

## **Task 17**

**Deploying Node.js App on  
Amazon EKS using Terraform**

**Umar Satti**

## Table of Contents

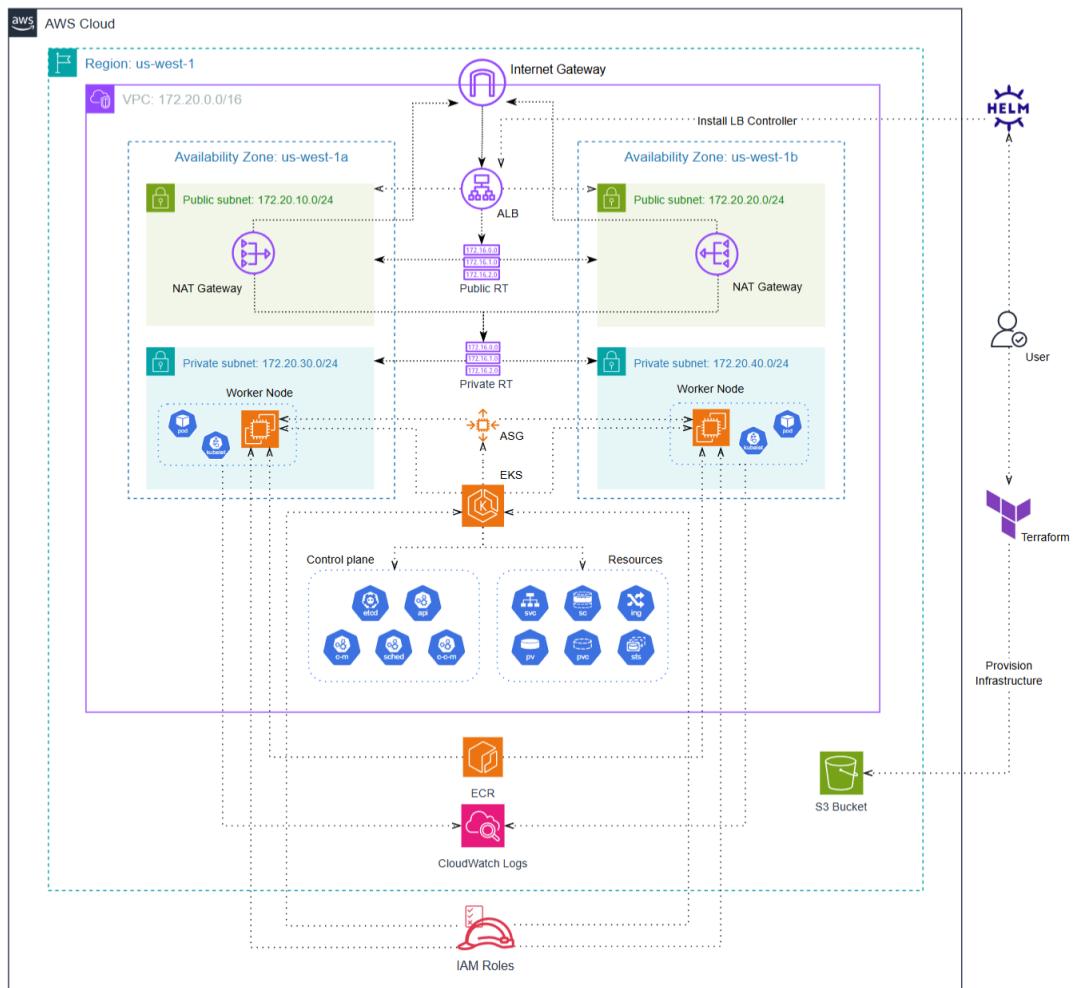
1. Task Description .....	3
2. Architecture Diagram .....	3
3. Project Structure.....	4
3.1 Application Files .....	4
3.2 Terraform Infrastructure Code.....	4
3.3 Terraform Commands .....	8
4. Validate Infrastructure on AWS .....	10
4.1 VPC and Networking Validation.....	10
4.2 IAM Validation.....	13
4.3 EKS Cluster Validation.....	18
4.4 Amazon ECR Validation .....	23
4.5 EC2 Instances (EKS Worker Nodes) Validation .....	23
4.6 Launch Template Validation.....	24
4.7 Auto Scaling Group Validation.....	24
4.8 CloudWatch Validation.....	25
5. Kubernetes Manifest and Configuration .....	30
5.1 StorageClass Configuration (storageclass.yaml) .....	30
5.2 StatefulSet Definition (statefulset.yaml) .....	31
5.3 Service Configuration (service.yaml) .....	32
5.4 Ingress Configuration (ingress.yaml).....	32
6. Post-Provisioning Kubernetes Deployment .....	34
6.1 Cluster Access and Initial Validation .....	34
6.2 Core System Components Validation .....	35
6.3 Storage Configuration .....	36
6.4 Container Image Management.....	36
6.5 Application Deployment.....	37
6.6 Service Configuration.....	38
6.7 Ingress and Load Balancer Configuration .....	39
6.8 Load Balancer Validation.....	39
6.9 Application Access Verification.....	41
7. Clean Up .....	42
8. Troubleshooting.....	45

# 1. Task Description

This project implements a fully automated AWS Kubernetes platform using Terraform and Amazon EKS. All core infrastructure components, including a custom VPC configuration, IAM roles and policies, and an EKS cluster with managed worker nodes, are provisioned using Terraform. EKS add-ons such as the AWS Load Balancer Controller, EBS CSI Driver, VPC CNI, and CloudWatch Observability are also configured.

A containerized Node.js application is deployed on the EKS cluster using Kubernetes manifests, including a StorageClass, StatefulSet, Service, and Ingress. Persistent storage is dynamically provisioned using Amazon EBS, while external access to the application is provided through an internet-facing Application Load Balancer managed by Kubernetes Ingress. CloudWatch Container Insights and log groups are used to collect metrics and logs from the cluster, nodes, and workloads, providing centralized observability.

# 2. Architecture Diagram



## 3. Project Structure

This project is organized into application source files and Terraform infrastructure code. The structure separates application logic, CI/CD configuration, and infrastructure provisioning for clarity and maintainability.

### 3.1 Application Files

These files define the Node.js application that is deployed on Worker node EC2s.

#### 3.1.1 app.js

- Main application entry point.
- Uses Express to serve static files and provide a /health endpoint.
- Listens on port 3000 or the PORT environment variable.

#### 3.1.2 package.json

- Defines application metadata, dependencies, and start scripts.
- Includes Express as a dependency.

#### 3.1.3 index.html

- Static page served by the application.
- Confirms successful deployment.

#### 3.1.4 Dockerfile

- Builds a lightweight Node.js container using the official Node 20 Alpine image.
- Installs dependencies, exposes port 3000, and starts the application.

### 3.2 Terraform Infrastructure Code

This directory contains Terraform configuration used to provision and manage all AWS infrastructure required for Jenkins, EC2, VPC, IAM, and CloudWatch. The setup follows a modular design to improve reusability and consistency.

#### 3.2.1 Root Terraform Files

These files act as the entry point and orchestrate all infrastructure modules.

##### **terraform.tf**

- Defines required Terraform providers (AWS, Kubernetes, Helm).
- Configures remote encrypted S3 backend with state locking.
- Sets AWS region and configures Kubernetes and Helm providers.

## **main.tf**

Serves as the central orchestration file and invokes the following Terraform modules:

- VPC: networking and security foundations
- IAM: roles and policies for EKS and add-ons
- ECR: container image repository
- EKS: Kubernetes cluster, node groups, and managed add-ons
- Add-ons: Helm-based Kubernetes controllers

## **variables.tf**

- Declares all input variables required by the root module.
- Groups variables logically by service (VPC, IAM, ECR, EKS, Add-ons).
- Enables environment flexibility without modifying Terraform code.

## **terraform.tfvars**

Supplies concrete values for all declared variables. Defines environment-specific configuration including:

- CIDR ranges and availability zones
- IAM role naming
- EKS version, node sizing, and add-on versions
- Allows reuse of the same code across environments.

### **3.2.2 Terraform Modules Directory**

#### **1. VPC Module**

The VPC module provisions the complete networking layer for the EKS cluster.

- Creates a custom VPC with DNS support enabled.
- Provisions:
  - Two public subnets
  - Two private subnets
  - Internet Gateway
  - Two NAT Gateways (one per AZ)
  - Four route tables (one for each subnet)
- Configures public and private route tables for internet and outbound access.
- Defines a shared security group for EKS worker nodes allowing:
  - Full intra-node communication
  - Outbound internet access
- Tags resources to integrate seamlessly with the EKS cluster.

## **2. IAM Module**

The IAM module provisions all roles and policies required by EKS, worker nodes, and cluster add-ons. This module also defines trust relationships for:

- EKS control plane
- EC2 worker nodes
- EKS Pod Identity

Creates IAM roles for:

- EKS cluster
- Managed node groups
- VPC CNI add-ons
- CloudWatch observability
- EBS CSI driver
- AWS Load Balancer Controller

## **3. ECR Module**

The ECR module provisions a private container image repository.

- Creates an Amazon ECR repository for application images.
- Enables image tag mutability configuration.
- Configures encryption at rest using AES-256.
- Outputs repository URI for use in Kubernetes deployments.

## **4. EKS Module**

The EKS module provisions the Kubernetes control plane, worker nodes, and managed add-ons.

### **eks/main.tf**

Creates an Amazon EKS cluster with both public and private API access. Enables control plane logging for operational visibility. Deploys a managed node group in private subnets with autoscaling. Associates a node security group and SSH access key.

Installs core EKS add-ons:

- VPC CNI
- CoreDNS
- kube-proxy
- CloudWatch Observability
- EBS CSI Driver
- Pod Identity Agent
- Configures IAM Pod Identity associations for add-ons.

## **eks/variables.tf**

Declares inputs for cluster configuration, networking, IAM roles, scaling, and add-on versions.

## **eks/outputs.tf**

Exposes cluster name, endpoint, and certificate authority data.

## **5. Add-ons Module**

The add-ons module installs Kubernetes controllers using Helm.

### **addons/main.tf**

- Deploys the AWS Load Balancer Controller via Helm.
- Configures cluster name, region, and VPC integration.
- Uses a Kubernetes service account mapped to IAM Pod Identity.



```
main.tf
1 # Install AWS Load Balancer Controller via Helm
2 resource "helm_release" "lb_controller" {
3     name      = "aws-load-balancer-controller"
4     repository = "https://aws.github.io/eks-charts"
5     chart      = "aws-load-balancer-controller"
6     namespace  = "kube-system"
7
8     set = [
9         {
10            name  = "clusterName"
11            value = var.cluster_name
12        },
13
14        {
15            name  = "serviceAccount.create"
16            value = "true"
17        },
18
19        {
20            name  = "serviceAccount.name"
21            value = "aws-load-balancer-controller"
22        },
23
24        {
25            name  = "region"
26            value = var.aws_region
27        },
28
29        {
30            name  = "vpcId"
31            value = var.vpc_id
32        }
33    ]
34 }
```

### **addons/variables.tf**

- Accepts cluster name, AWS region, and VPC ID.

```
variables.tf X
terraform > modules > addons > variables.tf > ...
1   variable "cluster_name" {
2     |   type = string
3   }
4
5   variable "aws_region" {
6     |   type = string
7   }
8
9   variable "vpc_id" {
10    |   type = string
11 }
```

### 3.2.3 Supporting Configuration

#### Load Balancer Controller IAM Policy (lbc-policy.json)

- The IAM policy for the AWS Load Balancer Controller is sourced directly from the official Kubernetes SIGs repository.
- Provides permissions required to manage Application Load Balancers, target groups, listeners, and security groups.
- Ensures alignment with supported controller versions and AWS best practices.

## 3.3 Terraform Commands

### 1. Terraform init

Initializes the Terraform working directory by downloading required providers and modules, configuring the remote backend, and preparing Terraform.

```
PS D:\Cloudelligent\Task-17\terraform> terraform init
Initializing the backend...
Initializing modules...
Initializing provider plugins...
- Reusing previous version of hashicorp/helm from the dependency lock file
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/kubernetes from the dependency lock file
- Using previously-installed hashicorp/kubernetes v3.0.1
- Using previously-installed hashicorp/helm v3.1.1
- Using previously-installed hashicorp/aws v6.28.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
PS D:\Cloudelligent\Task-17\terraform>
```

## 2. Terraform validate

Checks the Terraform configuration for syntax and logical errors without creating resources, ensuring the configuration is valid and consistent.

```
PS D:\Cloudelligent\Task-14\terraform> terraform validate
Success! The configuration is valid.

○ PS D:\Cloudelligent\Task-14\terraform>
```

## 3. Terraform plan

Creates an execution plan that previews the AWS resources Terraform will create and any changes that will be applied, allowing review before deployment.

## 4. Terraform apply

The terraform apply command provisions the AWS infrastructure as defined in the Terraform configuration. Upon confirmation, Terraform creates all required resources defined in VPC, IAM, EKS, ECR and Add-on modules.

```
Apply complete! Resources: 49 added, 0 changed, 0 destroyed.
○ PS D:\Cloudelligent\Task-17\terraform>
```

## 4. Validate Infrastructure on AWS

This section validates that all AWS resources created using Terraform are provisioned correctly and functioning as expected.

### 4.1 VPC and Networking Validation

#### 1. Verify VPC

In the AWS Console, navigate to **VPC** service. Select **Your VPCs** and verify the following:

- VPC created by Terraform exists.
- Correct IPv4 CIDR block as defined in Terraform.
- DNS hostnames and DNS resolution are enabled.
- VPC name matches the Terraform configuration.
- The following shows the VPC configuration:
  - **Name:** umarsatti-vpc
  - **VPC ID:** vpc-00b75666bbc3a1cb4
  - **IPv4 CIDR:** 172.20.0.0/16

The screenshot shows the AWS VPC dashboard. On the left, there's a sidebar with navigation links for VPC dashboard, AWS Global View, Filter by VPC, Virtual private cloud (with sub-links for Your VPCs, Subnets, Route tables, Internet gateways, Egress-only internet gateways, DHCP option sets, Elastic IPs, Managed prefix lists, NAT gateways, Peering connections, Route servers), Security (Network ACLs, Security groups), and PrivateLink and Lattice (Getting started, Endpoints, Endpoint services, Service networks). The main area is titled 'Your VPCs' and shows a table with one row. The table columns include VPC ID, Name, State, Encryption controls, Block Public Access, and IPv4 CIDR. The single row shows 'vpc-00b75666bbc3a1cb4' and 'umarsatti-vpc' as the Name. The state is 'Available'. The IPv4 CIDR is listed as '172.20.0.0/16'. Below this table, there's a detailed view for 'vpc-00b75666bbc3a1cb4 / umarsatti-vpc'. This view includes tabs for Details, Resource map, CIDRs, Flow logs, Tags, and Integrations. The 'Details' tab is selected and displays various configuration settings: VPC ID (vpc-00b75666bbc3a1cb4), State (Available), Main network ACL (acl-03c2b5392dbfe19e7), IPv6 CIDR (-), and many other fields like DNS resolution, Default VPC, Network Address Usage metrics, and Route 53 Resolver DNS Firewall rule groups, all of which are currently set to '-' or default values.

#### 2. Verify Subnets

Navigate to **Subnets** section in the VPC console. Confirm creation of public and private subnets. The following shows the subnets configuration:

##### Public Subnets

- **Names:** Public-Subnet-A & Public-Subnet-B
- **IPv4 CIDRs:** 172.20.10.0/24 & 172.20.20.0/24
- **Subnet IDs:** subnet-05215e5bcf8e3ede4 & subnet-09415a92f90887cad

## Private Subnets

- **Names:** Private-Subnet-A & Private-Subnet-B
- **IPv4 CIDRs:** 172.20.30.0/24 & 172.20.40.0/24
- **Subnet IDs:** subnet-01a55ce7a8a706414 & subnet-02641603a5370abec

The screenshot shows the AWS VPC Subnets console. On the left, there's a sidebar with 'Virtual private cloud' and 'Subnets' selected. The main area is titled 'Subnets (4) Info' with a search bar 'VPC : vpc-00b75666bbc3a1cb4'. A table lists four subnets:

Name	Subnet ID	State	VPC	Block Public...	IPv4 CIDR
Private-Subnet-B	subnet-02641603a5370abec	Available	vpc-00b75666bbc3a1cb4   umarsatti-vpc	Off	172.20.40.0/24
Public-Subnet-A	subnet-05215e5bcf8e3ede4	Available	vpc-00b75666bbc3a1cb4   umarsatti-vpc	Off	172.20.10.0/24
Private-Subnet-A	subnet-01a55ce7a8a706414	Available	vpc-00b75666bbc3a1cb4   umarsatti-vpc	Off	172.20.30.0/24
Public-Subnet-B	subnet-09415a92f90887cad	Available	vpc-00b75666bbc3a1cb4   umarsatti-vpc	Off	172.20.20.0/24

## 3. Verify Internet Gateway

Navigate to **Internet Gateways** in the VPC console.

- Confirm that Internet gateway exists and **attached** to VPC.
- The following shows the IGW configuration:
  - **Name:** umarsatti-igw
  - **Internet gateway ID:** igw-0b3ab5627a54c5cfe
  - **State:** Attached

The screenshot shows the AWS VPC Internet Gateways console. On the left, there's a sidebar with 'Virtual private cloud' and 'Internet gateways' selected. The main area is titled 'Internet gateways (1/1) Info' with a search bar 'VPC ID : vpc-00b75666bbc3a1cb4'. A table lists one internet gateway:

Name	Internet gateway ID	State	VPC ID	Owner
umarsatti-igw	igw-0b3ab5627a54c5cfe	Attached	vpc-00b75666bbc3a1cb4   umarsatti-vpc	504649076991

Below the table, there's a details section for 'igw-0b3ab5627a54c5cfe / umarsatti-igw' with tabs for 'Details' and 'Tags'. The 'Details' tab shows:

Internet gateway ID	State	VPC ID	Owner
igw-0b3ab5627a54c5cfe	Attached	vpc-00b75666bbc3a1cb4   umarsatti-vpc	504649076991

## 4. Verify NAT Gateway

Navigate to **NAT gateways** in the VPC console. Confirm that NAT Gateways exist in their respective public subnets. The following shows the NAT gateway configuration:

### NAT Gateway in Public subnet A:

- **Name:** Nat-EIP-Public-Subnet-A
- **Nat gateway ID:** nat-0873f7e437ed77e06
- **Elastic IP:** 52.9.78.230

## NAT Gateway in Public subnet B:

- **Name:** Nat-EIP-Public-Subnet-B
- **Nat gateway ID:** nat-0b2c1e4ec8fdbd3411
- **Elastic IP:** 52.52.200.142

The screenshot shows the AWS VPC dashboard with the 'NAT gateways' section selected. It displays two entries in a table:

Name	NAT gateway ID	Connectivity type	State	State message	Availability mode	Route table ID	Primary public IPv4 address
Nat-GW-Public-Subnet-B	nat-0b2c1e4ec8fdbd3411	Public	Available	-	Zonal	-	52.52.200.142
Nat-GW-Public-Subnet-A	nat-0873f7e437ed77e06	Public	Available	-	Zonal	-	52.9.78.230

## 5. Route Tables

Navigate to **Route Tables** in the VPC console. Confirm that public and private route tables are created. The following shows the configuration:

- **Public route table names:** Public-Subnet-A-RT & Public-Subnet-B-RT
- **Private route table names:** Private-Subnet-A-RT & Private-Subnet-B-RT

The screenshot shows the AWS VPC dashboard with the 'Route tables' section selected. It displays five entries in a table:

Name	Route table ID	Explicit subnet associations	Edge associations	Main	VPC
Public-Subnet-B-RT	rtb-0a20fd9571021f50f	subnet-09415a92f9087cad / Public-Subnet-B	-	No	vpc-00b75666bbc3a1cb4   umarsatti-vpc
Public-Subnet-A-RT	rtb-0221f67a4e03825	subnet-05215e5bcf8e3ede4 / Public-Subnet-A	-	No	vpc-00b75666bbc3a1cb4   umarsatti-vpc
Private-Subnet-A-RT	rts-031161c08bc4c6599	subnet-01a5ce7a8a706414 / Private-Subnet-A	-	No	vpc-00b75666bbc3a1cb4   umarsatti-vpc
Private-Subnet-B-RT	rts-04d688fa74145cd0	subnet-02641603a5370abec / Private-Subnet-B	-	No	vpc-00b75666bbc3a1cb4   umarsatti-vpc
-	rtb-0af564d3a8eca79d1	-	-	Yes	vpc-00b75666bbc3a1cb4   umarsatti-vpc

## 6. Security Groups

Navigate to **Security Groups** in the VPC console. Confirm that both the security groups are created. The following shows the security group configuration:

The screenshot shows the AWS VPC dashboard with the 'Security Groups' section selected. It displays four entries in a table:

Name	Security group ID	Security group name	VPC ID	Description
eks-cluster-sg-umarsatti-eks-cluster...	sg-02a2f75e877641039	eks-cluster-sg-umarsatti-eks-clu...	vpc-00b75666bbc3a1cb4	EKS created security group applied to ENI tha...
-	sg-018e65df0d16e6668	eks-remoteAccess-48cdde6-e0...	vpc-00b75666bbc3a1cb4	Security group for all nodes in the nodeGrou...
-	sg-067b1e9b43f353f	default	vpc-00b75666bbc3a1cb4	default VPC security group
umarsatti-eks-node-sg	sg-042dddebf6d4a50e8	umarsatti-eks-node-sg	vpc-00b75666bbc3a1cb4	Security group for all nodes in the cluster

## EKS worker node security group

- **Name:** umarsatti-eks-node-sg
- **Inbound rules:** Traffic on Port 22, 80, and 3000.
- **Outbound rules:** All traffic allowed (0.0.0.0/0)

The screenshot shows the AWS Security Groups console for the security group 'sg-042ddeb56fd4a50c8 - umarsatti-eks-node-sg'. The 'Inbound rules' tab is selected. There is one rule listed: 'sgr-0680e45051dee0861' which allows 'All traffic' from 'All' sources on 'All' ports. The 'Edit inbound rules' button is visible at the top right.

**Note:** Rest of the security groups were created automatically by EKS.

## 4.2 IAM Validation

This section validates that all IAM roles required for the EKS cluster, worker nodes, and add-ons have been created and configured correctly.

The screenshot shows the AWS IAM Roles page. A search bar at the top has 'uts-eks' entered, resulting in 6 matches. The table below lists the roles:

Role name	Trusted entities	Last activity
uts-eks-cloudwatch-addons-role	AWS Service: pods.eks	4 minutes ago
uts-eks-cluster-role	AWS Service: eks	13 minutes ago
uts-eks-ebs-addons-role	AWS Service: pods.eks	24 minutes ago
uts-eks-lbc-role	AWS Service: pods.eks	-
uts-eks-node-group-role	AWS Service: ec2	14 minutes ago

### 1. EKS Cluster IAM Role

Navigate to the **IAM** console and select **Roles**. Search for the role named **uts-eks-cluster-role**. Select this role and verify the following:

- The AWS managed policy **AmazonEKSClusterPolicy** is attached.
- The role is correctly referenced by the EKS cluster during creation.

This screenshot shows the AWS IAM Role configuration page for 'uts-eks-cluster-role'. The 'Permissions' tab is selected, displaying a table of attached policies. One policy, 'AmazonEKSClusterPolicy', is listed as an AWS managed policy. The ARN of the role is also visible.

This IAM Role should have trust relationship which allows the EKS service ([eks.amazonaws.com](https://eks.amazonaws.com)) to assume the role.

This screenshot shows the AWS IAM Role configuration page for 'uts-eks-cluster-role'. The 'Trust relationships' tab is selected, displaying a JSON representation of the trust policy. The policy document allows the principal 'eks.amazonaws.com' to assume the role using the 'sts:AssumeRole' action.

This confirms that the EKS control plane has the required permissions to manage cluster resources.

## 2. EKS Node Group IAM Role

In the **IAM** console, locate or search for the role named **uts-eks-node-group-role** and confirm the following.

- AWS managed policies are attached:
  - AmazonEC2ContainerRegistryReadOnly
  - AmazonEKS\_CNI\_Policy
  - AmazonEKSWorkerNodePolicy
  - AmazonSSMManagedInstanceCore
- The role is associated with the managed node group.

This screenshot shows the IAM Role configuration for 'uts-eks-node-group-role'. The 'Permissions' tab is active, displaying four attached policies:

- AmazonEC2ContainerRegistryReadOnly**: AWS managed, 7 entities
- AmazonEKS\_CNI\_Policy**: AWS managed, 17 entities
- AmazonEKSWorkerNodePolicy**: AWS managed, 14 entities
- AmazonSSMManagedInstanceCore**: AWS managed, 23 entities

This IAM Role should have trust relationship which allows the EC2 service ([ec2.amazonaws.com](https://ec2.amazonaws.com)) to assume the role.

```

1 = {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Effect": "Allow",
6       "Principal": {
7         "Service": "ec2.amazonaws.com"
8       },
9       "Action": "sts:AssumeRole"
10    }
11  ]
12}
  
```

This validates that worker nodes can join the cluster, pull container images, manage networking, and be accessed via AWS Systems Manager.

### 3. EBS CSI Driver IAM Role

Within the **IAM** console, search for **uts-eks-ebs-addons-role**. Verify the following:

- The AWS managed policy **AmazonEBSCSIDriverPolicy** is attached.
- The role is associated with the EBS CSI controller **service account**.

This screenshot shows the IAM role configuration for 'uts-eks-ebs-addons-role'. The 'Permissions' tab is active, displaying a single attached policy: 'AmazonEBSCSIDriverPolicy'. The ARN of the role is listed as arn:aws:iam::504649076991:role/uts-eks-ebs-addons-role.

This IAM Role should have trust relationship which allows EKS pods (**pods.eks.amazonaws.com**) to assume the role.

This screenshot shows the IAM role configuration for 'uts-eks-ebs-addons-role'. The 'Trust relationships' tab is active, displaying a JSON trust policy:

```
1: {
2:     "Version": "2012-10-17",
3:     "Statement": [
4:         {
5:             "Effect": "Allow",
6:             "Principal": {
7:                 "Service": "pods.eks.amazonaws.com"
8:             },
9:             "Action": [
10:                 "sts:TagSession",
11:                 "sts:AssumeRole"
12:             ]
13:         }
14:     ]
15: }
```

This ensures that Kubernetes workloads can dynamically provision and manage EBS volumes.

## 4. CloudWatch Observability IAM Role

Locate the IAM role named **uts-eks-cloudwatch-addons-role** and confirm:

- The **CloudWatchAgentServerPolicy** is attached.
- The role is linked to the CloudWatch observability add-on.

This screenshot shows the IAM Role configuration for 'uts-eks-cloudwatch-addons-role'. The 'Permissions' tab is active, displaying the attached policy 'CloudWatchAgentServerPolicy'. The ARN of the role is listed as arn:aws:iam::504649076991:role/uts-eks-cloudwatch-addons-role.

This IAM Role should have trust relationship which allows EKS pods (**pods.eks.amazonaws.com**) to assume the role.

This screenshot shows the IAM Role configuration for 'uts-eks-cloudwatch-addons-role'. The 'Trust relationships' tab is active, displaying the trust policy. The policy allows pods from the cluster to assume the role.

```
1 - {  
2 -   "Version": "2012-10-17",  
3 -   "Statement": [  
4 -     {  
5 -       "Effect": "Allow",  
6 -       "Principal": {  
7 -         "Service": "pods.eks.amazonaws.com"  
8 -       },  
9 -       "Action": [  
10 -         "sts:TagSession",  
11 -         "sts:AssumeRole"  
12 -       ]  
13 -     }  
14 -   ]  
15 - }
```

This validates that cluster and workload metrics and logs can be securely sent to Amazon CloudWatch.

## 5. Load Balancer Controller IAM Role

In the IAM console, search for the role **uts-eks-lbc-role** and verify:

- A custom inline policy named uts-aws-lb-controller-iam-policy is attached.
- The policy grants permissions required to create and manage AWS Application Load Balancers.

**uts-eks-lbc-role** [Info](#)

**Summary**

**Creation date**  
January 13, 2026, 19:31 (UTC+05:00)

**Last activity**  
-

**ARN**  
arn:aws:iam::504649076991:role/uts-eks-lbc-role

**Maximum session duration**  
1 hour

**Permissions** **Trust relationships** **Tags** **Last Accessed** **Revoke sessions**

**Permissions policies (1) [Info](#)**  
You can attach up to 10 managed policies.

**Filter by Type**  
All types

Policy name	Type	Attached entities
<a href="#">uts-aws-lb-controller-iam-policy</a>	Customer managed	1

**Permissions boundary (not set)**

This IAM Role should have trust relationship which allows EKS pods (**pods.eks.amazonaws.com**) to assume the role.

**uts-eks-lbc-role** [Info](#)

**Summary**

**Creation date**  
January 13, 2026, 19:31 (UTC+05:00)

**Last activity**  
-

**ARN**  
arn:aws:iam::504649076991:role/uts-eks-lbc-role

**Maximum session duration**  
1 hour

**Permissions** **Trust relationships** **Tags** **Last Accessed** **Revoke sessions**

**Trusted entities**

Entities that can assume this role under specified conditions.

```
1: {
2:     "Version": "2012-10-17",
3:     "Statement": [
4:         {
5:             "Effect": "Allow",
6:             "Principal": {
7:                 "Service": "pods.eks.amazonaws.com"
8:             },
9:             "Action": [
10:                 "sts:TagSession",
11:                 "sts:AssumeRole"
12:             ]
13:         }
14:     ]
15: }
```

This confirms that the AWS Load Balancer Controller can provision and manage load balancers for Kubernetes services.

## 4.3 EKS Cluster Validation

This section validates that the Amazon EKS cluster has been provisioned correctly, including compute, networking, add-ons, and observability.

### 1. EKS Cluster Status

Navigate to **Amazon EKS** within the **AWS Console**. Select **Clusters** located on the left navigation bar. Click on **umarsatti-eks-cluster**. Verify the following:

- Status:** Active
- Kubernetes version:** 1.33
- Platform:** EKS managed
- Cluster IAM role:** uts-eks-cluster-role
- API endpoint:** Public and private access enabled (with URL)
- OIDC provider URL:** Configured
- Certificate authority:** Valid and defined
- Support type:** Standard support

This confirms the EKS control plane is healthy and operational.

## 2. Managed Node Group Validation

Navigate to **Cluster** within the EKS console. Click on **Compute** tab. Verify the following:

- Managed node group named **umarsatti-node-group**
- Two active worker nodes managed by the same node group
- Status:** Ready
- Instance type:** t3.medium
- AMI:** Amazon Linux 2023
- Subnets:** Private subnets in separate Availability Zones

The screenshot shows the AWS EKS Compute tab with the following details:

- Nodes (2) Info**: Two nodes are listed:
  - ip-172-20-30-143.us-west-1.compute.internal: t3.medium, Node group: umarsatti-node-group, Created: 7 hours ago, Status: Ready
  - ip-172-20-40-81.us-west-1.compute.internal: t3.medium, Node group: umarsatti-node-group, Created: 7 hours ago, Status: Ready
- Node groups (1) Info**: One node group is listed:
  - umarsatti-node-group: Desired size: 2, AMI release version: 1.33.5-20260107, Launch template: -, Status: Active

## Detailed view of Worker Nodes:

Detailed view of the first worker node (ip-172-20-30-143.us-west-1.compute.internal):

- Details**:
 

Status: Ready	Kernel version: 6.12.58-82.121.amzn2023.x86_64	Created: an hour ago
Last transition time: an hour ago	Node group: umarsatti-node-group	Container runtime: containerd//2.1.5
OS (Architecture): linux (amd64)	Instance: i-0a375998137853b7a	Kubelet version: v1.33.5-eks-ecaa3a6
OS image: Amazon Linux 2023.9.20251208	Node auto repair: -	Node health issues: Info
Instance type: t3.medium		

Detailed view of the second worker node (ip-172-20-40-81.us-west-1.compute.internal):

- Details**:
 

Status: Ready	Kernel version: 6.12.58-82.121.amzn2023.x86_64	Created: an hour ago
Last transition time: an hour ago	Node group: umarsatti-node-group	Container runtime: containerd//2.1.5
OS (Architecture): linux (amd64)	Instance: i-06074daace86e5edf	Kubelet version: v1.33.5-eks-ecaa3a6
OS image: Amazon Linux 2023.9.20251208	Node auto repair: -	Node health issues: Info
Instance type: t3.medium		

## 3. Workload and System Pod Validation

Navigate to **Cluster** within the EKS console. Click on **Resources** tab. Select Pods and verify that system pods are running in the following namespaces:

- **kube-system:**
  - CoreDNS
  - aws-node (VPC CNI)
  - kube-proxy
  - EBS CSI controller
  - AWS Load Balancer Controller

- **amazon-cloudwatch**
  - CloudWatch Agent

Name	Created	Status
amazon-cloudwatch-observability-controller-manager-59555dfdgx12	36 minutes ago	Running
aws-load-balancer-controller-6486cccd8ff-7nj9f	30 minutes ago	Running
aws-load-balancer-controller-6486cccd8ff-lgs6b	30 minutes ago	Running
aws-node-lztjf	35 minutes ago	Running
aws-node-mhsnv	35 minutes ago	Running
cloudwatch-agent-tm4tl	35 minutes ago	Running
cloudwatch-agent-tz2w9	35 minutes ago	Running
coredns-dc7c47f6c-7v8k7	36 minutes ago	Running
coredns-dc7c47f6c-gpg74	36 minutes ago	Running
ebs-csi-controller-68f5f4f467-lsdnn	36 minutes ago	Running

All pods should be in **Running** state, confirming normal cluster operations.

## 4. EKS Add-ons Validation

Navigate to **Cluster** within the EKS console. Click on **Add-ons** tab and verify that the following add-ons are installed and in **Active** state:

- CoreDNS
- VPC CNI
- kube-proxy
- Amazon EBS CSI Driver
- Amazon CloudWatch Observability
- EKS Pod Identity Agent

Add-on	Category	Status	Version	EKS Pod Identity	IAM role for service account (IRSA)
CoreDNS	networking	Active	v1.12.4-eksbuild.1	Not required	Not required
Amazon EBS CSI Driver	storage	Active	v1.54.0-eksbuild.1	Not set	arn:aws:iam::504649076991:role/uts-eks-ebs-addons-role <a href="#">View in IAM</a>
Amazon CloudWatch Observability	observability	Active	v5.0.0-eksbuild.1	Not set	arn:aws:iam::504649076991:role/uts-eks-cloudwatch-addons-role <a href="#">View in IAM</a>
Amazon EKS Pod Identity Agent	security	Active	v1.3.10-eksbuild.2	Not required	Not required
kube-proxy	networking	Active	v1.33.5-eksbuild.2	Not required	Not required
Amazon VPC CNI	networking	Active	v1.21.1-eksbuild.1	Not set	Not set

Confirm IAM roles are associated with add-ons that require AWS permissions such as **EBS CSI Driver** and **CloudWatch Observability**.

## 5. Control Plane Logging

Navigate to **Cluster** within the EKS console. Click on **Observability** tab. Under **Control plane logs**, it should display the following log types enabled:

- API server
- Audit
- Authenticator
- Controller manager
- Scheduler

Log Type	Action
API server	<a href="#">View api logs</a>
Audit	<a href="#">View audit logs</a>
Authenticator	<a href="#">View authenticator logs</a>
Controller manager	<a href="#">View controllerManager logs</a>
Scheduler	<a href="#">View scheduler logs</a>

This confirms that control plane logs are being sent to Amazon CloudWatch.

**Note:** Click the link under each log type to view them in CloudWatch console.

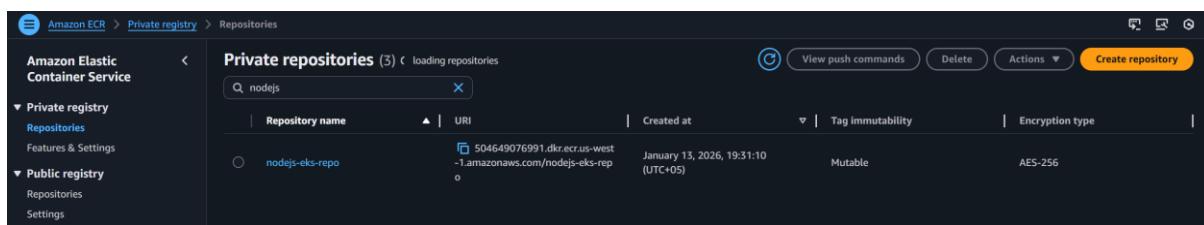
## Validation Summary

- EKS cluster is **Active** and **healthy**
- Node group is **highly available** and **private**
- All system workloads and add-ons are **running**
- IAM, logging, and networking are correctly configured

## 4.4 Amazon ECR Validation

Navigate to **Amazon ECR** and click **Repositories** under **Private registry**. Select **nodejs-eks-repo**. Verify the following:

- **Repository type:** Private
- **Region:** us-west-1
- **Encryption:** AES-256 (AWS managed)
- **Tag mutability:** Mutable



The screenshot shows the Amazon ECR console interface. On the left, there's a navigation sidebar with 'Amazon Elastic Container Service' at the top, followed by 'Private registry' and 'Public registry'. Under 'Private registry', there are 'Repositories' and 'Features & Settings'. The main area is titled 'Private repositories (3)' and shows a table with one item: 'nodejs-eks-repo'. The table columns include 'Repository name', 'URI', 'Created at', 'Tag immutability', and 'Encryption type'. The repository details are: nodejs-eks-repo, 504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-eks-repo, January 13, 2026, 19:31:10 (UTC+05), Mutable, AES-256. There are buttons for 'View push commands', 'Delete', 'Actions', and 'Create repository'.

This repository is used to store Docker images for the Node.js application.

EKS worker nodes can pull images using the IAM role **uts-eks-node-group-role** with the **AmazonEC2ContainerRegistryReadOnly** policy.

## 4.5 EC2 Instances (EKS Worker Nodes) Validation

Navigate to **EC2 console** and click **Instances**. Filter by the EKS cluster or node group or VPC ID.

Verify the following:

- **Total instances:** 2
- **Instance type:** t3.medium
- **State:** Running
- **Subnets:** Private subnets across two Availability Zones
- **Public IP:** None assigned
- **IAM role:** uts-eks-node-group-role
- **Security groups:** umarsatti-eks-node-sg & EKS-managed SGs

The screenshot shows the AWS EC2 Instances page with the following details:

- Instances (2) Info**
- Filter: Instance state = running, VPC ID = vpc-00b75666bbc3a1cb4
- Table headers: Name, Instance ID, Instance state, Instance type, Status check, Alarm status, Availability Zone, Public IPv4 DNS, Public IPv4 IP.
- Table data:
  - i-0a375998157853b... (Running, t3.medium, 3/3 checks passed, us-west-1a, -, -)
  - i-06074daace86e5edf (Running, t3.medium, 3/3 checks passed, us-west-1b, -, -)

This confirms worker nodes are privately deployed, healthy, and managed by EKS.

## 4.6 Launch Template Validation

Navigate to **EC2 console** and click **Launch Templates**. Select the template associated with the node group.

Verify the following:

- Launch template name:** eks-\*
- AMI:** Amazon Linux 2023
- Instance type:** t3.medium
- IAM instance profile:** uts-eks-node-group-role
- Security groups:** EKS-managed SGs + custom node SG
- Public IP:** Disabled

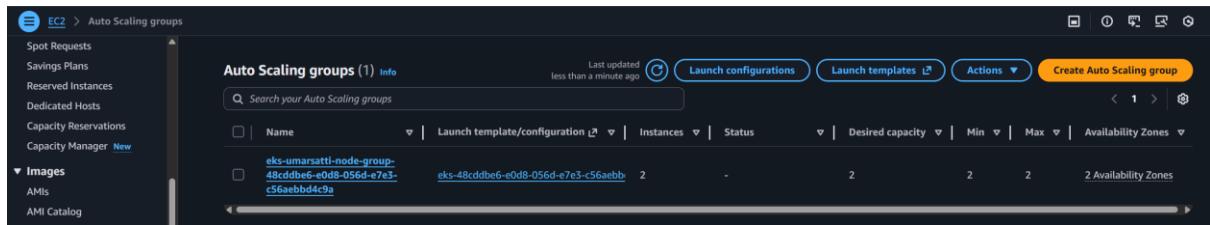
The screenshot shows the AWS EC2 Launch Templates page with the following details:

- Launch Templates (1/1) Info**
- Table headers: Launch Template ID, Launch Template Name, Default Version, Latest Version, Create Time, Created By, Mana.
- Table data:
  - lt-03cf801ce568f1861 (eks-48cddbe6-e0d8-056d-e7e3-c56aeabb4c9a, lt-03cf801ce568f1861, 1, 1, 2026-01-13T14:41:19.000Z, arn:aws:sts::504649076991:ass..., false)
- eks-48cddbe6-e0d8-056d-e7e3-c56aeabb4c9a (lt-03cf801ce568f1861)**
- Launch template details**
- Details:
  - Launch template ID: lt-03cf801ce568f1861
  - Launch template name: eks-48cddbe6-e0d8-056d-e7e3-c56aeabb4c9a
  - Default version: 1
  - Owner: arn:aws:sts::504649076991:assumed-role/AWSServiceRoleForAmazonEKSNodegroup/EKS

The launch template ensures all worker nodes are launched with a consistent and secure configuration.

## 4.7 Auto Scaling Group Validation

Navigate to **EC2 console** and click **Auto Scaling Groups**. Select the EKS-managed Auto Scaling Group.

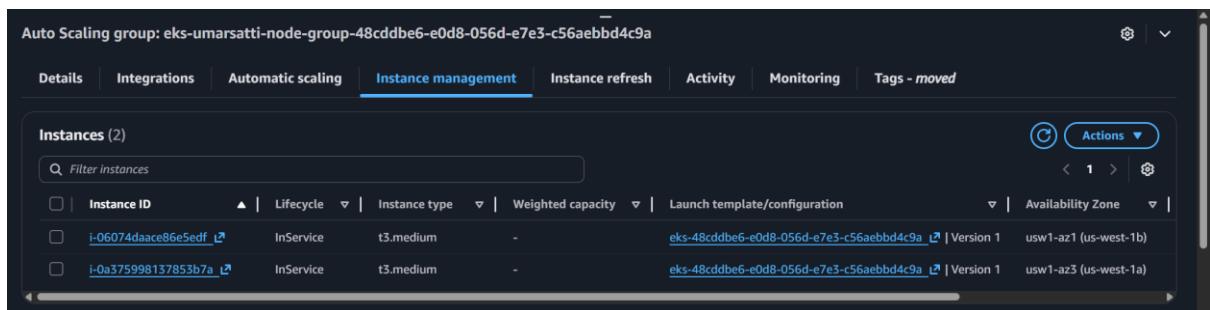


Verify the following:

- **Desired / Min / Max capacity:** 2 / 2 / 2
- **Running instances:** 2
- **Availability Zones:** us-west-1a and us-west-1b
- **Health status:** Healthy
- **Launch template:** Correct version attached

Navigate to the **Instance management** tab and confirm:

- All instances are **InService**
- Unhealthy instances would be automatically replaced



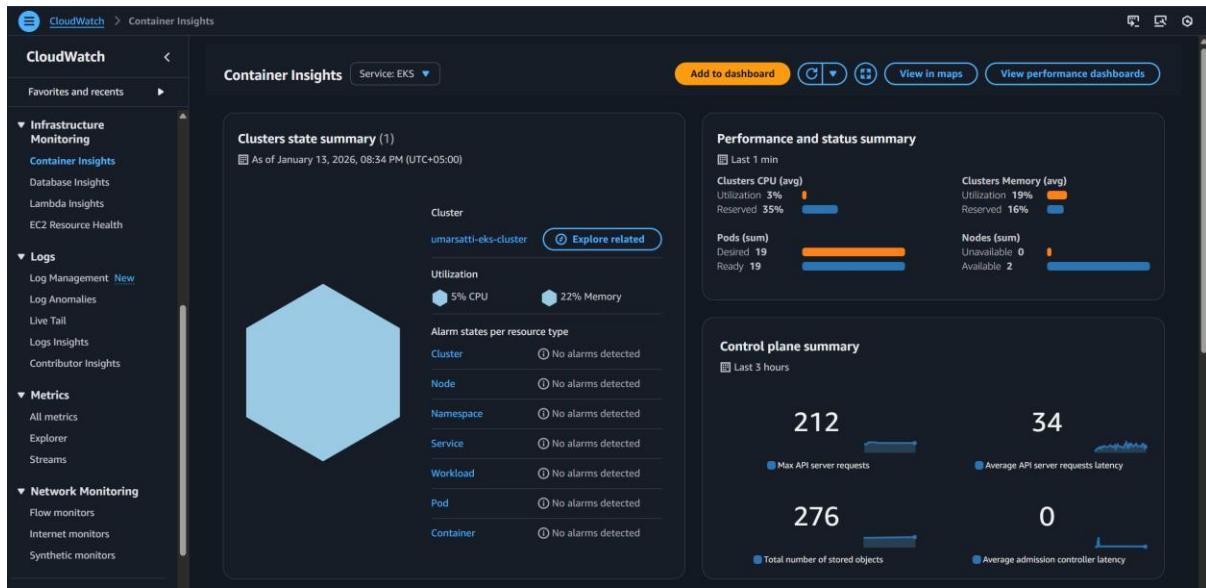
This validates automatic lifecycle management and high availability of worker nodes.

## 4.8 CloudWatch Validation

### 4.8.1 CloudWatch Container Insights

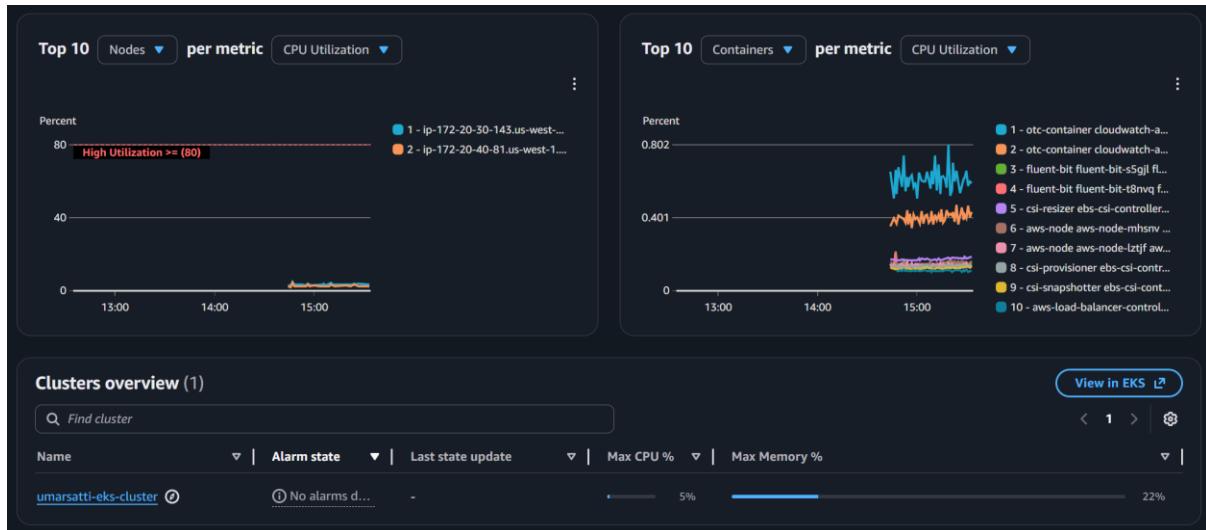
Navigate to **CloudWatch console** and click **Container Insights** under Infrastructure Monitoring. Select the EKS cluster **umarsatti-eks-cluster**.

#### Cluster Dashboard Validation



Verify the following from the **EKS Cluster dashboard**:

- Cluster status:** Active
- Nodes available:** Displays the expected number of worker nodes
- Pods ready:** All or most pods should be in a *Running* and *Ready* state
- CPU utilization:** Dashboard should display cluster CPU usage metrics
- Memory utilization:** Dashboard should display cluster memory usage metrics



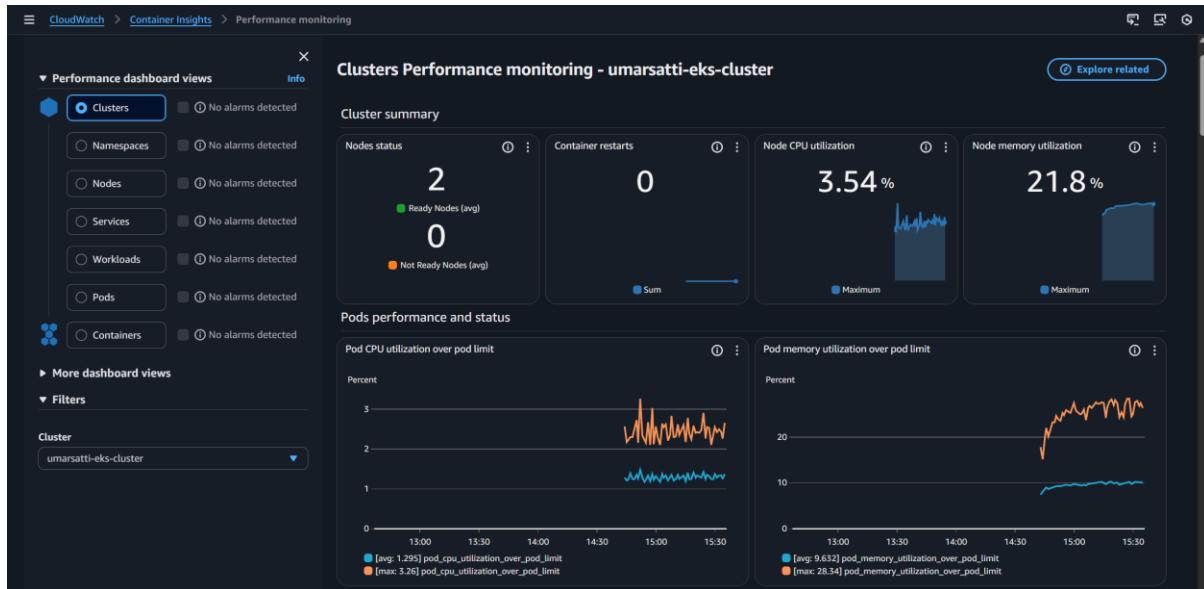
The presence of live CPU and memory metrics with no active alarms indicates normal cluster operation and sufficient capacity.

## Performance Monitoring

In the **Container Insights** dashboard, click **Performance monitoring** button on the top right and review the following:

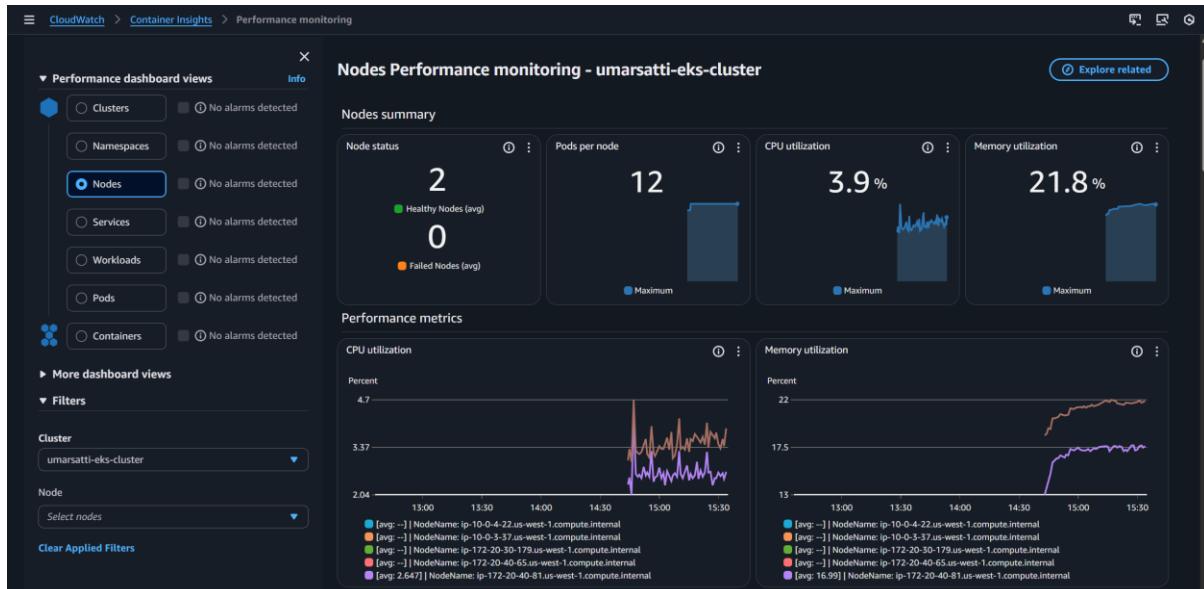
## Clusters View

- Ready node count is visible
- Container restart metrics are displayed
- CPU and memory utilization charts are populated



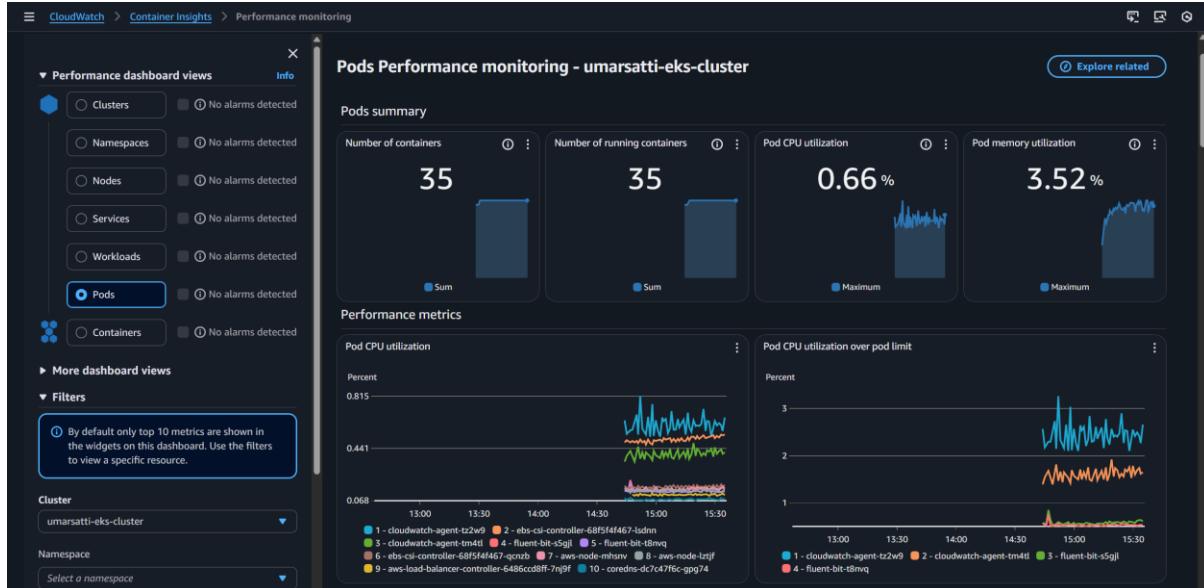
## Nodes View

- All nodes report a *Healthy* status
- Pod distribution across nodes is visible
- Node-level CPU and memory metrics are displayed



## Pods View

- All containers are in a **Running** state
- Pod-level CPU and memory utilization metrics are visible
- Resource usage appears within configured limits



These dashboards confirm that monitoring is functioning correctly and that cluster, node, and pod performance metrics are being collected.

## 4.8.2 CloudWatch Log Groups

In the **CloudWatch console**, click **Log Management** under Logs. Search for **umarsatti-eks-cluster** log groups.

### Log Groups Validation

Verify the following log groups exist and are receiving logs:

1. **/aws/eks/umarsatti-eks-cluster/cluster**
  - EKS control plane logs (API server, scheduler, controllers, authenticator)
2. **/aws/containerinsights/umarsatti-eks-cluster/performance**
  - Cluster, node, pod, and container performance metrics
3. **/aws/containerinsights/umarsatti-eks-cluster/application**
  - Application container stdout/stderr logs (via Fluent Bit)
4. **/aws/containerinsights/umarsatti-eks-cluster/dataplane**
  - Kubelet, container runtime, and CNI logs

## 5. /aws/containerinsights/umarsatti-eks-cluster/host

- o EC2 worker node system and OS logs

The screenshot shows the AWS CloudWatch Log Management interface. On the left, there's a sidebar with 'CloudWatch' selected, followed by 'Favorites and recents', 'Logs' (with 'Log Management New' highlighted), 'Metrics', and 'Metrics' again. The main area is titled 'Log groups (18)' and contains a search bar with 'umarsatti' and a dropdown menu for 'Exact match'. Below the search bar is a table with columns: 'Log group', 'Log class', 'Anomaly d...', 'Deletion pr...', 'Data pro...', 'Sensitive...', and 'Retention'. There are five entries listed:

Log group	Log class	Anomaly d...	Deletion pr...	Data pro...	Sensitive...	Retention
/aws/containerinsights/umarsatti-eks-cluster/application	Standard	Configure	Off	-	-	Never
/aws/containerinsights/umarsatti-eks-cluster/dataplane	Standard	Configure	Off	-	-	Never
/aws/containerinsights/umarsatti-eks-cluster/host	Standard	Configure	Off	-	-	Never
/aws/containerinsights/umarsatti-eks-cluster/performance	Standard	Configure	Off	-	-	Never

All log groups are:

- **Log class:** Standard
- **Retention:** Never expire
- **Status:** Actively receiving logs

## 5. Kubernetes Manifest and Configuration

This section documents the Kubernetes YAML manifest files used to deploy the application on Amazon EKS. Each manifest defines a specific Kubernetes resource and its intended behavior. Screenshots of these files are captured to demonstrate configuration correctness prior to deployment.

### 5.1 StorageClass Configuration (storageclass.yaml)

The StorageClass defines how persistent storage is dynamically provisioned for the application using Amazon EBS.

```
! storageclass.yaml X
!
storageclass.yaml
1  apiVersion: storage.k8s.io/v1
2  kind: StorageClass
3  metadata:
4    name: ebs-sc
5  provisioner: ebs.csi.aws.com
6  reclaimPolicy: Delete
7  volumeBindingMode: WaitForFirstConsumer
8  parameters:
9    type: gp3
10
```

#### Key Configuration:

- Provisioner: ebs.csi.aws.com (AWS EBS CSI Driver)
- Volume Type: gp3
- Reclaim Policy: Delete
- Volume Binding Mode: WaitForFirstConsumer

#### Purpose:

- Enables dynamic provisioning of Amazon EBS volumes
- Ensures volumes are created only after pods are scheduled, aligning storage with availability zones
- Automatically deletes EBS volumes when their PersistentVolumeClaims (PVCs) are removed

This **StorageClass** is later referenced by the **StatefulSet** to provide persistent storage for each pod.

## 5.2 StatefulSet Definition (statefulset.yaml)

The StatefulSet manages the deployment of the Node.js application pods with stable identities and persistent storage.

```
! statefulset.yaml ×
!
statefulset.yaml
1  apiVersion: apps/v1
2  kind: StatefulSet
3  metadata:
4    name: node-app
5  spec:
6    serviceName: node-app
7    replicas: 2
8    selector:
9      matchLabels:
10     app: node-app
11    template:
12      metadata:
13        labels:
14          app: node-app
15      spec:
16        topologySpreadConstraints:
17          - maxSkew: 1
18            topologyKey: topology.kubernetes.io/zone
19            whenUnsatisfiable: DoNotSchedule
20        labelSelector:
21          matchLabels:
22            app: node-app
23        containers:
24          - name: node-app
25            image: 504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-eks-repo:latest
26            ports:
27              - containerPort: 3000
28            volumeMounts:
29              - name: data
30                mountPath: /usr/src/app/data
31        volumeClaimTemplates:
32          - metadata:
33            name: data
34            spec:
35              accessModes: ["ReadWriteOnce"]
36              storageClassName: ebs-sc
37            resources:
38              requests:
39                storage: 4Gi
40
```

### Key Configuration:

- Replicas: 2
- Pod Naming: node-app-0, node-app-1
- Container Image: Stored in Amazon ECR
- Container Port: 3000
- Topology Spread Constraints: Evenly distributes pods across availability zones
- Volume Mount: /usr/src/app/data

### Persistent Storage:

- Each pod defines a volumeClaimTemplate
- Requests 4 GiB of storage per pod
- Uses the ebs-sc StorageClass
- Access mode: ReadWriteOnce

### Purpose:

- Provides stable pod identities and ordered startup
- Automatically creates PersistentVolumeClaims (PVCs) for each pod
- Triggers dynamic provisioning of PersistentVolumes (PVs) backed by Amazon EBS
- Ensures data persistence across pod restarts and rescheduling

## 5.3 Service Configuration (service.yaml)

The Service provides stable internal networking and exposes the application to other Kubernetes components.

```
! service.yaml X
!
! service.yaml
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: node-app-service
5  spec:
6    selector:
7      app: node-app
8    ports:
9      - protocol: TCP
10     port: 80
11     targetPort: 3000
12   type: NodePort
13
```

### Key Configuration:

- Type: NodePort
- Service Port: 80
- Target Port: 3000
- Selector: Matches pods labeled app: node-app

### Purpose:

- Enables internal traffic routing to StatefulSet pods
- Exposes the application on a static port on each worker node
- Acts as the backend service for the Ingress and Application Load Balancer

## 5.4 Ingress Configuration (ingress.yaml)

The Ingress resource integrates with the AWS Load Balancer Controller to expose the application externally.

```

! ingress.yaml ×
  ! ingress.yaml
  1   apiVersion: networking.k8s.io/v1
  2   kind: Ingress
  3   metadata:
  4     name: node-app-ingress
  5     annotations:
  6       alb.ingress.kubernetes.io/scheme: internet-facing
  7       alb.ingress.kubernetes.io/target-type: ip
  8       alb.ingress.kubernetes.io/healthcheck-path: /health
  9   spec:
10     ingressClassName: alb
11     rules:
12       - http:
13         paths:
14           - path: /
15             pathType: Prefix
16             backend:
17               service:
18                 name: node-app-service
19                 port:
20                   number: 80
21

```

### Key Configuration:

- Ingress Class: alb
- Scheme: Internet-facing
- Target Type: IP
- Health Check Path: /health
- Routing: Forwards traffic from / to node-app-service on port 80

### Purpose:

- Automatically provisions an AWS Application Load Balancer (ALB)
- Routes external HTTP traffic to the NodePort service
- Enables scalable and highly available access to the application

## 5.5 Resource Creation Summary

When these manifests are applied, Kubernetes automatically creates the following resources:

- Pods: Two application pods managed by the StatefulSet
- PersistentVolumeClaims (PVCs): One PVC per pod
- PersistentVolumes (PVs): Dynamically provisioned EBS volumes via CSI
- Service: Internal and NodePort-based traffic routing
- Ingress & ALB: External access via AWS Application Load Balancer

This layered resource creation ensures a fully functional, scalable, and persistent application architecture on EKS.

# 6. Post-Provisioning Kubernetes Deployment

This section validates the successful deployment of a Node.js application on the EKS cluster after infrastructure provisioning. It covers cluster access, node readiness, storage configuration, container image management, workload deployment, networking, and external access through an AWS Application Load Balancer.

## 6.1 Cluster Access and Initial Validation

### 1. Configure kubectl Access

Before interacting with the cluster, the local kubeconfig is updated to include the EKS cluster endpoint and authentication details.

- `aws eks update-kubeconfig --region us-west-1 --name umarsatti-eks-cluster`

This command retrieves cluster metadata from AWS and configures **kubectl** to authenticate using AWS IAM credentials.

```
● PS D:\Cloudelligent\Task-17> aws eks update-kubeconfig --region us-west-1 --name umarsatti-eks-cluster
  Updated context arn:aws:eks:us-west-1:504649076991:cluster/umarsatti-eks-cluster in C:\Users\umars\.kube\config
○ PS D:\Cloudelligent\Task-17>
```

#### Validation:

- kubeconfig updated successfully
- kubectl context set to umarsatti-eks-cluster
- The cluster is accessible for management operations.

### 2. Verify Worker Nodes

- `kubectl get nodes -o wide`

```
PS D:\Cloudelligent\Task-17> kubectl get nodes -o wide
NAME           STATUS  ROLES   AGE    VERSION      INTERNAL-IP     EXTERNAL-IP   OS-IMAGE          KERNEL-VERSION   CONTAINER-RUNTIME
ip-172-30-38-143.us-west-1.compute.internal   Ready   <none>   87m   v1.33.5-eks-ecaa3a6  172.30.30.143   <none>        Amazon Linux 2023.9.20251208  6.12.58-82.121.amzn2023.x86_64  containerd://2.1.5
ip-172-30-40-81.us-west-1.compute.internal   Ready   <none>   87m   v1.33.5-eks-ecaa3a6  172.30.40.81   <none>        Amazon Linux 2023.9.20251208  6.12.58-82.121.amzn2023.x86_64  containerd://2.1.5
PS D:\Cloudelligent\Task-17>
```

#### Validation Checks:

- All worker nodes appear in the output
- Node status is **Ready**
- Nodes have no public IPs (private subnet deployment)
- Nodes report the expected Kubernetes version and container runtime

This confirms that the managed node group has successfully joined the cluster and is ready to schedule workloads.

## 6.2 Core System Components Validation

### 1. Verify DaemonSets

- `kubectl get daemonsets -A`

DaemonSets ensure critical services run on every worker node. The following must be running with desired replicas equal to available replicas:

- **aws-node**: VPC CNI for pod networking
- **kube-proxy**: Service networking and routing
- **ebs-csi-node**: EBS volume attachment
- **cloudwatch-agent**: Metrics collection
- **fluent-bit**: Log forwarding
- **eks-pod-identity-agent**: IAM role assumption for pods

NAMESPACE	NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
amazon-cloudwatch	cloudwatch-agent	2	2	2	2	2	kubernetes.io/os=linux	86m
amazon-cloudwatch	cloudwatch-agent-windows	0	0	0	0	0	kubernetes.io/os=windows	86m
amazon-cloudwatch	cloudwatch-agent-windows-container-insights	0	0	0	0	0	kubernetes.io/os=windows	86m
amazon-cloudwatch	dcm-exporter	0	0	0	0	0	kubernetes.io/os=linux	86m
amazon-cloudwatch	fluent-bit	2	2	2	2	2	kubernetes.io/os=linux	87m
amazon-cloudwatch	fluent-bit-windows	0	0	0	0	0	kubernetes.io/os=windows	87m
amazon-cloudwatch	neuron-monitor	0	0	0	0	0	kubernetes.io/os=linux	86m
kube-system	aws-node	2	2	2	2	2	<none>	89m
kube-system	ebs-csi-node	2	2	2	2	2	kubernetes.io/os=linux	87m
kube-system	ebs-csi-node-windows	0	0	0	0	0	kubernetes.io/os=windows	87m
kube-system	eks-pod-identity-agent	2	2	2	2	2	<none>	87m
kube-system	kube-proxy	2	2	2	2	2	<none>	89m

#### Validation:

All required DaemonSets are running on both nodes with no missing pods. Node-level services are functioning correctly.

### 2. Verify System Pods

- `kubectl get pods -n kube-system`

#### Key Observations:

- All pods are in **Running** state
- No container restarts observed
- High availability achieved for critical components (CoreDNS, EBS CSI controller, Load Balancer Controller)

```

● PS D:\Cloudelligent\Task-17> kubectl get pods -n kube-system
  NAME                               READY   STATUS    RESTARTS   AGE
  aws-load-balancer-controller-6486ccd8ff-7nj9f   1/1     Running   0          79m
  aws-load-balancer-controller-6486ccd8ff-lgs6b   1/1     Running   0          79m
  aws-node-1ztjf                         2/2     Running   0          84m
  aws-node-mhsnv                         2/2     Running   0          84m
● coredns-dc7c47f6c-7v8k7
  coredns-dc7c47f6c-gpg74                 1/1     Running   0          85m
  ebs-csi-controller-68f5f4f467-1sdnn      6/6     Running   0          85m
  ebs-csi-controller-68f5f4f467-qcnzb      6/6     Running   0          85m
  ebs-csi-node-qjxvs                      3/3     Running   0          84m
  ebs-csi-node-ztnkt                      3/3     Running   0          84m
  eks-pod-identity-agent-djnxt           1/1     Running   0          84m
  eks-pod-identity-agent-lrfcp           1/1     Running   0          84m
  kube-proxy-cdg6z                        1/1     Running   0          84m
  kube-proxy-krm25                        1/1     Running   0          84m
○ PS D:\Cloudelligent\Task-17>

```

This confirms cluster stability and readiness for application workloads.

## 6.3 Storage Configuration

### Create and Validate StorageClass

- *kubectl apply -f storageclass.yaml*
- *kubectl get sc*

### StorageClass Highlights:

- Uses **EBS CSI driver**
- Volumes provisioned dynamically
- **WaitForFirstConsumer** ensures volumes are created in the correct AZ

```

● PS D:\Cloudelligent\Task-17> kubectl apply -f storageclass.yaml
storageclass.storage.k8s.io/ebs-sc created
● PS D:\Cloudelligent\Task-17> kubectl get sc
  NAME      PROVISIONER            RECLAIMPOLICY  VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
  ebs-sc    ebs.csi.aws.com       Delete          WaitForFirstConsumer  false                8s
  gp2      kubernetes.io/aws-ebs Delete          WaitForFirstConsumer  false                69m
● PS D:\Cloudelligent\Task-17>

```

This StorageClass is used by StatefulSets to provision persistent EBS volumes automatically.

## 6.4 Container Image Management

### Build and Push Application Image to ECR

The Node.js application image is built locally and pushed to Amazon ECR using repository-specific push commands.

```

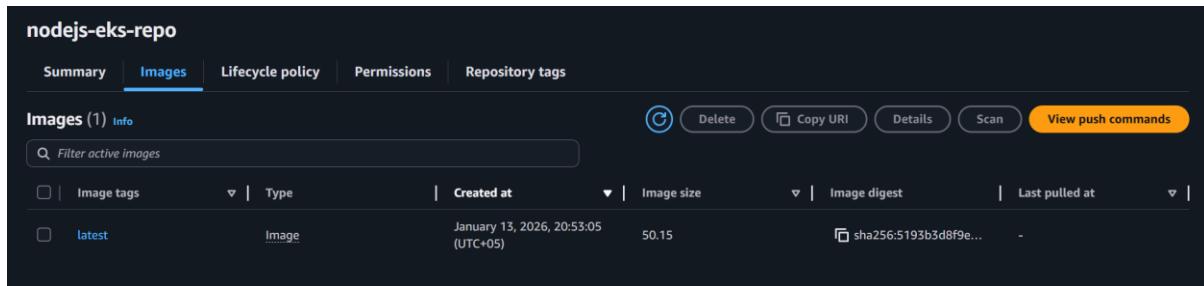
● PS D:\Cloudelligent\Task-17\nodejs-app> aws ecr get-login-password --region us-west-1 | docker login --username AWS --password-stdin 504649076991.dkr.ecr.us-west-1.amazonaws.com
Login Succeeded
● PS D:\Cloudelligent\Task-17\nodejs-app> docker build -t nodejs-eks-repo .
[+] Building 0.5s (10/10) FINISHED
   => [internal] load build definition from Dockerfile
   => => transferring dockerfile: 350B
   => [internal] load metadata for docker.io/library/node:20-alpine
   => [internal] load .dockerignore
   => => transferring context: 28
   => [1/5] FROM docker.io/library/node:20-alpine@sha256:658d8f63e501824ddc23e06d4bb95c71e7d704537c9d9272f488ac83a370d448
   => [internal] load build context
   => => transferring context: 120B
   => CACHED [2/5] WORKDIR /usr/src/app
   => CACHED [3/5] COPY package.json .
   => CACHED [4/5] RUN npm install
   => CACHED [5/5] COPY . .
   => exporting to image
   => => exporting layers
   => => writing image sha256:82e10e587208869322d23e1b6818fdaf56ccc12ffe7248c282ebc20ac8d3f4f5eb
   => => naming to docker.io/library/nodejs-eks-repo

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/lbtvvtvn78ry7j7krjgeoisp8a
●
What's next:
  View a summary of image vulnerabilities and recommendations - docker scout quickview
● PS D:\Cloudelligent\Task-17\nodejs-app> docker tag nodejs-eks-repo:latest 504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-eks-repo:latest
The push refers to repository [504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-eks-repo]
18fe8347e7d5: Pushed
5fad28fa7f6: Pushed
65f51e52e4b: Pushed
3c664c8c6c9d: Pushed
6be78693c259: Pushed
571b9a15087: Pushed
c7e70c519c8c: Pushed
7b28cf5e767: Pushed
latest: digest: sha256:5193b3d8f9ec826c66ab6f074f1a42f6c122ab0369449cfcc60e07b88d84716b5 size: 1991
● PS D:\Cloudelligent\Task-17\nodejs-app>

```

### Validation Steps:

- Docker authentication to ECR succeeds
- Image build completes without errors
- Image appears in the ECR repository with the expected tag (latest)



This ensures the EKS worker nodes can securely pull the application image during deployment.

## 6.5 Application Deployment

### 1. Deploy StatefulSet

- `kubectl apply -f statefulset.yaml`
- `kubectl get pods`

### StatefulSet Validation:

- Desired replicas are running
- Pods start sequentially
- Each pod reaches **Ready** state

- No restarts detected

```
● PS D:\Cloudelligent\Task-17> kubectl apply -f statefulset.yaml
statefulset.apps/node-app created
● PS D:\Cloudelligent\Task-17> kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
node-app-0 1/1     Running   0          75s
node-app-1 1/1     Running   0          58s
● PS D:\Cloudelligent\Task-17> █
```

StatefulSet is used to provide stable pod identities and persistent storage.

## 2. Verify Persistent Volumes

- *kubectl get pv*
- *kubectl get pvc*

### Observations:

- One PVC created per pod
- PVCs are bound to dynamically provisioned EBS volumes
- Volumes persist independently of pod lifecycle

```
● PS D:\Cloudelligent\Task-17> kubectl get pv
NAME           CAPACITY   ACCESS MODES  RECLAIM POLICY  STATUS   CLAIM                                     STORAGECLASS  VOLUMEATTRIBUTESCLASS  REASON  AGE
pvc-1bf555ca-0cc2-45d8-91ce-cbd7f875a2a9  4Gi        RWO          Delete        Bound   default/data-node-app-0                   ebs-sc       <unset>                         3m44s
pvc-3a9dc6db-9b34-4323-b91d-9e488489f8ac  4Gi        RWO          Delete        Bound   default/data-node-app-1                   ebs-sc       <unset>                         3m27s
● PS D:\Cloudelligent\Task-17> kubectl get pvc
NAME           STATUS   VOLUME                                     CAPACITY   ACCESS MODES  STORAGECLASS  VOLUMEATTRIBUTESCLASS  AGE
● data-node-app-0  Bound   pvc-1bf555ca-0cc2-45d8-91ce-cbd7f875a2a9  4Gi        RWO          ebs-sc       <unset>                         3m54s
● data-node-app-1  Bound   pvc-3a9dc6db-9b34-4323-b91d-9e488489f8ac  4Gi        RWO          ebs-sc       <unset>                         3m37s
○ PS D:\Cloudelligent\Task-17> █
```

This confirms correct integration between Kubernetes storage and AWS EBS.

## 6.6 Service Configuration

### Deploy Kubernetes Service

- *kubectl apply -f service.yaml*
- *kubectl get svc*

### Service Characteristics:

- Type: NodePort
- Provides stable internal access to pods
- Acts as the backend for the Ingress resource

```

● PS D:\Cloudelligent\Task-17> kubectl apply -f service.yaml
  service/node-app-service created
● PS D:\Cloudelligent\Task-17> kubectl get svc
  NAME           TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
  kubernetes     ClusterIP  10.100.0.1  <none>        443/TCP   78m
  node-app-service NodePort   10.100.41.218 <none>        80:31777/TCP 8s
● PS D:\Cloudelligent\Task-17> █

```

This service enables traffic routing from the load balancer to application pods.

## 6.7 Ingress and Load Balancer Configuration

### Deploy Ingress Resource

- *kubectl apply -f ingress.yaml*
- *kubectl get ingress*

### Ingress Validation:

- AWS Load Balancer Controller detects the resource
- Application Load Balancer is provisioned automatically
- ALB DNS name is assigned

```

● PS D:\Cloudelligent\Task-17> kubectl apply -f ingress.yaml
  ingress.networking.k8s.io/node-app-ingress created
● PS D:\Cloudelligent\Task-17> kubectl get ingress
  NAME           CLASS   HOSTS   ADDRESS          PORTS   AGE
  node-app-ingress alb     *       k8s-default-nodeappi-15eb8af526-1755730335.us-west-1.elb.amazonaws.com  80      9s
○ PS D:\Cloudelligent\Task-17> █

```

**Traffic flow:** Client → ALB → NodePort → Service → Pods

## 6.8 Load Balancer Validation

### 1. Target Group Validation

Navigate to **EC2** console. Select **Target Groups** located in the left navigation bar.

The screenshot shows the AWS CloudWatch Metrics Target groups (1/2) page. It displays two target groups in a table:

Name	ARN	Port	Protocol	Target type	Load balancer	VPC ID
k8s-default-nodeapps-c37a0bf5e8	arn:aws:elasticloadbalancing:us-west-1:504649076991:loadbalancer/app/k8s-default-nodeappi-15eb8af526	3000	HTTP	IP	k8s-default-nodeappi-15eb8af526	vpc-00b7566bbc3a1cb4
task7-tg	arn:aws:elasticloadbalancing:us-west-1:504649076991:loadbalancer/app/task7-tg	3000	HTTP	IP	None associated	vpc-052f3413af93fb8fc

Below the table, the details for the target group "k8s-default-nodeapps-c37a0bf5e8" are shown. The "Targets" tab is selected. It lists two registered targets:

IP address	Port	Zone	Health status	Administrative override	Override details	Anomaly detection
172.20.40.36	3000	us-west-1b (usw1-az1)	Healthy	No override	No override is currently active on target	Normal
172.20.30.145	3000	us-west-1a (usw1-az3)	Healthy	No override	No override is currently active on target	Normal

## Checks:

- Worker node IPs registered
- All targets marked **Healthy**
- Health checks passing

## 2. Application Load Balancer Validation

Navigate to **EC2** console. Select **Load Balancers** located in the left navigation bar.

The screenshot shows the AWS EC2 Load balancers (1/1) page. It displays one active application load balancer in a table:

Name	Status	Type	Scheme	IP address type	VPC ID	Availability Zones	Security group
k8s-default-nodeappi-15eb8af526	Active	application	Internet-facing	IPv4	vpc-00b7566bbc3a1cb4	2 Availability Zones	2 Security groups

Below the table, the details for the load balancer "k8s-default-nodeappi-15eb8af526" are shown. The "Details" tab is selected. It provides the following information:

Details			
Load balancer type Application	Status <span style="color: green;">Active</span>	VPC vpc-00b7566bbc3a1cb4	Load balancer IP address type IPv4
Scheme Internet-facing	Hosted zone Z368ELLRRE2KJ0	Availability Zones subnet-05215e5bcf8e3ede4 us-west-1a (usw1-az3) subnet-09415a92f90887cad us-west-1b (usw1-az1)	Date created January 13, 2026, 20:59 (UTC+05:00)
Load balancer ARN arn:aws:elasticloadbalancing:us-west-1:504649076991:loadbalancer/app/k8s-default-nodeappi-15eb8af526/206800f213f0f92a	DNS name info k8s-default-nodeappi-15eb8af526-1755730335.us-west-1.elb.amazonaws.com (A Record)		

## Checks:

- Load balancer is Active
- Internet-facing
- Deployed across multiple AZs
- Listener configured on port 80

This confirms high availability and fault tolerance.

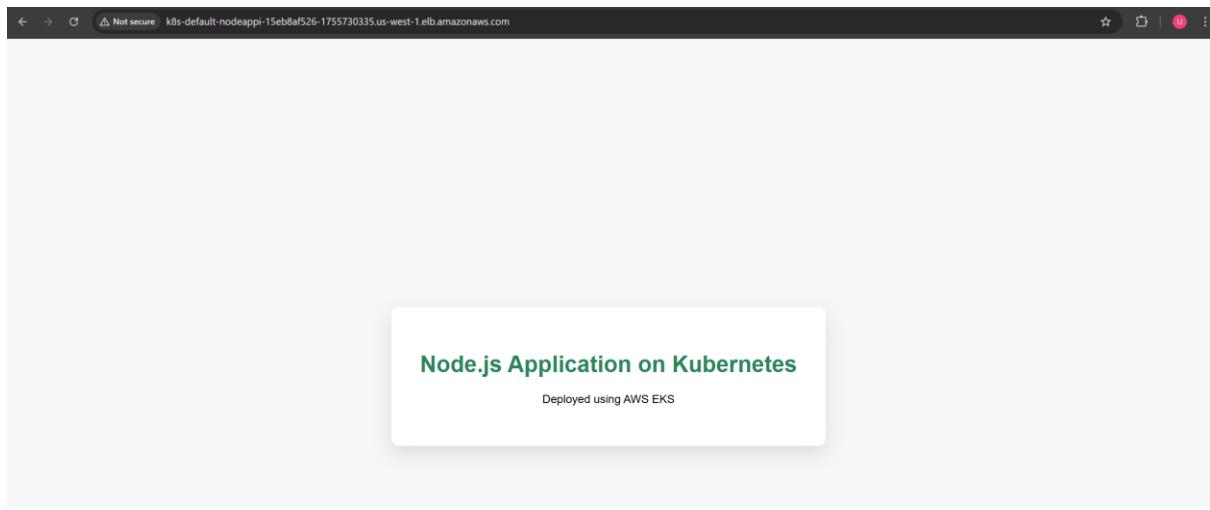
## 6.9 Application Access Verification

### Access Application

Using the ALB DNS name in a browser:

#### Validation Outcome:

- Application loads successfully
- Requests are served without errors
- Traffic is balanced across multiple pods



As shown above, the application is accessible using ALB DNS name.

### Overall Post-Provisioning Summary

- EKS cluster accessible and stable
- Nodes, system components, and storage validated
- Application image managed via ECR
- Stateful application deployed with persistent storage
- External access enabled using AWS ALB

## 7. Clean Up

This section documents the removal of all Kubernetes workloads and AWS infrastructure resources created during the project. Performing a clean-up ensures that no unused resources remain running and prevents unnecessary AWS costs.

### 7.1 Delete Ingress Resource

The Ingress resource is removed first to ensure that the Application Load Balancer is deprovisioned correctly.

- *kubectl get ingress*

#### Ingress Resource Details:

- **Ingress name:** node-app-ingress
- **Port:** 80
- ALB address assigned
- **Hosts:** \*

#### Delete command:

- *kubectl delete ingress node-app-ingress*

```
PS D:\Cloudelligent\Task-17> kubectl get ingress
NAME      CLASS   HOSTS   ADDRESS                                     PORTS   AGE
node-app-ingress  alb     *    k8s-default-nodeappi-15eb8af526-1755730335.us-west-1.elb.amazonaws.com  80      7h36m
PS D:\Cloudelligent\Task-17> kubectl delete ingress node-app-ingress
ingress.networking.k8s.io "node-app-ingress" deleted
PS D:\Cloudelligent\Task-17>
```

#### Result:

The Ingress resource is successfully deleted, triggering the cleanup of the associated AWS Application Load Balancer.

### 7.2 Delete Service

The Kubernetes Service is deleted after the Ingress to cleanly remove backend routing.

- *kubectl get service*

#### Service Resource Details:

- **Service name:** node-app-service
- **Type:** NodePort
- Cluster IP assigned
- No external IP

- **Port mapping:** 80:31777/TCP

#### **Delete command:**

- *kubectl delete service node-app-service*

```
● PS D:\Cloudelligent\Task-17> kubectl get service
  NAME           TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
  kubernetes     ClusterIP  10.100.0.1  <none>        443/TCP     8h
  node-app-service  NodePort   10.100.41.218 <none>        80:31777/TCP 7h39m
● PS D:\Cloudelligent\Task-17> kubectl delete service node-app-service
  service "node-app-service" deleted
○ PS D:\Cloudelligent\Task-17>
```

#### **Result:**

The NodePort service is removed and no longer exposes the application internally.

### **7.3 Delete StatefulSet**

The application StatefulSet is deleted to terminate running pods and detach associated volumes.

- *kubectl get statefulset*

#### **StatefulSet Resource Details:**

- StatefulSet name: node-app
- READY: 2/2

#### **Delete command:**

- *kubectl delete statefulset node-app*

```
● PS D:\Cloudelligent\Task-17> kubectl get statefulset
  NAME    READY   AGE
  node-app  2/2    7h43m
● PS D:\Cloudelligent\Task-17> kubectl delete statefulset node-app
  statefulset.apps "node-app" deleted
○ PS D:\Cloudelligent\Task-17>
```

#### **Result:**

All application pods are terminated in an orderly manner.

## 7.4 Delete StorageClass

The custom StorageClass is deleted after application workloads are removed.

- *kubectl get storageclass*

### StorageClass Resource Details:

- StorageClass name: ebs-sc
- Provisioner: ebs.csi.aws.com
- Reclaim policy: Delete
- Volume binding mode: WaitForFirstConsumer

### Delete Command:

- *kubectl delete storageclass ebs-sc*

```
● PS D:\Cloudelligent\Task-17> kubectl get storageclass
  NAME      PROVISIONER          RECLAIMPOLICY  VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION  AGE
  ebs-sc    ebs.csi.aws.com     Delete         WaitForFirstConsumer  false                7h49m
  gp2       kubernetes.io/aws-ebs Delete         WaitForFirstConsumer  false                8h
● PS D:\Cloudelligent\Task-17> kubectl delete storageclass ebs-sc
  storageclass.storage.k8s.io "ebs-sc" deleted
○ PS D:\Cloudelligent\Task-17>
```

### Result:

The StorageClass is removed. Any dynamically provisioned EBS volumes are cleaned up according to the reclaim policy.

## 7.5 Destroy Terraform Infrastructure

After all Kubernetes resources are removed, the underlying AWS infrastructure is destroyed using Terraform.

- *terraform destroy -auto-approve*

### Result:

- Terraform successfully destroyed all managed resources
- Total resources destroyed: **49**

```
Destroy complete! Resources: 49 destroyed.
○ PS D:\Cloudelligent\Task-17\terraform>
```

This confirms that the EKS cluster, networking components, IAM roles, add-ons, and supporting AWS services have been fully removed.

## 8. Troubleshooting

The following issues were encountered during the Amazon EKS cluster provisioning, add-on configuration, IAM Pod Identity integration, application deployment, and load balancer setup. Each issue documents the observed problem, root cause, and the corrective action taken.

### **Issue 1: AWS Load Balancer Controller fails with IMDSv2 metadata timeout**

#### **Problem Description**

The AWS Load Balancer Controller failed to start with the following error:

*"unable to initialize AWS cloud""error":"failed to get VPC ID: failed to fetch VPC ID from instance metadata... context deadline exceeded"*

#### **Root Cause**

Modern EKS worker nodes enforce **IMDSv2**, which limits metadata access using hop count restrictions. The controller pod runs multiple network hops away from the node's hardware, preventing it from accessing instance metadata such as VPC ID and region.

#### **Solution**

Explicitly provided the **VPC ID** and **region** during Helm installation so the controller no longer relied on instance metadata.

### **Issue 2: Missing Service Account causing “Unauthorized” errors**

#### **Problem Description**

The controller failed to start with the following errors:

*serviceaccounts "aws-load-balancer-controller" not found  
"error":"Unauthorized"*

#### **Root Cause**

The Helm installation was configured with `serviceAccount.create=false`, but the service account was not created manually. As a result, the controller pod had no Kubernetes or AWS identity.

#### **Solution**

Re-ran the Helm upgrade with:

`serviceAccount.create=true`

This created the service account and linked it to the IAM role using **EKS Pod Identity Association**.

## **Issue 3: Pod Identity Association already exists**

### **Problem Description**

Attempting to create a Pod Identity Association failed with an error indicating the association already existed.

### **Root Cause**

The association had already been created in AWS and could not be duplicated.

### **Solution**

Confirmed the existing association and executed:

*kubectl rollout restart*

This forced the pods to reload and successfully assume the existing IAM role.

## **Issue 4: Ingress creation fails with YAML unmarshalling error**

### **Problem Description**

Applying the Ingress manifest resulted in:

*BadRequest: cannot unmarshal object into Go struct*

### **Root Cause**

The ingress.yaml file was missing a list indicator (-) under the paths section, violating Kubernetes API schema rules.

### **Solution**

Corrected the YAML indentation and added the required list markers, then re-applied the manifest.

## **Issue 5: Target group registration limited to a single Availability Zone**

### **Problem Description**

Application Load Balancer target groups registered pods only from one private subnet.

### **Root Cause**

EBS volumes are **Availability Zone-locked**. When using a Deployment with persistent storage, Kubernetes was forced to schedule all pods in the same AZ where the volume existed.

### **Solution**

Migrated to a **StatefulSet**, which provisions one EBS volume per pod, allowing pods to be evenly distributed across multiple Availability Zones.

Alternatively, EFS CSI Driver could be used for multi-AZ storage.

## **Issue 6: Worker nodes created but not joining the cluster**

### **Problem Description**

Worker nodes were launched, but CoreDNS and CloudWatch add-ons showed a **Degraded** status. Kubernetes access errors indicated insufficient permissions.

### **Root Cause**

The current IAM principal did not have Kubernetes RBAC permissions and was not mapped in the cluster authentication configuration.

### **Solution**

Updated cluster authentication to grant the IAM principal access to Kubernetes resources, allowing worker nodes and system add-ons to initialize correctly.

## **Issue 7: AmazonEBSCSIDriverPolicy ARN does not exist**

### **Problem Description**

Terraform failed when referencing:

*arn:aws:iam::aws:policy/AmazonEBSCSIDriverPolicy*

### **Root Cause**

The policy exists under the **service-role** path, not the root managed policy path.

### **Solution**

Updated the ARN to:

*arn:aws:iam::aws:policy/service-role/AmazonEBSCSIDriverPolicy*

## **Issue 8: Managed Node Group creation fails due to VPC CNI add-on configuration**

### **Problem Description**

Managed Node Group creation failed, and VPC networking logs showed:

*NetworkPluginNotReady*

*cni plugin not initialized*

### **Root Cause**

A Pod Identity IAM role was incorrectly attached to the **VPC CNI add-on**, which does not support **Pod Identity**.

### **Solution**

Removed the Pod Identity role from the VPC CNI add-on. Once corrected, node groups were created successfully and networking initialized properly.

## **Issue 10: CloudWatch Container Insights active but metrics unavailable**

### **Problem Description**

CloudWatch Container Insights console returned:

*Log group '/aws/containerinsights/umarsatti-eks-cluster/performance' does not exist*

No log groups were created despite the add-on being active.

### **Root Cause**

The CloudWatch Observability add-on creates a service account (cloudwatch-agent) in the amazon-cloudwatch namespace. While an IAM role was provided, no **Pod Identity Association** existed for this service account.

### **Solution**

Created a Pod Identity Association linking:

- Service Account: cloudwatch-agent
- Namespace: amazon-cloudwatch
- IAM Role: CloudWatch permissions role

After applying the association, log groups and metrics were created successfully.