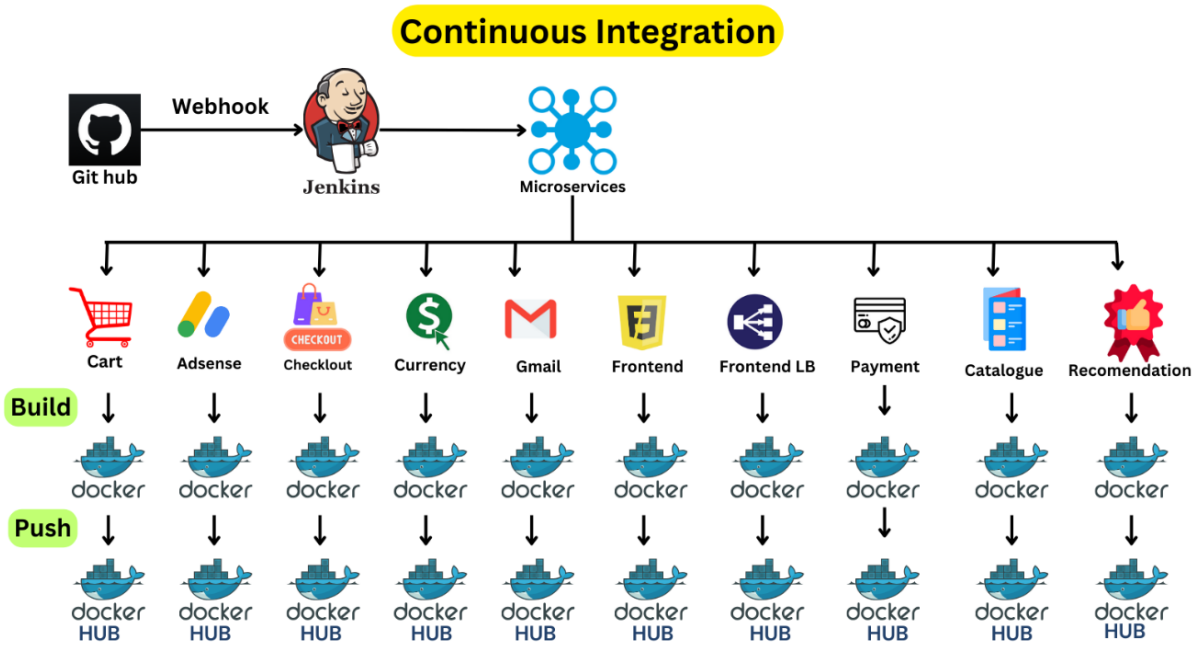
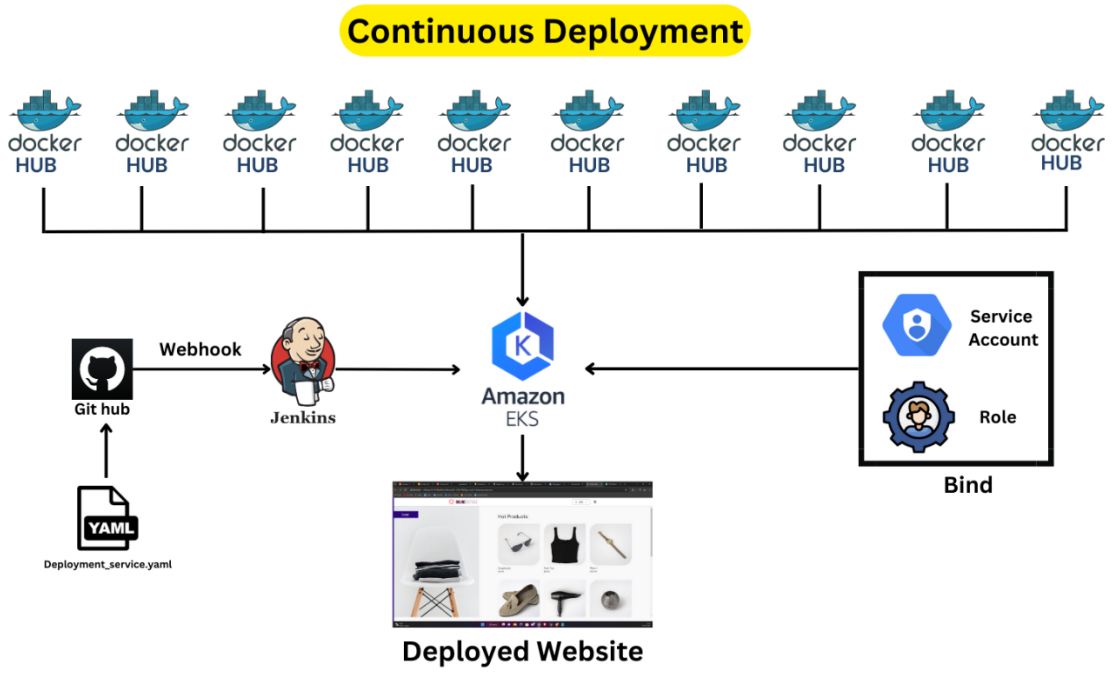
11 Microservice CI/CD Pipeline

E-Commerce Website

Why Microservices?

Microservices is an architectural style that structures an application as a collection of small, autonomous services modeled around a business domain. Each microservice is self-contained and implements a single business capability. They communicate with each other over well-defined APIs.





Project Overview:

Project Name: 11 Microservice CI/CD Pipeline

Deployment Platform: AWS EKS (Elastic Kubernetes Service)

Application Type: E-Commerce Website

Microservices Overview

Microservices Implemented:

1. Ad Sense Service
2. Cart Service
3. Checkout Service
4. Currency Service
5. Email Service
6. Frontend Service
7. External Frontend (for load balancing)
8. Payment Service
9. Product Catalogue Service
10. Recommendation Service

Reasons for Choosing Microservices:

1. Scalability:

Microservices allow individual services to scale independently based on their specific demand. For instance, the Cart Service can be scaled during high traffic periods without affecting other services.

2. Resilience:

In a microservices architecture, the failure of one service does not necessarily impact the entire system. Each service can be designed to handle failures gracefully, thereby improving the overall resilience of the application.

3. Development Speed:

Development teams can work on different services simultaneously without waiting for other teams. This parallel development accelerates the overall development process and allows for quicker releases.

4. Technology Diversity:

Each microservice can be developed using the most suitable technology stack for its specific needs. For example, the Product Catalogue Service can use a NoSQL database for better performance, while the Checkout Service might use a relational database for transaction consistency.

5. Isolation and Maintenance:

Microservices encapsulate their own logic, making it easier to understand, develop, test, and maintain. Changes in one service do not directly affect others, reducing the risk and complexity of updates.

Project Components and Pipeline

CI/CD Pipeline Using Jenkins:

Jenkinsfiles: Each microservice has its own Jenkinsfile defining the steps for building, testing, and deploying the service.

Docker: Microservices are containerized using Docker, allowing for consistent environments from development to production.

AWS EKS: The microservices are deployed on AWS EKS, a managed Kubernetes service, which orchestrates and manages the lifecycle of the containers.

Github Repository link :-

Phase 1 : Infrastructure Setup

Create a user in AWS IAM with any name

Attach Policies to the newly created user.

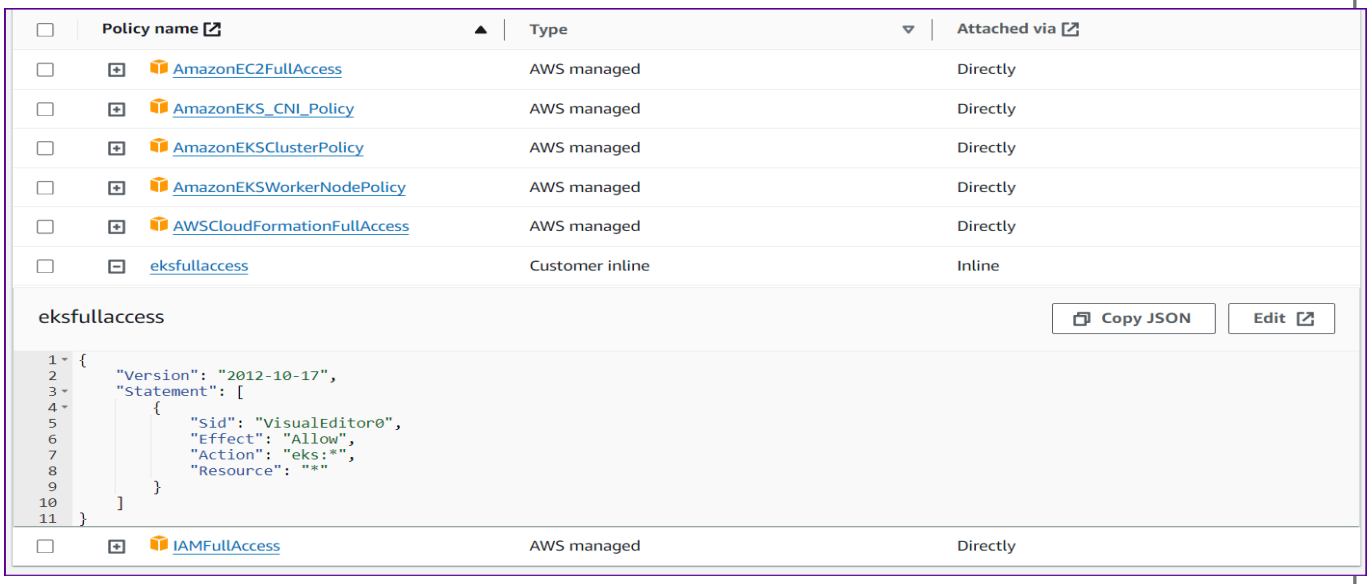
below policies:

- *AmazonEC2FullAccess*
- *AmazonEKS_CNI_Policy*
- *AmazonEKSClusterPolicy*
- *AmazonEKSWorkerNodePolicy*
- *AWSCloudFormationFullAccess*
- *IAMFullAccess*

One more **inline policy** we need to create with content as below:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": "eks:*",
      "Resource": "*"
    }
  ]
}
```

Attach this policy to your user as well.



Once IAM User is created, Create its Secret Access Key and download the **credentials.csv** file .

Launch Virtual Machine using AWS EC2

Here is a detailed list of the basic requirements and setup for the EC2 instance i have used for running Jenkins, including the specifics of the instance type, AMI, and security groups.

EC2 Instance Requirements and Setup:

1. Instance Type

- Instance Type: `t2.large`

  - vCPUs: 2

  - Memory: 8 GB

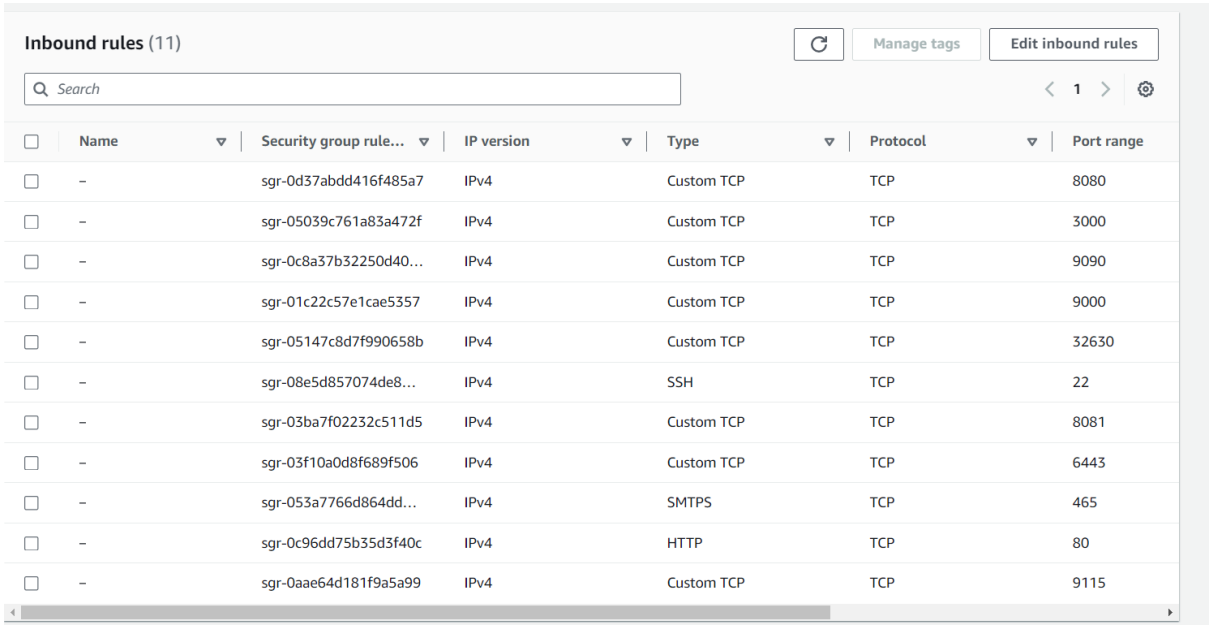
  - Network Performance: Moderate

2. Amazon Machine Image (AMI)

- AMI: Ubuntu Server 20.04 LTS (Focal Fossa)

3. Security Groups

Security groups act as a virtual firewall for your instance to control inbound and outbound traffic.



Inbound rules (11)

Manage tags | Edit inbound rules

Search

1

Name	Security group rule...	IP version	Type	Protocol	Port range
-	sgr-0d37abdd416f485a7	IPv4	Custom TCP	TCP	8080
-	sgr-05039c761a83a472f	IPv4	Custom TCP	TCP	3000
-	sgr-0c8a37b32250d40...	IPv4	Custom TCP	TCP	9090
-	sgr-01c22c57e1cae5357	IPv4	Custom TCP	TCP	9000
-	sgr-05147c8d7f990658b	IPv4	Custom TCP	TCP	32630
-	sgr-08e5d857074de8...	IPv4	SSH	TCP	22
-	sgr-03ba7f02232c511d5	IPv4	Custom TCP	TCP	8081
-	sgr-03f10a0d8f689f506	IPv4	Custom TCP	TCP	6443
-	sgr-053a7766d864dd...	IPv4	SMTPS	TCP	465
-	sgr-0c96dd75b35d3f40c	IPv4	HTTP	TCP	80
-	sgr-0aae64d181f9a5a99	IPv4	Custom TCP	TCP	9115

After Launching your Virtual machine ,**SSH** into the Server.

Install AWS CLI , EKSCTL & KUBECTL on VM Server

AWSCLI

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
sudo apt install unzip
unzip awscliv2.zip
sudo ./aws/install
aws configure
```

KUBECTL

```
curl -o kubectl https://amazon-eks.s3.us-west-2.amazonaws.com/1.19.6/2021-01-05/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin
kubectl version --short --client
```

EKSCTL

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

Save all the script in a file, for example, ctl.sh, and make it executable using:

```
chmod +x ctl.sh
```

Then, you can run the script using:

```
./ctl.sh
```

Installing Jenkins on Ubuntu

Execute these commands on Jenkins Server

```
#!/bin/bash
# Install OpenJDK 17 JRE Headless
sudo apt install openjdk-17-jre-headless -y
# Download Jenkins GPG key
sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \
https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
# Add Jenkins repository to package manager sources
echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \
https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
/etc/apt/sources.list.d/jenkins.list > /dev/null
# Update package manager repositories
sudo apt-get update
# Install Jenkins
sudo apt-get install jenkins -y
```

Save this script in a file, for example, install_jenkins.sh, and make it executable using:

```
chmod +x install_jenkins.sh
```

Then, you can run the script using:

```
./install_jenkins.sh
```

This script will automate the installation process of OpenJDK 17 JRE Headless and Jenkins.

Create EKS Cluster

```
eksctl create cluster --name=EKS-1 \
                --region=ap-south-1 \
                --zones=ap-south-1a,ap-south-1b \
                --without-nodegroup
```

Open ID Connect

```
eksctl utils associate-iam-oidc-provider \
  --region ap-south-1 \
  --cluster EKS-1 \
  --approve
```

Create node Group

```
eksctl create nodegroup --cluster=EKS-1 \
                --region=ap-south-1 \
                --name=node2 \
                --node-type=t3.medium \
                --nodes=3 \
                --nodes-min=2 \
                --nodes-max=4 \
                --node-volume-size=20 \
                --ssh-access \
                --ssh-public-key=DevOps \
                --managed \
                --asg-access \
                --external-dns-access \
                --full-ecr-access \
                --appmesh-access \
                --alb-ingress-access
```

Make sure to change the name of **ssh-public-Key** with your SSH key.

Phase 2 : Multi-branch Pipeline Setup

Step 1: Install Jenkins Plugins

To get started, you need to install the required Jenkins plugins. Follow these steps to install the plugins:

Access Jenkins Dashboard:

Open a web browser and navigate to your Jenkins instance (e.g., http://your-instance-public-dns:8080).

Log in with your Jenkins credentials. (cat address provided on Jenkins)

Install Plugins:

-Go to Manage Jenkins > Manage Plugins.

-Click on the Available tab.

Search for and install the following plugins:

- **Docker**: Enables Jenkins to use Docker containers.
- **Docker Pipeline**: Allows Jenkins to use Docker containers in pipeline jobs.
- **Kubernetes**: Provides support for Kubernetes in Jenkins.
- **Kubernetes CLI**: Allows Jenkins to interact with Kubernetes clusters.
- **Multibranch Scan Webhook Trigger:** Adds webhook trigger functionality for multibranch projects.

Step 2: Create Credentials

You need to create credentials for Docker and GitHub access.

Create Docker Credentials:

-Go to Manage Jenkins > Manage Credentials > (global) > Add Credentials.

-Choose Username with password as the kind.

-ID: docker-cred

-Username: Your Docker Hub username.

-Password: Your Docker Hub password.

-Click OK.

Create GitHub Credentials:

-Go to Manage Jenkins > Manage Credentials > (global) > Add Credentials.

-Choose Secret text as the kind.

-ID: git-cred

-Secret: Your GitHub Personal Access Token.

-Click OK.

Step 3: Configure Multibranch Pipeline

Create a New Multibranch Pipeline:

-Go to New Item.

-Enter a name for your project (e.g., microservice-pipeline).

-Select Multibranch Pipeline and click OK.

-Configure Branch Sources:

-Under Branch Sources, click Add source.

-Choose Git.

Project Repository: Enter the repository URL

Credentials: Select the git-cred credentials.

Configure Build Configuration:

Under Build Configuration, ensure by **Jenkinsfile** is selected.

Script Path: Leave it as the default (**Jenkinsfile**).

Configure Webhook Trigger:

-Under Scan Repository Triggers, click Add.

-Select Multibranch Scan Webhook Trigger.

Trigger Token: Enter a token name (e.g., **xxx**).

Webhook URL:

Note the webhook URL: http://your-jenkins-instance:8080/multibranch-webhook-trigger/invoke?token=**xxx**

Replace your-jenkins-instance with the public DNS or IP address of your Jenkins instance.

Replace **xxx** with the token name you provided.

Step 4: Set Up GitHub Webhook

Go to GitHub Repository Settings:

-Navigate to your repository on GitHub.

-Go to Settings > Webhooks.

Add Webhook:

-Click Add webhook.

-Payload URL: Enter the webhook URL you noted earlier (http://your-jenkins-instance:8080/multibranch-webhook-trigger/invoke?token= **xxx**).

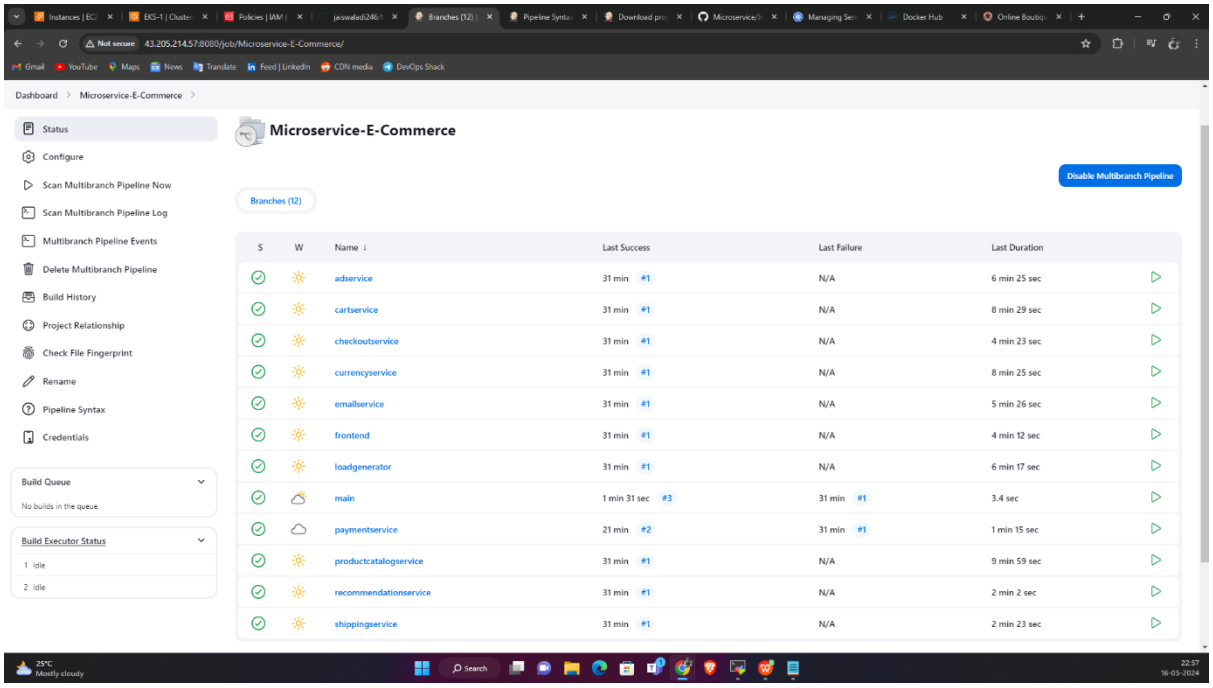
-Content type: Select application/json.

-Secret: Leave it blank.

-Select Let me select individual events and choose Push events.

-Click Add webhook.

After completing this you will see your Jenkins multi-branch Pipeline starts build automatically.



Phase 3 : Continuous Deployment

Run these commands on Server

Create Service Account, Role & Assign that role, And create a secret for Service Account and generate a Token. We will Deploy our Application on the main branch .

Create a file : Vim **svc.yml**

Creating Service Account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins
  namespace: webapps
```

To run the svc.yml : kubectl apply -f svc.yaml

Similarly create a **role.yml** file

Create Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: app-role
  namespace: webapps
rules:
  - apiGroups:
      - ""
      - apps
      - autoscaling
```

```
    - batch
    - extensions
    - policy
    - rbac.authorization.k8s.io
  resources:
    - pods
    - componentstatuses
    - configmaps
    - daemonsets
    - deployments
    - events
    - endpoints
    - horizontalpodautoscalers
    - ingress
    - jobs
    - limitranges
    - namespaces
    - nodes
    - pods
    - persistentvolumes
    - persistentvolumeclaims
    - resourcequotas
    - replicasets
    - replicationcontrollers
    - serviceaccounts
    - services
```

```
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

To run the role.yaml file: `kubectl apply -f role.yaml`

Similarly create a **bind.yml** file

Bind the role to service account

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-rolebinding
  namespace: webapps
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: app-role
subjects:
- namespace: webapps
  kind: ServiceAccount
  name: jenkins
```

To run the bind.yaml file: `kubectl apply -f bind.yaml`

Create Token

Similarly create a secret.yml file

```
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: mysecretname
  annotations:
    kubernetes.io/service-account.name: Jenkins
```

To run the secret.yml file: `kubectl apply -f secret.yml -n webapps`

Save the token .

-Create a dummy job in your Jenkins with Pipeline job and go to the **pipeline syntax** and select **With Kubernetes:Configure Kubernetes**

1. **Credentials** – Provide the Token that you have saved .
2. **Kubernates Endpoint API**- You can find it in your AWS EKS cluster.
3. **Cluster name**- Provide any name.
4. **NameSpace** – webapps

Click on Generate Syntax.

You will get pipeline syntax :-

```
withKubeCredentials(kubectlCredentials: [[caCertificate: '', clusterName: 'EKS-1', contextName: '', credentialsId: 'k8-token', namespace: 'webapps', serverUrl: 'https://B7C7C20487B2624AAB0AD54DF1469566.yl4.ap-south-1.eks.amazonaws.com']]) {
//block of code
        }
```

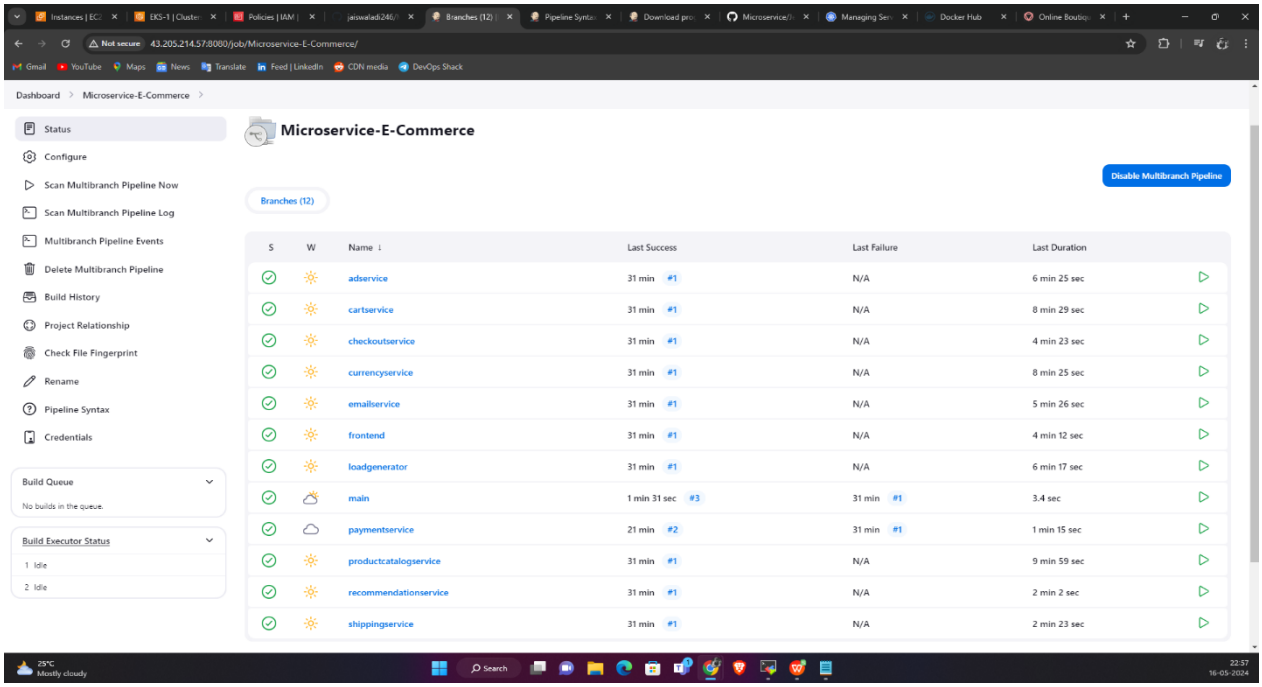
Create a Jenkinsfile on the main branch .

```
pipeline {
   agent any

   stages {
      stage('Deploy To Kubernetes') {
         steps {
            withKubeCredentials(kubectlCredentials: [[caCertificate: '', clusterName: 'EKS-1', contextName: '', credentialsId: 'k8-token', namespace: 'webapps', serverUrl: 'https://B7C7C20487B2624AAB0AD54DF1469566.yl4.ap-south-1.eks.amazonaws.com']]) {
               sh "kubectl apply -f deployment-service.yml"

            }
         }
      }

      stage('verify Deployment') {
         steps {
            withKubeCredentials(kubectlCredentials: [[caCertificate: '', clusterName: 'EKS-1', contextName: '', credentialsId: 'k8-token', namespace: 'webapps', serverUrl: 'https://B7C7C20487B2624AAB0AD54DF1469566.yl4.ap-south-1.eks.amazonaws.com']]) {
               sh "kubectl get svc -n webapps"
            }
         }
      }
   }
}
```

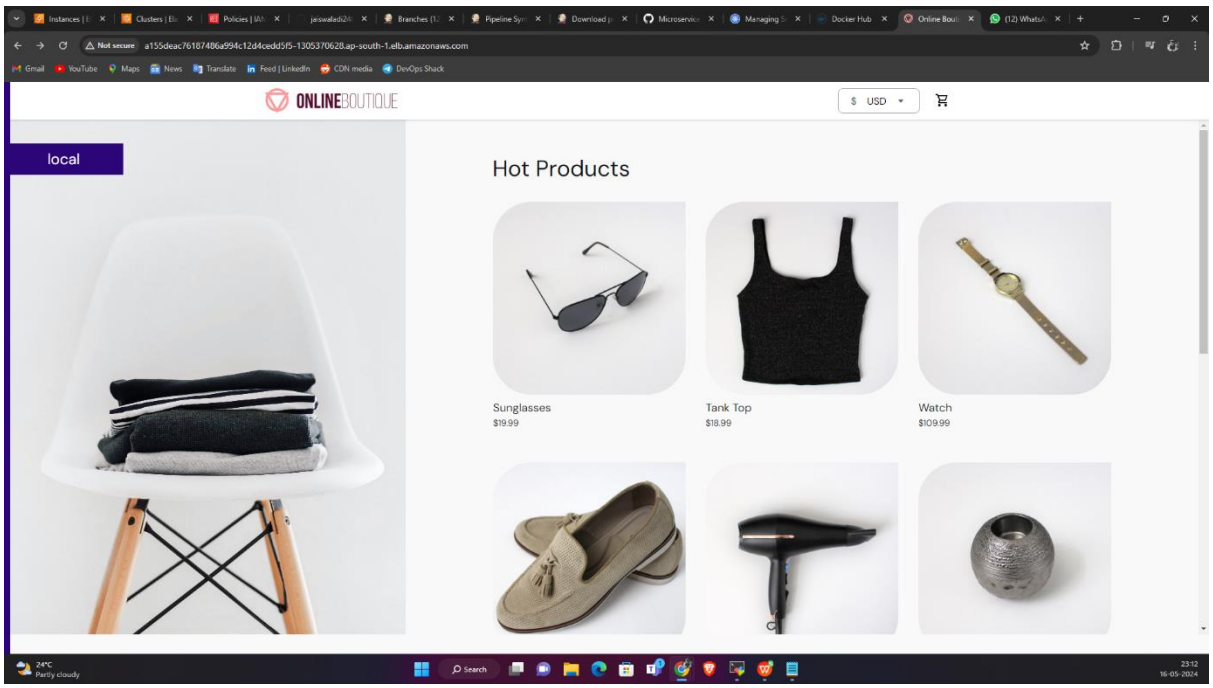
As you Jenkinsfile is created your Pipeline will automatically start the build and Deployment.

Results

PIPELINE



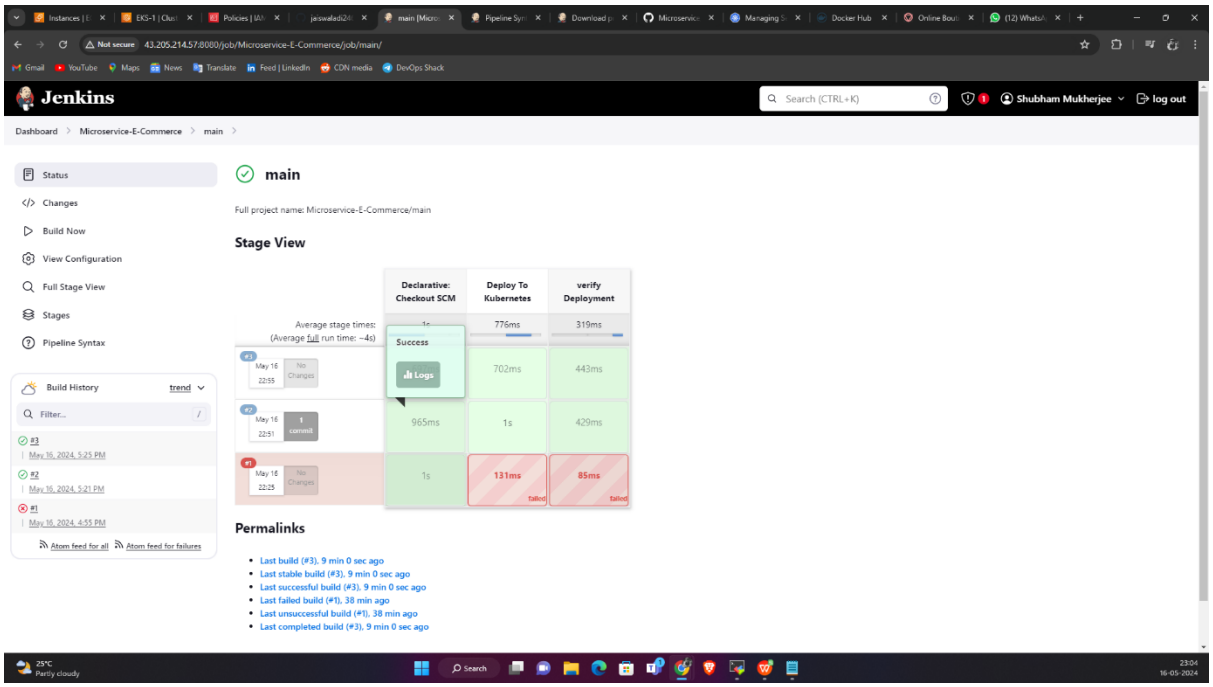
APPLICATION



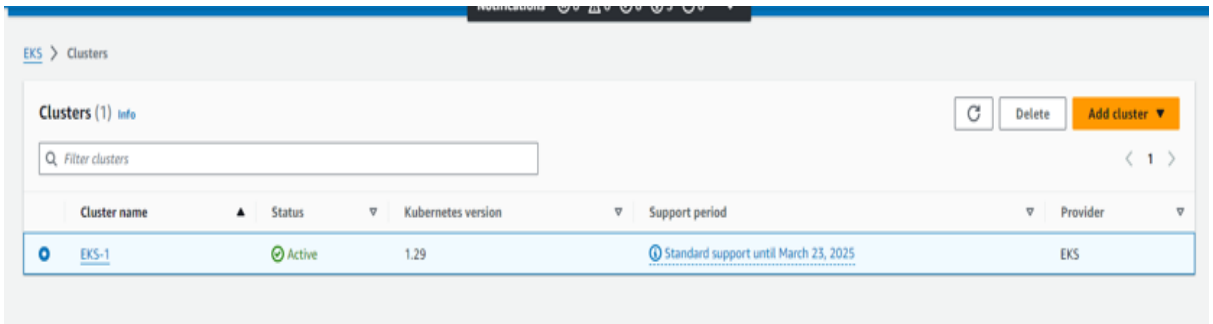
CONSOLE OUTPUT

```
+ kubectl get svc -n webapps
NAME                  TYPE           CLUSTER-IP       EXTERNAL-IP                                                              PORT(S)        AGE
adservice             ClusterIP      10.100.103.174   <none>                                                                   9555/TCP       3m52s
cartservice           ClusterIP      10.100.237.236   <none>                                                                   7070/TCP       3m53s
checkoutservice       ClusterIP      10.100.102.76    <none>                                                                   5050/TCP       3m53s
currencyservice       ClusterIP      10.100.9.201     <none>                                                                   7000/TCP       3m53s
emailservice          ClusterIP      10.100.140.105   <none>                                                                   5000/TCP       3m53s
frontend              NodePort       10.100.19.87     <none>                                                                   80:32183/TCP   3m53s
frontend-external     LoadBalancer   10.100.146.35    a155deac76187486a994c12d4cedd5f5-1305370628.ap-south-1.elb.amazonaws.com  80:32195/TCP   3m53s
paymentservice        ClusterIP      10.100.227.91    <none>                                                                   50051/TCP      3m53s
productcatalogservice ClusterIP      10.100.107.131   <none>                                                                   3550/TCP       3m53s
recommendationservice ClusterIP      10.100.146.47    <none>                                                                   8080/TCP       3m53s
redis-cart            ClusterIP      10.100.7.161     <none>                                                                   6379/TCP       3m53s
shippingservice       ClusterIP      10.100.23.22     <none>                                                                   50051/TCP      3m53s
[Pipeline] }
```

MAIN BRANCH PIPELINE



EKS CLUSTER



Project Impact

The implementation of this 11 Microservice CI/CD Pipeline project provides several significant benefits:

Improved Efficiency: Automation of the build and deployment process reduces the time and effort required for manual deployments, leading to faster delivery cycles.

Enhanced Reliability: Automated testing and deployment help in identifying and resolving issues early, improving the overall reliability and quality of the application.

Scalability: The microservices architecture and Kubernetes orchestration enable the application to handle varying loads efficiently, ensuring consistent performance and availability.

Maintainability: Independent development and deployment of services simplify the maintenance and updates, reducing the complexity and risk of making changes.

Acknowledgment

Special thanks to Aditya Jaiswal of "DevOps Shack" on YouTube for his invaluable guidance and tutorials, which were instrumental in the successful completion of this project. Thank you, Aditya!

Conclusion

The successful deployment of the e-commerce website using a microservices architecture and an automated CI/CD pipeline on AWS EKS exemplifies the advantages of modern software development practices. This project not only achieves the goals of scalability, reliability, and efficiency but also sets a strong foundation for future enhancements and growth. The methodologies and technologies employed in this project can serve as a blueprint for similar projects aiming to leverage microservices and CI/CD pipelines for continuous innovation and delivery.