone.

What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. Companies and developers use Kubernetes to simplify operations, ensure application uptime, and improve resource utilization. It handles scheduling containers across clusters, restarts failed containers, and scales apps based on demand. With features like self-healing, service discovery, and automated rollouts, Kubernetes reduces the manual work needed to manage complex apps. It supports consistent environments across development, testing, and production. Kubernetes is cloud-agnostic, running on AWS, Azure, GCP, or onpremises. This portability, combined with its automation capabilities, makes Kubernetes a vital tool for modern DevOps and microservices architectures. Developers can focus more on building features and less on infrastructure.

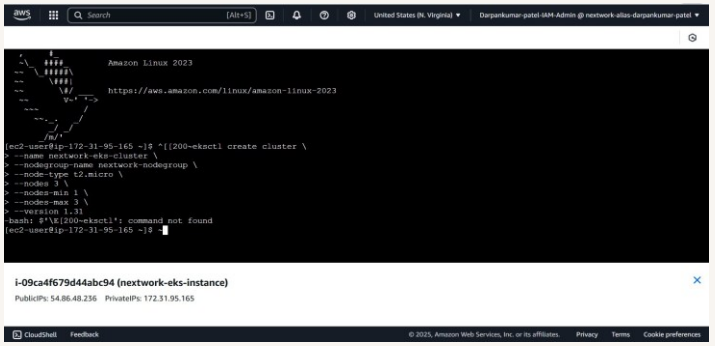
I used eksctl to create and manage my Amazon EKS (Elastic Kubernetes Service) cluster from the command line with a single, simple command. The create cluster command I ran defined the cluster name, node group name, instance type, number of nodes, Kubernetes version, and AWS region. Specifically, I used: eksctl create cluster \ --name nextwork-eks-cluster \ --nodegroup-name nextwork-nodegroup \ --nodetype t2.micro \ --nodes 3 \ --nodes-min 1 \ --nodes-max 3 \ --version 1.31 \ --region us-east-1 This command launched a fully managed Kubernetes control plane and a node group of EC2 instances to run the workloads. It abstracted away complex tasks like creating VPCs, security groups, IAM roles, and node provisioning. Using eksctl drastically simplified the cluster setup compared to manually configuring everything with the AWS CLI.

Darpan Patel

placeholder

I initially ran into two errors while using eksctl. The first one was because `eksctl` wasn't installed on the EC2 instance yet. When I tried to run the command, the terminal didn't recognize it as a valid command since the binary wasn't present.

I resolved this by downloading the latest `eksctl` release from GitHub and moving it to `/usr/local/bin` so it could be accessed system-wide. The second one was because the EC2 instance didn't have permission to create EKS resources. By default, EC2 instances don't have access to AWS services unless explicitly granted. I resolved this by creating an IAM role with `AdministratorAccess` permissions and attaching it to the EC2 instance. This allowed the instance to authenticate and interact with EKS and other AWS services successfully.

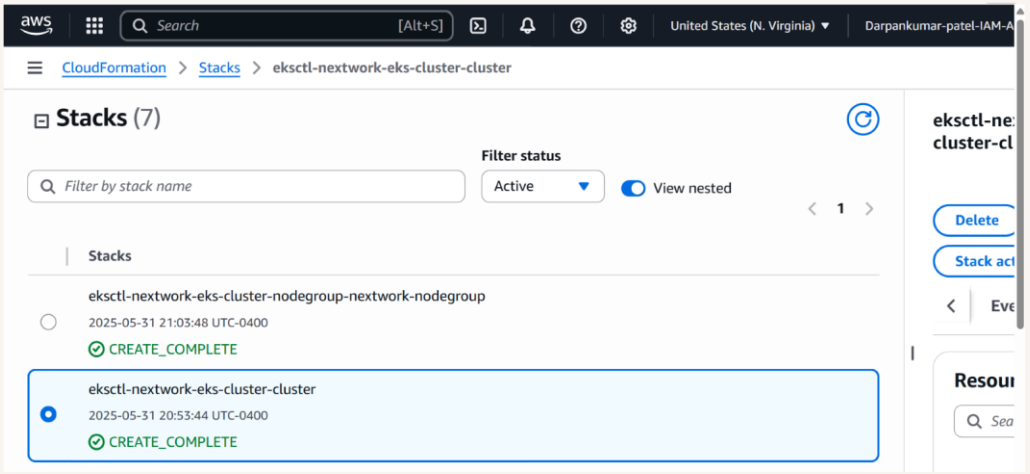


eksctl and CloudFormation

CloudFormation helped create my EKS cluster because `eksctl` uses it behind the scenes to automate the provisioning of all the AWS resources needed for the cluster. Instead of creating each component manually, `eksctl` generates a CloudFormation template and launches a stack to build everything. It created VPC resources because a Kubernetes cluster needs a private, secure network to manage communication between worker nodes and to safely expose services. CloudFormation set up components like subnets, route tables, NAT gateways, internet gateways, and security groups to ensure the EKS cluster is properly networked and accessible while maintaining security and scalability.

Darpan Patel

There was also a second CloudFormation stack for the node group, which is the collection of EC2 instances that will run the containers. The difference between a cluster and a node group is that the cluster represents the entire Kubernetes environment, including the control plane that manages the orchestration and the overall network setup, while the node group consists of the actual worker nodes (servers) that run the containerized applications. Separating these into two stacks allows for easier management and troubleshooting of the control plane and the worker nodes independently.

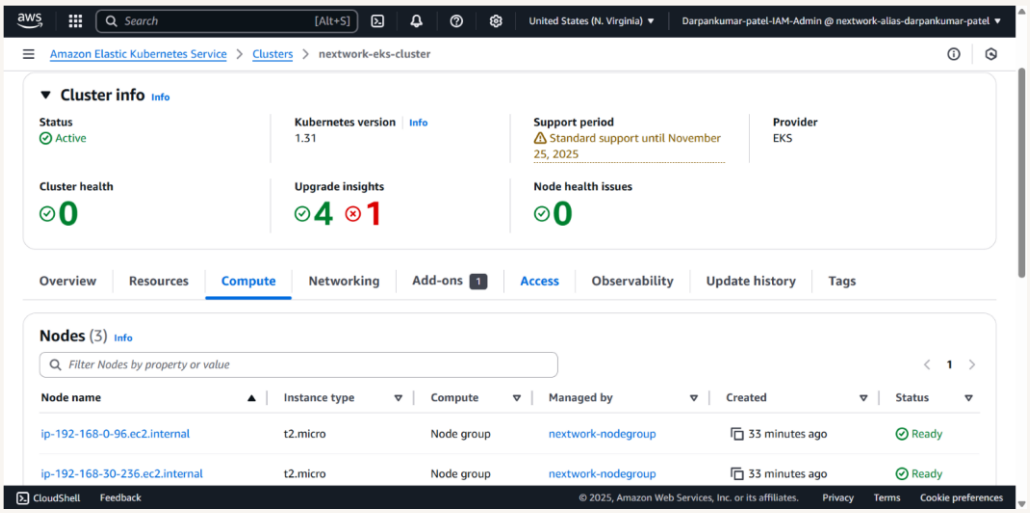


Darpan Patel

The EKS console

I had to create an IAM access entry in order to grant my AWS IAM user the necessary permissions to interact with the Kubernetes cluster inside EKS. An access entry is a way to link AWS IAM users with Kubernetes' Role-Based Access Control (RBAC), which controls access within the cluster itself. I set it up by selecting my IAM user's ARN in the EKS console, then attaching the AmazonEKSClusterAdminPolicy, which gives full administrative rights to manage and view all parts of the cluster, including the nodes. This ensures I can properly manage the cluster despite having AWS AdministratorAccess, which alone doesn't grant Kubernetes permissions.

It took about 10 to 15 minutes to create my cluster. Since I'll create this cluster again in the next project of this series, maybe this process could be sped up if I reuse an existing node group or leverage pre-configured CloudFormation templates to avoid creating all the networking resources from scratch each time.



EXTRA: Deleting nodes

Did you know you can find your EKS cluster's nodes in Amazon EC2? This is because each node in an EKS cluster is actually an EC2 instance running under the hood. Kubernetes uses the term "node" to refer to the servers that run your containers, and on AWS, these servers are EC2 instances. So when you create or manage node groups in EKS, you're actually launching and controlling EC2 instances that act as the worker machines for your Kubernetes workloads.

When I deleted my EC2 instances, Kubernetes automatically launched new nodes to replace them and maintain the desired number of nodes in the cluster. This is because Kubernetes is designed to maintain the cluster's desired state by continuously monitoring node health and availability—so when nodes are terminated, it uses the Auto Scaling Group behind the node group to spin up replacements.