

Task 12

Jenkins Pipeline Setup for ECS Deployment

Umar Satti

Table of Contents

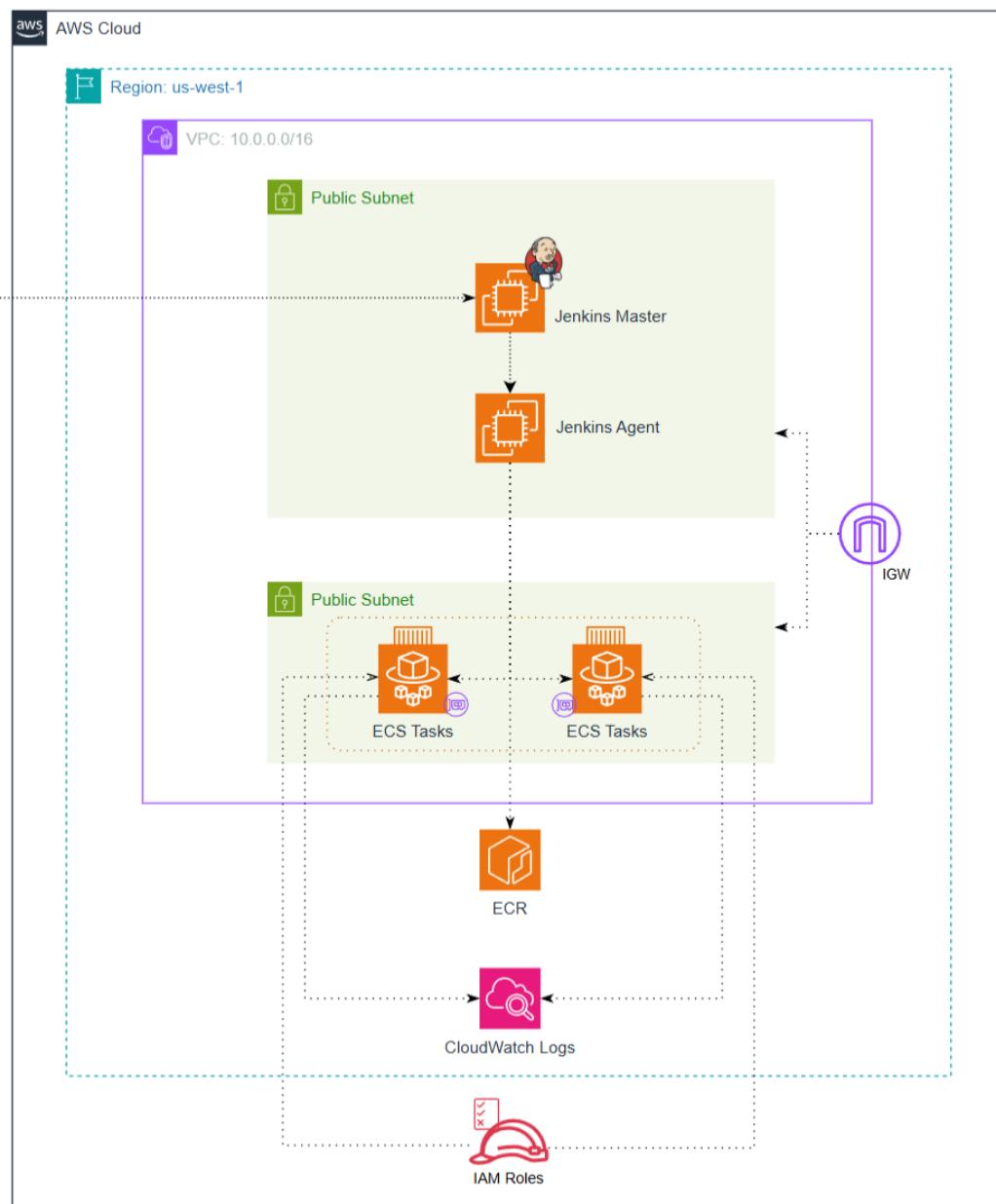
Task Description	4
Architecture Diagram	4
1. Test Application Locally	5
2. VPC Configuration	7
2.1: Create VPC.....	7
2.2: Create Internet Gateway	7
2.3: Create Subnets.....	8
2.4: Create Route Tables.....	9
2.5: Setting up Routes for Route tables.....	10
2.6: Route Table Association	11
2.7: Security Groups	11
3. IAM Roles.....	13
3.1: IAM Role for Jenkins EC2 Instances	13
3.2: ECS Task Execution Role	14
4. EC2 Instances for Jenkins	15
4.1: EC2 Instance Configuration.....	15
4.2: Jenkins Master EC2 Installation	16
4.3: Jenkins Agent EC2 Installation.....	17
4.4: Accessing Jenkins UI and Initial Setup	17
4.5: Connecting Jenkins Agent to Master	18
4.6: Jenkins Plugin Installation	19
4.7: Jenkins AWS Credentials Configuration	19
5. Elastic Container Registry (ECR)	20
5.1: Create ECR Repository.....	20
5.2: Push Docker Image to ECR	20
6. Elastic Container Service (ECS).....	22
6.2 Create ECS Cluster.....	23
6.3: Create ECS Service	24
7. Jenkins Pipeline Using Jenkinsfile.....	25
7.1: Pipeline Overview	25
7.2: Jenkins Agent Configuration	25
7.3: Environment Variables	26

7.4: Pipeline Stages Explanation	26
8. Running and Triggering the Jenkins CI/CD Pipeline	29
8.1: Commit and Push Jenkinsfile to GitHub	29
8.2: Create a Jenkins Pipeline Job.....	29
8.3: GitHub Webhook Configuration (Optional).....	31
8.4: Test the Current Application Deployment.....	31
8.5: Trigger the Jenkins Pipeline.....	32
8.6: Jenkins Console Output	33
8.7: ECS Service Events	36
8.8: Application Access	37
9. Troubleshooting	38

Task Description

This project implements a CI/CD pipeline using Jenkins on an EC2 agent to deploy a Dockerized Node.js application to AWS ECS Fargate. The pipeline integrates with GitHub webhooks to automatically build Docker images, push them to Amazon ECR, and update the ECS service. AWS CLI and scoped IAM roles manage infrastructure updates securely, while the pipeline handles testing, deployment, and rollback.

Architecture Diagram



1. Test Application Locally

Before integrating the application into the CI/CD pipeline and deploying it to AWS, the Node.js application was tested locally using Docker. This step ensures that the application builds correctly, runs as expected inside a container, and exposes the intended port before moving to a cloud-based deployment.

i. Build Docker image

- Use **docker build -t nodejs-app:latest**.
- Validates the Dockerfile and application dependencies.
- Packages the Node.js application into a Docker image.
- Produces a container image ready to be pushed to ECR and deployed to ECS.

```
PS D:\Cloudelligent\Task-12> docker build -t nodejs-app:latest .
[+] Building 8.5s (13/13) FINISHED
   => [internal] load build definition from Dockerfile
   => [internal] load metadata for docker.io/library/node:20-alpine
   => [internal] load metadata for docker.io/library/node:20-alpine
   => [auth] library/node:pull token for registry-1.docker.io
   => [internal] load .dockerignore
   => [internal] transfer context: 2B
[build 1/4] FROM docker.io/library/node:20-alpine@sha256:658d0f63e501824d6c23e06d4bb95c71e7d704537c9d9272f488ac03a370d448
[internal] load build context
   => transferring context: 1.49kB
   => CACHED [build 2/4] WORKDIR /app
   => [build 3/4] COPY package*.json ./
   => [build 4/4] RUN npm install --production
   => CACHED [stage-1 3/5] COPY --from=build /app/node_modules ./node_modules
   => CACHED [stage-1 4/5] COPY app.js ./
   => [stage-1 5/5] COPY public ./public
   => exporting to image
   => => exporting layers
   => => writing image sha256:11cefa6c926387e6e2af05d9a8acc3177d1ae821223c3bc27526feade2514
   => => naming to docker.io/library/nodejs-app:latest

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/3011b895592x3y5ueyb59kct7

What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview
PS D:\Cloudelligent\Task-12>
```

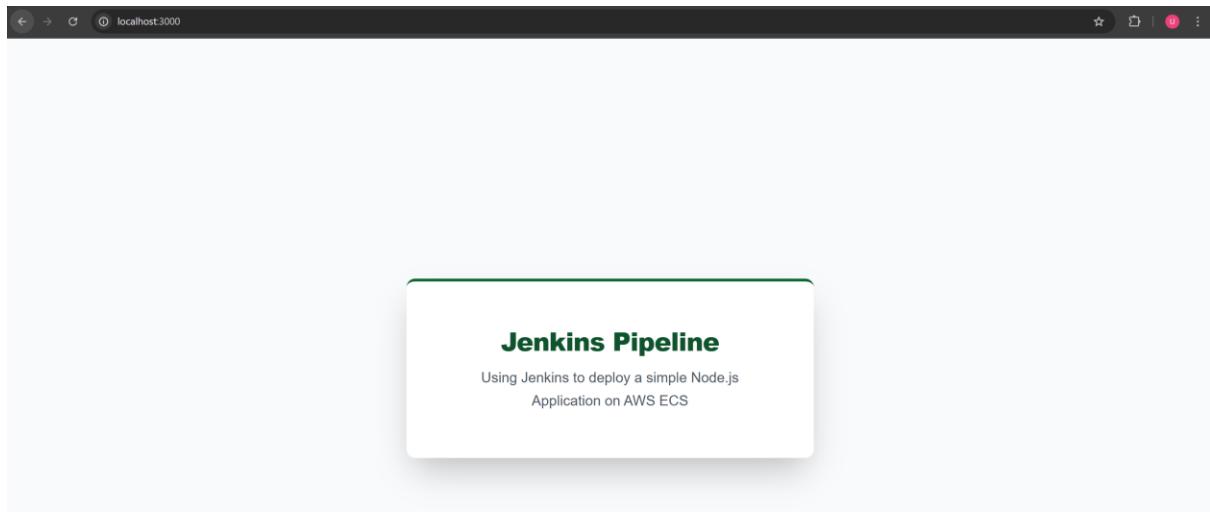
ii. Run the Docker container

- docker run -d -p 3000:3000 nodejs-app:latest
- Runs the container in detached mode.
- Maps local port **3000** to the container's exposed port **3000**.
- Ensures the application starts correctly inside the container.

```
PS D:\Cloudelligent\Task-12> docker run -d -p 3000:3000 nodejs-app:latest
e90d43cdab8183e71b3454912d8fd5fddd897bd238bf1a2233299f929c48b02f
PS D:\Cloudelligent\Task-12> docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
e90d43cdab81 nodejs-app:latest "docker-entrypoint.s..." 5 seconds ago Up 4 seconds 0.0.0.0:3000->3000/tcp, [::]:3000->3000/tcp naughty_vaughan
PS D:\Cloudelligent\Task-12>
```

iii. Verify the application

- Confirm the container is running using docker ps -a.
- Open **http://localhost:3000** in a web browser to verify the Node.js application loads successfully.



Accessing the application through **http://localhost:3000** verified that:

- The Docker image was built correctly
- The application started successfully inside the container
- The exposed port configuration worked as expected

2. VPC Configuration

2.1: Create VPC

A **Virtual Private Cloud (VPC)** is a logically isolated network environment within AWS, allowing the creation of secure and private infrastructure.

Steps:

1. Sign in to the AWS Management Console.
2. Navigate to VPC service using the search bar at the top.
3. In the VPC Console, click **Your VPCs**.
4. Click **Create VPC** button on the top right.
5. Choose **VPC only** option for ‘Resources to create’.
6. Choose a name for the VPC and add an IPv4 CIDR block.
7. Leave the **Tenancy** as **Default**.

As shown in the image below, under **Details** tab, a VPC has been created with the following configuration:

The screenshot shows the AWS VPC console's 'Your VPCs' page. On the left, there's a sidebar with various VPC-related options like Subnets, Route tables, Internet gateways, and Network ACLs. The main area displays a table of VPCs. A single row is selected for 'umarsatti-vpc'. Below the table, there are tabs for 'Details', 'Resource map', 'CIDRs', 'Flow logs', 'Tags', and 'Integrations'. The 'Details' tab is active, showing the following configuration:

Setting	Value
VPC ID	vpc-003aa34be02d5bc8a
DNS resolution	Enabled
Main network ACL	ad-03ba5b17ec715bf58
IPv6 CIDR	-
Encryption control ID	-
State	Available
Tenancy	default
Default VPC	No
Network Address Usage metrics	Disabled
Encryption control mode	-
Block Public Access	Off
DHCP option set	doopt-0e71024114ad5a935
IPv4 CIDR	10.0.0.0/16
Route 53 Resolver DNS Firewall rule groups	-
DNS hostnames	Disabled
Main route table	rts-011be02108e9000de
IPv6 pool	-
Owner ID	504649076991

- **Name:** umarsatti-vpc
- **IPv4 CIDR range:** 10.0.0.0/16
- **VPC ID:** vpc-003aa34be02d5bc8a

2.2: Create Internet Gateway

An **Internet Gateway (IGW)** enables communication between instances in the VPC and the public internet.

Steps:

1. Within the VPC Console, click **Internet gateways**.

2. Click on **Create internet gateway** button.
3. Choose a **Name** and add optional Tags.
4. Click **Create internet gateway** button. This creates an Internet gateway outside the VPC, and it needs to attach to the VPC created earlier.
5. Select **Actions** on the top right and click **Attach to VPC**. Choose the VPC ID (created earlier) and click **Attach internet gateway**. This attaches the Internet gateway to the VPC, allowing internet access.

As shown in the image below under **Details** tab, an internet gateway has been created with the following configuration:

- **Name:** umarsatti-igw
- **State:** Attached
- **VPC ID:** vpc-003aa34be02d5bc8a
- **Internet gateway ID:** igw-0c7828f1b891f54c8

The screenshot shows the AWS VPC dashboard with the 'Internet gateways' section selected. A table lists one internet gateway entry:

Name	Internet gateway ID	State	VPC ID	Owner
umarsatti-igw	igw-0c7828f1b891f54c8	Attached	vpc-003aa34be02d5bc8a umarsatti-vpc	504649076991

Below the table, the details for the gateway are expanded, showing its ID as 'igw-0c7828f1b891f54c8 / umarsatti-igw'. The 'Details' tab is selected, displaying the following information:

- Internet gateway ID:** igw-0c7828f1b891f54c8
- State:** Attached
- VPC ID:** vpc-003aa34be02d5bc8a | umarsatti-vpc
- Owner:** 504649076991

2.3: Create Subnets

Steps:

1. Within the VPC Console, click **Subnets**.
2. Click on **Create subnet** button.
3. Choose a **Subnet name**, **Availability Zone**, **IPv4 VPC CIDR block**, **IPv4 subnet CIDR Block** and optional **Tags**.
4. Repeat this process by clicking on **Add new subnet** button at the bottom to add more subnets (Public and Private).
5. After adding the required subnets, click on **Create subnet** button at the bottom right.

As shown in the image below under **Subnets** section, four subnets have been created with the following configuration:

The screenshot shows the AWS VPC Subnets console. On the left, there's a navigation sidebar with sections like VPC dashboard, Virtual private cloud, Subnets, Security, and PrivateLink and Lattice. The main area shows a table of subnets with columns: Name, Subnet ID, State, VPC, Block Public Access, IPv4 CIDR, and IPv6 CIDR. One subnet, 'uts-public-subnet-a', is selected. A detailed view panel on the right shows the subnet's ARN, Availability Zone (us-west-1a), and various configuration details such as Route table, Network ACL, and DNS64 settings.

Public Subnet in us-east-1a:

- **Name:** uts-public-subnet-a
- **IPv4 CIDR:** 10.0.1.0/24

Note: The **Public Subnet** hosts the Jenkins Master and Jenkins Agent. It is not a secure approach and is only suitable for development or testing environment.

2.4: Create Route Tables

Route Tables define how network traffic moves within the VPC. Each route table directs traffic either internally (within the VPC) or externally (to the internet or a NAT gateway).

Steps:

1. Within the VPC Console, click **Route tables**.
2. Click on **Create route table** button on the top right.
3. Choose a **Name**, **VPC**, and optional **Tags**.
4. When done, click on **Create route table** button to create the resource.

As shown in the image below under **Route tables** section, two route tables have been created (excluding default route table) with the following configuration:

The screenshot shows the AWS VPC Route Tables console. On the left, there's a sidebar with navigation links like 'VPC dashboard', 'AWS Global View', 'Virtual private cloud', 'Route tables', 'Security', and 'Network ACLs'. The main area has a header 'Route tables (1/7) Info' with a search bar. Below is a table with columns: Name, Route table ID, Explicit subnet assoc..., Edge associations, Main, VPC, and Owner ID. One row is selected: 'uts-public-rt' (rtb-07880238e1b4d289b). A detailed view panel on the right shows 'Details' for this route table, including its ID, VPC (vpc-003aa34be02d5bc8a), and owner (umarsatti-vpc). It also lists 'Explicit subnet associations' (subnet-030e70046420a08a / uts-public-subnet-a) and 'Edge associations'.

Name	Route table ID	Explicit subnet assoc...	Edge associations	Main	VPC	Owner ID
rtb-05769252b20b81fc7	-	-	-	Yes	vpc-0787f1b2fc819a76e	504649076991
uts-public-rt	rtb-07880238e1b4d289b	subnet-030e70046420a08a	-	No	vpc-003aa34be02d5bc8a umarsatti-vpc	504649076991
rtb-010254d7b0e8f3f1a1	-	-	-	Yes	vpc-0a59169e8549cad test-hammad...	504649076991
rtb-0cc23ca16eaef8472	subnet-06248549e6204...	-	-	No	vpc-003aa34be02d5bc8a umarsatti-vpc	504649076991
rtb-07880238e1b4d289b	rtb-07880238e1b4d289b / uts-public-rt	-	-	No	vpc-003aa34be02d5bc8a umarsatti-vpc	504649076991

Public Route table:

- **Name:** uts-public-rt
- **Route table ID:** rtb-07880238e1b4d289b

In this configuration, the public route table directs internet-bound traffic to the **Internet Gateway**.

2.5: Setting up Routes for Route tables

Steps:

1. Within the Route tables console, select a Route table (example: Public-Route-Table) and click **Edit routes** on the right.
2. Add a new route by choosing a **Destination** and **Target**.
3. Click **Save changes**. This adds a route to this route table.

Public Routes (Public-Route-Table):

- **Destination:** 0.0.0.0/0
- **Target:** umarsatti-igw | igw-0c7828f1b891f54c8 (Internet gateway)

The screenshot shows the 'Routes' tab for the 'uts-public-rt' route table. It displays two routes in a table with columns: Destination, Target, Status, Propagated, and Route Origin. The first route has a Destination of '0.0.0.0/0' and a Target of 'igw-0c7828f1b891f54c8'. The second route has a Destination of '10.0.0.0/16' and a Target of 'local'. Both routes are marked as 'Active'. There are buttons for 'Edit routes' and 'Create Route' at the bottom right.

Destination	Target	Status	Propagated	Route Origin
0.0.0.0/0	igw-0c7828f1b891f54c8	Active	No	Create Route
10.0.0.0/16	local	Active	No	Create Route Table

This step defines how traffic flows between subnets and external networks. The **public route table** sends all internet-bound traffic (0.0.0.0/0) to the **Internet Gateway**, enabling access for public resources.

2.6: Route Table Association

Steps:

1. Within the Route tables console, select a Route table (example: Public-Route-Table) and click **Subnet associations** tab under the details section.
2. Click **Edit subnet associations** and select the specific subnet (e.g. nginx-public-subnet)
3. Click **Save associations** button on the bottom right.

Public Route table Subnet associations

- **Names:** uts-public-rt
- **Subnets:** uts-public-subnet-a
- **IPv4 CIDR:** 10.0.1.0/24

The screenshot shows the AWS Route Tables console for route table rtb-07880238e1b4d289b. The 'Subnet associations' tab is selected. In the 'Explicit subnet associations' section, there is one entry: Name 'uts-public-subnet-a' associated with Subnet ID 'subnet-030e70046420a086a' and IPv4 CIDR '10.0.1.0/24'. The 'Subnets without explicit associations' section shows '0' subnets.

Public subnet uses the **Internet Gateway** for external communication. Without this association, subnets would rely on the default main route table, potentially misrouting traffic.

2.7: Security Groups

Security Groups act as virtual firewalls that regulate inbound and outbound traffic at the instance level.

Steps:

1. Within the VPC Console, click **Security Groups**.
2. Click on **Create security gateway** button on the top right.
3. Choose a **Security group name**, **Description**, and **VPC** (umarsatti-vpc).
4. Add Inbound rules by selecting **Type**, **Protocol**, **Port range**, **Source**, and **Description** (optional).
5. Leave Outbound rules as default (All traffic).

The screenshot shows the AWS VPC Security Groups page. On the left, there's a sidebar with options like 'AWS Global View', 'Virtual private cloud', 'Security', and 'PrivateLink and Lattice'. The main area displays a table of security groups with columns for Name, Security group ID, Security group name, VPC ID, and Description. One row is selected: 'sg-0aab15691d0e8ccf3 - uts-public-sg'. Below the table, there are tabs for Details, Inbound rules, Outbound rules, Sharing, VPC associations, and Tags. The 'Details' tab is active, showing the security group's name, ID, owner (504649076991), and its association with VPC ID vpc-003aa34be02d5bc8a.

Security Group for Instances and ECS Tasks

Here is the configuration for the public subnet security group:

- Security group name:** uts-public-sg
- Security group ID:** sg-0aab15691d0e8ccf3
- Inbound Rules:**

Type	Protocol	Port range	Source
HTTP	TCP	80	0.0.0.0/0
Custom TCP	TCP	3000	0.0.0.0/0
Custom TCP	TCP	8080	0.0.0.0/0

The screenshot shows the 'Inbound rules' tab for the 'sg-0aab15691d0e8ccf3 - uts-public-sg' security group. It lists three rules:

- Rule 1: Name: sgr-049e9dc84c9d2db31, Security group rule ID: sg-r-049e9dc84c9d2db31, IP version: IPv4, Type: HTTP, Protocol: TCP, Port range: 80, Source: 0.0.0.0/0
- Rule 2: Name: sgr-0cd7b555f69fa5b3b, Security group rule ID: sg-r-0cd7b555f69fa5b3b, IP version: IPv4, Type: Custom TCP, Protocol: TCP, Port range: 8080, Source: 0.0.0.0/0
- Rule 3: Name: sgr-070c63d35dba5b7a2, Security group rule ID: sg-r-070c63d35dba5b7a2, IP version: IPv4, Type: Custom TCP, Protocol: TCP, Port range: 3000, Source: 0.0.0.0/0

 There are buttons for 'Manage tags' and 'Edit inbound rules' at the top right of the table.

In this configuration, the **uts-public-sg** security group allows HTTP traffic and TCP port 5000/8080 traffic from internet. Port 3000 is for Node.js application and port 8080 is for Jenkins access.

3. IAM Roles

IAM roles are used to securely grant AWS permissions to services without storing access keys. In this project, IAM roles allow Jenkins to interact with ECR and ECS, and enable ECS tasks to pull images and send logs.

3.1: IAM Role for Jenkins EC2 Instances

An IAM role was created and attached to the Jenkins Master and Agent EC2 instances to allow Jenkins to deploy resources in AWS.

IAM Role Details

- **Role Name:** uts-jenkins-ec2-role
- **Trusted Service:** EC2

Attached Managed Policies

- **AmazonSSMManagedInstanceCore:** Enables secure instance management using AWS Systems Manager.
- **AmazonEC2ContainerRegistryPowerUser:** Allows Jenkins to authenticate with ECR and push/pull Docker images.

Inline Policy: Jenkins-EC2-ECS-Policy

- Grants permissions to register task definitions
- Update ECS services.
- Create and update CloudWatch Log stream.

The screenshot shows the AWS IAM Roles page with the role 'uts-jenkins-ec2-role' selected. The left sidebar shows navigation options like Dashboard, Access management, and Access reports. The main panel displays the role's summary, including its ARN, creation date (December 18, 2025), last activity (14 minutes ago), and maximum session duration (1 hour). It also shows the Instance profile ARN. Below this, the 'Permissions' tab is active, showing three attached managed policies: AmazonSSMManagedInstanceCore, AmazonEC2ContainerRegistryPowerUser, and Jenkins-EC2-ECS-Policy. A table lists these policies with their types (AWS managed or Customer inline) and the number of entities they are attached to. The Jenkins policy is listed as Customer inline with 0 entities attached.

Policy name	Type	Attached entities
AmazonSSMManagedInstanceCore	AWS managed	20
AmazonEC2ContainerRegistryPowerUser	AWS managed	7
Jenkins-EC2-ECS-Policy	Customer inline	0

This role enables Jenkins to use AWS CLI securely without hardcoding credentials.

3.2: ECS Task Execution Role

ECS tasks use the default execution role to access AWS services at runtime.

Role Name: ecsTaskExecutionRole

Purpose

- Pull container images from Amazon ECR.
- Send application logs to Amazon CloudWatch Logs.

This role is referenced in the ECS task definition and is required for Fargate tasks to start successfully.

4. EC2 Instances for Jenkins

Two Ubuntu EC2 instances were launched to host the Jenkins Master and Jenkins Agent. This separation allows the master to manage pipelines while the agent performs builds and deployments.

4.1: EC2 Instance Configuration

The following configuration was used for **both** Jenkins Master and Jenkins Agent:

- **Number of instances:** 2
- **Operating System:** Ubuntu (x86_64)
- **Instance type:** t3.small
- **Key pair:** None (AWS Systems Manager used for access)
- **VPC:** Custom VPC created earlier
- **Subnet:** Public subnet
- **Security Group:** uts-public-sg
- **IAM Role:** uts-jenkins-ec2-role
- **Storage:** 20 GB gp3 EBS volume

This setup provides sufficient resources for Jenkins operations in a development/testing environment.

Jenkins Master EC2 Instance

The screenshot shows the AWS EC2 Instances page with the following details for the Jenkins Master instance:

Attribute	Value
Name	Jenkins-Master
Instance ID	i-0a0d95b7686d55c8
Instance state	Running
Instance type	t3.small
Status check	3/3 checks passed
Alarm status	View alarms
Availability Zone	us-west-1a
Public IPv4 DNS	-
Public IP address	52.55.213.42
Private IP address	10.0.1.23
Private IP DNS name (IPv4 only)	ip-10-0-1-23.us-west-1.compute.internal
Instance type	t3.small
VPC ID	vpc-003aa34be02d5bc8a (umarsatti-vpc)
Subnet ID	subnet-030e70046420a086a (uts-public-subnet-a)
Instance ARN	arn:aws:ec2:us-west-1:504649076991:instance/i-0a0d95b7686d55c8
Managed	false

Jenkins Agent EC2 Instance

The screenshot shows the AWS EC2 Instances page. On the left, there's a sidebar with navigation links like Dashboard, AWS Global View, Events, Instances, Instance Types, Launch Templates, Spot Requests, Savings Plans, Reserved Instances, Dedicated Hosts, Capacity Reservations, Capacity Manager, Images, AMIs, AMI Catalog, Elastic Block Store, Volumes, Snapshots, Lifecycle Manager, Network & Security, Security Groups, Elastic IPs, Placement Groups, Key Pairs, and more. The main area displays two instances:

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
Jenkins-Master	i-0a0d95b7686d55c8	Running	t3.small	3/3 checks passed	View alarms	us-west-1a	-
Jenkins-Agent	i-09912790e9bbb9bde	Running	t3.small	3/3 checks passed	View alarms	us-west-1a	-

Below the instances, there's a detailed view for the Jenkins-Agent instance:

Details	Status and alarms	Monitoring	Security	Networking	Storage	Tags
Instance summary						
Instance ID i-09912790e9bbb9bde	Public IPv4 address 54.183.238.96 [open address]	Private IPv4 address 10.0.1.181				
IPv6 address -	Instance state Running	Public DNS -				
Hostname type IP name: ip-10-0-1-181.us-west-1.compute.internal	Private IP DNS name (IPv4 only) ip-10-0-1-181.us-west-1.compute.internal	Elastic IP addresses -				
Answer private resource DNS name -	Instance type t3.small	AWS Compute Optimizer finding No recommendations available for this instance.				
Auto-assigned IP address 54.183.238.96 [Public IP]	VPC ID vpc-003aa34be02d5bc8a (umarsatti-vpc)	Subnet ID subnet-030e70046420a086a (uts-public-subnet-a)	Auto Scaling Group name -			
IAM Role uts-jenkins-ec2-role	Instance ARN arn:aws:ec2:us-west-1:504649076991:instance/i-09912790e9bbb9bde	Managed false				

4.2: Jenkins Master EC2 Installation

The Jenkins Master instance hosts the Jenkins UI and controls pipeline execution.

Access Method

- Connected to the instance using **AWS Systems Manager Session Manager**, avoiding SSH key usage.

Java Installation

- Installed OpenJDK 21 to meet Jenkins runtime requirements.
- Verified installation using `java -version`.

```
root@ip-10-0-1-23:~# java -version
openjdk version "21.0.9" 2025-10-21
OpenJDK Runtime Environment (build 21.0.9+10-Ubuntu-124.04)
OpenJDK 64-Bit Server VM (build 21.0.9+10-Ubuntu-124.04, mixed mode, sharing)
root@ip-10-0-1-23:~#
```

Jenkins Installation

- Added the official Jenkins repository and signing key.
- Installed Jenkins using the system package manager.
- Enabled the Jenkins service and verified it was running using `systemctl`.

```

root@ip-10-0-1-23:~# systemctl status jenkins
● jenkins.service - Jenkins Continuous Integration Server
  Loaded: loaded (/usr/lib/systemd/system/jenkins.service; enabled; preset: enabled)
  Active: active (running) since Thu 2025-12-18 14:48:12 UTC; 9h ago
    Main PID: 4316 (java)
      Tasks: 50 (limit: 2204)
        Memory: 23.1M (peak: 903.8M)
          CPU: 8min 35.693s
        CGroup: /system.slice/jenkins.service
              └─4316 /usr/bin/java -Djava.awt.headless=true -jar /usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war --httpPort=8080

Dec 18 23:00:45 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:00:45.282+0000 [id=1383] INFO o.j.p.g.w.s.DefaultPushGHEventSubscriber#onEvent: Received PushEvent for https://github.com/
Dec 18 23:18:20 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:18:20.963+0000 [id=1385] INFO o.j.p.g.w.s.DefaultPushGHEventSubscriber#onEvent: Received PushEvent for https://github.com/
Dec 18 23:18:20 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:18:20.975+0000 [id=1385] INFO o.j.p.g.w.s.DefaultPushGHEventSubscriber$!#run: Poked jenkins-nodejs-pipeline
Dec 18 23:18:20 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:18:20.984+0000 [id=2794] INFO c.c.jenkins.GitHubPushTrigger$!#run: SCM change detected in jenkins-nodejs-pipeline. Triggered
Dec 18 23:18:20 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:18:20.990+0000 [id=385] INFO o.j.p.g.w.s.DefaultPushGHEventSubscriber#onEvent: Received PushEvent for https://github.com/
Dec 18 23:25:52 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:25:52.547+0000 [id=1385] INFO o.j.p.g.w.s.DefaultPushGHEventSubscriber$!#run: Poked jenkins-nodejs-pipeline
Dec 18 23:29:53 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:29:53.290+0000 [id=9201] INFO c.c.jenkins.GitHubPushTrigger$!#run: SCM changes detected in jenkins-nodejs-pipeline. Triggered
Dec 18 23:46:28 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:46:28.634+0000 [id=1387] INFO o.j.p.g.w.s.DefaultPushGHEventSubscriber$!#run: Received PushEvent for https://github.com/
Dec 18 23:46:28 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:46:28.635+0000 [id=1387] INFO o.j.p.g.w.s.DefaultPushGHEventSubscriber#onEvent: Poked jenkins-nodejs-pipeline
Dec 18 23:46:29 ip-10-0-1-23 jenkins[4316]: 2025-12-18 23:46:29.336+0000 [id=3067] INFO c.c.jenkins.GitHubPushTrigger$!#run: SCM changes detected in jenkins-nodejs-pipeline. Triggered

```

4.3: Jenkins Agent EC2 Installation

The Jenkins Agent instance is responsible for executing build and deployment tasks.

Java Installation

- Installed OpenJDK 21, required to run the Jenkins agent process.

```

root@ip-10-0-1-181:~# java -version
openjdk version "21.0.9" 2025-10-21
OpenJDK Runtime Environment (build 21.0.9+10-Ubuntu-124.04)
OpenJDK 64-Bit Server VM (build 21.0.9+10-Ubuntu-124.04, mixed mode, sharing)
root@ip-10-0-1-181:~#

```

AWS CLI Installation

- Installed AWS CLI v2 to allow Jenkins jobs to authenticate with AWS services.
- Required for pushing Docker images to Amazon ECR and updating ECS services.

To install the AWS CLI, run the following commands.

```

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

```

Docker Installation

- Installed Docker to build, tag, and push container images during pipeline execution.

4.4: Accessing Jenkins UI and Initial Setup

- Accessed Jenkins via `http://<Jenkins-Master-Public-IP>:8080`
- Retrieved the initial admin password from:
 - `/var/lib/jenkins/secrets/initialAdminPassword`
- Installed suggested plugins during setup.
- Created an admin user with basic details.

- **Username:** admin
- **Password:** admin
- **Full name:** Admin
- **Email:** admin@example.com

Agent Configuration

Configured a new node:

- **Node name:** Slave
- **Type:** Permanent Agent
- **Remote root directory:** /opt/build

4.5: Connecting Jenkins Agent to Master

The agent was connected to the Jenkins Master using the JNLP method.

- Downloaded the Jenkins agent JAR from the master.
- Executed the agent command with:
 - Jenkins URL
 - Secret token
 - Agent name
 - Agent Label: slave-node
 - Working directory (/opt/build)

Agent Slave

This is Jenkins agent
Agent is connected.

Labels

slave-node

Projects tied to Slave

None

Build Executor Status

(0 of 2 executors busy)

This established a persistent connection between the master and agent.

4.6: Jenkins Plugin Installation

Required plugins were installed on the Jenkins Master to support GitHub integration, Docker builds, and AWS deployments.

Installed Plugins

- **Git & Node.js:** Git plugin, NodeJS plugin
- **AWS & ECR:** AWS SDK, AWS Credentials, Amazon ECR, Pipeline AWS Steps
- **ECS:** AWS SDK for ECS, Amazon ECS/Fargate plugin
- **Docker:** Docker, Docker Pipeline, Docker API, Docker Commons

These plugins enable end-to-end CI/CD functionality.

4.7: Jenkins AWS Credentials Configuration

Configure Jenkins to authenticate with AWS using IAM roles.

- Navigate to **Manage Jenkins → Credentials**
- Added **AWS Credentials** with:
 - No access keys provided
 - **IAM Role:** Add role ARN
 - **STS Token Duration:** 3600 seconds

ID	Name	Kind	Description
9027c7d5-3b02-4c8e-b9a5-80a6a23b7511	arnawsiam:504649076991:instance-profile/uts-jenkins-ec2-role	AWS Credentials	

This allows Jenkins to securely access AWS services without storing static credentials.

5. Elastic Container Registry (ECR)

Amazon Elastic Container Registry (ECR) is used to store, manage, and version Docker container images. In this project, ECR serves as the central image repository from which ECS pulls the application image during deployment.

5.1: Create ECR Repository

A private ECR repository was created to store the Node.js application image.

Navigate to the **ECR console** using the search bar. On the left navigation bar, select **Private repository**. Click the **Create** button to configure ECR.

Repository Configuration

- **Repository name:** nodejs-app
- **Visibility:** Private
- **Tag mutability:** Mutable
- **Encryption:** AES-256
- **Repository URI:** 504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-app

This repository stores Docker images and allows controlled access via IAM roles.

5.2: Push Docker Image to ECR

The Docker image was built locally and pushed to the ECR repository using AWS CLI and Docker. Click the **View push commands** button and execute the commands to push docker image to ECR.

1. Authenticate Docker with ECR

- Uses AWS CLI to retrieve a temporary authentication token.
- Allows Docker to interact with the private ECR repository.

2. Build the Docker image

- Builds the Node.js application image using the local Dockerfile.
- Tags the image as nodejs-app.

3. Tag the image for ECR

- Retags the local image with the ECR repository URI.
- Prepares the image for upload to ECR.

4. Push the image to ECR

- Uploads the Docker image to the private ECR repository.
- Makes the image available for ECS deployments.

After the push is completed successfully, the Docker image appears in the ECR repository and is ready to be referenced in the ECS task definition.

6. Elastic Container Service (ECS)

Amazon Elastic Container Service (ECS) is used to run and manage the Dockerized Node.js application on AWS Fargate. This setup includes creating a task definition, an ECS cluster, and an ECS service to ensure scalable and highly available deployments.

6.1: Create ECS Task Definition

The task definition defines how the container runs, including resource requirements, networking, and logging. Navigate to the **ECS** console using the search bar at the top. Select **Task Definition** and add the configuration defined below.

Task Definition Configuration

- **Family name:** nodejs-app-task-definition
- **Launch type:** AWS Fargate
- **OS / Architecture:** Linux, x86_64
- **Network mode:** awsvpc
- **Task size:**
 - CPU: 0.5 vCPU
 - Memory: 1 GB
- **Task role / Execution role:** ecsTaskExecutionRole

Container Configuration

- **Container name:** nodejs
- **Essential:** Yes
- **Image URI:** 504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-app
- **Port mapping:**
 - **Container port:** 3000
 - **Protocol:** TCP
 - **App protocol:** HTTP
- **Logging Configuration**
- **Log driver:** Amazon CloudWatch Logs
- **Log group:** /ecs/jenkins-nodejs-app-task-definition
- **Region:** us-west-1
- **Stream prefix:** ecs
- **Auto-create log group:** Enabled

The screenshot shows the 'Containers' tab selected. Under 'Task size', there are two bar charts. The first chart for 'Task CPU' shows 512 units (0.5 vCPU) allocated to the 'nodejs' container. The second chart for 'Task memory' shows 1,024 MB (1 GiB) allocated to the 'nodejs' container.

Default settings were used for storage, monitoring, and tags.

6.2 Create ECS Cluster

An ECS cluster was created to host and manage the Fargate tasks.

Navigate to the **ECS** console using the search bar at the top. Select **Cluster** and add the configuration defined below.

Cluster Configuration

- Cluster name:** nodejs-ecs-cluster
- Compute capacity:** Fargate only
- Monitoring:**
 - Container Insights enabled with enhanced observability
 - ECS Exec logging enabled
- Encryption:** Default (none)
- Tags:** None

The screenshot shows the 'Clusters' table with one entry: 'nodejs-ecs-cluster'. The cluster status is 1 service, 0 pending, 2 running, and 0 EC2 instances. A note indicates 'Container Insights with enhanced observability' is enabled.

This cluster provides the runtime environment for the application containers.

6.3: Create ECS Service

The ECS service ensures that the desired number of application tasks are running and handles deployments and rollbacks. Within the cluster console, Click the **Create service** button and add the following configurations.

Service Details

- **Service name:** nodejs-app-task-definition-service
- **Task definition:** nodejs-app-task-definition (LATEST revision)

Compute and Environment

- **Capacity provider:** FARGATE
- **Platform version:** LATEST
- **ECS Exec:** Enabled

Deployment Configuration

- **Scheduling strategy:** Replica
- **Desired tasks:** 2
- **Deployment strategy:** Rolling update
- **Minimum healthy tasks:** 50%
- **Maximum running tasks:** 100%
- **Deployment circuit breaker:** Enabled with automatic rollback

Networking Configuration

- **VPC:** Custom VPC created earlier
- **Subnets:** Previously created subnets
- **Security groups:** Existing security group with required inbound rules

All remaining settings were left at their default values.

The screenshot shows the AWS Elastic Container Service (ECS) Cluster Overview page for the 'nodejs-ecs-cluster'. The page displays the following information:

- Cluster overview:** Shows the ARN (arn:aws:ecs:us-west-1:504649076991:cluster/nodejs-ecs-cluster), Status (Active), CloudWatch monitoring (Container Insights with enhanced observability), and Registered container instances (none).
- Services:** A table showing the status of services. One service, 'nodejs-app-task-definition-service', is listed as Active with 1 Pending task and 2 Running tasks.
- Services (1/1) info:** A detailed view of the service. It includes a search bar for 'Filter services by value', and columns for Service name (nodejs-app-task-definition-service), ARN (arn:aws:ecs:us-west-1:504649076991:cluster/nodejs-ecs-cluster:nodejs-app-task-definition-service), Status (Active), Task definition (REPLICAS), and Deployments and tasks (2/2 Tasks running). Filter options for launch type, scheduling strategy, resource management type, and deployment status are also present.

7. Jenkins Pipeline Using Jenkinsfile

The Jenkins pipeline automates the complete CI/CD workflow for the Node.js application. It pulls source code from GitHub, builds and tests a Docker image, pushes the image to Amazon ECR, and deploys the updated image to Amazon ECS. The pipeline is defined declaratively using a Jenkinsfile stored in the GitHub repository.

7.1: Pipeline Overview

Stage	Description
Checkout	Pulls application source code and Jenkinsfile from GitHub
Build	Builds the Docker image for the Node.js application
Test	Runs a basic container test to validate the image
Push to ECR	Tags and pushes the Docker image to Amazon ECR
Deploy	Triggers ECS service update with the new image
Rollback	Handled automatically by ECS deployment circuit breaker

7.2: Jenkins Agent Configuration

The pipeline is configured to run on a dedicated Jenkins agent node:

- **Agent label:** slave-node
- Ensures all build and deployment steps execute on the Agent EC2 instance.
- Keeps the Jenkins Master lightweight and focused on orchestration.



```
Jenkinsfile X
Jenkinsfile
1 pipeline {
2     // Jenkins agent node label
3     agent {
4         label 'slave-node'
5     }
}
```

A screenshot of a code editor showing a Jenkinsfile. The file contains a pipeline block with an agent label 'slave-node'. The code is syntax-highlighted with colors for different keywords and comments.

7.3: Environment Variables

Global environment variables are defined to make the pipeline reusable and configurable.

- **AWS_DEFAULT_REGION:** Specifies the AWS region for ECR and ECS operations
- **ECR_REPO:** Stores the full ECR repository URI
- **IMAGE_TAG:** Uses the latest tag for the Docker image

```
// Environment variables for AWS and Docker
environment {
    // AWS region where ECS and ECR are located
    AWS_DEFAULT_REGION = 'us-west-1'

    // ECR repository URI for storing Docker images
    ECR_REPO = '504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-app'

    // Use "latest" tag for Docker image
    IMAGE_TAG = "latest"
}
```

These variables are referenced across multiple pipeline stages to maintain consistency.

7.4: Pipeline Stages Explanation

Stage 1: Source

- Pulls the application source code and Jenkinsfile from the GitHub repository.
- Pipeline always runs against the latest committed code on the main branch.

```
stages {
    // Stage 1: Pull source code from GitHub repository
    stage('Source') {
        steps {
            git branch: 'main', url: 'https://github.com/Umarsattii1/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git'
        }
    }
}
```

Stage 2: Build

- Builds the Docker image using the application's Dockerfile.
- Tags the image locally using the defined IMAGE_TAG.
- Verifies that the application can be successfully containerized.

```
// Stage 2: Build Docker image for the application
stage('Build') {
    steps {
        script {
            sh "docker build -t nodejs-app:${IMAGE_TAG} ."
        }
    }
}
```

Stage 3: Test

- Runs the Docker container temporarily to validate startup.
- Confirms that the Node.js application launches correctly inside the container.
- Stops and removes the container automatically after verification.

```
// Stage 3: Optional test to run container briefly
stage('Test') {
    steps {
        script {
            sh "docker run --rm nodejs-app:${IMAGE_TAG} node app.js & sleep 5; docker ps -q | xargs docker stop || true"
        }
    }
}
```

This stage provides basic validation even in the absence of formal unit tests.

Stage 4: Push to ECR

- Authenticates Docker with Amazon ECR using AWS CLI.
- Tags the locally built image with the ECR repository URI.
- Pushes the Docker image to the private ECR repository.

```
// Stage 4: Push Docker image to Amazon ECR
stage('Push') {
    steps {
        script {
            sh """
                aws ecr get-login-password --region ${AWS_DEFAULT_REGION} | docker login --username AWS --password-stdin ${ECR_REPO}
                docker tag nodejs-app:${IMAGE_TAG} ${ECR_REPO}:${IMAGE_TAG}
                docker push ${ECR_REPO}:${IMAGE_TAG}
            """
        }
    }
}
```

Authentication relies on the IAM role attached to the EC2 instance, eliminating the need for static credentials.

Stage 5: Deploy to ECS

- Triggers a new deployment of the ECS service using AWS CLI.
- Forces ECS to pull the latest Docker image from ECR.
- ECS performs a rolling update based on the service configuration.

```
// Stage 5: Deploy new Docker image to ECS service
stage('Deploy') {
    steps {
        script {
            sh """
                aws ecs update-service \
                    --cluster nodejs-ecs-cluster \
                    --service nodejs-app-task-definition-service \
                    --force-new-deployment \
                    --region ${AWS_DEFAULT_REGION}
            """
        }
    }
}
```

Deployment failures are automatically handled using the ECS deployment circuit breaker.

Post-Build Actions

- **On Success:** Displays a successful ECS deployment message.
- **On Failure:** Displays an error message prompting investigation of Jenkins and ECS logs.

```
// Post-build actions: notify success or failure
post {
    success {
        echo 'Deployment succeeded! ECS service has been updated with the latest image.'
    }
    failure {
        echo 'Deployment failed. Please check Jenkins and ECS logs for errors.'
    }
}
```

These post-build steps provide immediate feedback on pipeline execution status.

This Jenkins pipeline enables fully automated CI/CD, ensuring that every code change pushed to GitHub results in a new Docker image build and a controlled ECS deployment.

8. Running and Triggering the Jenkins CI/CD Pipeline

This section describes how the Jenkins pipeline is executed, triggered by GitHub events, and verified after deployment to Amazon ECS.

8.1: Commit and Push Jenkinsfile to GitHub

The Jenkins pipeline configuration is stored as code using a Jenkinsfile in the GitHub repository.

- Added the Jenkinsfile to version control.
- Committed the pipeline configuration.
- Pushed changes to the main branch.

This ensures the CI/CD pipeline is reproducible and versioned alongside the application code.

8.2: Create a Jenkins Pipeline Job

A new Jenkins pipeline job was created to execute the CI/CD workflow.

- **Job Configuration**
- **Job name:** jenkins-nodejs-pipeline
- **Job type:** Pipeline
- **Description:** Pipeline for building, testing, and deploying Node.js app to ECS using Jenkins, Docker, and ECR.

Pipeline Settings

- **Definition:** Pipeline script from SCM
- **SCM:** Git
- **Repository URL:** <https://github.com/Umarsatti1/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git>
- **Branch specifier:** */main
- **Script path:** Jenkinsfile
- **Lightweight checkout:** Enabled

Pipeline

Define your Pipeline using Groovy directly or pull it from source control.

Definition

Pipeline script from SCM

SCM ?

Git

Repositories ?

Repository URL ?
https://github.com/Umarsatti1/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git

Credentials ?
- none -

Advanced ▾

+ Add Repository

Branches to build ?

Branch Specifier (blank for 'any') ?
*/main

+ Add Branch

Repository browser ?
(Auto)

Additional Behaviours

+ Add

Script Path ?
Jenkinsfile

Lightweight checkout ?

Pipeline Syntax

Triggers

- GitHub hook trigger for GITScm polling
- Poll SCM

Triggers

Set up automated actions that start your build based on specific events, like code changes or scheduled times.

- Build after other projects are built ?
- Build periodically ?
- GitHub hook trigger for GITScm polling ?
- Poll SCM ?

Schedule ?

No schedules so will only run due to SCM changes if triggered by a post-commit hook

- Ignore post-commit hooks ?

- Trigger builds remotely (e.g., from scripts) ?

After configuration, the job was saved and made ready for execution.

8.3: GitHub Webhook Configuration (Optional)

To enable automatic pipeline execution on every code push, a GitHub webhook was configured.

- **Payload URL:** http://<Jenkins-Master-IP>:8080/github-webhook/
- **Content type:** application/json
- **Trigger event:** Push events only
- **Status:** Active

The screenshot shows the 'Add webhook' form on GitHub. It includes fields for 'Payload URL' (http://52.53.213.42:8080/github-webhook/), 'Content type' (application/json), and a 'Secret' field. Under 'SSL verification', 'Enable SSL verification' is selected. In the 'Which events would you like to trigger this webhook?' section, 'Just the push event.' is selected. A checkbox for 'Active' is checked, and the 'Add webhook' button is visible at the bottom.

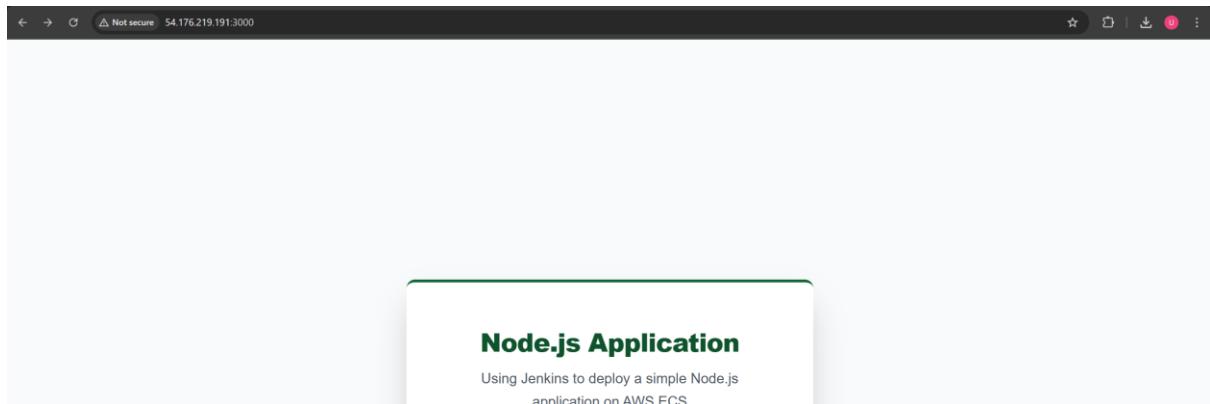
The screenshot shows the 'Webhooks' list page on GitHub. It displays a single webhook entry with the URL http://52.53.213.42:8080/github-we... (push). The status is 'Last delivery was successful.' and there are 'Edit' and 'Delete' buttons.

This webhook notifies Jenkins whenever changes are pushed to the repository.

8.4: Test the Current Application Deployment

Before triggering an update, the currently deployed application was verified:

- Retrieved the **Elastic Network Interface (ENI)** IP associated with ECS tasks.
- Accessed the application using: http://<ENI-IP>:3000
- Example: http://54.175.219.191:3000



This confirmed the existing ECS deployment was running successfully.

8.5: Trigger the Jenkins Pipeline

The pipeline is automatically triggered on every new commit to the GitHub repository.

Upon trigger, Jenkins performs the following steps:

- Checks out the latest source code
- Builds a new Docker image
- Pushes the image to Amazon ECR
- Deploys the updated image to the ECS service

To validate automation:

- Application code (e.g., index.html) was modified.
- Changes were pushed to GitHub.
- Jenkins pipeline executed successfully without manual intervention.

```
● PS D:\Cloudelligent\Task-12> git add .
● PS D:\Cloudelligent\Task-12> git commit -m "Testing Jenkins pipeline"
[main 653c36c] Testing Jenkins pipeline
 1 file changed, 5 insertions(+), 9 deletions(-)
● PS D:\Cloudelligent\Task-12> git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 424 bytes | 424.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/Umarsattii/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git
  f9212f2..653c36c main -> main
○ PS D:\Cloudelligent\Task-12> █
```

8.6: Jenkins Console Output

Confirmed successful execution of each pipeline stage.

The screenshot shows the Jenkins Pipeline configuration page for the 'jenkins-nodejs-pipeline'. The top navigation bar includes links for Status, Changes, Build Now, Configure, Delete Pipeline, Stages, Rename, Pipeline Syntax, GitHub Hook Log, and Git Polling Log. The main content area displays the pipeline name 'jenkins-nodejs-pipeline' with a green checkmark icon, a brief description 'Pipeline for building, testing, and deploying Node.js app to ECS using Jenkins, Docker, and ECR.', and a 'Permalinks' section listing recent builds. Below this is a 'Builds' section containing a table with three rows: #3 (11:46 PM, successful), #2 (11:30 PM, successful), and #1 (11:18 PM, failed). A 'Filter' input field is also present.

#	Build Time	Status
#3	11:46 PM	Success
#2	11:30 PM	Success
#1	11:18 PM	Failure

1. Pipeline Initialization & Checkout

```
Started by GitHub push by Umarsattii
Obtained Jenkinsfile from git https://github.com/Umarsattii/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Slave in /opt/build/workspace/jenkins-nodejs-pipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Declarative: Checkout SCM)
[Pipeline] checkout
Selected Git installation does not exist. Using Default
The recommended git tool is: NONE
No credentials specified
Fetching changes from the remote Git repository
Checking out Revision 653c36c72e467efc7969b95161d96fc242d95f35 (refs/remotes/origin/main)
Commit message: "Testing Jenkins pipeline"
> git rev-parse --resolve-git-dir /opt/build/workspace/jenkins-nodejs-pipeline/.git # timeout=10
> git config remote.origin.url https://github.com/Umarsattii/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git # timeout=10
Fetching upstream changes from https://github.com/Umarsattii/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git
> git --version # timeout=10
> git --version # 'git version 2.43.0'
> git fetch --tags --progress -- https://github.com/Umarsattii/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git +refs/heads/*:refs/remotes/origin/* # timeout=10
> git rev-parse refs/remotes/origin/main^{commit} # timeout=10
> git config core.sparsecheckout # timeout=10
> git checkout -f 653c36c72e467efc7969b95161d96fc242d95f35 # timeout=10
> git rev-list --no-walk f9212f2af9c98d432dc11a3c77c1e88bf4273812 # timeout=10
```

The pipeline starts on a slave node. It performs a declarative checkout from Git, fetching the Jenkinsfile from the GitHub repository. The system checks out a specific commit (653c36c72e467efc7969b95161d96fc242d95f35) with the message "Testing Jenkins pipeline" and configures Git with sparse checkout for efficiency.

2. Source Stage

```
[Pipeline] }
[Pipeline] // stage
[Pipeline] withEnv
[Pipeline] {
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Source)
[Pipeline] git
Selected Git installation does not exist. Using Default
The recommended git tool is: NONE
No credentials specified
Fetching changes from the remote Git repository
Checking out Revision 653c36c72e467efc7969b95161d96fc242d95f35 (refs/remotes/origin/main)
Commit message: "Testing Jenkins pipeline"
```

This shows the Source stage where Jenkins detects that no Git installation is configured (using default). It successfully fetches changes from the remote repository and checks out the main branch commit, preparing the workspace for the build process.

3. Build Stage

```
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ docker build -t nodejs-app:latest .
#0 building with "default" instance using docker driver

#1 [internal] load build definition from Dockerfile
> git rev-parse --resolve-git-dir /opt/build/workspace/jenkins-nodejs-pipeline/.git # timeout=10
> git config remote.origin.url https://github.com/Umarsattil/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git # timeout=10
Fetching upstream changes from https://github.com/Umarsattil/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git
> git --version # timeout=10
> git --version # 'git version 2.43.0'
> git fetch --tags --force --progress -- https://github.com/Umarsattil/Task-12-Jenkins-Setup-and-ECS-Deployment-Pipeline.git +refs/heads/*:refs/remotes/origin/* # timeout=10
> git rev-parse refs/remotes/origin/main^{commit} # timeout=10
> git config core.sparsecheckout # timeout=10
> git checkout -f 653c36c72e467efc7969b95161d96fc242d95f35 # timeout=10
> git branch -a -v --no-abbrev # timeout=10
> git branch -D main # timeout=10
> git checkout -b main 653c36c72e467efc7969b95161d96fc242d95f35 # timeout=10
#1 transferring dockerfile: 534B done
#1 DONE 0.0s

#2 [internal] load metadata for docker.io/library/node:20-alpine
#2 DONE 0.6s

#3 [internal] load .dockerignore
#3 transferring context: 2B done
#3 DONE 0.0s

#4 [internal] load build context
#4 transferring context: 123B done
#4 DONE 0.0s
```

The Build stage executes a Docker build command (docker build -t nodejs-app:latest). It creates a Docker image using the default driver, transfers the Dockerfile, loads Docker metadata and context, and completes the build process.

4. Test Stage

```
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ sleep 5
+ docker run --rm nodejs-app:latest node app.js
Server is now serving static files from the 'public' folder.
Access the application at http://localhost:3000
+ docker ps -q
+ xargs docker stop
a9572b959df9
```

The **Test stage** runs the application in a Docker container with docker run --rm nodejs-app:latest node app.js. The output confirms the Node.js server is running and serving static files from the 'public' folder, accessible at http://localhost:3000. It then lists and stops the Docker container.

5. Push Stage

```
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (push)
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ aws ecr get-login-password --region us-west-1
+ docker login --username AWS --password-stdin 504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-app

WARNING! Your credentials are stored unencrypted in '/root/.docker/config.json'.
Configure a credential helper to remove this warning. See
https://docs.docker.com/go/credential-store/

Login Succeeded
+ docker tag nodejs-app:latest 504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-app:latest
+ docker push 504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-app:latest
The push refers to repository [504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-app]
f9e26c244f7f: Waiting
f8bc6c62be16: Layer already exists
fd1849a5c548: Layer already exists
d62659c90dbd: Layer already exists
1074353eec0d: Layer already exists
8d06ba6946d1: Layer already exists
fcb8b8ee4622: Layer already exists
cf4ab8ea3408: Layer already exists
cb3325e64457: Layer already exists
f9e26c244f7f: Pushed
latest: digest: sha256:0e7638478c970433d9259fdb45264961467d3afce35077af391125d406098f7 size: 856
```

The Push stage authenticates with AWS ECR using aws ecr get-login-password, then tags and pushes the Docker image to the ECR repository (504649076991.dkr.ecr.us-west-1.amazonaws.com/nodejs-app). "Layer already exists" indicates efficient caching, with the final image digest and size displayed after successful push.

6. Deploy Stage

```
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ aws ecs update-service --cluster nodejs-ecs-cluster --service nodejs-app-task-definition-service --force-new-deployment --region us-west-1
```

The **Deploy stage** executes an AWS ECS update-service command to deploy the new image to the ECS cluster named "nodejs-ecs-cluster" in the us-west-1 region, forcing a new deployment of the service "nodejs-app-task-definition-service."

7. Pipeline Success

```
Deployment succeeded! The ECS service has been updated with the new image.
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

The pipeline concludes with "Deployment succeeded! The ECS service has been updated with the new image." The pipeline finishes with a SUCCESS status, confirming all stages (checkout, build, test, push, deploy) completed successfully.

8.7: ECS Service Events

Verified new task revisions and successful rolling deployment.

The screenshot shows the AWS ECS Cluster Overview page for the 'nodejs-ecs-cluster'. The top navigation bar includes 'Amazon Elastic Container Service > Clusters > nodejs-ecs-cluster > Tasks'. On the left, a sidebar lists 'Express Mode', 'Clusters' (selected), 'Namespaces', 'Task definitions', 'Account settings', 'Amazon ECR', 'Repositories', 'AWS Batch', 'Documentation', 'Discover products', and 'Subscriptions'. The main content area is titled 'nodejs-ecs-cluster' and shows 'Cluster overview' with ARN 'arn:aws:ecs:us-west-1:504649076991:cluster/nodejs-ecs-cluster' and Status 'Active'. It also shows 'CloudWatch monitoring' and 'Registered container instances'. Below this, the 'Services' section shows 'Draining' and 'Active' status with 1 task. The 'Tasks' section shows 'Pending' and 'Running' status with 2 tasks. A 'Tasks (6)' table is displayed, listing six tasks with columns for Task ID, Last status, Desired status, Task definition, Health status, Created at, Started by, Started at, and Group. The tasks are listed with their status (Running or Stopped) and the corresponding task definition ID.

Task	Last status	Desired status	Task definition	Health status	Created at	Started by	Started at	Group
21da2ac38ad94...	Running	Running	nodejs-app-task-definition:1	Unknown	5 minutes ago	ecs-svc/57626389151...	5 minutes ago	service:nodejs-ap
8cad0d9bd9104...	Running	Running	nodejs-app-task-definition:1	Unknown	3 minutes ago	ecs-svc/57626389151...	2 minutes ago	service:nodejs-ap
14c249c4c9314a...	Stopped	Stopped	nodejs-app-task-definition:1	Unknown	2 hours ago	ecs-svc/50086675882...	2 hours ago	service:nodejs-ap
29d0bbe5b1934...	Stopped	Stopped	nodejs-app-task-definition:1	Unknown	19 minutes ago	ecs-svc/8242589728...	19 minutes ago	service:nodejs-ap
35070892be24...	Stopped	Stopped	nodejs-app-task-definition:1	Unknown	21 minutes ago	ecs-svc/8242589728...	21 minutes ago	service:nodejs-ap
78c491ca25c043...	Stopped	Stopped	nodejs-app-task-definition:1	Unknown	2 hours ago	ecs-svc/30086675882...	2 hours ago	service:nodejs-ap

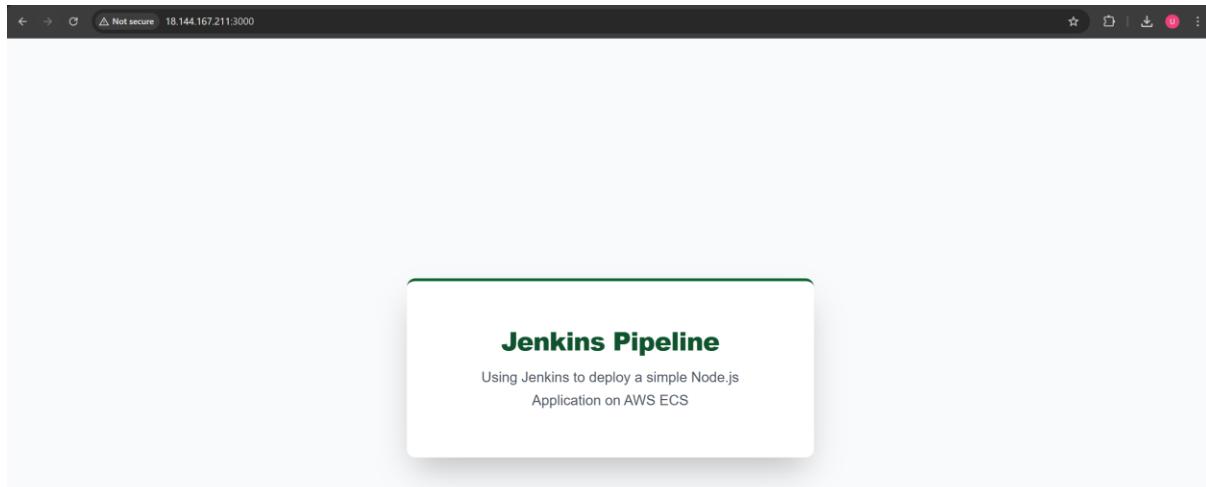
Events (16) Info		Last updated Dec 19, 2025, 04:55 (UTC+5:00) Edit
Started at	Message	Event type
December 19, 2025, 04:52 (UTC+5:00)	service nodejs-app-task-definition-service has reached a steady state.	All events
December 19, 2025, 04:52 (UTC+5:00)	service nodejs-app-task-definition-service deployment ecs-svc/5762638915142188496 deployment completed.	
December 19, 2025, 04:50 (UTC+5:00)	service nodejs-app-task-definition-service has started 1 tasks: task 8cad0dbd91042f3b5801dc2eb0e701 .	
December 19, 2025, 04:49 (UTC+5:00)	service nodejs-app-task-definition-service has stopped 1 running tasks: task 35070892be2f4d7fa5d16ec61ccb1b79 .	
December 19, 2025, 04:48 (UTC+5:00)	service nodejs-app-task-definition-service has started 1 tasks: task 21da2ac38ad948418614e81bd4ea63ac .	
December 19, 2025, 04:47 (UTC+5:00)	service nodejs-app-task-definition-service has stopped 1 running tasks: task 29d8be5b193e4b53897d9258738481a9 .	
December 19, 2025, 04:36 (UTC+5:00)	service nodejs-app-task-definition-service has reached a steady state.	
December 19, 2025, 04:36 (UTC+5:00)	service nodejs-app-task-definition-service deployment ecs-svc/8242538972861294843 deployment completed.	
December 19, 2025, 04:34 (UTC+5:00)	service nodejs-app-task-definition-service has started 1 tasks: task 29d8be5b193e4b53897d9258738481a9 .	
December 19, 2025, 04:33 (UTC+5:00)	service nodejs-app-task-definition-service has stopped 1 running tasks: task 78c491ce25c043b4ba00d11c5944d69b .	
December 19, 2025, 04:32 (UTC+5:00)	service nodejs-app-task-definition-service has started 1 tasks: task 35070892be2f4d7fa5d16ec61ccb1b79 .	
December 19, 2025, 04:30 (UTC+5:00)	service nodejs-app-task-definition-service has stopped 1 running tasks: task 14c249c4c9314aedae6bc4fd7beac56e .	
December 19, 2025, 02:47 (UTC+5:00)	service nodejs-app-task-definition-service has reached a steady state.	
December 19, 2025, 02:47 (UTC+5:00)	service nodejs-app-task-definition-service deployment ecs-svc/3008667588251526730 deployment completed.	
December 19, 2025, 02:44 (UTC+5:00)	service nodejs-app-task-definition-service has started 1 tasks: task 78c491ce25c043b4ba00d11c5944d69b .	
December 19, 2025, 02:44 (UTC+5:00)	service nodejs-app-task-definition-service has started 1 tasks: task 14c249c4c9314aedae6bc4fd7beac56e .	

8.8: Application Access

Accessed the application using the new ECS task ENI IP.

Updated Application Confirmation

- The updated UI confirmed the new Docker image was deployed.
- ECS launched new tasks, resulting in a different ENI IP.
- Old tasks were terminated as part of the rolling update process.



This completes the end-to-end CI/CD workflow, demonstrating automated build, image management, and deployment to Amazon ECS using Jenkins.

9. Troubleshooting

The following issues were encountered during the setup and execution of the Jenkins CI/CD pipeline, Docker image deployment, and ECS service updates. Each issue outlines the observed problem, root cause, and the steps taken to resolve it.

Issue 1: Jenkins Pipeline Failed During ECR Authentication

Problem Description

During pipeline execution, the build failed at the ECR authentication stage with the following error:

```
aws: not found
error: cannot perform an interactive login from a non TTY device
```

Root Cause

The Jenkins Agent EC2 instance did not have the AWS CLI installed. Since Jenkins uses the AWS CLI to authenticate with Amazon ECR and push Docker images, the aws ecr get-login-password command could not be executed.

Solution

Installed AWS CLI v2 on the Jenkins Agent EC2 instance:

- Downloaded and installed AWS CLI using the official AWS installer.
- Verified installation using aws --version.

After installing AWS CLI, the pipeline was able to authenticate with ECR and successfully push Docker images.

Issue 2: ECS Service Continued Running Old Application Version

Problem Description

After a successful Jenkins pipeline run and Docker image push to ECR, the application UI did not reflect recent changes. ECS was still running the old application version.

Root Cause

The ECS service was still using **task definition revision 1**, which referenced an older container image. Although new Docker images were being built and pushed to ECR using \${BUILD_NUMBER}, ECS does not automatically deploy new images unless a new task definition revision is created and applied to the service.

As a result, new images existed in ECR but ECS continued running the old task definition. Hence, the old container image and HTML content remained active.

Solution

Updated the Jenkins pipeline to use the latest image tag instead of \${BUILD_NUMBER}.

- ECS pulls the latest image on every forced deployment.
- Ensures ECS always runs the most recently pushed image without requiring task definition updates.

This change resolved the issue and allowed ECS to consistently deploy updated application versions.

Issue 3: GitHub Webhook Trigger Not Working

Problem Description

GitHub pushes were not triggering the Jenkins pipeline automatically.

Root Cause

The Jenkins Master EC2 instance IP address changed after stopping and restarting the instance. Since the GitHub webhook was configured using the old public IP, webhook requests were no longer reaching Jenkins.

Solution

Assigned an **Elastic IP (EIP)** to the Jenkins Master EC2 instance.

- Ensured a static public IP address for Jenkins.
- Updated the GitHub webhook to use the Elastic IP.

After assigning the Elastic IP, GitHub webhook triggers functioned reliably.

Issue 4: Jenkins Agent Showing Offline After Restarting Jenkins Master

Problem Description

After restarting the Jenkins Master EC2 instance, the Jenkins Agent appeared as **offline** in the Jenkins UI.

Root Cause

The Jenkins Master IP address changed after restart. The agent was still attempting to connect to the old master IP, causing the connection to fail.

Solution

Reconnected the Jenkins Agent using the updated Jenkins Master IP.

- Downloaded the agent JAR from the new Jenkins Master URL.
- Re-ran the agent connection command with the updated IP address.

```
curl -sO http://<NEW-JENKINS-MASTER-IP>:8080/jnlpJars/agent.jar  
java -jar agent.jar -url http://<NEW-JENKINS-MASTER-IP>:8080/ \  
-secret <SECRET> -name Slave -webSocket -workDir "/opt/build"
```

After re-running the command, the agent successfully reconnected and became online.