

TASK 8

**Deploying an Application
using AWS CodePipeline on
ECS EC2 with Terraform**

Umar Satti

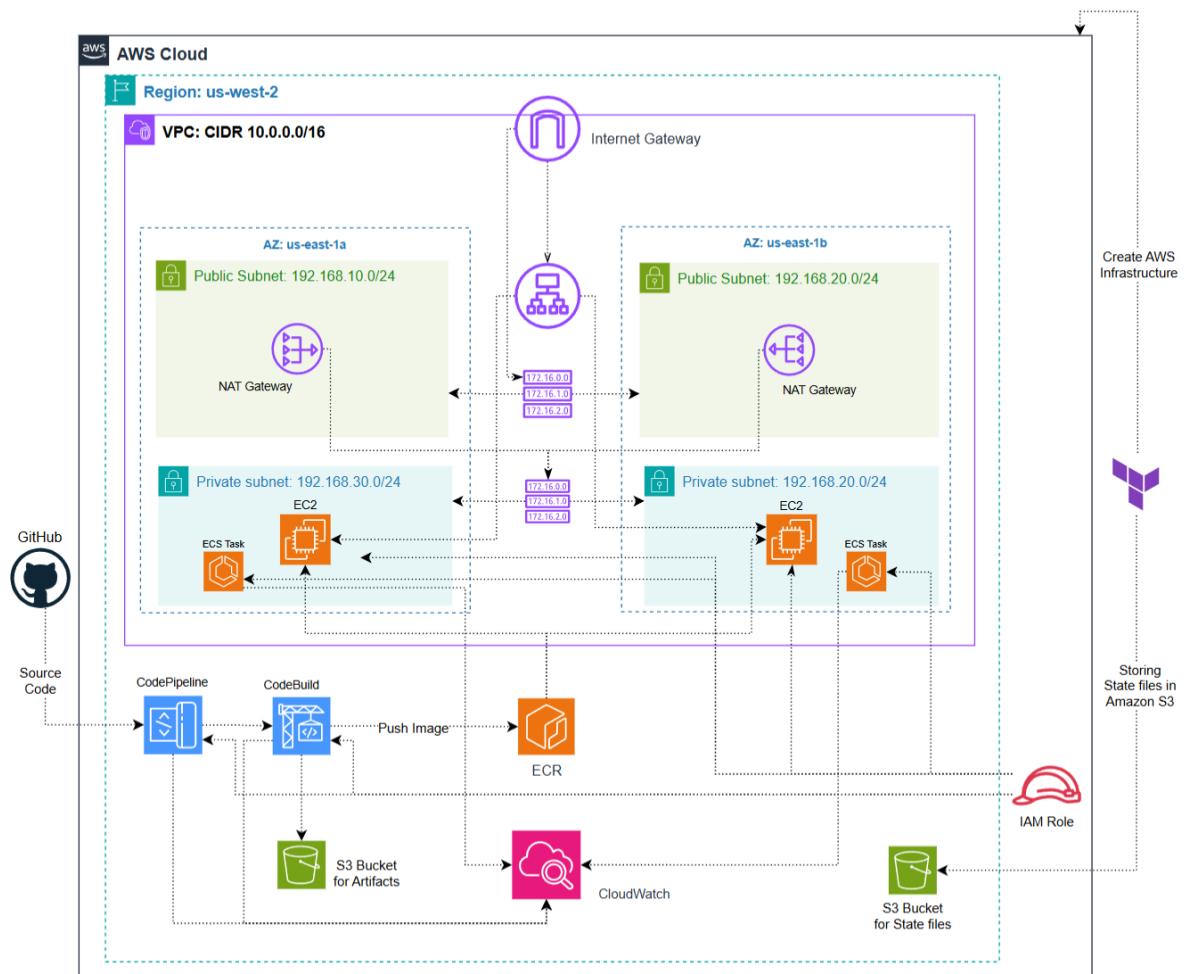
Table of Contents

Task Description	3
Architecture Diagram.....	3
Tasks	4
Task 1.1: Flask Application and Dockerfile Configuration	4
Task 1.2: Terraform Structure.....	6
Task 1.3: S3 Bucket for Terraform Remote Backend.....	7
Task 1.4: Root Directory Files.....	8
Task 1.5: VPC Module	12
Task 1.6: ECR Module	13
Task 1.7: Task Definition Module	14
Task 1.8: ECS Module.....	16
Task 1.9: ALB Module	17
Task 1.10: CodeBuild Module.....	18
Task 1.11: CodePipeline Module.....	20
Task 1.12: Execute Terraform Commands	21
Task 1.13: Validate Infrastructure in AWS Console	22
Task 1.14: Deploy Flask Application via CodePipeline	33
Task 1.15: Confirm Flask Application is Accessible.....	33
Task 1.16: Clean Up	33
Task 1.17: Troubleshooting	34

Task Description

This project demonstrates a complete CI/CD pipeline for deploying a Python Flask web application on Amazon ECS using EC2 launch type with AWS CodePipeline. The infrastructure includes a highly available VPC with public/private subnets, Application Load Balancer (ALB), private ECR repository, ECS cluster with Auto Scaling Group (ASG), CodeBuild for Docker image building, and CodePipeline orchestrated from GitHub source code. All resources are provisioned using Terraform with modular architecture.

Architecture Diagram



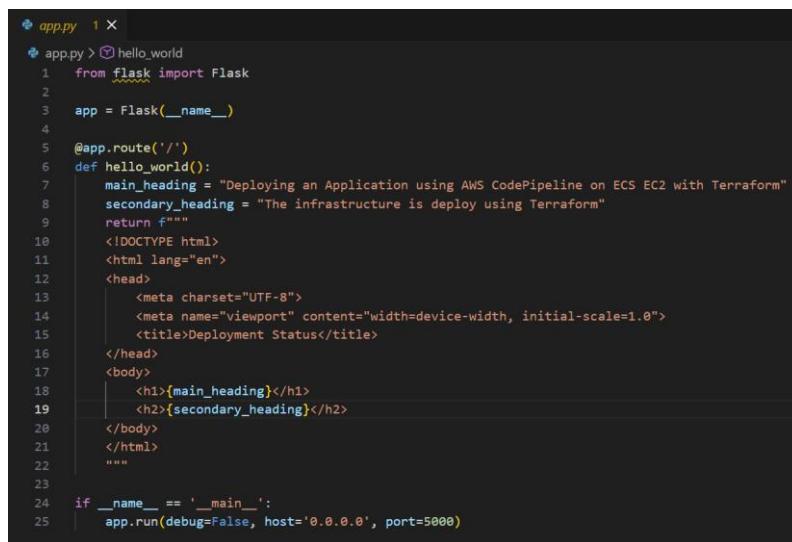
Tasks

Task 1.1: Flask Application and Dockerfile Configuration

The first step of the project is containerizing a **Python Flask application** that serves a webpage on port 5000. This image will be uploaded to ECR and deployed on ECS EC2.

Step 1: Create Flask Application (app.py)

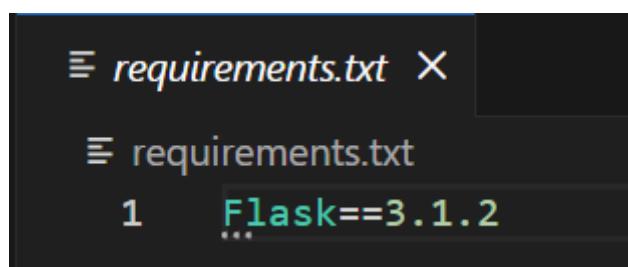
- Implements a simple Flask app with one route (/).
- Returns an HTML page with main heading and secondary heading.
- Runs on **port 5000** and binds to 0.0.0.0 so ECS can route traffic to the container.



```
❸ app.py 1 ×
❹ app.py > ⚏ hello_world
  1  from flask import Flask
  2
  3  app = Flask(__name__)
  4
  5  @app.route('/')
  6  def hello_world():
  7      main_heading = "Deploying an Application using AWS CodePipeline on ECS EC2 with Terraform"
  8      secondary_heading = "The infrastructure is deploy using Terraform"
  9      return f"""
 10         <!DOCTYPE html>
 11         <html lang="en">
 12             <head>
 13                 <meta charset="UTF-8">
 14                 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 15                 <title>Deployment Status</title>
 16             </head>
 17             <body>
 18                 <h1>{main_heading}</h1>
 19                 <h2>{secondary_heading}</h2>
 20             </body>
 21         </html>
 22     """
 23
 24  if __name__ == '__main__':
 25      app.run(debug=False, host='0.0.0.0', port=5000)
```

Step 2: Create requirements.txt

- Lists all Python dependencies.
- Contains **Flask==3.1.2** to ensure the Flask app runs inside the container.
- Keeps builds reproducible across environments.



```
☰ requirements.txt ×
☰ requirements.txt
  1  Flask==3.1.2
```

Step 3: Create Dockerfile

- **Base image:** python:3.13-alpine for a lightweight Python environment.
- **Working directory:** /app inside the container.

- **Copy and install dependencies:**
 - requirements.txt is copied and installed via pip.
 - **Copy application code:** app.py is added to the container.
 - **Expose port 5000:** Matches the Flask app, allowing ALB and ECS to route traffic.
 - **Run the app:** CMD ["python", "app.py"] starts Flask in the container.

```
  Dockerfile X

  Dockerfile
1  #Use the official Python base image
2  FROM python:3.13-alpine
3
4  #Set the working directory inside the container
5  WORKDIR /app
6
7  #Copy the requirements file into the container
8  COPY requirements.txt .
9
10 #Install the Python dependencies (Flask)
11 RUN pip install --no-cache-dir -r requirements.txt
12
13 #Copy the application code into the container
14 COPY app.py .
15
16 #Expose the port that the Flask app runs on
17 EXPOSE 5000
18
19 #Define the command to run the application
20 CMD ["python", "app.py"]
```

Step 4: Test Locally

1. Build the Docker image using `docker build -t flask-app:latest`.
 - o Validates the Dockerfile.
 - o Produces a container image ready for ECS deployment.

```

PS D:\Cloudelligent\Task-B
PS D:\Cloudelligent\Task-B> docker build -t flask-app:latest .
[*] [internal] load build definition from Dockerfile
=> => transferring dockerfile: 53B
[*] [internal] load metadata for docker.io/library/python:3.13-alpine
=> [auth] library/python:pull token for registry-1.docker.io
[*] [internal] load .dockerignore
=> [internal] transfer context: 0B
[*] [internal] load build context
=> => transferring context: 765B
[*] [internal] resolve docker.io/library/python:3 15-alpine@sha256:239fb5a53e308f7928c4690a6df1f0b9f11b525f3
=> sha256:239fb5a53e308f7928c4690a6df1f0b9f11b525f3 0.0s
=> sha256:e511ae37c0152d2ed2a71c45b670d28773c3e9a1d8ff5542f9aae5223bf8 5.29Kb 0.0s
[*] sha256:239fb5a53e308f7928c4690a6df1f0b9f11b525f3 10.36B / 10.36B
=> sha256:e511ae37c0152d2ed2a71c45b670d28773c3e9a1d8ff5542f9aae5223bf8 1.74Kb / 1.74Kb 0.0s
=> sha256:56082da5225e9a8919814e1a1b567881e1a845e1b84ee1d8b6deec2a2 456.93M / 456.93M 1.6s
=> sha256:316e4d4045e548e78a97337d1ca8e7547d5038824a186ef7c401f5e37 12.44Kb / 12.44Kb 5.7s
[*] extracting sha256:56082da5225e9a8919814e1a1b567881e1a845e1b84ee1d8b6deec2a2 0.2s
[*] extracting sha256:316e4d4045e548e78a97337d1ca8e7547d5038824a186ef7c401f5e37 3.1s
[*] extracting sha256:b9768522e5642680d9a7b0e8f643e1a6dc3c7493b663e093e250c2236185890da 0.0s
[*] [internal] load build context
=> [internal] transferring context: 765B
[*] [2/2] WORKDIR /app
[*] [3/3] COPY requirements.txt .
[*] [4/4] RUN pip install --no-cache-dir -r requirements.txt
[*] [5/5] COPY app.py .
[*] exporting image
=> => exporting layers
[*] writing image sha256:2c56a8cb237b3054ea609696061adaf4591ffd5f233ea6f24aa82b2ff323ac3
=> naming to docker.io/library/flask-app:latest 0.0s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/omc4x081rldn7h7yjm801

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

```

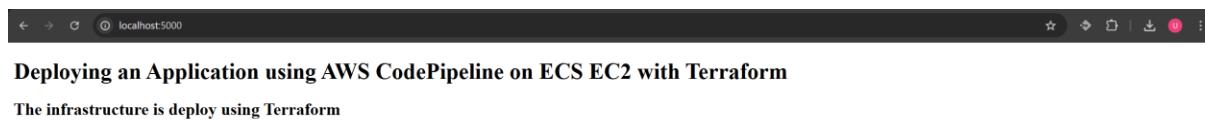
- Run container using `docker run -d -p 5000:5000 --name flask-app flask-app:latest`
 - Maps local port 5000 to container port 5000.
 - Confirms the Flask app is accessible locally.

```
● PS D:\Cloudelligent\Task-8> docker run -d -p 5000:5000 --name flask-app flask-app:latest
255a19373c12e72be74c8984584be4cc0fe244a84c798b445a6d13f9636403d0
○ PS D:\Cloudelligent\Task-8> █
```

3. Verify:

- **docker ps -a** command shows the container is running.
- Open <http://localhost:5000> to confirm the webpage displays correctly.

```
● PS D:\Cloudelligent\Task-8> docker ps -a
CONTAINER ID   IMAGE       COMMAND      CREATED     STATUS      PORTS          NAMES
255a19373c12   flask-app:latest "python app.py"  35 seconds ago   Up 35 seconds   0.0.0.0:5000->5000/tcp, [::]:5000->5000/tcp   flask-app
○ PS D:\Cloudelligent\Task-8> █
```



Task 1.2: Terraform Structure

Steps:

1. Create a `terraform.tf` file in the root directory. This file defines the AWS provider, provider version, AWS region, and the S3 backend for storing Terraform state files.
2. Create an S3 bucket in the same AWS region to store Terraform state files (explained in Task 1.3 below).
3. Create a `main.tf` file in the root directory. This connects modules together, passes outputs from one module to another, and passes variables between them.
4. Create a `variables.tf` file in the root directory. This file defines variables by description and type as key-value pairs. These contain arguments for parameters such as VPC name, CIDR block, ECR URIs, and more.
5. Create an `outputs.tf` file in the root directory. This file exposes important module outputs such as ALB DNS name after deployment.
6. Create a `modules` directory that contains 7 total modules including VPC, Task definition, ECR, ECS, ALB, CodePipeline, and CodeBuild. Each module contains its own individual `main.tf`, `variables.tf`, and `outputs.tf` files.

```
D:.
  .terraform.lock.hcl
  main.tf
  outputs.tf
  terraform.tf
  terraform.tfvars
  variables.tf

  .terraform
    terraform.tfstate

    modules
      modules.json

    providers
      registry.terraform.io
        hashicorp
          aws
            6.23.0
              windows_amd64
                LICENSE.txt
                terraform-provider-aws_v6.23.0_x5.exe

    modules
      alb
        main.tf
        outputs.tf
        variables.tf

      codebuild
        main.tf
        outputs.tf
        variables.tf

      codepipeline
        main.tf
        outputs.tf
        variables.tf

      ecr
        main.tf
        outputs.tf
        variables.tf

      ecs
        main.tf
        outputs.tf
        variables.tf

        task_definition
          main.tf
          outputs.tf
          variables.tf

      vpc
        main.tf
        outputs.tf
        variables.tf
```

Task 1.3: S3 Bucket for Terraform Remote Backend

Steps:

1. Log in to the AWS Management Console. Navigate to S3 using the search bar at the top of the console page.
2. Click on **Create Bucket** button.
3. Choose **General Purpose**, add a globally unique bucket name, and make sure the AWS Region is the same as Terraform.
4. Click **Create Bucket**.
5. Update **terraform.tf** file in root directory to reference this S3 bucket in the backend block.

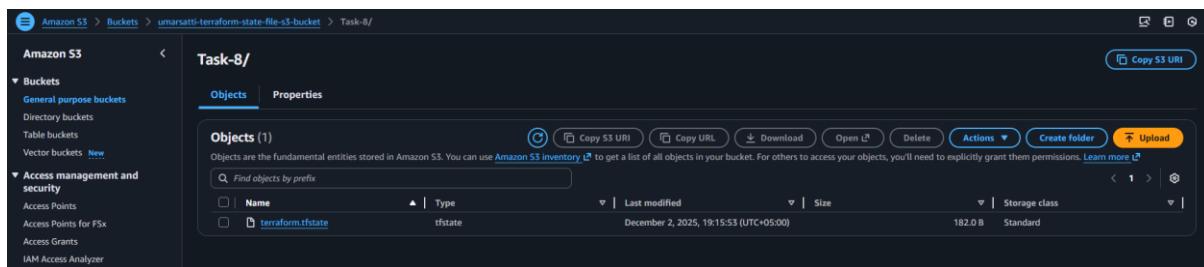
```

backend "s3" {
  bucket      = "umarsatti-terraform-s3-bucket-state-file"
  key         = "Task-5/terraform.tfstate"
  region      = "us-west-2"
  encrypt     = true
  use_lockfile = true
}

```

Once the S3 bucket is created in the AWS Management Console and referenced in the Terraform backend configuration, Terraform automatically begins storing and versioning state files in this bucket. In this case, the S3 bucket named **umarsatti-terraform-s3-bucket-state-file** is used as the remote backend, as defined in the **provider.tf** file. The backend block ensures all state information is centralized, secure, and persistent across multiple users or workstations.

The screenshot shown below shows the exact file path inside the S3 bucket:
S3 > Buckets > umarsatti-terraform-state-file-s3-bucket > Task-8 > terraform.tfstate



This confirms that:

- Terraform successfully initialized the backend and wrote the state file to the S3 bucket.
- The **terraform.tfstate** file contains metadata about all deployed AWS resources (VPC, Subnets, Security groups, etc.).
- Every terraform plan, apply, or destroy operation reads and updates this file automatically.
- The locking mechanism (enabled by **use_lockfile = true**) ensures that no two processes modify the state simultaneously, preventing state corruption.

Task 1.4: Root Directory Files

Terraform root directory acts as the orchestration layer for all AWS infrastructure.

While each AWS service (VPC, ALB, ECS, ECR, Task Definitions, CodeBuild, CodePipeline) is defined in its own module, the root directory ties everything together by:

- Defining the backend and AWS provider.
- Supplying variables into each module.
- Connecting module outputs to other modules (e.g., ECR → Task Definition → ECS).

- Managing environment-specific configuration with `terraform.tfvars`.
- Exposing global outputs after deployment.

This keeps the project modular, reusable, and maintainable.

terraform.tf

This file configures how Terraform runs and where it stores its state.

Purpose

- Declares the required AWS provider version.
- Defines the remote backend (S3 bucket) for storing the Terraform state file.
- Enables state locking to prevent concurrent modifications.
- Sets the deployment region.

Key points

- Using an S3 backend ensures collaboration and prevents state loss.
- State locking improves safety when multiple people run Terraform.
- The AWS provider block ensures all modules use `us-east-1`.

main.tf

This is the central “orchestration” file that wires all modules together.

Purpose

- Loads modules for **VPC**, **ECR**, **Task Definition**, **ECS**, **ALB**, **CodeBuild**, and **CodePipeline**.
- Passes variables to each module.
- Feeds outputs (e.g., subnets, image URI, target group ARN) into downstream modules.
- Automatically creates dependency order between infrastructure components.

High-level flow

1. **VPC module** builds networking (VPC, public/private subnets, IGW, security groups).
2. **ECR module** creates private container registry.
3. **Task Definition module** builds the ECS Task Definition using the ECR image and CloudWatch Logs.
4. **ALB module** creates an Application Load Balancer and Target Group.
5. **ECS module** deploys the ECS Cluster and Service, wiring in VPC networking, Task Definition ARN, and ALB target group.
6. **CodeBuild module** sets up the CI build environment inside the VPC.
7. **CodePipeline module** creates the end-to-end deployment pipeline, linking GitHub → CodeBuild → ECS.

This structure ensures clean separation of responsibilities while keeping the root module in control of the overall architecture.

variables.tf

Defines all input variables used across the root and module configurations.

Purpose

- Makes the project configurable, reusable, and environment-agnostic.
- Provides typed variables with descriptions, improving validation and readability.
- Allows module inputs to be managed from a central place.

Examples of variable groups:

- **Networking variables** (VPC CIDR ranges, names).
- **ECR variables** (repo name, encryption options).
- **Task Definition variables** (CPU, memory, container name, logs).
- **ECS variables** (cluster/service names, instance type, IAM roles).
- **ALB variables** (ports, names, target type).
- **Pipeline variables** (pipeline name, GitHub connection, S3 bucket).

terraform.tfvars

Provides the actual values for the variables defined in variables.tf.

```

terraform > terraform.tfvars > ...
terraform > terraform.tfvars > ...
1   # VPC Variables
2   vpc_cidr      = "192.168.0.0/16"
3   vpc_name       = "Umarsatti-VPC"
4   igw_name       = "Umarsatti-IGW"
5   eip_domain     = "vpc"
6   public_route_cidr = "0.0.0.0/0"
7
8   # ECR Variables
9   ecr_repository_name = "python-ecr"
10  ecr_mutability      = "MUTABLE"
11  ecr_encryption      = "AES256"
12
13  # Task Definition Variables
14  task_exec_name    = "ecs-ec2-task-execution-iam-role"
15  log_group_name    = "ecs-task-definition-log-group"
16  task_definition_name = "python-flask-task-definition"
17  network_mode      = "bridge"
18  launch_type        = "EC2"
19  task_cpu           = "256"
20  task_memory        = "512"
21  container_name     = "flask"
22
23  # ECS Variables
24  ec2_iam_role_name = "ec2-instance-role-for-ecs"
25  ec2_instance_profile_name = "amazon-ec2-instance-profile"
26  instance_type      = "t3.micro"
27  cluster_name        = "Flask-ECS-Cluster"
28  service_name        = "Flask-ECS-Service"
29
30  # ALB Variables
31  target_type        = "instance"
32  target_group_port  = 5000
33  listener_port      = 80
34  alb_name           = "ecs-ec2-alb"
35  lb_type             = "application"
36  tg_name             = "ecs-ec2-target-group"
37
38  # CodeBuild Variables
39  codebuild_iam_role  = "CodeBuild-iam-role-ecs-ec2"
40  codebuild_project_name = "codebuild-ecs-ec2-project"
41  account_id          = "730335208305"
42  region               = "us-east-1"
43  codebuild_logs        = "codebuild-ecs-ec2-log-group"
44
45  # CodePipeline Variables
46  bucket_name          = "codepipeline-umarsatti-ecs-ec2-bucket"
47  connection            = "aws-github-connection"
48  provider_type         = "GitHub"
49  codepipeline_role_name = "CodePipeline-iam-role-ecs-ec2"
50  pipeline_name          = "codePipeline-ecs-ec2-project"
51  github_repo_url       = "Umarsattii/Task-8-AWS-CodePipeline-on-ECS-EC2-with-Terraform"
52

```

Purpose

- Keeps configuration separate from logic.
- Allows easy switching between environments (e.g., dev, test, prod).
- Supplies concrete values for all networking, compute, container, ALB, CI/CD, and IAM parameters.

This enables the same Terraform code to deploy different setups simply by swapping tfvars files.

outputs.tf

Exposes key information after deployment.

Common outputs

- **ECR repository URI** – used for pushing application images.
- **ALB DNS name** – used for accessing the application through the load balancer.
- **Cluster and service identifiers** – useful for integration or debugging.

Outputs make it easy to retrieve essential resource details without browsing the AWS console.

Task 1.5: VPC Module

This section explains each Terraform configuration file located inside the **VPC module** (modules/vpc).

main.tf

Local Variables

The module defines three maps to dynamically build networking resources:

- **public_subnets** – CIDRs and AZs for all public subnets (used for ALB and NAT Gateways).
- **private_subnets** – CIDRs and AZs for all private subnets (used for ECS instances running tasks).
- **private_to_nat** – Maps each private subnet to the NAT Gateway in its corresponding public subnet, ensuring correct outbound routing per AZ.

These locals allow scalable subnet creation with **for_each** meta-argument.

VPC

Creates the main VPC using the CIDR passed from the root module.

DNS support and hostnames are enabled for service discovery and internal communication.

Public & Private Subnets

Both subnet types are generated dynamically from locals:

- Public subnets map public IPs on launch and host ALB and NAT Gateways.
- Private subnets remain isolated with no public IPs and are used for ECS compute.

Tags use the subnet names from locals for clear identification.

Internet Gateway (IGW)

Provides internet access to all public subnets. Required for NAT Gateways to function.

Elastic IPs and NAT Gateways

Each public subnet gets:

- Its own **Elastic IP**
- Its own **NAT Gateway**

This guarantees high availability and ensures private subnets can reach the internet (e.g., pull images from ECR) without being publicly exposed.

Route Tables & Routes

- **Public Route Tables:** One per public subnet, routing 0.0.0.0/0 through the IGW.
- **Private Route Tables:** One per private subnet, routing outbound traffic through the NAT Gateway mapped via `private_to_nat`.

Associations link each subnet to the correct route table.

Security Groups

Two security groups support ALB → ECS traffic flow:

1. ALB Security Group

- Allows inbound HTTP (port 80) from anywhere.
- Egress is fully open.
- Used by the ALB.

2. ECS/EC2 Security Group

- Allows inbound traffic only from the ALB security group on:
 - Port **5000** (your Flask app)
 - Ephemeral ports (32768–65535) for ALB health checks
- Outbound traffic is fully open.
- Used by ECS EC2 instances and tasks.

This ensures ECS tasks are private and only reachable through the ALB.

variables.tf

The VPC module accepts five variables:

- **vpc_cidr** – The VPC’s IP range.
- **vpc_name** – Tag for the VPC.
- **igw_name** – Tag for the Internet Gateway.
- **eip_domain** – Domain setting for NAT EIPs.
- **public_route_cidr** – Destination CIDR for public and private routes (usually 0.0.0.0/0).

These variables keep the module flexible and reusable.

outputs.tf

The module exposes essential network details for downstream modules:

- **vpc_id** – Needed by ALB, ECS, and CodeBuild modules.
- **public_subnets** – Used by ALB, NAT Gateways, and ECS if required.
- **private_subnets** – Used by ECS tasks and CodeBuild.
- **alb_sg_id** – Passed to the ALB module.
- **ec2_ecs_sg_id** – Passed to the ECS module for instance/task security.

These outputs allow the VPC module to integrate cleanly with the rest of the infrastructure.

Task 1.6: ECR Module

This section explains each Terraform configuration file located inside the **ECR module** (modules/ecr).

main.tf

The module creates a single **Amazon ECR repository** used to store container images for the ECS service.

Key points:

- **name** comes from `ecr_repository_name`, allowing different repositories per environment (dev, test, prod).
- **image_tag_mutability** is controlled by `ecr_mutability` (e.g., MUTABLE/IMMUTABLE), defining whether image tags can be overwritten.
- **force_delete = true** ensures the repository can be destroyed even if images exist, preventing cleanup issues during iterative development.
- **encryption_configuration** uses the value in `ecr_encryption` (such as AES256) to secure all images at rest.
- **image_scanning_configuration** disables scan-on-push, simplifying CI/CD pipelines but can be enabled later for security hardening.

This repository serves as the private storage location for the ECS application image pulled during deployments.

variables.tf

The module defines three input variables:

- **ecr_repository_name** – sets the name of the ECR repo.
- **ecr_mutability** – controls tag mutability.
- **ecr_encryption** – defines encryption type for stored images.

These inputs make the module fully reusable with different naming conventions or security settings.

outputs.tf

Two outputs provide essential information:

- **ecr_image_uri** – the full repository URL (e.g., `account.dkr.ecr.region.amazonaws.com/repo`), required by the Task Definition.
- **ecr_repository_name** – exposes the repo name for use in CodeBuild or pipelines.

These outputs allow other modules (Task Definition, CodeBuild, CodePipeline) to seamlessly reference the container registry.

Task 1.7: Task Definition Module

This section explains each Terraform configuration file located inside the **Task Definition module** (`modules/task_definition`).

main.tf

This module creates everything required for an ECS Task Definition to run your container on Fargate.

IAM Role (Task Execution Role)

- The role (`ecs_task_execution_role`) is assumed by ECS tasks and allows the service to pull images and write logs.
- The AWS-managed policy **AmazonECSTaskExecutionRolePolicy** is attached, giving access to ECR, CloudWatch Logs, and Secrets Manager.

CloudWatch Log Group

- Creates a log group defined by `log_group_name`.
- Retains logs for 7 days to keep storage costs low.
- The container writes its stdout/stderr here.

ECS Task Definition

- Defines how the application container runs on ECS Fargate.
- `family`, `cpu`, `memory`, `network_mode`, and `launch_type` all come from variables to keep the module reusable.
- Both `execution_role_arn` and `task_role_arn` use the same role for simplicity.
- The container uses the ECR image passed in (`ecr_image_uri:latest`).
- Exposes port **5000** to the ALB.
- Configures AWS Logs driver so all container logs stream to the created CloudWatch log group.

This task definition becomes the blueprint ECS uses when running or updating your service.

variables.tf

This file defines all parameters required to customize the task definition:

- **ecr_image_uri** – the repository URL from the ECR module.
- **task_exec_name**, **task_definition_name**, **container_name** – naming inputs for IAM and ECS resources.
- **network_mode**, **launch_type** – Fargate compatibility settings.
- **task_cpu**, **task_memory** – resource configuration.
- **log_group_name** – CloudWatch log group name.

All variables make the module fully adaptable for different services or environments.

outputs.tf

- **task_definition_arn** – the full ARN needed by the ECS Service module when referencing this task.
- **container_name** – used by load balancer targets and ECS service configuration.

These outputs help connect this module to the ECS Service and ALB modules later in the deployment workflow.

Task 1.8: ECS Module

This section explains each Terraform configuration file located inside the **ECS module** (modules/ecs).

main.tf

This module creates a complete **ECS EC2 cluster** environment, including compute capacity, IAM roles, launch template, Auto Scaling Group, capacity provider, and ECS service.

AMI Lookup

- Fetches the latest **ECS-Optimized Amazon Linux 2023 AMI**.
- Ensures new EC2 instances always run the latest ECS-optimized image.

IAM Role and Instance Profile (For EC2 Instances)

- The EC2 IAM role is assumed by the instances running in the cluster.
- Attached policies allow:
 - ECS agent communication (AmazonEC2ContainerServiceforEC2Role)
 - SSM access for remote management (AmazonSSMManagedInstanceCore)
- The instance profile binds the IAM role to EC2 instances launched via the ASG.

Launch Template

Defines the configuration for EC2 instances inside the cluster:

- Uses the ECS-optimized AMI.
- Applies the instance profile.
- Assigns the security group passed into the module.
- Injects **ECS_CLUSTER=cluster_name** into userdata so the instance automatically joins the ECS Cluster.

Auto Scaling Group

- Launches and maintains **2 EC2 instances** across private subnets.
- Protects instances from scale-in to avoid accidental task disruption.
- Tags instances with AmazonECSManaged = true, enabling ECS-managed instance lifecycle.

ECS Cluster

- Creates the cluster with **Container Insights (enhanced)** enabled for monitoring.
- Enables ECS Exec for secure per-container shell access.

Capacity Provider

- Connects the Auto Scaling Group to ECS via a capacity provider.
- Enables **managed scaling**, allowing ECS to automatically adjust EC2 capacity based on running tasks.
- The cluster is configured to use this provider by default.

ECS Service

Runs your application tasks on EC2 capacity:

- Uses the task definition ARN passed from the Task Definition module.
- Maintains **2 running tasks** with rolling deployments.
- Registers the container with the ALB target group on port **5000**.
- Supports ECS Exec for troubleshooting.

variables.tf

This file defines all required inputs:

- Naming inputs: IAM role, instance profile, cluster name, service name.
- Settings for compute: instance type, private subnets, security group.
- References from other modules: task definition ARN, container name, target group ARN.

These variables keep the ECS module reusable and environment-agnostic.

outputs.tf

- **ecs_cluster_name** – Allows other modules or pipelines to reference the ECS cluster.
- **ecs_service_name** – Useful for deployment automation and ECS Exec commands.

Task 1.9: ALB Module

This section explains each Terraform configuration file located inside the **ALB module** (modules/alb).

main.tf

This module provisions an **Application Load Balancer (ALB)** that distributes traffic to your ECS service.

Application Load Balancer

- Creates an internet-facing ALB using the name in alb_name.
- Placed in **public subnets** so external users can reach the application.
- Uses the security group passed into the module, allowing controlled inbound HTTP traffic.
- Deletion protection is disabled so Terraform can recreate the ALB when needed.

Target Group

- Creates a target group named with tg_name.
- Uses **HTTP** on the port specified by target_group_port.
- target_type is configurable (e.g., instance for ECS EC2 mode, ip for Fargate).
- Associated with the provided VPC.

This is where ECS tasks register as traffic targets.

Listener

- Creates an HTTP listener on listener_port (typically port 80).
- Forwards all incoming requests to the target group.
- Acts as the entry point for all traffic to the web application.

variables.tf

Defines all required ALB inputs:

- Networking inputs: **vpc_id**, public subnets, ALB security group.
- ALB settings: name, load balancer type (usually application).
- Target group and listener settings: ports, target type, target group name.

These keep the module flexible for multiple environments and architectures.

outputs.tf

- **alb_arn** – used by monitoring or IAM policies.
- **target_group_arn** – required by the ECS Service module to register tasks.
- **alb_dns_name** – the public URL users use to access the application.

Task 1.10: CodeBuild Module

This section explains each Terraform configuration file located inside the **CodeBuild module** (modules/codebuild).

main.tf

This module creates the **CodeBuild project** responsible for building the Docker image, pushing it to ECR, and sending build artifacts to CodePipeline.

IAM Role for CodeBuild

- The IAM role (codebuild_role) is assumed by CodeBuild.
- Allows the project to access logs, ECR, S3, VPC resources, and networking APIs.

- Each inline policy provides specific permissions:

CloudWatch Logs Policy

- Allows creating log groupsstreams and writing logs from CodeBuild.

S3 Policy

- Allows retrieving and uploading files used during builds (mainly for CodePipeline integration).

ECR Policy

- Enables CodeBuild to authenticate to ECR, upload image layers, and push built Docker images.

VPC Policy

- Required for CodeBuild to run inside private subnets (network interface creation, VPC lookups, etc.).
- Restricts network interface permissions to the provided private subnets.

CodeBuild Project

- Defines the build environment and links it to CodePipeline.
- Uses **amazonlinux-x86_64-standard:5.0**, with Docker enabled (`privileged_mode = true`) so it can build images.
- Environment variables supply:
 - AWS Account ID
 - Region
 - ECR repository name
- Sends logs to CloudWatch in the log group defined via variable `codebuild_logs`.

VPC Configuration

- CodeBuild runs entirely inside the **private subnets**, using the provided security group.
- Ensures builds happen securely without exposing the build environment to the public internet.

variables.tf

Defines all necessary inputs:

- Networking: VPC ID, private subnets, private security group.
- Build settings: IAM role name, project name, account ID, region, ECR repo name, and log group.
- These parameters keep the module flexible for any environment.

outputs.tf

- **codebuild_project_arn** – useful for IAM policies or pipeline integrations.

- **codebuild_project_name** – used by CodePipeline when connecting pipeline stages.

Task 1.11: CodePipeline Module

This section explains each Terraform configuration file located inside the **CodePipeline module** (modules/codepipeline).

main.tf

This module provisions a **fully managed CI/CD pipeline** using AWS CodePipeline, integrated with GitHub (via CodeStar Connections), CodeBuild, and ECS.

S3 Bucket for Artifacts

- Creates an S3 bucket (`bucket_name`) to store pipeline artifacts.
- Public access is fully blocked to ensure security.
- `force_destroy = true` allows the bucket to be deleted by Terraform even if it contains artifacts.

CodeStar Connection

- Establishes a connection to a GitHub repository (`github_connection`) using the specified provider type.
- Enables CodePipeline to pull source code automatically from GitHub.

IAM Role and Policies

- Creates an IAM role for CodePipeline (`codepipeline_role_name`).
- Policies grant permissions to:
 - Access the S3 artifact bucket.
 - Use the CodeStar GitHub connection.
 - Trigger CodeBuild projects.
 - Interact with ECS (describe/update services and register task definitions).
- Ensures the pipeline can fully manage CI/CD flow without external IAM dependencies.

CodePipeline

Defines a three-stage pipeline:

1. Source Stage

- Pulls code from GitHub via the CodeStar connection.
- Detects changes automatically on the main branch.
- Outputs SourceArtifact for the next stage.

2. Build Stage

- Uses the CodeBuild project to build the Docker image.
- Takes SourceArtifact as input and outputs BuildArtifact.
- Ensures images are built and pushed to ECR.

3. Deploy Stage

- Deploys the built image to ECS.
- Uses the cluster and service names provided.
- Reads imagedefinitions.json from the build artifact to update the ECS service.

variables.tf

- Inputs are fully parameterized to integrate with other modules:
 - **CodeBuild**: project ARN and name.
 - **ECS**: cluster name and service name.
 - **Pipeline**: S3 bucket, CodeStar connection, provider type, pipeline name, GitHub repo URL.
- Keeps the module reusable across multiple environments.

Task 1.12: Execute Terraform Commands

This task deploys the entire Flask application environment using Terraform. Ensure you are in the root Terraform directory.

1. **terraform init**

Initializes providers and backend (S3).

2. **terraform validate**

Checks syntax and logic for errors.

3. **terraform plan**

Shows planned resource creation and updates (52 to add).

```
Plan: 52 to add, 0 to change, 0 to destroy.
```

```
Changes to Outputs:
```

```
+ alb_dns_name = (known after apply)
```

4. **terraform apply --auto-approve**

Creates all resources, including:

- VPC, public/private subnets, IGW, NAT gateways
- Security groups (ALB-SG, EC2-ECS-SG)
- ECR repository (flask-app)
- ECS Cluster and Service

- ALB and target group
- CodeBuild and CodePipeline

```
Apply complete! Resources: 52 added, 0 changed, 0 destroyed.

Outputs:

alb_dns_name = "ecs-ec2-alb-56126343.us-east-1.elb.amazonaws.com"
○ PS D:\Cloudelligent\Task-8\terraform>
```

Task 1.13: Validate Infrastructure in AWS Console

After running `terraform apply`, verify that Terraform successfully provisioned all AWS resources required for the Flask application deployment pipeline. This section walks through each area of the infrastructure and what you should expect to see.

VPC & Networking

1. Verify VPC

- Go to AWS Console → **VPC** → **Your VPCs**
- Check that the VPC created by Terraform exists.

Expected configuration:

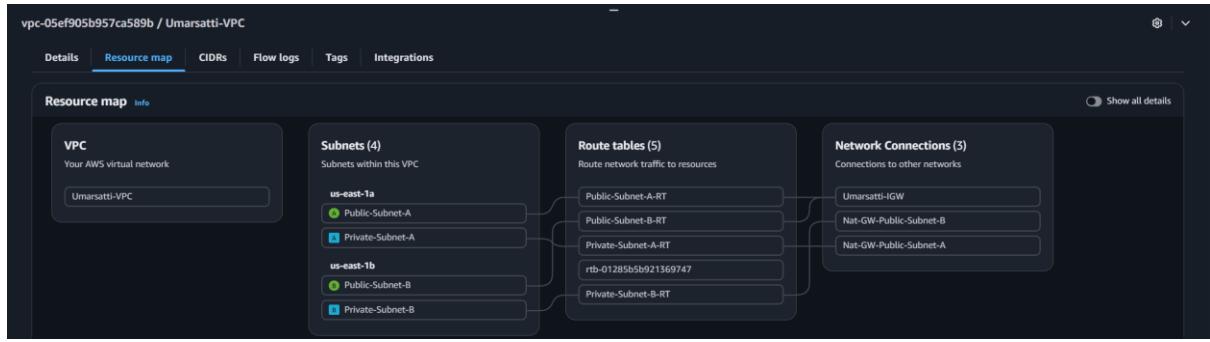
- Correct IPv4 CIDR block
- **DNS hostnames enabled** (required for ALB & ECS)

The screenshot shows the AWS VPC dashboard with the 'Your VPCs' table. The table has columns: Name, VPC ID, State, Block Public Access, IPv4 CIDR, IPv6 CIDR, and DHCP option set. One row is selected: 'Umarsatti-VPC' (vpc-05ef905b957ca589b), which is 'Available', has 'Off' for Block Public Access, '192.168.0.0/16' for IPv4 CIDR, and 'dopt-088ee356fa06392dc' for DHCP option set. Below the table, the details for 'vpc-05ef905b957ca589b / Umarsatti-VPC' are shown, including the VPC ID, State (Available), DNS resolution (Enabled), Main network ACL (acl-0277875b2afee376), IP6 CIDR (Network border group), Encryption control ID, State (Available), Tenancy (default), Default VPC (No), Network Address Usage metrics (Disabled), and Encryption control mode (Disabled). The 'Block Public Access' is Off, and the 'DHCP option set' is dopt-088ee356fa06392dc. The 'IPv4 CIDR' is 192.168.0.0/16, and the 'Route 53 Resolver DNS Firewall rule groups' is empty. The 'DNS hostnames' is Enabled, and the 'Main route table' is rtb-01285b5b921369747. The 'IPv6 pool' is empty, and the 'Owner ID' is 730335208305.

Name	VPC ID	State	Block Public Access	IPv4 CIDR	IPv6 CIDR	DHCP option set
Umarsatti-VPC	vpc-05ef905b957ca589b	Available	Off	192.168.0.0/16	-	dopt-088ee356fa06392dc

Details

VPC ID vpc-05ef905b957ca589b	State Available	Block Public Access Off	DNS hostnames Enabled
DNS resolution Enabled	Tenancy default	DHCP option set dopt-088ee356fa06392dc	Main route table rtb-01285b5b921369747
Main network ACL acl-0277875b2afee376	Default VPC No	IPv4 CIDR 192.168.0.0/16	IPv6 pool -
IP6 CIDR (Network border group) -	Network Address Usage metrics Disabled	Route 53 Resolver DNS Firewall rule groups -	Owner ID 730335208305
Encryption control ID -	Encryption control mode -		



2. Verify Subnets

Go to AWS Console → VPC → Subnets

Public Subnets

- 2 public subnets (Public-Subnet-A and Public-Subnet-B)
- Used for **Application Load Balancer**

Private Subnets

- 2 private subnets (Private-Subnet-A and Private-Subnet-B)
- Used by **ECS EC2 instances**
- Used by **CodeBuild ENIs** (if pipeline uses VPC mode)

Subnets (4) Info							
Find subnets by attribute or tag <input type="text"/> Actions Create subnet							
VPC dashboard	AWS Global View						
Filter by VPC <input type="button" value="VPC"/>							Last updated 1 minute ago
Virtual private cloud	Your VPCs Subnets Route tables Internet gateways Egress-only internet gateways Carrier gateways						
Name	Subnet ID	State	VPC	Block Publi...	IPv4 CIDR		
Public-Subnet-B	subnet-05453c11a1df4dc28	Available	vpc-05ef905b957ca589b Umarsatti-VPC	Off	192.168.20.0/24		
Private-Subnet-A	subnet-09a9b4a505946143b	Available	vpc-05ef905b957ca589b Umarsatti-VPC	Off	192.168.30.0/24		
Private-Subnet-B	subnet-08d5af6d6ec53b382	Available	vpc-05ef905b957ca589b Umarsatti-VPC	Off	192.168.40.0/24		
Public-Subnet-A	subnet-03333ea5039b63f117	Available	vpc-05ef905b957ca589b Umarsatti-VPC	Off	192.168.10.0/24		

3. Internet Gateway & NAT Gateway

Internet Gateway

- Should exist and be **attached to the VPC**

NAT Gateway

- Should exist (depending on your architecture)
- Located in a public subnet
- Status: **Available**

Allows CodeBuild to reach ECR / S3 from private subnets.

The image shows two screenshots of the AWS VPC console.

Internet Gateways:

- Left Panel:** Shows the VPC dashboard with a sidebar for Virtual private cloud, including Internet gateways, Subnets, Route tables, and other VPC components.
- Right Panel:** Shows the "Internet gateways (1/1) Info" page. A search bar filters by VPC ID: `vpc-05ef905b957ca589b`. A table lists one gateway:

Name	Internet gateway ID	State	VPC ID	Owner
Umarsatti-IGW	igw-0737911105fb8b8bc	Attached	vpc-05ef905b957ca589b Umarsatti-VPC	730355208305

NAT Gateways:

- Left Panel:** Shows the VPC dashboard with a sidebar for Virtual private cloud, including NAT gateways, Subnets, Route tables, and other VPC components.
- Right Panel:** Shows the "NAT gateways (2) Info" page. A search bar filters by VPC ID: `vpc-05ef905b957ca589b`. A table lists two gateways:

Name	NAT gateway ID	Connectivity...	State	Primary public I...	Primary private ...	Primary network interfa...
Nat-GW-Public-Subn...	nat-010dedaa9dd4a656	Public	Available	100.30.130.200	192.168.20.21	eni-0792bbb9b4110f9e9
Nat-GW-Public-Subn...	nat-074ddb5f6f2ad631f	Public	Available	34.237.103.52	192.168.10.81	eni-0f56212751f52e104

4. Route Tables

Go to AWS Console → VPC → Route Tables

The image shows the AWS VPC Route Tables management page.

Route tables (5) Info:

- Left Panel:** Shows the VPC dashboard with a sidebar for Virtual private cloud, including Route tables, Internet gateways, Subnets, Route tables, and other VPC components.
- Right Panel:** Shows the "Route tables (5) Info" page. A search bar filters by VPC ID: `vpc-05ef905b957ca589b`. A table lists five route tables:

Name	Route table ID	Explicit subnet associations	Edge associati...	Main	VPC
Public-Subnet-A-RT	rtb-0b18bac526f801683	subnet-0333ea5039b65f17 / Public-Subnet-A	-	No	vpc-05ef905b957ca589b
Public-Subnet-B-RT	rtb-0c90b238856203380	subnet-05453c11a1df4dc28 / Public-Subnet-B	-	No	vpc-05ef905b957ca589b
Private-Subnet-A-RT	rtb-08379b8911f98d1b1	subnet-09a9b4a505946143b / Private-Subnet-A	-	No	vpc-05ef905b957ca589b
-	rtb-01285b5b921369747	-	-	Yes	vpc-05ef905b957ca589b
Private-Subnet-B-RT	rtb-0a2151901db4d2a83	subnet-08d56f6d6ec53b382 / Private-Subnet-B	-	No	vpc-05ef905b957ca589b

Public Route Table

- Associated with public subnets
- Has route: `0.0.0.0/0 → Internet Gateway`

The image shows the details of the Public-Subnet-A-RT route table.

rtb-0b18bac526f801683 / Public-Subnet-A-RT:

- Left Panel:** Shows the VPC dashboard with a sidebar for Virtual private cloud, including Route tables, Internet gateways, Subnets, Route tables, and other VPC components.
- Right Panel:** Shows the "Routes (2)" section of the route table. A search bar filters routes. A table lists two routes:

Destination	Target	Status	Propagated	Route Origin
0.0.0.0/0	igw-0737911105fb8b8bc	Active	No	Create Route
192.168.0.0/16	local	Active	No	Create Route Table

rtb-0c90b238856203380 / Public-Subnet-B-RT					
Details	Routes	Subnet associations	Edge associations	Route propagation	Tags
Routes (2)					
<input type="button" value="Filter routes"/> Both ▾ Edit routes					
Destination	Target	Status	Propagated	Route Origin	
0.0.0.0/0	igw-0737911105fb8b8bc	Active	No	Create Route	
192.168.0.0/16	local	Active	No	Create Route Table	

Private Route Table(s)

- Associated with private subnets
- Has route: 0.0.0.0/0 → NAT Gateway

rtb-08379b8911f98d1b1 / Private-Subnet-A-RT																	
Details	Routes	Subnet associations	Edge associations	Route propagation	Tags												
Details <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 25%;">Route table ID</td> <td>rtb-08379b8911f98d1b1</td> <td style="width: 25%;">Main</td> <td><input checked="" type="checkbox"/> No</td> <td style="width: 25%;">Explicit subnet associations</td> <td>subnet-09a9b4a505946143b / Private-Subnet-A</td> </tr> <tr> <td>VPC</td> <td>vpc-05ef905b957ca589b Umarsatti-VPC</td> <td>Owner ID</td> <td>730335208305</td> <td>Edge associations</td> <td>-</td> </tr> </table>						Route table ID	rtb-08379b8911f98d1b1	Main	<input checked="" type="checkbox"/> No	Explicit subnet associations	subnet-09a9b4a505946143b / Private-Subnet-A	VPC	vpc-05ef905b957ca589b Umarsatti-VPC	Owner ID	730335208305	Edge associations	-
Route table ID	rtb-08379b8911f98d1b1	Main	<input checked="" type="checkbox"/> No	Explicit subnet associations	subnet-09a9b4a505946143b / Private-Subnet-A												
VPC	vpc-05ef905b957ca589b Umarsatti-VPC	Owner ID	730335208305	Edge associations	-												

rtb-0a2151901db4d2a83 / Private-Subnet-B-RT																								
Details	Routes	Subnet associations	Edge associations	Route propagation	Tags																			
Routes (2)																								
<input type="button" value="Filter routes"/> Both ▾ Edit routes	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th>Destination</th> <th>Target</th> <th>Status</th> <th>Propagated</th> <th>Route Origin</th> </tr> <tr> <td>0.0.0.0/0</td> <td>nat-010dedaaaf9dd4a656</td> <td>Active</td> <td>No</td> <td>Create Route</td> </tr> </table>	Destination	Target	Status	Propagated	Route Origin	0.0.0.0/0	nat-010dedaaaf9dd4a656	Active	No	Create Route	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th>Destination</th> <th>Target</th> <th>Status</th> <th>Propagated</th> <th>Route Origin</th> </tr> <tr> <td>192.168.0.0/16</td> <td>local</td> <td>Active</td> <td>No</td> <td>Create Route Table</td> </tr> </table>	Destination	Target	Status	Propagated	Route Origin	192.168.0.0/16	local	Active	No	Create Route Table		
Destination	Target	Status	Propagated	Route Origin																				
0.0.0.0/0	nat-010dedaaaf9dd4a656	Active	No	Create Route																				
Destination	Target	Status	Propagated	Route Origin																				
192.168.0.0/16	local	Active	No	Create Route Table																				

Security Groups

Go to AWS Console → VPC → Security Groups

Security Groups (3) Info					
<input type="button" value="Actions ▾"/> Export security groups to CSV Create security group					
<input type="text" value="VPC ID = vpc-05ef905b957ca589b"/> Clear filters					
Name	Security group ID	Security group name	VPC ID	Description	
<input type="checkbox"/> EC2-ECS-SG	sg-01c283fbcd8beba23	EC2-ECS-SG	vpc-05ef905b957ca589b	Allows Custom TCP traffic on port 500...	
<input type="checkbox"/> -	sg-0fbed0b85f022e8da	default	vpc-05ef905b957ca589b	default VPC security group	
<input type="checkbox"/> ALB-SG	sg-0689df1a67e0295d2	ALB-SG	vpc-05ef905b957ca589b	Allows HTTP traffic from the internet	

ALB Security Group (ALB-SG)

Name	IP version	Type	Protocol	Port range	Source
sgr-0583c29fb28db4e0f	IPv4	HTTP	TCP	80	0.0.0.0/0

ECS EC2 Security Group

- Internet → ALB → ECS tasks (Flask on port 5000)

Name	IP version	Type	Protocol	Port range	Source
sgr-06a9d0fe9c1740da3	-	Custom TCP	TCP	5000	sg-0689df1a67e9295d2 / ALB-SG
sgr-01bb43cfabaaf89e5	-	Custom TCP	TCP	32768 - 65535	sg-0689df1a67e9295d2 / ALB-SG

ECR Repository

Go to AWS Console → ECR → Private Repositories

Verify:

- Flask repository exists**
- Repository URI looks like:
<Account_ID>.dkr.ecr.<region>.amazonaws.com/flask-app

Initially the image list may be empty until CodePipeline runs.

Repository name	URI	Created at	Tag immutability	Encryption type
python-ecr	750335208305.dkr.ecr.us-east-1.amazonaws.com/python-ecr	December 04, 2025, 04:03:18 (UTC+05)	Mutable	AES-256

ECS: Cluster, Service, and Tasks

Go to AWS Console → ECS → Clusters

Verify:

ECS Cluster

- Cluster created by Terraform exists (flask-ecs-cluster)
- Launch type: **EC2**

Last updated December 4, 2025, 04:16 (UTC+5:00)

Create cluster

Clusters (1) Info

Cluster flask-ecs-cluster Services Tasks Container instances CloudWatch monitoring Capacity provider strat

No tasks running 2 EC2 Container Insights with enhanced observability ASG

ECS Service

- Service exists inside the cluster
- Status: **Active**

Last updated December 4, 2025, 04:16 (UTC+5:00)

Actions Create with Express Mode

flask-ecs-cluster ASG

Cluster overview

ARN arn:aws:ecs:us-east-1:730335208305:cluster/flask-ecs-cluster **Status** Active

CloudWatch monitoring Container Insights with enhanced observability View in CloudWatch

Registered container instances 2

Services

Draining	Active	Pending	Running
-	1	-	-

Tasks

Services (1) Info

Filter services by value

Service name	ARN	Status	Scheduling strategy	Lau...	Task definiti...	Deployments and tasks
flask-ecs-service	arn:aws:ecs:us-e...	Active	REPLICA	EC2	python-flask-t...	0/2 Tasks

ECS Tasks

Before the first deployment, tasks may show:

- **Stopped** (because no Docker image exists yet)

After CodePipeline deploys (later):

- **Running tasks: 2/2** (or your desired count)

Last updated December 4, 2025, 04:17 (UTC+5:00)

Capacity providers (1) Info

Filter capacity providers by property or value

Capacity provider	Scaling type	Provisioning model	Update status	Update status reason
flask-ecs-cluster-capacity-provider	EC2 Auto Scaling	-	-	-

You can also use FARGATE and FARGATE_SPOT capacity providers to launch tasks on AWS Fargate.

Last updated December 4, 2025, 04:17 (UTC+5:00)

Container instances (2) Info

Register external instances Actions

Filter container instances by property or value

Container instance	Instance type	Status	Statu...	Type	Instance ID	Capac...	Availability zo...	Running
896bc75617d0448a923b6cae48d935...	t3.micro	Active	-	EC2	i-01f7513b1b2f...	flask-ecs-...	us-east-1a	0
999cc9782e69480dab2c149cb87d0f76	t3.micro	Active	-	EC2	i-03c60dc721d8...	flask-ecs-...	us-east-1b	0

Application Load Balancer

Go to AWS Console → EC2 → Load Balancers

Verify:

ALB

- ALB exists
- Assigned to both public subnets
- Uses **ALB-SG**

Listener

- HTTP on port **80**
- Default action: forward → Flask target group

The screenshot shows the AWS EC2 Load Balancers console. In the top navigation bar, 'EC2' is selected, followed by 'Load balancers'. Below the navigation, there's a search bar and filters for 'Name', 'State', 'Type', 'VPC ID', 'Availability Zones', and 'DNS name'. A single load balancer entry is listed: 'ecs-ec2-alb' (Active, application type, VPC ID: vpc-05ef905b957ca589b, 2 Availability Zones: us-east-1a, us-east-1b), with a DNS name of 'ecs-ec2-alb-56126343.us-east-1.elb.amazonaws.com'. The main content area is titled 'Load balancer: ecs-ec2-alb' and contains tabs for 'Details', 'Listeners and rules', 'Network mapping', 'Resource map', 'Security', 'Monitoring', 'Integrations', 'Attributes', 'Capacity', and 'Tags'. The 'Details' tab is selected, displaying information such as Load balancer type (Application), Status (Active), Hosted zone (Z355XDTRQ7X7K), VPC (vpc-05ef905b957ca589b), Availability Zones (us-east-1a, us-east-1b, us-east-1c), and DNS name (ecs-ec2-alb-56126343.us-east-1.elb.amazonaws.com).

Target Group

Go to AWS Console → EC2 → Target Groups

Verify:

Target Group

- Target type: **instance**
- Protocol: **HTTP**
- Port: **5000**

Health Checks

Before first image push:

- **0 healthy targets**

Note: After CodePipeline deploys and tasks run:

- **Healthy targets = number of running ECS tasks**

CodePipeline & CodeBuild

1. CodeBuild

AWS Console → **CodeBuild** → Build Projects

Verify:

- Build project exists
- VPC configuration is set (if enabled)
- Buildspec includes steps for:
 - building Docker image
 - logging into ECR
 - pushing image
 - returning image tag

2. CodePipeline

AWS Console → **CodePipeline**

Verify:

- Pipeline exists with 3 stages:
 1. **Source (GitHub)**
 2. **Build (CodeBuild)**
 3. **Deploy (ECS)**

The screenshot shows the AWS CodePipeline Pipelines page. On the left, there's a sidebar with 'Developer Tools' and 'CodePipeline'. The main area has a table titled 'Pipelines' with one row. The row for 'codePipeline-ecs-ec2-project' shows 'Failed' under 'Latest execution status' and '16 minutes ago' under 'Latest execution started'. There are buttons for 'View history', 'Release change', 'Delete pipeline', and 'Create pipeline'.

Execute CodePipeline

Step 1: Update GitHub Connection

Go to AWS → CodePipeline → Connections → GitHub Connection → Update Connection

Note: This is a mandatory first-time step

The screenshot shows the AWS Connections page. It lists a connection named 'aws-github-connection' with a provider of 'GitHub'. There are two entries for this connection. The top entry is labeled 'Pending' and has an ARN: arn:aws:codestar-connections:us-east-1:730335208305:connection/b28d9d8d-b7ce-4439-abba-e65cdf712363. The bottom entry is labeled 'Available' and has an ARN: arn:aws:codestar-connections:us-east-1:730335208305:connection/b28d9d8d-b7ce-4439-abba-e65cdf712363.

Step 2: Run CodePipeline

- Run pipeline manually the first time
- Wait for the pipeline stages to complete successfully.

The screenshot shows the AWS CodePipeline Pipeline page for 'codePipeline-ecs-ec2-project'. The pipeline consists of three stages: 'Source', 'Build', and 'Deploy'. Each stage has a green circular icon with a checkmark, indicating successful completion. The 'Source' stage shows 'All actions succeeded.' and a 'GitHub' action from 5 minutes ago. The 'Build' stage shows 'All actions succeeded.' and an 'AWS CodeBuild' action from 3 minutes ago. The 'Deploy' stage shows 'All actions succeeded.' and an 'Amazon ECS' action from 1 minute ago. There are tabs for 'Pipeline', 'Executions', 'Triggers', 'Settings', 'Tags', and 'Stage'.

Step 3: Check Target Group Health

Go to ALB → Target Groups → Health check

Target group health should transition from **unhealthy** to **healthy**

The screenshot shows the AWS CloudWatch Metrics console with the 'Target groups' page. It displays a single target group named 'ecs-ec2-target-group'. The target group details show the following configuration:

- Target type:** Instance
- Protocol:** Port
- Port:** 5000
- Protocol version:** HTTP1
- VPC:** vpc-05ef905b957ca589b

The target group status indicates 2 total targets, with 2 healthy and 0 unhealthy. There are 0 anomalous targets, 0 unused targets, 0 initial targets, and 0 draining targets.

Distribution of targets by Availability Zone (AZ): Select values in this table to see corresponding filters applied to the Registered targets table below.

The screenshot shows the 'Target group: ecs-ec2-target-group' details page. It lists two registered targets:

Instance ID	Name	Port	Zone	Health status	Admin... Override	Overri...	Launch... Override	Anomaly detect...
i-03c60dc721d826c06		32776	us-east-1b ...	Healthy	-	No override.	No overri...	Normal
i-01f751381b2feaf87		32773	us-east-1a ...	Healthy	-	No override.	No overri...	Normal

Step 4: Verify ECS Tasks Running

Go to ECS → Cluster

The screenshot shows the 'flask-ecs-cluster' cluster overview page. It displays the following information:

- ARN:** arn:aws:ecs:us-east-1:730335208305:cluster/flask-ecs-cluster
- Status:** Active
- CloudWatch monitoring:** Container Insights with enhanced observability
- Registered container instances:** 2

The 'Services' section shows one service named 'flask-ecs-service' with the following details:

Service name	ARN	Status	Scheduling strategy	Task definition	Deployments and tasks	Last de...
flask-ecs-service	arn:aws:ecs:us-east-1:730335208305:task-definition/python-flask-t...	Active	REPLICA	EC2	python-flask-t...	2/2 Tasks running

Then go to ECS → Cluster

Should show 2/2 running tasks:

flask-ecs-service [Info](#)

Last updated December 4, 2025, 04:26 (UTC+5:00) [Delete service](#) [Update service](#)

Service overview [Info](#)

Status [Active](#) Tasks (2 Desired) 0 Pending | 2 Running Task definition: revision python-flask-task-definition:11 Deployment status [Success](#)

Health and metrics [Tasks](#) Logs Deployments Events Configuration and networking Service auto scaling Event history Tags

Tasks (1/2) Last updated December 4, 2025, 04:27 (UTC+5:00) [Stop](#)

Task	Last status	Desired st...	Task definition	Health sta...	Created at	Started by
3d561c3a41d7404a859847f74f0d03a9	Running	Running	python-flask-task-definiti...	Unknown	3 minutes ago	ecs-svc/25483351234...
f43c9c3e3b7e4406a053f12d9346df5c	Running	Running	python-flask-task-definiti...	Unknown	2 minutes ago	ecs-svc/25483351234...

Step 5: Test Application

Retrieve ALB DNS from AWS Console → EC2 → Load Balancers → ALB → DNS name

Open in browser:

<http://<alb-dns-name>>

A simple flask application should be displayed as shown below.

Deploying an Application using AWS CodePipeline on ECS EC2 with Terraform

The infrastructure is deploy using Terraform

Task 1.14: Deploy Flask Application via CodePipeline

1. Navigate to **CodePipeline** in the AWS Console.
2. Locate the pipeline created for your Flask app (e.g., flask-app-pipeline).
3. Click **Release change** or **Start pipeline execution**.
 - o This triggers the pipeline:
 - **Source stage:** Pulls the code from your GitHub repository.
 - **Build stage:** CodeBuild builds the Docker image and pushes it to ECR.
 - **Deploy stage:** ECS service updates tasks with the new image.
4. Monitor each stage to ensure all steps succeed.
 - o **Success:** The Build stage shows the Docker image pushed to ECR.
 - o **Deploy stage:** ECS tasks are updated with the new Flask image.

Task 1.15: Confirm Flask Application is Accessible

1. Navigate to **EC2 → Load Balancers**.
2. Copy the **ALB DNS Name** assigned.
3. Open a browser and paste the DNS.
 - o You should see a simple Flask page.

Task 1.16: Clean Up

To destroy all resources, run the following command

- *terraform destroy --auto-approve*

This removes VPC, subnets, ECS, ALB, ECR, CodeBuild, and CodePipeline resources.

Troubleshooting

Issue 1: VPC Client Error – DHCP Options

Problem Description:

CodeBuild or ECS provisioning failed with the following error:

- *VPC_CLIENT_ERROR: Unexpected EC2 error: error while getting DHCP options for VPC*

Root Cause:

The IAM role for CodeBuild did not include the "ec2:DescribeDhcpOptions" permission. CodeBuild needs this to query VPC DHCP settings for network interface creation.

Solution:

Add "ec2:DescribeDhcpOptions" to the CodeBuild VPC policy in Terraform. After updating and applying, the provisioning succeeded.

Issue 2: VPC Client Error – UnauthorizedOperation

Problem Description:

CodeBuild execution failed with:

- *VPC_CLIENT_ERROR: Unexpected EC2 error: UnauthorizedOperation*

Root Cause:

CodeBuild did not have sufficient permissions to create or attach network interfaces in the private subnets. When running in a VPC, CodeBuild must provision ENIs (Elastic Network Interfaces) to access resources, and it can only do this if the IAM role allows it to create network interfaces **and** authorize them in the specific private subnets.

Solution:

Update the CodeBuild IAM role to allow the creation of network interfaces and explicitly permit CodeBuild to attach them to the private subnets. This ensures CodeBuild can launch build containers within the VPC and communicate with other resources securely.

Issue 3: ECS Permissions Error in CodePipeline

Problem Description:

CodePipeline failed during the Deploy stage with a message:

The provided role does not have sufficient permissions to access ECS

Root Cause:

CodePipeline was trying to update the ECS service, which requires passing the ECS task execution role. Without "iam:PassRole", CodePipeline cannot allow ECS to assume the role.

Solution:

Add "iam:PassRole" to the CodePipeline IAM role policy. After updating, ECS tasks updated successfully with the new Docker image.

Issue 4: GitHub Connection Requires Manual Update

Problem Description:

Pipeline execution failed at the Source stage. The AWS CodeStar GitHub connection showed **inactive** and required clicking **Update Connection** manually.

Root Cause:

CodeStar connections sometimes require token refresh to remain active; Terraform cannot automatically refresh GitHub OAuth tokens.

Solution:

Currently, manually click **Update Connection** in the AWS Developer Tools Console page to reactivate. Future improvements could involve storing GitHub tokens in **Secrets Manager** and automating token refresh, but this is not yet fully automated.