

Container Orchestration and Infrastructure Management

For
Online Shopping Grocery App

Submitted By

Specialization	SAP ID	Name
BTech CSE (CC&VT) NH	500091910	Hitendra Sisodia
BTech CSE (CC&VT) NH	500091882	Ujesh Sisodia
BTech CSE (CC&VT) NH	500091901	Priyanshu Tandon
BTech CSE (CC&VT) NH	500091736	Anirudh Srinivasan



Department of Systemics

School Of Computer Science

UNIVERSITY OF PETROLEUM & ENERGY STUDIES,

DEHRADUN- 248007. Uttarakhand

Submitted To:

Ms.Avita Katal

Assistant Professor (SG)

School of Computer Science

Submitted By:

Ujesh Sisodia(500091882)

Hitendra Sisodia(500091910)

Priyanshu Tandon (500091901)

Anirudh Srinivasan (500091736)

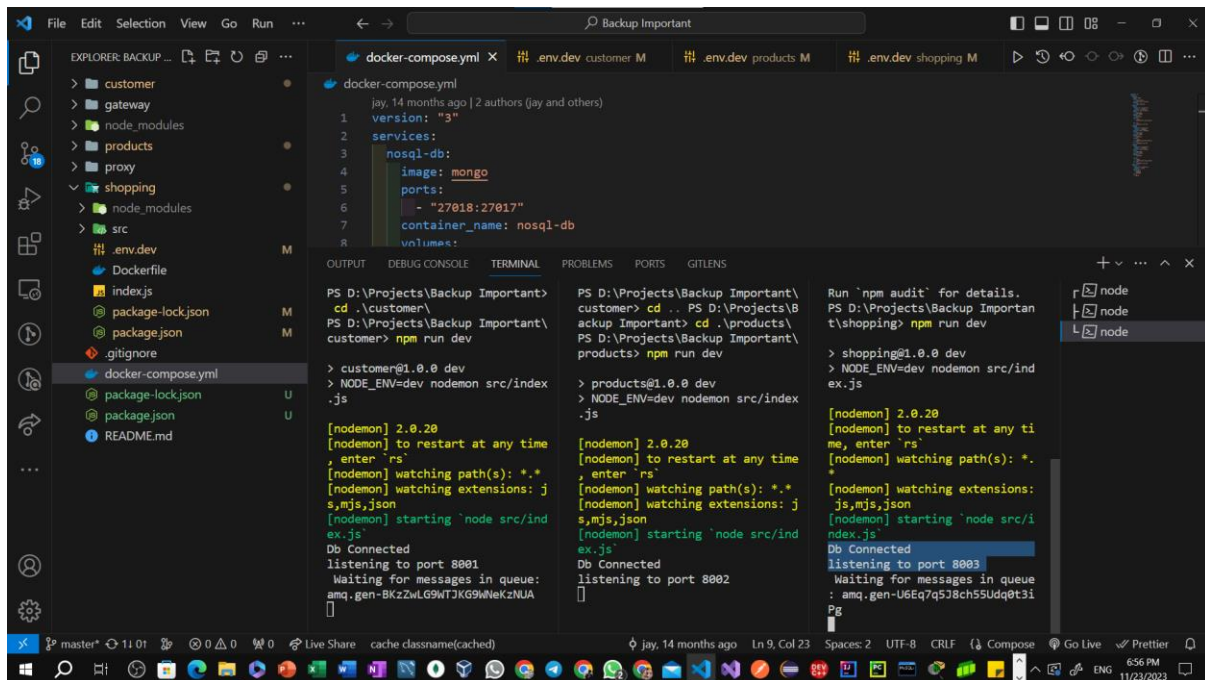
Container Orchestration and Infrastructure Automation

Assignment – 1 Recap

Microservices created for the project

- 1) Customer: 8001
- 2) Products: 8002
- 3) Shopping: 8003
- 4) Proxy: 80 (Nginx Default Port No)

Each Microservices running individually at port 8001, 8002, 8003.



The screenshot shows a VS Code editor with a project named 'Backup Important'. The Explorer pane on the left shows a directory structure with folders for 'customer', 'gateway', 'products', 'proxy', and 'shopping', each containing 'node_modules', 'package-lock.json', 'package.json', and 'README.md'. The 'docker-compose.yml' file is open in the editor, showing a configuration for three services: 'nosql-db' (mongo), 'customer' (node), 'products' (node), and 'shopping' (node). The terminal pane at the bottom shows the output of running 'npm run dev' for each service. The 'customer' service is listening on port 8001, 'products' on port 8002, and 'shopping' on port 8003. All services are using Node.js 2.0.28 and Nodemon.

```
docker-compose.yml
version: "3"
services:
  nosql-db:
    image: mongo
    ports:
      - "27018:27017"
    container_name: nosql-db
    volumes:
      - ./data/db
  customer:
    build:
      dockerfile: Dockerfile
    ports:
      - "8001:8001"
  products:
    build:
      dockerfile: Dockerfile
    ports:
      - "8002:8002"
  shopping:
    build:
      dockerfile: Dockerfile
    ports:
      - "8003:8003"
```

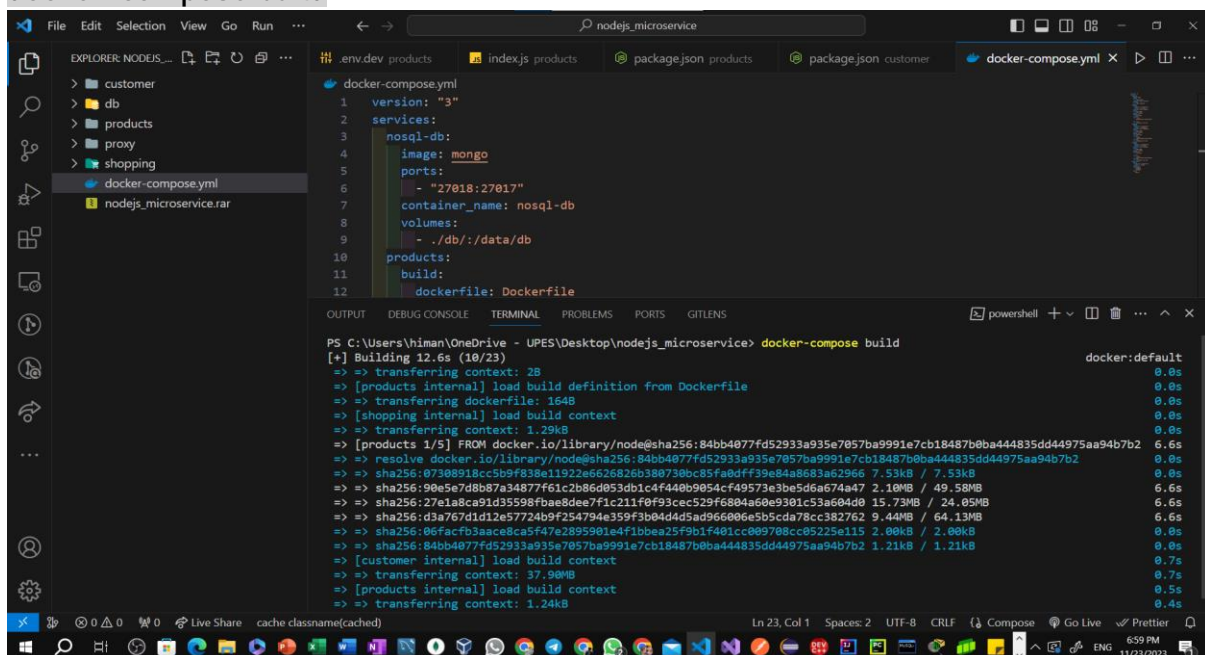
```
PS D:\Projects\Backup Important> cd .\customer\
PS D:\Projects\Backup Important\customer> npm run dev
> customer@1.0.0 dev
> NODE_ENV=dev nodemon src/index.js
[nodemon] 2.0.28
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
Db Connected
listening to port 8001
Waiting for messages in queue:
amq.gen-BkZwL6G9WTJ3KG9WNeKzNUA

PS D:\Projects\Backup Important\customer> cd ..\products\
PS D:\Projects\Backup Important\products> npm run dev
> products@1.0.0 dev
> NODE_ENV=dev nodemon src/index.js
[nodemon] 2.0.28
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
Db Connected
listening to port 8002

PS D:\Projects\Backup Important\products> cd ..\shopping\
PS D:\Projects\Backup Important\t\shopping> npm run dev
> shopping@1.0.0 dev
> NODE_ENV=dev nodemon src/index.js
[nodemon] 2.0.28
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
Db Connected
listening to port 8003
Waiting for messages in queue:
amq.gen-U6Eq7q538ch55Udq0t3iPg
```

Executing the below command initiates the Docker Compose build process, ensuring that each microservice is built with its dependencies and configurations.

docker-compose build



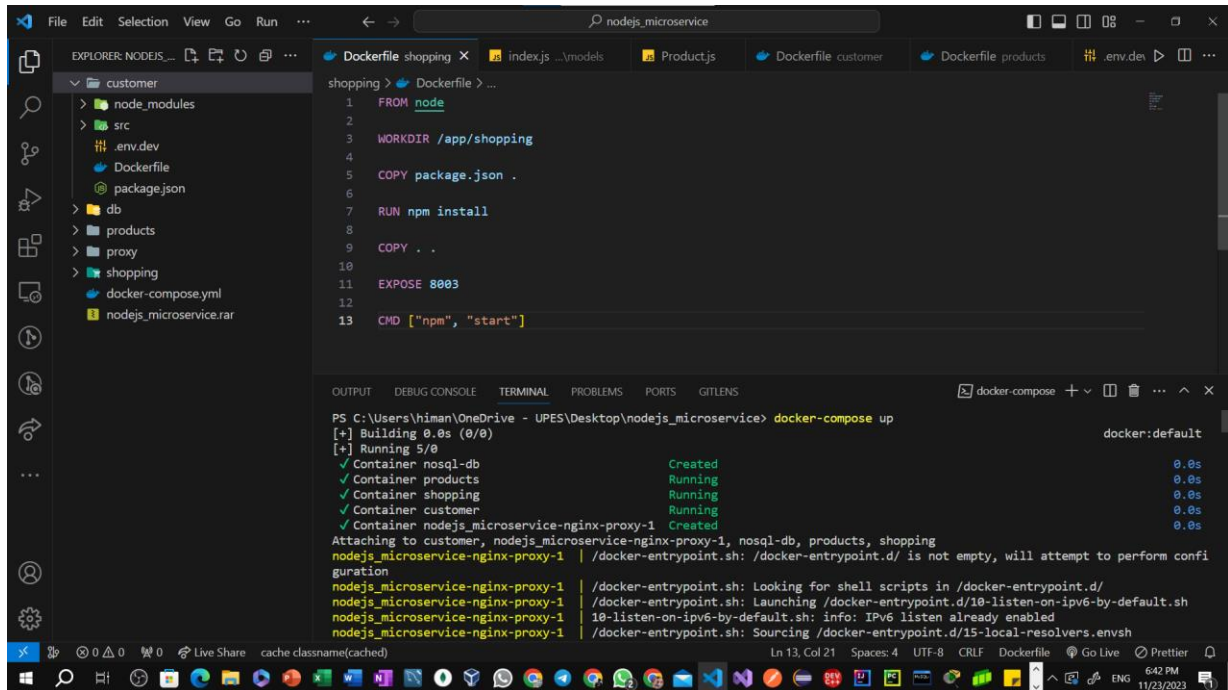
The screenshot shows a VS Code editor with a project named 'nodejs_microservice'. The Explorer pane on the left shows a directory structure with folders for 'customer', 'db', 'products', 'proxy', and 'shopping', each containing 'node_modules', 'package-lock.json', 'package.json', and 'README.md'. The 'docker-compose.yml' file is open in the editor, showing a configuration for three services: 'nosql-db' (mongo), 'customer' (node), 'products' (node), and 'shopping' (node). The terminal pane at the bottom shows the output of running 'docker-compose build'. The output indicates that the build process is successful, with all services built and ready to run.

```
PS C:\Users\himan\OneDrive - UPES\Desktop\nodejs_microservice> docker-compose build
[*] Building 12.6s (10/23)
=> transferring context: 28
=> [products internal] load build definition from Dockerfile
=> transferring dockerfile: 164B
=> [shopping internal] load build context
=> transferring context: 1.29kB
=> [products 1/5] FROM docker.io/library/node@sha256:84bb4077fd52933a935e7057ba9991e7cb18487b0ba444835dd44975aa94b7b2
=> resolve docker.io/library/node@sha256:84bb4077fd52933a935e7057ba9991e7cb18487b0ba444835dd44975aa94b7b2
=> sha256:07308918c5b9f830e11922e626826b380730bc85fa0df739e84a8683a62966 7.53kB / 7.53kB
=> sha256:905e740807a34877f61c2b86d53d1c4f40e9854cf49973e3bc5d6a674e47 2.10MB / 40.58MB
=> sha256:27e1a83c91d85598fbae8de7f1c211f093ce529f6804a6e9301c3a604d6 15.73MB / 24.05MB
=> sha256:d3a767d1d12e57724b9f25479a359f3b04d4d5ad966006a5b5cda78cc382762 9.44MB / 64.13MB
=> sha256:06facfb3aace8ca5f47e2895901e4f1bba25f9b1f401cc009708cc05225e115 2.00kB / 2.00kB
=> sha256:84bb4077fd52933a935e7057ba9991e7cb18487b0ba444835dd44975aa94b7b2 1.21kB / 1.21kB
=> [customer internal] load build context
=> transferring context: 37.90MB
=> [products internal] load build context
=> transferring context: 1.24kB
```

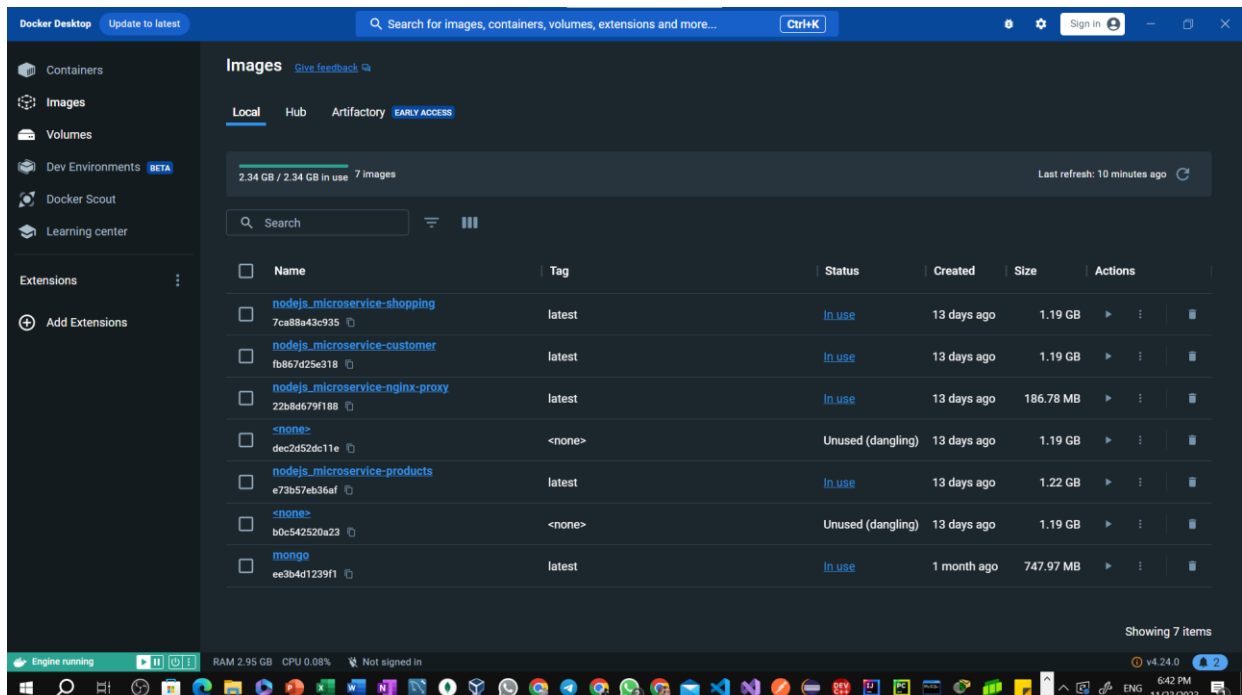
Container Orchestration and Infrastructure Automation

Running All Microservices using single command.

```
docker-compose up
```



Docker Images



Container Orchestration and Infrastructure Automation

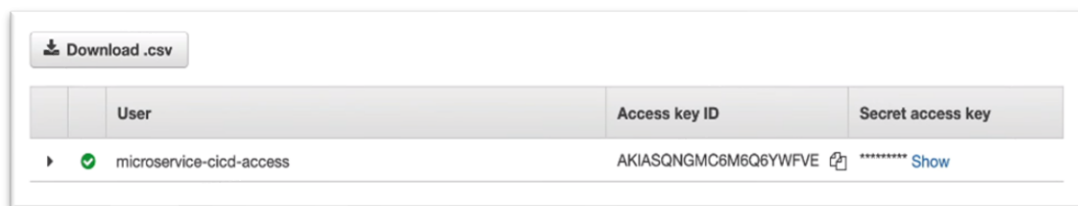
Assignment 2 - Part 2: Deployment and Scaling (50 marks)

Cloud Platform Selection

1. Justify the choice of a specific cloud platform (e.g., AWS, Azure, Google Cloud) based on its suitability for the project.

In the case of our "Grocery Online Shopping App," the justification of choosing of AWS (Amazon Web Services) based on its suitability for the project are as follows:

1. **Microservices Architecture Support:** AWS offers containerization services like Amazon ECS (Elastic Container Service) and Kubernetes-based Amazon EKS (Elastic Kubernetes Service), which are well-suited for deploying and managing microservices. These services provide the flexibility and scalability needed for a microservices-based architecture.
2. **Security and Compliance:** AWS places a strong emphasis on security, providing features like Identity and Access Management (IAM), encryption services, and compliance certifications. These features are crucial for handling sensitive customer data, especially in an online shopping app.



Creating user with help of IAM service in AWS

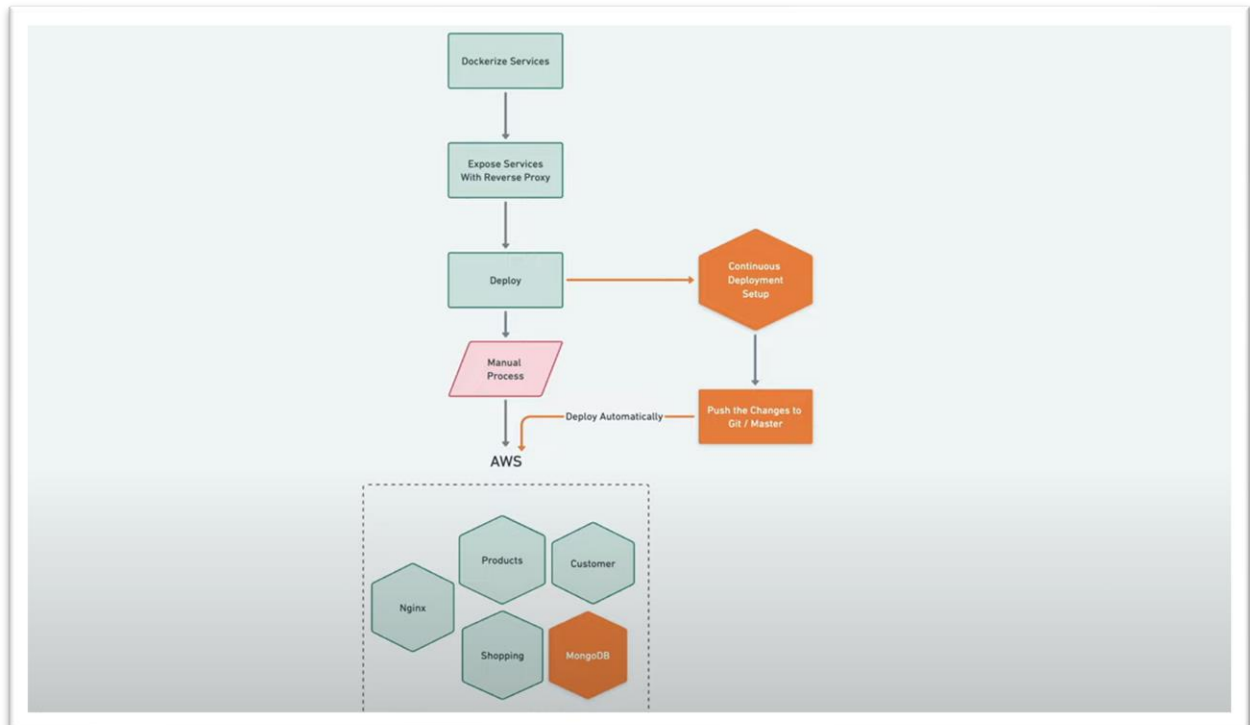
3. **Scalability and Elasticity:** AWS offers auto-scaling capabilities that allow our application to automatically adjust its capacity based on demand. This ensures that our online grocery shopping app can handle varying levels of user traffic efficiently, providing a seamless experience for customers.
4. **DevOps and CI/CD Support:** AWS integrates well with DevOps practices, providing tools like AWS CodePipeline and AWS CodeDeploy for continuous integration and deployment. This facilitates automated testing, building, and deployment of our microservices, streamlining the development and release process.

The choice of AWS for our "Grocery Online Shopping App" is justified by its comprehensive service offerings, strong support for microservices architecture, security features, scalability options, global presence, and integration with DevOps practices. It provides a robust and flexible foundation for building, deploying, and managing the various components of our online grocery shopping application.

Container Orchestration and Infrastructure Automation

Container Orchestration

2. Utilize a container orchestration platform (e.g., Kubernetes) to manage and deploy the microservices.



Pre-requisites:

Kubernetes Cluster: Ensure we have access to a Kubernetes cluster, either locally using tools like Minikube.

Minikube, a local Kubernetes solution, simplifies Kubernetes learning and development by requiring only Docker or a compatible container, making Kubernetes just a single command away with 'minikube start'.

Deployment Steps

Step 1. Containerize Microservices

Ensure that each microservice (product, shopping, payments) is containerized using Docker. Each service should have its own Dockerfile. (already done)

Step 2. Create Kubernetes Deployment Files

For each microservice, create Kubernetes Deployment YAML files. These files describe how Kubernetes should create and manage instances of our containers. (Need to be done)

Container Orchestration and Infrastructure Automation

Manifest Creation:

Deployment Manifest: # customer-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: customer-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: customer
  template:
    metadata:
      labels:
        app: customer
    spec:
      containers:
        - name: products
          image: demo/ customer -service:latest
          ports:
            - containerPort: 8001
          env:
            - name: DATABASE_URL
              value: mongodb://127.0.0.1:27017/msytt_customer
```

Deployment Manifest: # products-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: products-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: products
  template:
    metadata:
      labels:
        app: products
    spec:
      containers:
        - name: products
          image: demo/products-service:latest
          ports:
            - containerPort: 8002
          env:
            - name: DATABASE_URL
              value: mongodb://127.0.0.1:27017/msytt_products
```

Container Orchestration and Infrastructure Automation

Deployment Manifest: `# shopping-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shopping-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: shopping
  template:
    metadata:
      labels:
        app: shopping
    spec:
      containers:
        - name: shopping
          image: demo/shopping-service:latest
          ports:
            - containerPort: 8003
          env:
            - name: DATABASE_URL
              value: mongodb://127.0.0.1:27017/msytt_shopping
```

Step 3. Create Kubernetes Services

Create Kubernetes Service YAML files for each microservice. Services allow communication between different microservices.

Service YAML: Defines how other services or external clients can access our microservices. This includes details like ports, protocols, and selectors.

Service Manifest: `# customer-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: customer-service
spec:
  selector:
    app: products
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8001
```

Container Orchestration and Infrastructure Automation

Service Manifest: # products-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: products-service
spec:
  selector:
    app: products
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8002
```

Service Manifest: # shopping-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: shopping-service
spec:
  selector:
    app: products
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8003
```

Step 4. Apply Deployments and Services

Use kubectl to apply the Deployment and Service YAML files for each microservice:

```
kubectl apply -f <microservice-deployment.yaml>
kubectl apply -f <microservice-service.yaml>
```

Step 5. Verify Deployments and Check if our microservices are running:

```
kubectl get pods          # Check pods
kubectl get services      # Check services
kubectl describe pod <pod-name> # Detailed info about a pod
```


Container Orchestration and Infrastructure Automation

3. Describe how container orchestration simplifies deployment, scaling, and management.

In the context of our "Grocery Online Shopping App" project, container orchestration plays a crucial role in simplifying deployment, scaling, and management of the microservices architecture. Here's how container orchestration addresses these aspects:

Container orchestration simplifies deployment, scaling, and management by automating the processes of deploying and managing containers at scale. It abstracts complexities, streamlines resource allocation, automates scaling based on demand, and provides centralized management, making it easier to maintain, update, and scale applications in a containerized environment.

Deployment Simplification: With the transition to a microservices architecture, our application is likely composed of multiple services running in separate containers. Container orchestration tools, such as Kubernetes or Docker Swarm, simplify the deployment process by automating the distribution of containerized microservices across a cluster of machines.

Scaling: If the load on our "Grocery Online Shopping App" increases, we can easily scale specific microservices or the entire application. Kubernetes, for example, supports horizontal scaling, where we can dynamically adjust the number of running instances of a microservice based on factors like CPU usage or incoming requests.

Management Efficiency: Container orchestration tools provide centralized management and monitoring capabilities, allowing us to oversee the health and performance of our microservices.

Resource Optimization: Container orchestration optimizes resource utilization by efficiently distributing microservices across the available infrastructure. Containers are isolated from each other, preventing interference and ensuring that each microservice runs in its own environment.

Load Balancing: Container orchestration tools often include built-in load balancing capabilities, distributing incoming traffic across multiple instances of a microservice. This ensures even distribution of requests, preventing any single microservice instance from becoming a bottleneck.

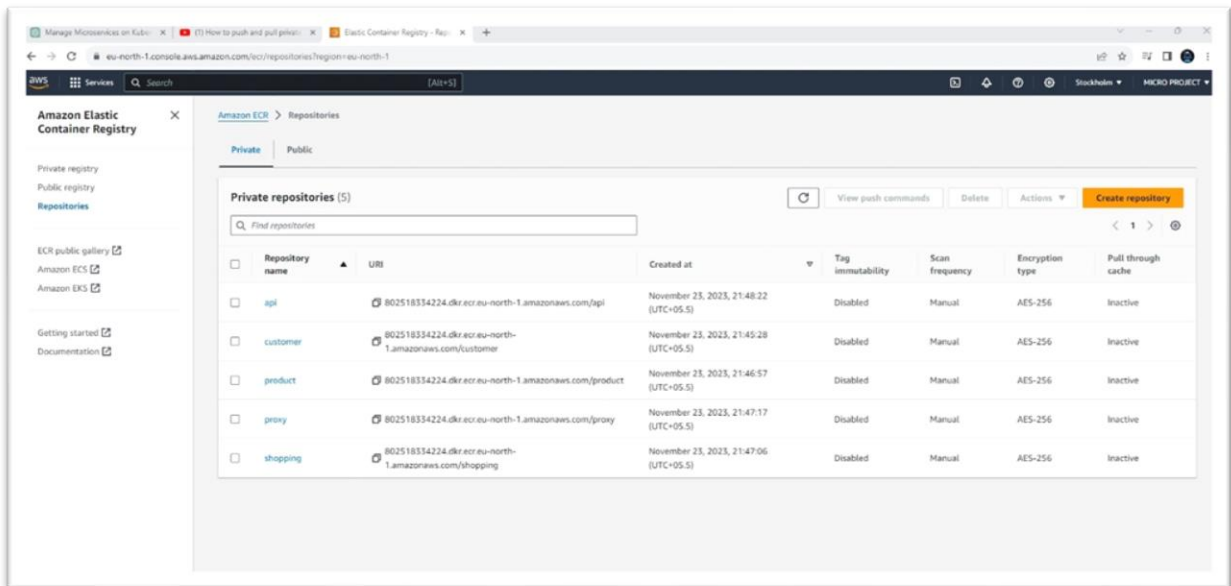
Container Orchestration and Infrastructure Automation

Deployment Process

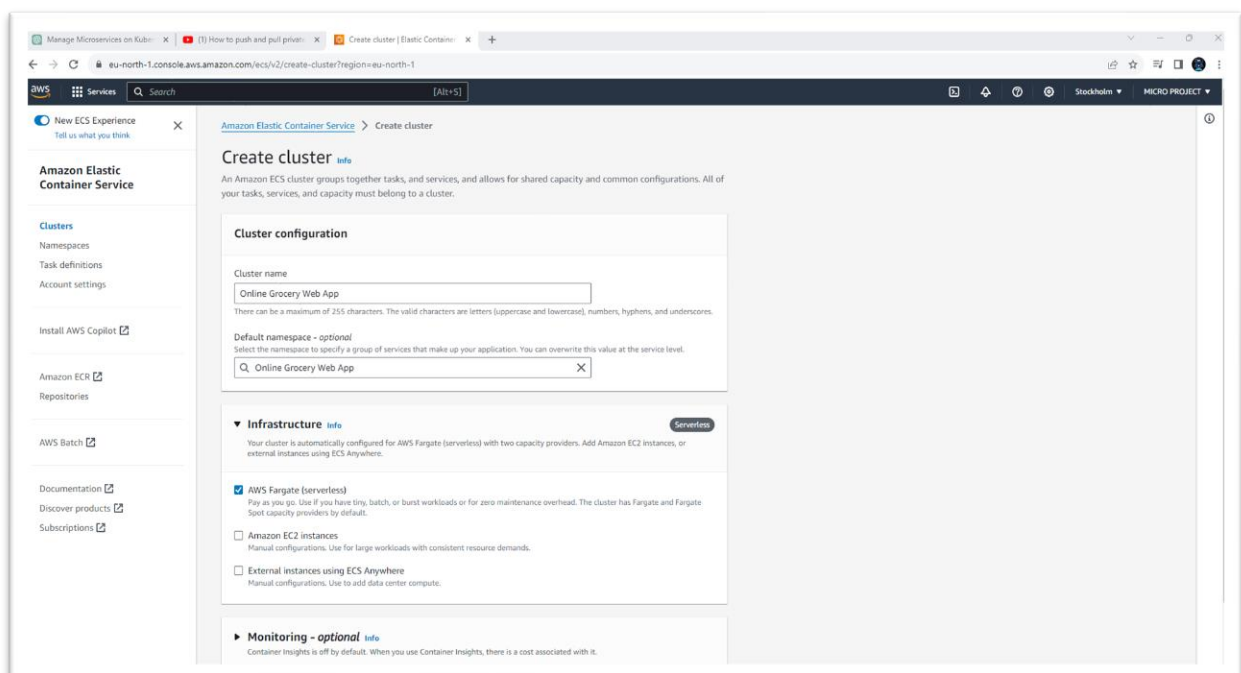
4. Provide scripts or configurations for deploying the microservices on the chosen cloud platform.

AWS ECS Deployment

Step 1: Push the images from your local machine to Amazon ECR (Elastic Container Registry):



Step 2: Create a cluster using Amazon ECS (Elastic Container Service): In this case, I opted for AWS Fargate over Amazon EC2 as it simplifies various complexities and decision-making, offering a faster and more straightforward approach. Additionally, I enabled monitoring during the cluster creation process, eliminating the need for a separate monitoring setup.



Container Orchestration and Infrastructure Automation

Step3: Task- A task is the instantiation of a task definition within a cluster. After you create a task definition for your application within Amazon ECS, you can specify the number of tasks to run on your cluster. An Amazon ECS service runs and maintains your desired number of tasks simultaneously in an Amazon ECS cluster. After creating a task, we had to wait a couple of minutes for its status to turn active.

The screenshot shows the 'Create task definition' page in the AWS Management Console. The 'Container - 1' tab is active. The 'Container details' section shows the container name 'customer', the image URI 'https://docker.io/ujeshisodia/microservice-backend-cu', and the 'Essential container' checkbox is checked. The 'Port mappings' section shows two mappings: container port 80 to host port 80 (TCP) and container port 8080 to host port 8080 (TCP). The 'Resource allocation limits' section shows CPU set to 1 vCPU, GPU set to 1, and memory limits set to 1 GB for both hard and soft limits. The 'Environment variables' section is currently empty.

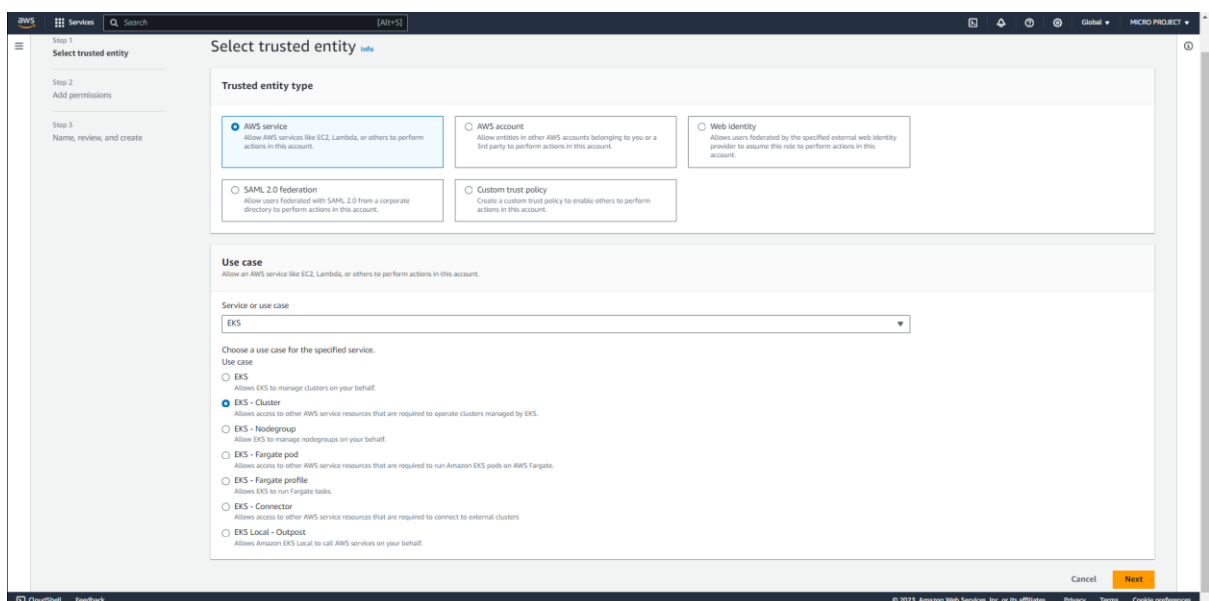
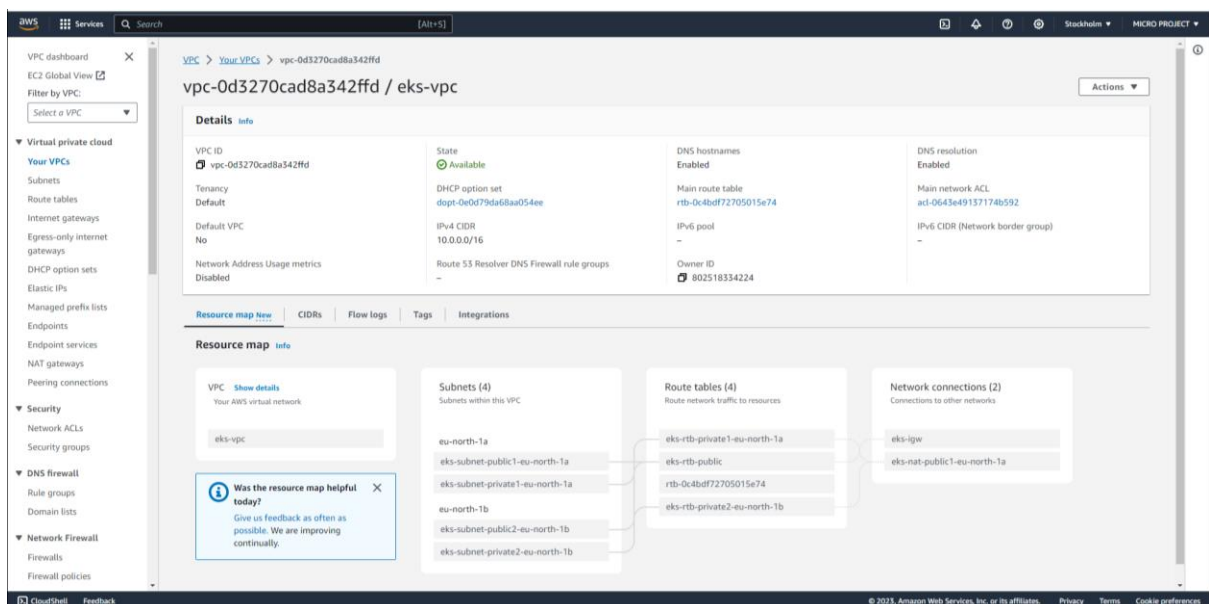
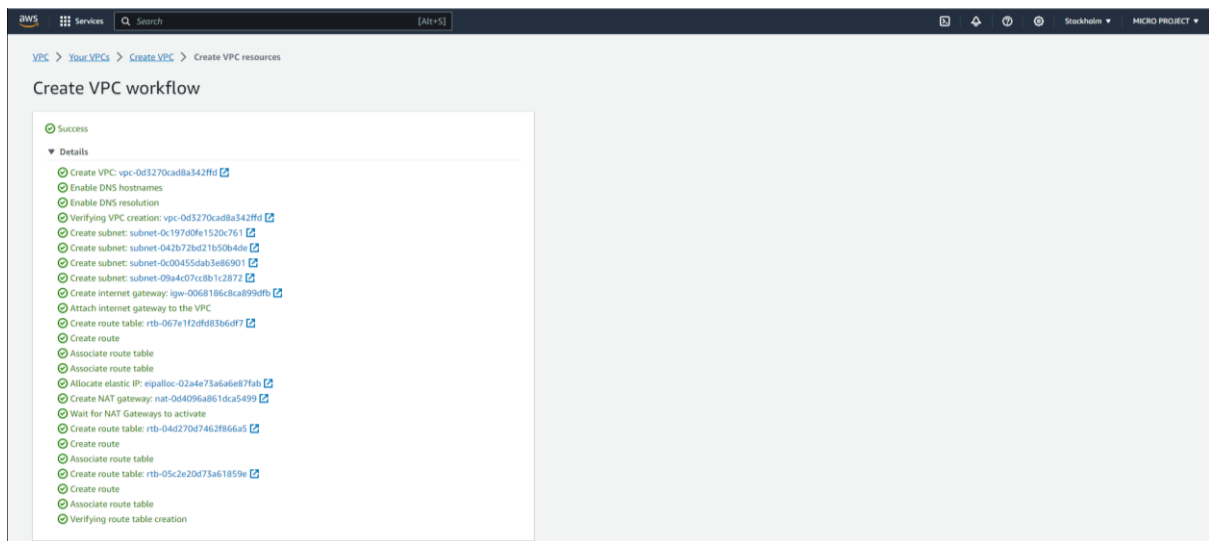
The screenshot shows the 'Task definition' page in the AWS Management Console. The 'Overview' tab is active. The 'ARN' is 'arn:aws:ecs:eu-central-1:6373947936:task-definition/TrackInventoryProject:1'. The 'Status' is 'ACTIVE'. The 'Time created' is '2023-11-11T12:34:33.627Z'. The 'App environment' is 'FARGATE'. The 'Task role' is 'ecsTaskExecutionRole'. The 'Task size' section shows 'Task CPU' as '.5 vCPU' and 'Task memory' as '1 GB'.

Step 4: To create the containers

The screenshot shows the 'Clusters' page in the AWS Management Console. The 'MicroCluster' cluster is listed with 0 services, 0 tasks, and 0 registered container instances. The 'Create cluster' button is visible in the top right corner.

Container Orchestration and Infrastructure Automation

AWS EKS Deployment



Container Orchestration and Infrastructure Automation

This screenshot shows the AWS IAM console interface for the 'eks-cluster-role'. The left sidebar contains navigation links for Identity and Access Management (IAM), Access reports, and Related services. The main content area displays the role's summary, including its creation date (November 23, 2023) and ARN (arn:aws:iam:802518334224:role/eks-cluster-role). Below the summary, there are tabs for Permissions, Trust relationships, Tags, Access Advisor, and Revoke sessions. The 'Permissions' tab is active, showing a list of permissions policies attached to the role. The list includes 'AmazonEKSClusterPolicy' and 'AmazonEKSVPCResourceController'. A 'Generate policy based on CloudTrail events' section is also visible at the bottom.

eks-cluster-role

Allows access to other AWS service resources that are required to operate clusters managed by EKS.

Summary

Creation date: November 23, 2023, 20:35 (UTC+05:30)
Last activity: -
ARN: arn:aws:iam:802518334224:role/eks-cluster-role
Maximum session duration: 1 hour

Permissions policies (1)

You can attach up to 10 managed policies.

Policy name	Type	Attached entities
AmazonEKSClusterPolicy	AWS managed	1

Permissions boundary (not set)

Generate policy based on CloudTrail events

You can generate a new policy based on the access activity for this role, then customize, create, and attach it to this role. AWS uses your CloudTrail events to identify the services and actions used and generate a policy. [Learn more](#)

[Generate policy](#)

No requests to generate a policy in the past 7 days.

This screenshot shows the 'Attach policy to eks-cluster-role' screen in the AWS IAM console. It displays a list of 'Other permissions policies' that can be attached to the role. The list is filtered by 'All types' and shows 7 matches. The policies listed include 'AmazonEKSCNIPolicy', 'AmazonEKSFargatePodExecutionRolePolicy', 'AmazonEKSLocalOutpostClusterPolicy', 'AmazonEKSServicePolicy', 'AmazonEKSVPCResourceController', 'AmazonEKSWorkerNodePolicy', and 'AWSFaultInjectionSimulatorEKSAccess'. The 'AmazonEKSVPCResourceController' policy is selected. The 'Add permissions' button is highlighted in orange.

Attach policy to eks-cluster-role

Current permissions policies (1)

Other permissions policies (1/888)

Filter by Type: All types 7 matches

Policy name	Type	Description
AmazonEKSCNIPolicy	AWS managed	This policy provides the Amazon VPC CNIP...
AmazonEKSFargatePodExecutionRolePolicy	AWS managed	Provides access to other AWS service res...
AmazonEKSLocalOutpostClusterPolicy	AWS managed	This policy provides permissions to EKS lo...
AmazonEKSServicePolicy	AWS managed	This policy allows Amazon Elastic Contain...
AmazonEKSVPCResourceController	AWS managed	Policy used by VPC Resource Controller L...
AmazonEKSWorkerNodePolicy	AWS managed	This policy allows Amazon EKS worker no...
AWSFaultInjectionSimulatorEKSAccess	AWS managed	This policy grants the Fault Injection Sim...

[Cancel](#) [Add permissions](#)

This screenshot shows the 'Name, review, and create' screen in the AWS IAM console for creating a new role. The 'Role name' field is filled with 'eks-node-role'. The 'Description' field contains 'Allows EC2 instances to call AWS services on your behalf.' The 'Step 1: Select trusted entities' section shows a trust policy for the role, which allows the role to assume the 'sts:AssumeRole' action on the 'eks2.amazonaws.com' service. The 'Step 2: Add permissions' section is also visible, showing a permissions policy summary.

Name, review, and create

Role details

Role name
Enter a meaningful name to identify this role.
eks-node-role
Maximum 64 characters. Use alphanumeric and "+-._@>" characters.

Description
Add a short explanation for this role.
Allows EC2 instances to call AWS services on your behalf.
Maximum 1000 characters. Use alphanumeric and "+-._@>" characters.

Step 1: Select trusted entities

Trust policy

```
1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Effect": "allow",
6       "Action": [
7         "sts:AssumeRole"
8       ],
9       "Principal": {
10        "Service": [
11          "eks2.amazonaws.com"
12        ]
13      }
14    ]
15  }
16 }
```

Step 2: Add permissions

Permissions policy summary

Container Orchestration and Infrastructure Automation

The image displays three sequential screenshots of the AWS Management Console, illustrating the process of creating an Amazon EKS (Elastic Kubernetes Service) cluster.

Top Screenshot: Specify networking

This screen shows the "Specify networking" step of the "Create EKS cluster" wizard. The left sidebar lists the steps: Step 1: Configure cluster, Step 2: Specify networking (current), Step 3: Configure observability, Step 4: Select add-ons, Step 5: Configure selected add-ons settings, and Step 6: Review and create.

The "Networking" section includes the following configuration options:

- VPC:** Select a VPC to use for your EKS cluster resources. To create a new VPC, go to the [VPC console](#). The selected VPC is `vpc-0d3270cad8a342fd` | `eks-vpc`.
- Subnets:** Choose the subnets in your VPC where the control plane may place elastic network interfaces (ENIs) to facilitate communication with your cluster. To create a new subnet, go to the corresponding page in the [VPC console](#). The selected subnets are:
 - `subnet-0c00455dab3a86901` | `eks-subnet-private-1-eu-north-1a` (eu-north-1a | 10.0.128.0/20)
 - `subnet-09a4c07cc8b1c2872` | `eks-subnet-private-2-eu-north-1b` (eu-north-1b | 10.0.144.0/20)
- Security groups:** Choose the security groups to apply to the EKS-managed Elastic Network Interfaces that are created in your worker node subnets. To create a new security group, go to the corresponding page in the [VPC console](#). The selected security group is `sg-06c4d17bf5e4a0bdd`.
- Choose cluster IP address family:** Specify the IP address type for pods and services in your cluster.
 - ☒ IPv4
 - ☐ IPv6
- ☐ **Configure Kubernetes service IP address range:** Specify the range from which cluster services will receive IP addresses.

Middle Screenshot: Configure cluster

This screen shows the "Configure cluster" step of the "Create EKS cluster" wizard. The left sidebar lists the steps: Step 1: Configure cluster (current), Step 2: Specify networking, Step 3: Configure observability, Step 4: Select add-ons, Step 5: Configure selected add-ons settings, and Step 6: Review and create.

The "Cluster configuration" section includes the following configuration options:

- Name:** Enter a unique name for this cluster. This property cannot be changed after the cluster is created. The cluster name should begin with letter or digit and can have any of the following characters: the set of Unicode letters, digits, hyphens and underscores. Maximum length of 100. The entered name is `eks-cluster`.
- Kubernetes version:** Select Kubernetes version for this cluster. The selected version is `1.28`. A warning message states: "Kubernetes version 1.28 reaches the end of standard support on November 2024. If you don't update your cluster to a later version before that date, it will automatically enter extended support. After the extended support preview ends, clusters on versions in extended support will be subject to additional fees. [Learn more](#)."
- Cluster service role:** Select the IAM role to allow the Kubernetes control plane to manage AWS resources on your behalf. This property cannot be changed after the cluster is created. To create a new role, follow the instructions in the [Amazon EKS User Guide](#). The selected role is `eks-cluster-role`.
- Secrets encryption:** Once turned on, secrets encryption cannot be modified or removed.
 - ☒ **Turn on envelope encryption of Kubernetes secrets using KMS**
Envelope encryption provides an additional layer of encryption for your Kubernetes secrets.

Bottom Screenshot: eks-cluster details

This screenshot shows the details page for the created EKS cluster, named `eks-cluster`. The left sidebar shows the "Amazon Elastic Kubernetes Service" section with links to "Clusters", "Amazon EKS Anywhere", "Enterprise Subscriptions", "Related services" (Amazon ECR, AWS Batch), "Documentation", and "Submit feedback".

The main content area shows the cluster status as `Active`. The "Cluster info" section displays the following details:

- Status:** Active
- Kubernetes version:** 1.28
- Support type:** Standard support until November 2024
- Provider:** EKS

The "Details" section provides further information:

- API server endpoint:** `https://812335E39A839B79AE4595299DF7A3.gr7.eu-north-1.eks.amazonaws.com`
- OpenID Connect provider URL:** `https://oidc.eks.eu-north-1.amazonaws.com/id/812335E39A839B79AE4595299DF7A3`
- Cluster IAM role ARN:** `arn:aws:iam::802518334224:role/eks-cluster-role`
- Created:** 32 minutes ago
- Cluster ARN:** `arn:aws:eks:eu-north-1:802518334224:cluster/eks-cluster`
- Platform version:** eks-4

The "Secrets encryption" section shows the option to **Enable** envelope encryption.

Container Orchestration and Infrastructure Automation

The screenshot displays the Amazon Elastic Kubernetes Service (EKS) console. The left sidebar shows the 'Amazon Elastic Kubernetes Service' menu with options like 'Clusters', 'Amazon ECR', and 'AWS Batch'. The main content area is titled 'eks-node-group' and shows the 'Node group configuration' tab. The configuration details are as follows:

Property	Value	Status
Kubernetes version	1.28	Active
AMI type	AL2_x86_64	
AMI release version	1.28.3-20231116	
Instance types	t3.medium	
Disk size	20 GiB	

Below the configuration, the 'Nodes (1)' tab is selected, showing a table of nodes:

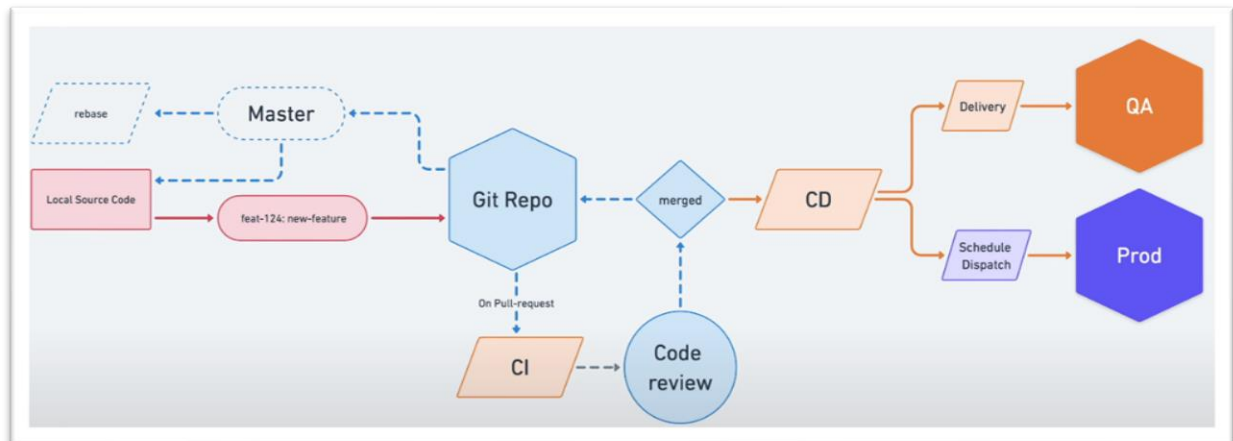
Name	Instance type	Node group	Created	Status
ip-10-0-145-30-eu-north-1-compute.internal	t3.medium	eks-node-group	Created a minute ago	Ready

The screenshot displays the Amazon Elastic Kubernetes Service (EKS) console, specifically the 'Resources' tab for the 'eks-cluster'. The left sidebar shows the 'Amazon Elastic Kubernetes Service' menu. The main content area is titled 'Resources' and shows the 'Workloads: Pods (4)' section. The pods are listed in a table:

Name	Age
aws-node-x48ft	Created 5 minutes ago
coredns-864f5db5d4-6fw6	Created 38 minutes ago
coredns-864f5db5d4-bq22h	Created 38 minutes ago
kube-proxy-xfw8	Created 5 minutes ago

Container Orchestration and Infrastructure Automation

5. Demonstrate the ability to deploy the application in a cloud environment consistently.



First of all, the Local Source code is pointed to Master. And when we working in the new feature then we check out to the new branch and that specific branch is going to work on whatever the changes we are going to do. Once it is done then we will push the source code to git repository. Afterwards once we have pushed the changes then we are going to pull the request.

And once we have pulled the request then there is a kind of settings that is Continuous Integration. This setting will take care all the test cases that what we are adding to new feature.

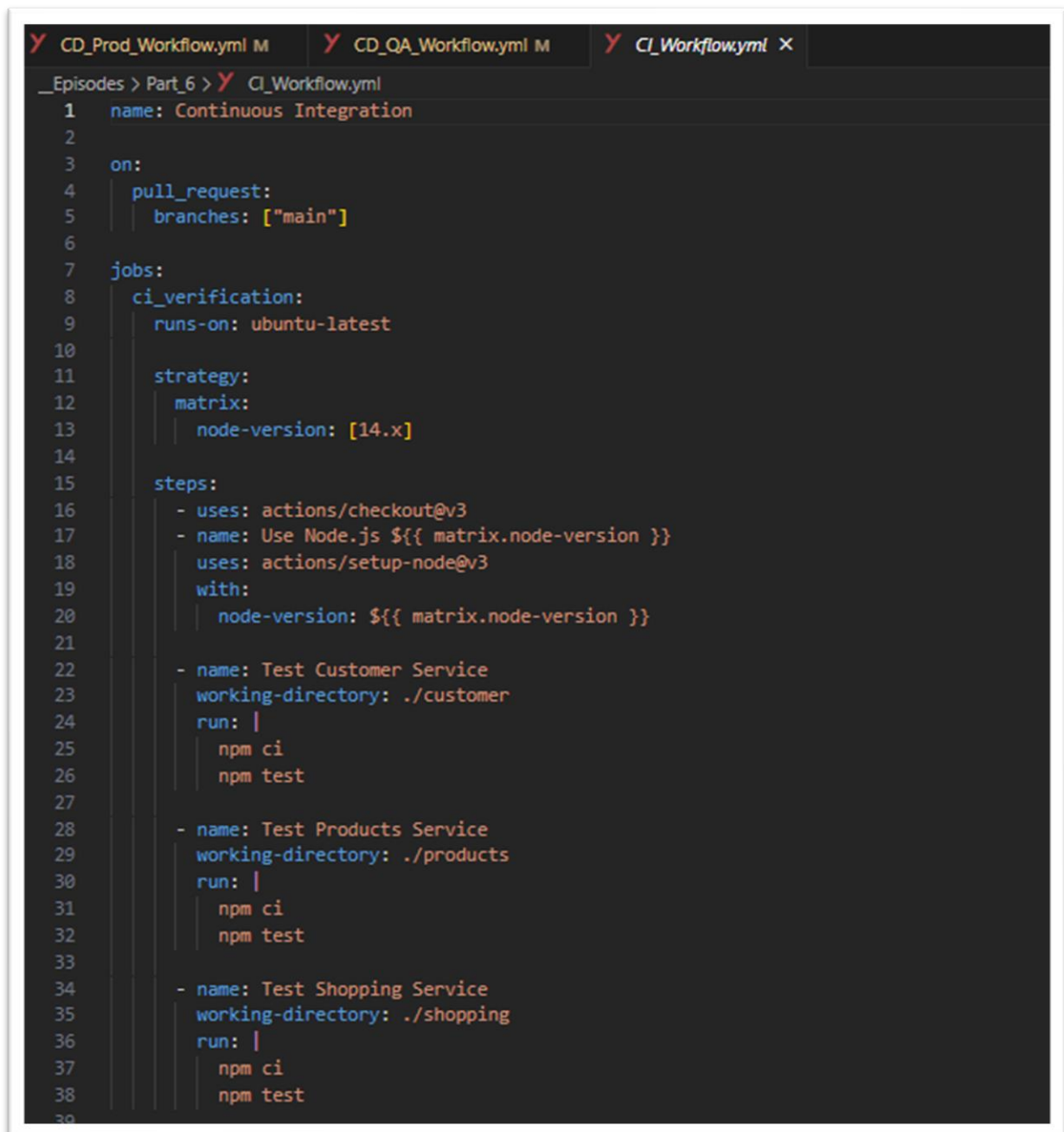
After passing all the test cases we can raise a kind of Code Review request. If the code review request is approved by other team member, then we can merge the source code to the repository. As soon as we have merged the source code it will run the **Continuous Deployment** pipeline.

It depends upon which pipeline that which **pipeline** is going to accommodate by default and depends on that it will automatically to the QA environment or maybe production environment.

Demonstration:

Container Orchestration and Infrastructure Automation

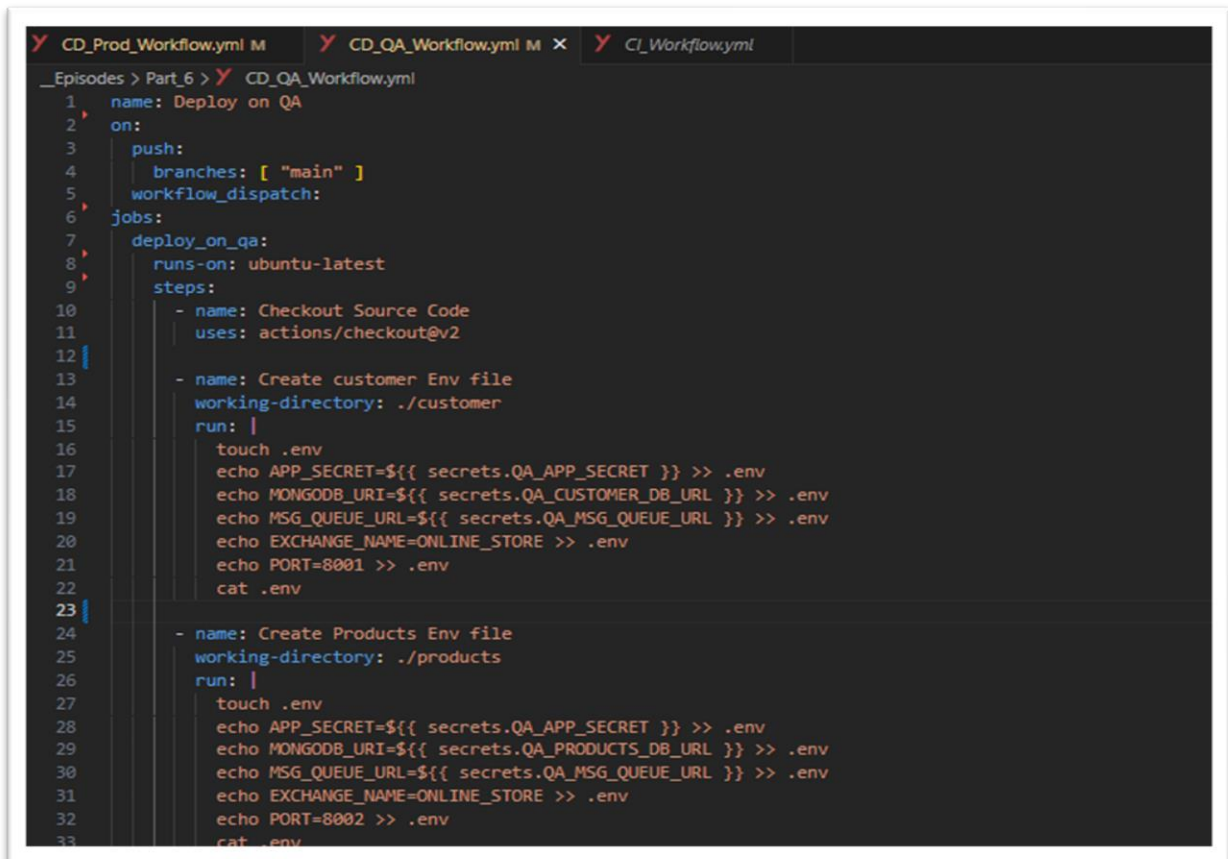
1. `CI_Workflow.yml`: this file contains all the configuration for the Continuous Improvement.



```
1 name: Continuous Integration
2
3 on:
4   pull_request:
5     branches: ["main"]
6
7 jobs:
8   ci_verification:
9     runs-on: ubuntu-latest
10
11     strategy:
12       matrix:
13         node-version: [14.x]
14
15     steps:
16       - uses: actions/checkout@v3
17       - name: Use Node.js ${ matrix.node-version }
18         uses: actions/setup-node@v3
19         with:
20           node-version: ${ matrix.node-version }
21
22       - name: Test Customer Service
23         working-directory: ./customer
24         run: |
25           npm ci
26           npm test
27
28       - name: Test Products Service
29         working-directory: ./products
30         run: |
31           npm ci
32           npm test
33
34       - name: Test Shopping Service
35         working-directory: ./shopping
36         run: |
37           npm ci
38           npm test
39
```

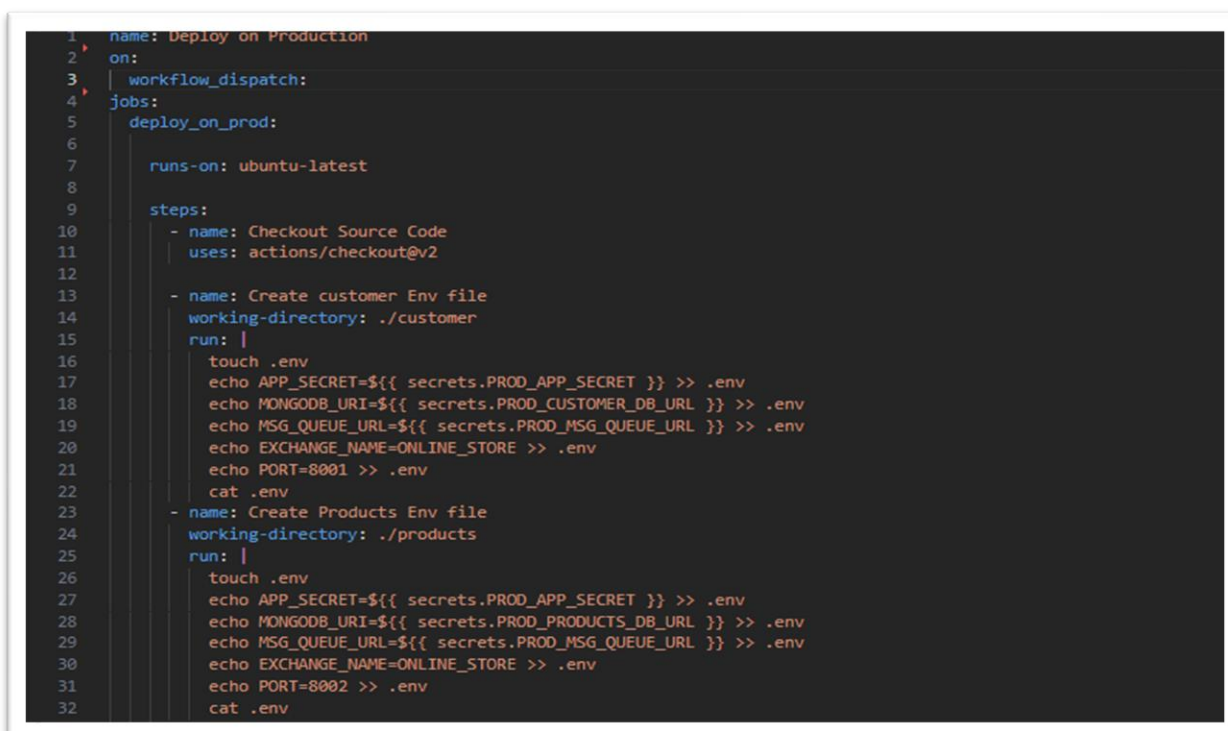
Container Orchestration and Infrastructure Automation

2. **CD_QA_Workflow.yml**: this file contains all the configuration for the Continuous Deployment.



```
1 name: Deploy on QA
2 on:
3   push:
4     branches: [ "main" ]
5   workflow_dispatch:
6 jobs:
7   deploy_on_qa:
8     runs-on: ubuntu-latest
9     steps:
10      - name: Checkout Source Code
11        uses: actions/checkout@v2
12
13      - name: Create customer Env file
14        working-directory: ./customer
15        run: |
16          touch .env
17          echo APP_SECRET=${{ secrets.QA_APP_SECRET }} >> .env
18          echo MONGODB_URI=${{ secrets.QA_CUSTOMER_DB_URL }} >> .env
19          echo MSG_QUEUE_URL=${{ secrets.QA_MSG_QUEUE_URL }} >> .env
20          echo EXCHANGE_NAME=ONLINE_STORE >> .env
21          echo PORT=8001 >> .env
22          cat .env
23
24      - name: Create Products Env file
25        working-directory: ./products
26        run: |
27          touch .env
28          echo APP_SECRET=${{ secrets.QA_APP_SECRET }} >> .env
29          echo MONGODB_URI=${{ secrets.QA_PRODUCTS_DB_URL }} >> .env
30          echo MSG_QUEUE_URL=${{ secrets.QA_MSG_QUEUE_URL }} >> .env
31          echo EXCHANGE_NAME=ONLINE_STORE >> .env
32          echo PORT=8002 >> .env
33          cat .env
```

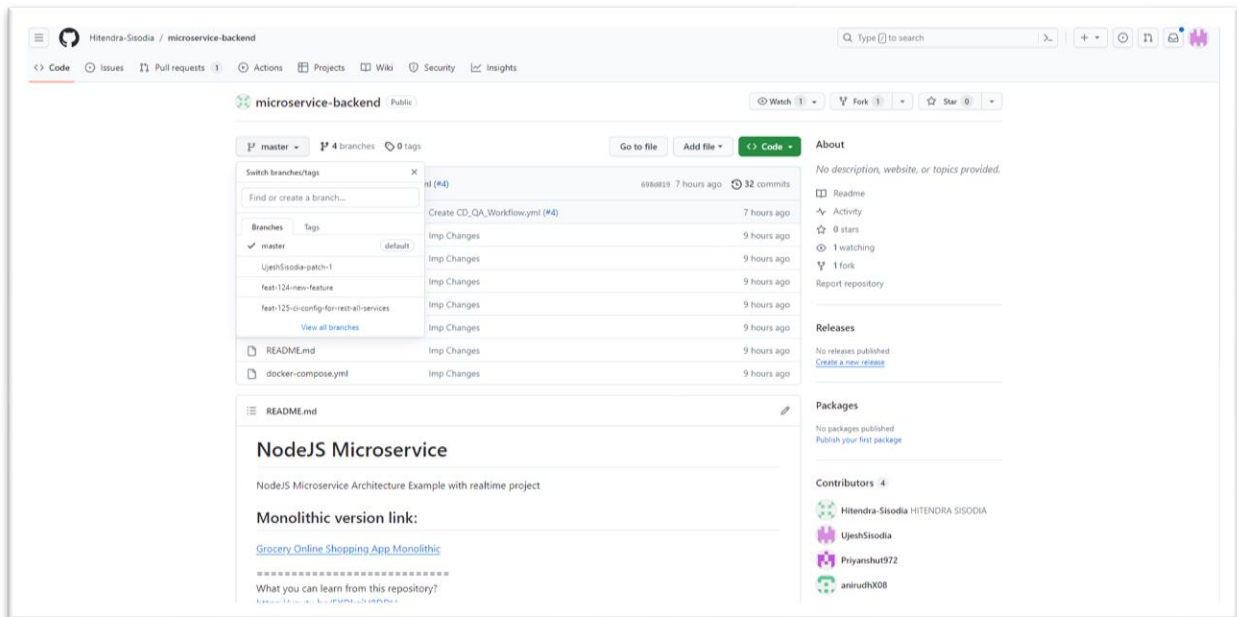
3. **CD_Prod_Workflow.yml**: this file contains all the configuration for the Continuous Deployment.



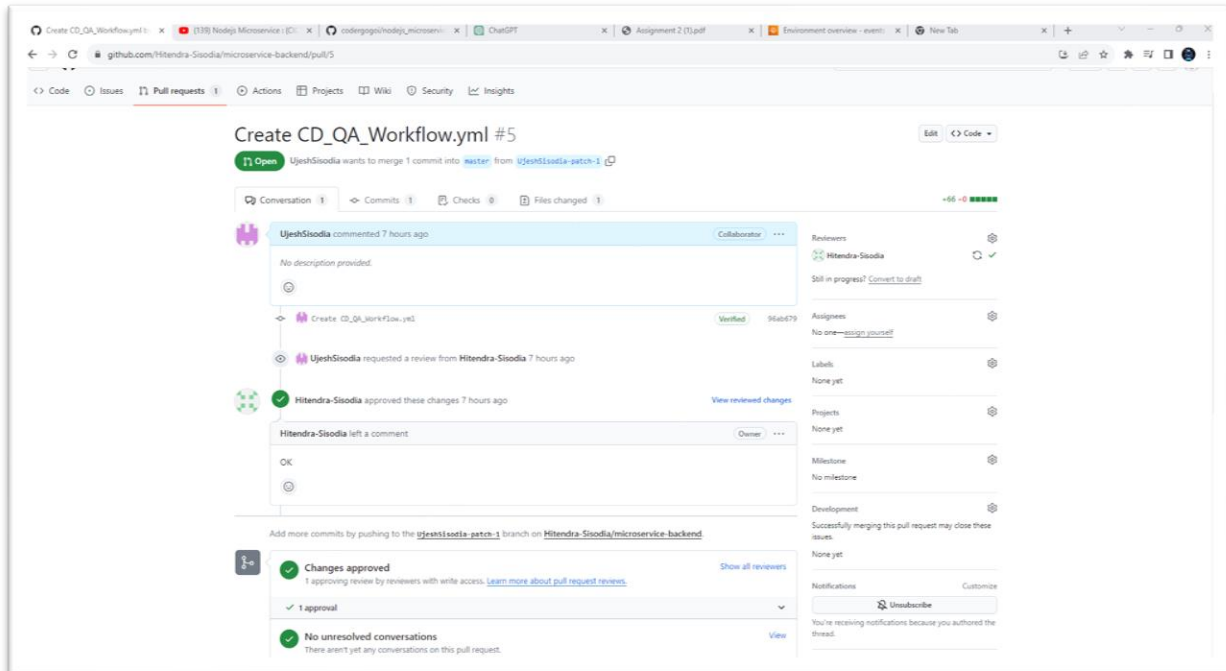
```
1 name: Deploy on Production
2 on:
3   push:
4     branches: [ "main" ]
5   workflow_dispatch:
6 jobs:
7   deploy_on_prod:
8     runs-on: ubuntu-latest
9     steps:
10      - name: Checkout Source Code
11        uses: actions/checkout@v2
12
13      - name: Create customer Env file
14        working-directory: ./customer
15        run: |
16          touch .env
17          echo APP_SECRET=${{ secrets.PROD_APP_SECRET }} >> .env
18          echo MONGODB_URI=${{ secrets.PROD_CUSTOMER_DB_URL }} >> .env
19          echo MSG_QUEUE_URL=${{ secrets.PROD_MSG_QUEUE_URL }} >> .env
20          echo EXCHANGE_NAME=ONLINE_STORE >> .env
21          echo PORT=8001 >> .env
22          cat .env
23
24      - name: Create Products Env file
25        working-directory: ./products
26        run: |
27          touch .env
28          echo APP_SECRET=${{ secrets.PROD_APP_SECRET }} >> .env
29          echo MONGODB_URI=${{ secrets.PROD_PRODUCTS_DB_URL }} >> .env
30          echo MSG_QUEUE_URL=${{ secrets.PROD_MSG_QUEUE_URL }} >> .env
31          echo EXCHANGE_NAME=ONLINE_STORE >> .env
32          echo PORT=8002 >> .env
33          cat .env
```

Container Orchestration and Infrastructure Automation

1. We have default branch i.e., master. We have created new branch feature i.e., feat-124-new-feature and feat-125-ci-config-for-all-services.

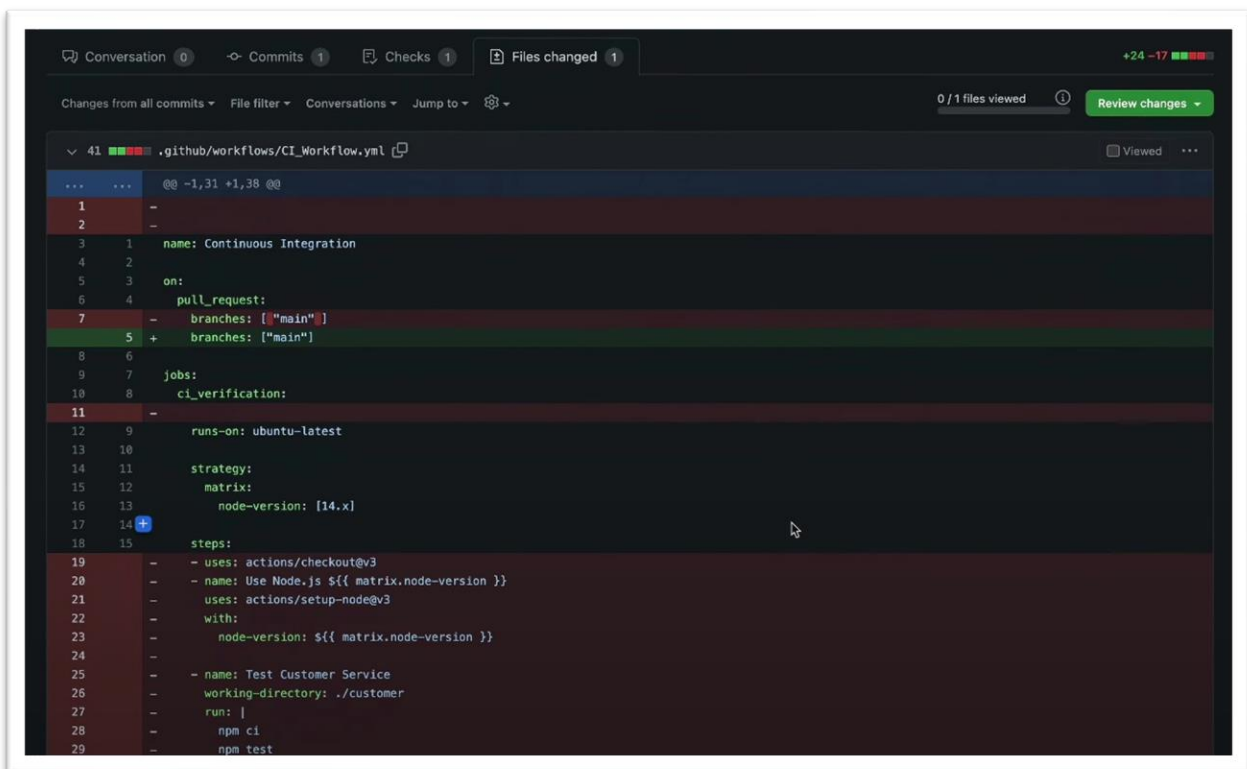


2. The changes which we have made are reviewed by our team member.

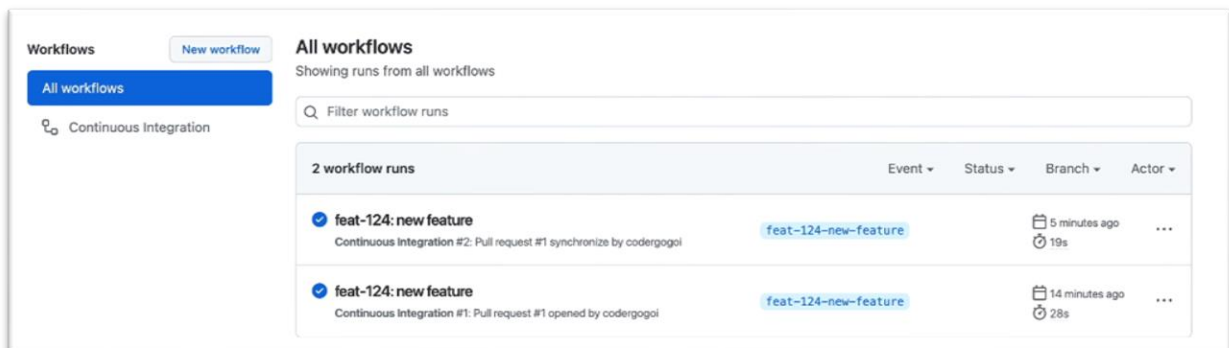


Container Orchestration and Infrastructure Automation

3. The changes which I made was adding configuration files for each microservice.



4. These are the Continuous integration changes which we have made in two new branch which we created.



Container Orchestration and Infrastructure Automation

5. The verification are as follows.

The screenshot shows the GitHub Actions interface for a workflow named **ci_verification (14.x)**. The workflow is in a "Completed" state, indicated by a green checkmark. The summary shows it succeeded in 17s. The workflow steps are listed on the right, with their respective durations:

- Set up job: 1s
- Run actions/checkout@v3: 1s
- Use Node.js 14.x: 0s
- Test Customer Service: 5s
- Test Products Service: 2s
- Test Shopping Service: 3s
- Post Use Node.js 14.x: 0s
- Post Run actions/checkout@v3: 0s
- Complete job: 0s

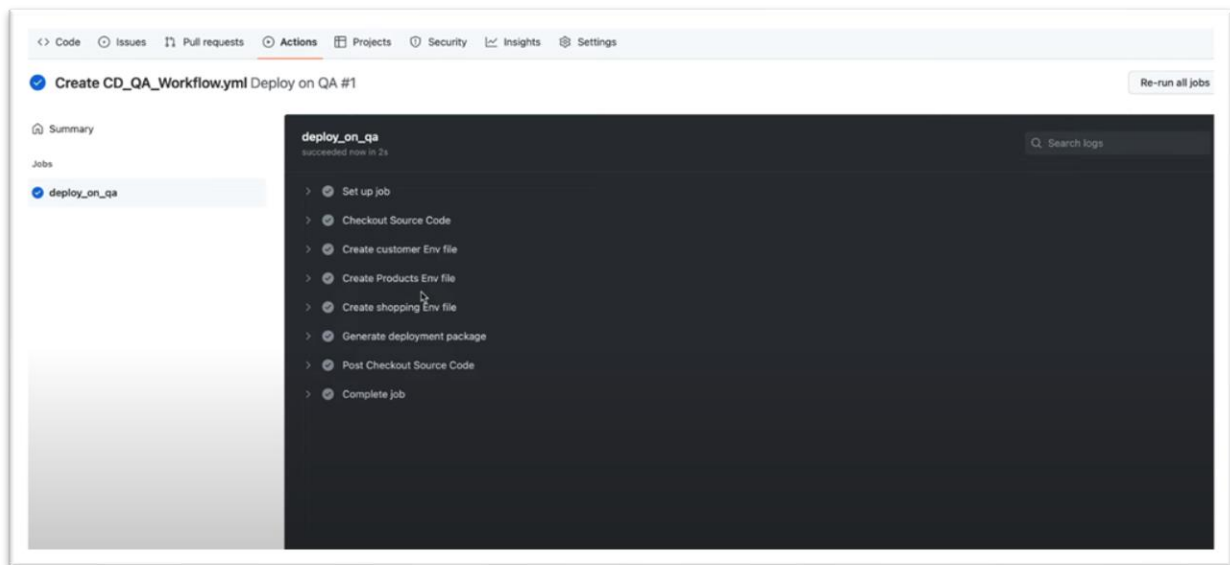
6. Creating Continuous Deployment Workflow.

The screenshot shows the GitHub Actions interface for workflow runs. The "All workflows" section is active, showing a list of workflow runs. The table below summarizes the runs:

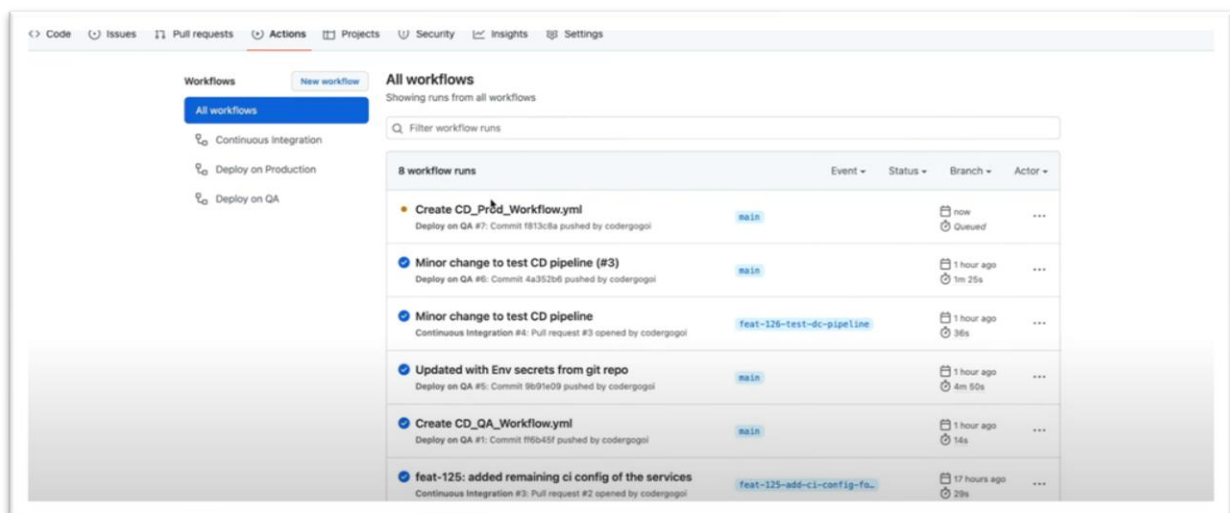
Workflow Name	Branch	Status	Event	Actor
Create CD_QA_Workflow.yml	main	Queued	now	...
feat-125: added remaining ci config of the services	feat-125-add-ci-config-fe...	Completed	16 hours ago	...
feat-124: new feature	feat-124-new-feature	Completed	16 hours ago	...
feat-124: new feature	feat-124-new-feature	Completed	16 hours ago	...

Container Orchestration and Infrastructure Automation

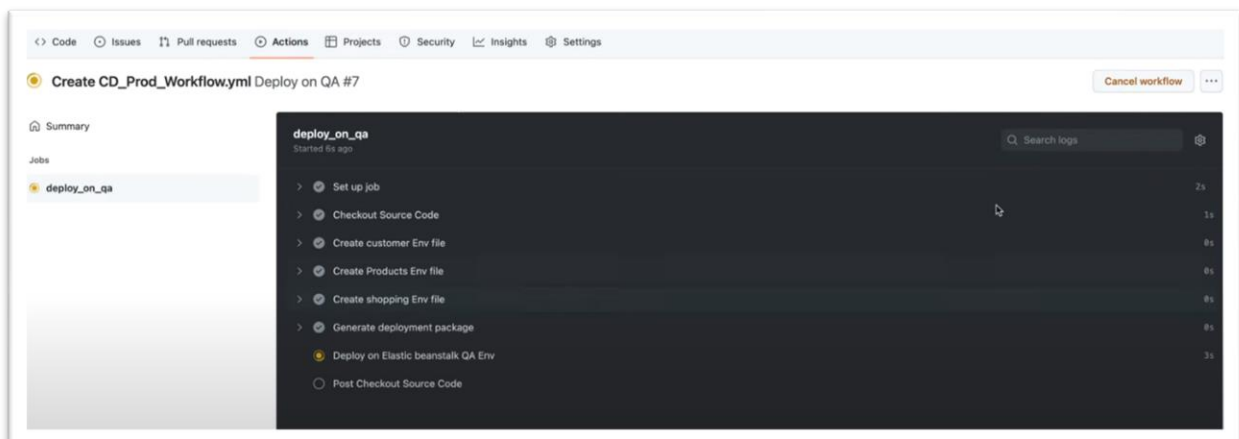
7. Now we will verify the Continuous Deployment



8. Updating the Continuous Deployment production environment in the workflow.



9. And verification are as follows



Container Orchestration and Infrastructure Automation

Scaling and Load Balancing

6. Implementing automatic scaling mechanisms for microservices based on resource usage is essential for ensuring optimal performance and resource utilization. Here's a general approach to implementing automatic scaling, assuming we're working with a container orchestration platform like Kubernetes:

Monitoring Resource Metrics: Set up monitoring for key resource metrics such as CPU utilization, memory usage, and network traffic. Tools like Prometheus and Grafana can be integrated into our Kubernetes cluster to collect and visualize these metrics.

Define Scaling Policies: Define scaling policies based on the monitored metrics. For example, we might decide to scale up the number of instances when CPU utilization exceeds a certain threshold or scale down when it falls below another threshold. Similar policies can be defined for memory usage and other relevant metrics.

Horizontal Pod Autoscaler (HPA): In Kubernetes, use the Horizontal Pod Autoscaler (HPA) to automatically adjust the number of replicas (instances) of a microservice based on observed metrics. Here's an example YAML configuration for an HPA that scales based on CPU utilization.

This HPA configuration specifies scaling behaviour based on CPU utilization, maintaining a minimum of 2 replicas and scaling up to a maximum of 10 replicas.

Customer HPA

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: customer-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: customer-service-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
```

In this example, the HPA targets a deployment named my-microservice and scales the number of replicas to maintain an average CPU utilization of 50%.

Container Orchestration and Infrastructure Automation

Products HPA

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: products-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: products-service-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
```

Shopping HPA

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: shopping-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: shopping-service-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
```


Container Orchestration and Infrastructure Automation

Vertical Pod Autoscaler (VPA): Consider using the Vertical Pod Autoscaler (VPA) if our microservices have variable resource requirements. VPA adjusts the resource requests and limits for containers based on historical usage patterns.

Customer VPA

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: customer-service-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: "Deployment"
    name: "customer-service-deployment"
  updatePolicy:
    updateMode: "Auto"
```

Products VPA

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: products-service-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: "Deployment"
    name: "customer-service-deployment"
  updatePolicy:
    updateMode: "Auto"
```

Shopping VPA

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: shopping-service-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: "Deployment"
    name: "customer-service-deployment"
  updatePolicy:
    updateMode: "Auto"
```

Container Orchestration and Infrastructure Automation

Cluster Autoscaler: For automatic scaling of nodes in our Kubernetes cluster based on overall resource demand, use the Cluster Autoscaler. This ensures that there are enough nodes to accommodate the scaling of individual microservices.

Prometheus Alert Manager: Set up alerts based on resource metrics using Prometheus Alert Manager. Define alert rules that trigger notifications when resource usage surpasses defined thresholds. These alerts can be used for both manual intervention and triggering automatic scaling.

Integration with Cloud Provider Auto Scaling: If our microservices run on a cloud platform (e.g., AWS, Google Cloud, Azure), leverage the cloud provider's auto-scaling features. For example, AWS Auto Scaling Groups can automatically adjust the number of EC2 instances based on predefined policies.

Continuous Monitoring and Tuning: Regularly review and adjust our scaling policies based on the changing characteristics of our microservices. Continuous monitoring helps identify trends and patterns that might require adjustments to scaling thresholds.

Load Testing: Conduct load testing to simulate different levels of user traffic and observe how our scaling mechanisms respond. This helps ensure that our system can handle varying workloads effectively.

Logging and Auditing: Implement logging and auditing mechanisms to capture scaling events and changes in resource usage. This information is valuable for post-event analysis and troubleshooting.

Container Orchestration and Infrastructure Automation

7. Following are the ways for achieving load balancing:

1. **Container Orchestration Platforms:** Container orchestration platforms, such as Kubernetes and Docker Swarm, provide built-in support for load balancing. These platforms manage the deployment, scaling, and operation of containers, including distributing traffic among container instances.
2. **Load balancing** in a containerized environment is a fundamental aspect of maintaining application availability, scalability, and reliability, and container orchestration platforms provide the necessary tools and abstractions to simplify its implementation.
3. **In ECS (Amazon Elastic Container Service)**, load balancing is achieved through the integration with Elastic Load Balancing (ELB). ECS allows you to define a service that manages the deployment and scaling of containers. When using a load balancer, ECS distributes incoming traffic across the tasks within the service. ELB automatically registers and deregisters container instances as they are started or stopped, ensuring a balanced distribution of traffic. This dynamic scaling and registration process allows ECS to efficiently manage containerized workloads, ensuring optimal resource utilization and high availability.
4. **Service Discovery:** Load balancing in a containerized environment starts with service discovery. Container orchestration platforms maintain a service registry that keeps track of the IP addresses and ports of running containers.
5. **Load Balancer Configuration:** Container orchestration platforms often include an internal load balancer or integrate with external load balancers.
6. **Dynamic Scaling:** Container orchestration platforms support dynamic scaling, allowing the automatic adjustment of the number of container instances based on demand.
7. **External Load Balancers:** In a cloud environment, external load balancers provided by cloud service providers (e.g., AWS Elastic Load Balancer, Google Cloud Load Balancer) can be used to distribute traffic to container instances.
8. **Load Balancing for Microservices:** In a microservices architecture, where services communicate with each other, load balancing is also applied.

Container Orchestration and Infrastructure Automation

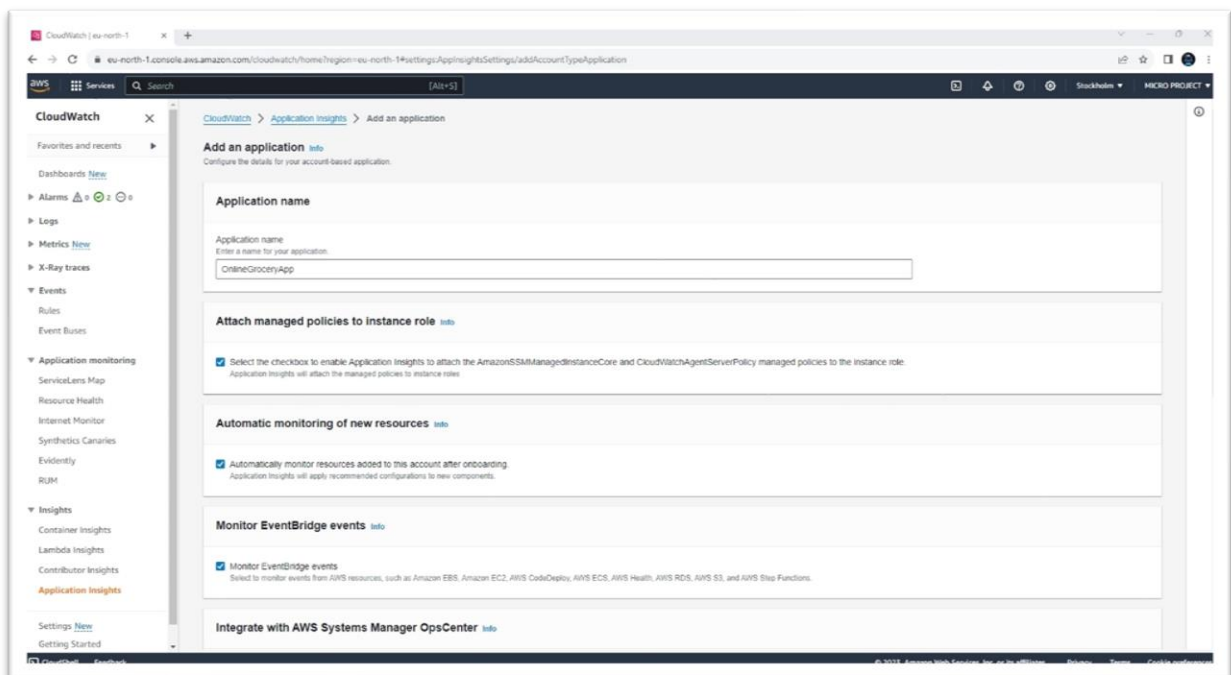
Monitoring and Logging

8. Monitoring and logging for the deployed microservices.

In the deployment of microservices, establishing robust monitoring and logging practices is paramount to ensuring the optimal performance and reliability of the application. For monitoring, services such as Amazon CloudWatch can be employed to capture real-time metrics, including CPU utilization, memory usage, and network activity. By setting up custom CloudWatch Alarms, teams can receive timely notifications or trigger automated actions when predefined thresholds are breached, enabling proactive responses to potential issues. In tandem with monitoring, logging mechanisms play a crucial role in troubleshooting and gaining insights into the microservices' behaviour. **Leveraging CloudWatch Logs**, developers can centralize log management and store logs generated by each microservice. Integrating CloudWatch Logs with additional tools like Amazon Elasticsearch and Kibana facilitates advanced log analysis, enabling efficient troubleshooting of issues and identification of performance bottlenecks.

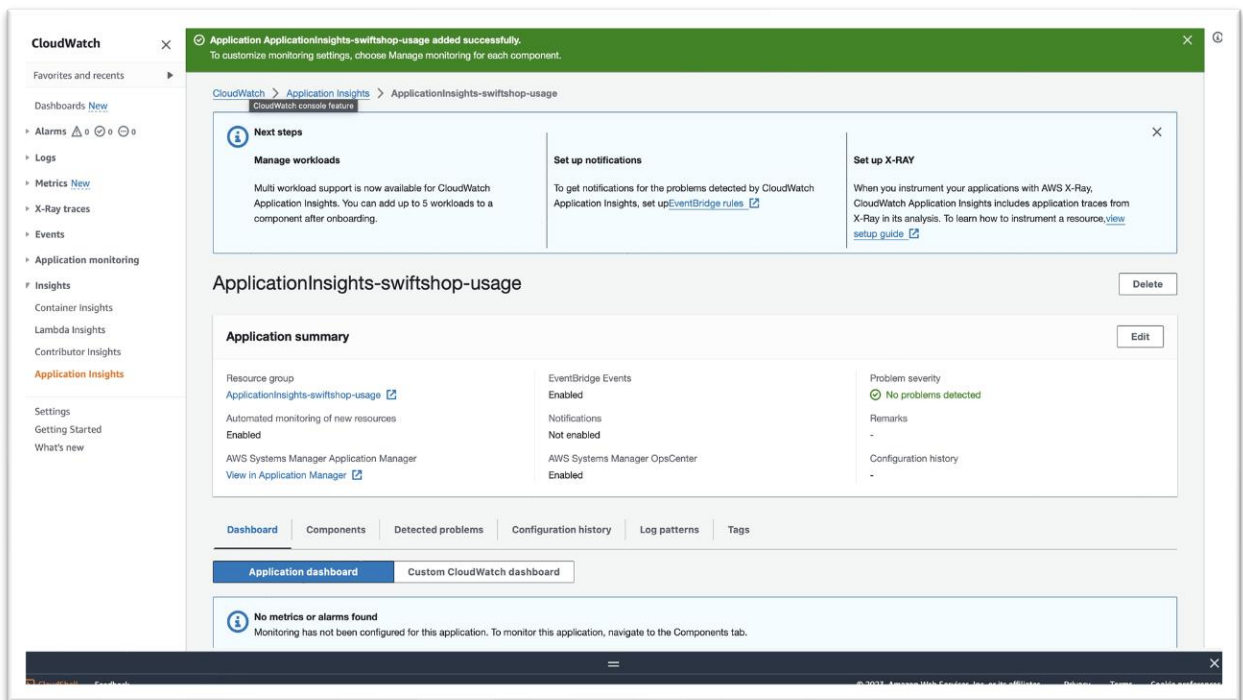
The ability to monitor application performance and troubleshoot issues is not only about collecting data but also about utilizing these insights to iterate and optimize. Continuous refinement of monitoring and logging configurations ensures that the deployed microservices remain resilient, scalable, and responsive to evolving demands, ultimately contributing to the overall health and effectiveness of the application architecture.

Application insight creation



Container Orchestration and Infrastructure Automation

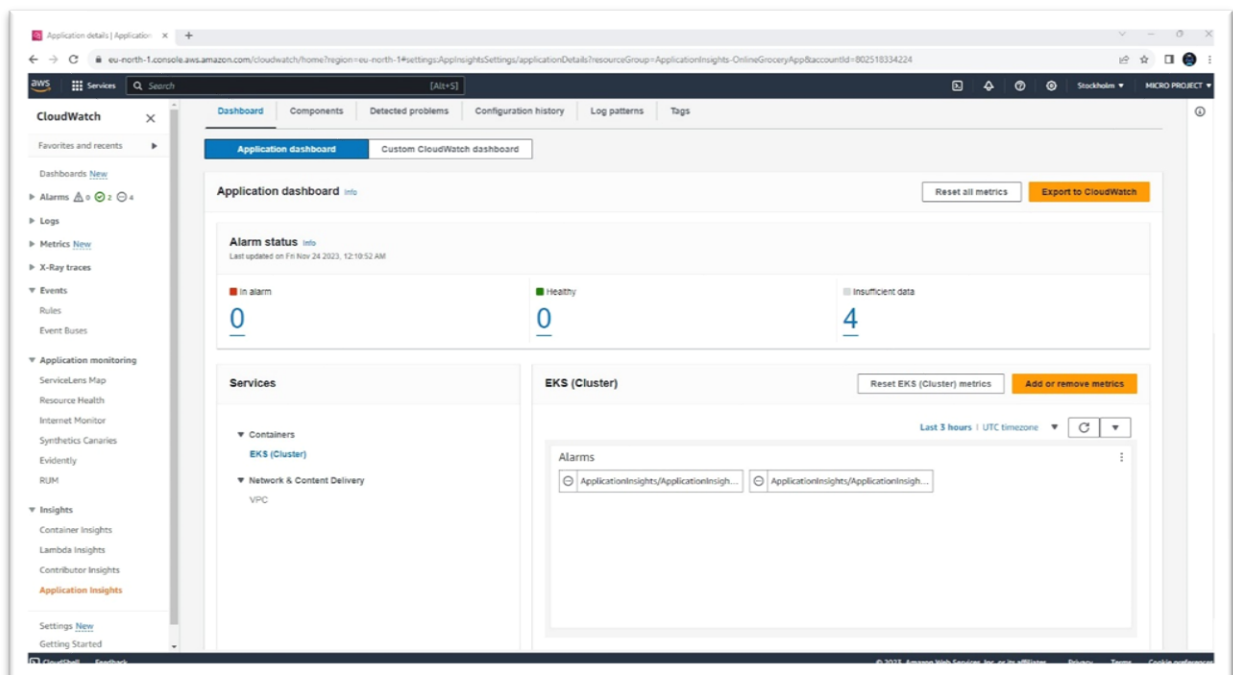
Instance creation:



9. Showcase the ability to monitor the application's performance and troubleshoot issues.

Troubleshooting:

- Using `kubectl logs` to inspecting container logs.
- Monitoring resource utilization with `kubectl top`.
- Implementing health probes for application reliability.



Container Orchestration and Infrastructure Automation

GitHub Link: <https://github.com/Hitendra-Sisodia/microservice-backend>

Overview

