You are required to deploy a Flask backend and an Express frontend application using **AWS** and **Terraform**. Follow the steps below to deploy the applications in different configurations.

#### **Part 1: Deploy Both Flask and Express on a Single EC2 Instance**

1. **Objective**: Deploy both the Flask backend and the Express frontend on a **single EC2 instance** using Terraform.
   * Provision the EC2 instance with Terraform.
   * Configure the instance with:
     + A user data script or a configuration management tool (e.g., Ansible or Cloud-Init) to install dependencies (Python, Node.js).
     + Scripts to start the Flask backend and the Express frontend.
   * Ensure both applications are running on different ports (e.g., Flask on port 5000 and Express on port 3000).
2. **Expected Deliverables**:
   * Terraform configuration files (main.tf, variables.tf, etc.).
   * A working EC2 instance with Flask and Express running and accessible via the instance's public IP.

#### **Part 2: Deploy Flask and Express on Separate EC2 Instances**

1. **Objective**: Deploy the Flask backend and the Express frontend on **two separate EC2 instances** using Terraform.
   * Provision two EC2 instances using Terraform:
     + One for the Flask backend.
     + One for the Express frontend.
   * Configure security groups to:
     + Allow communication between the two instances.
     + Expose both applications to the internet on their respective ports.
2. **Configuration**:
   * Use Terraform to define networking resources, such as VPC, subnets, and security groups.
   * Use user data scripts to automate the installation and startup of both applications.
3. **Expected Deliverables**:
   * Terraform configuration files.
   * Two working EC2 instances: one running Flask and one running Express.
   * Security groups configured to allow proper communication and public access.

#### **Part 3: Deploy Flask and Express Using Docker and AWS Services**

1. **Objective**: Deploy Flask and Express as **Docker containers** using **AWS ECR**, **ECS**, and **VPC** with Terraform.
2. **Steps**:
   * **ECR**:
     + Use Terraform to create two Elastic Container Registry (ECR) repositories: one for the Flask backend and one for the Express frontend.
     + Build Docker images for both applications and push them to their respective ECR repositories.
   * **VPC**:
     + Use Terraform to create a VPC with subnets, route tables, and security groups.
   * **ECS**:
     + Use Terraform to set up an ECS cluster.
     + Create two ECS services:
       - One for the Flask backend.
       - One for the Express frontend.
     + Use ECS Fargate or EC2 launch type to deploy the containers.
   * **Load Balancer**:
     + Use Terraform to provision an Application Load Balancer (ALB).
     + Configure ALB listeners to route requests to the appropriate ECS service.
3. **Expected Deliverables**:
   * Terraform configuration files for ECR, ECS, VPC, and ALB setup.
   * Docker images pushed to ECR.
   * ECS services running and accessible via the ALB.

EXPLANATION:-

1. Install & configure tools:  
   * Terraform (v1.0+)
   * AWS CLI (configured with aws configure)
   * Git (for EC2 to clone repo) or make sure you can scp code
2. Prepare your app repo(s):  
   * frontend/ (Express) with package.json, index.js (listen:3000)
   * backend/ (Flask) with requirements.txt, app.py (listen:5000)
   * Add simple health endpoints /health.
   * Make repo public or ensure EC2 has access (or use S3 to transfer).
3. Terraform best practice here:  
   * Use variables.tf, outputs.tf, main.tf.
   * Use S3 backend + DynamoDB for state locking (examples below).

## **PART 1 — Both Flask & Express on a single EC2 instance**

### **Goal**

Provision a single EC2 instance with Terraform and use a user-data script to install Python & Node, clone the repo, and start both apps on ports 5000 (Flask) and 3000 (Express). Expose via Security Group on those ports.

### **Files (minimum):**

#### **variables.tf**

variable "aws\_region" { default = "us-east-1" }

variable "instance\_type" { default = "t3.micro" }

variable "ssh\_key\_name" { description = "Existing EC2 key pair name" }

variable "github\_repo" { description = "Git repo with frontend & backend", default = "https://github.com/YOUR\_USERNAME/YOUR\_REPO.git" }

variable "public\_key\_path" { description = "Path to public key (optional)" , default = "~/.ssh/id\_rsa.pub" }

#### **provider.tf (and backend)**

terraform {

required\_providers {

aws = { source = "hashicorp/aws" }

}

backend "s3" {

bucket = "YOUR\_TERRAFORM\_STATE\_BUCKET"

key = "project/part1/terraform.tfstate"

region = "us-east-1"

dynamodb\_table = "terraform-locks"

}

}

provider "aws" {

region = var.aws\_region

}

#### **main.tf**

resource "aws\_vpc" "main" {

cidr\_block = "10.0.0.0/16"

tags = { Name = "single-ec2-vpc" }

}

resource "aws\_subnet" "public" {

vpc\_id = aws\_vpc.main.id

cidr\_block = "10.0.1.0/24"

map\_public\_ip\_on\_launch = true

availability\_zone = data.aws\_availability\_zones.available.names[0]

}

data "aws\_availability\_zones" "available" {}

resource "aws\_internet\_gateway" "igw" {

vpc\_id = aws\_vpc.main.id

}

resource "aws\_route\_table" "public" {

vpc\_id = aws\_vpc.main.id

route { cidr\_block = "0.0.0.0/0"; gateway\_id = aws\_internet\_gateway.igw.id }

}

resource "aws\_route\_table\_association" "a" {

subnet\_id = aws\_subnet.public.id

route\_table\_id = aws\_route\_table.public.id

}

resource "aws\_security\_group" "ec2\_sg" {

name = "single-ec2-sg"

vpc\_id = aws\_vpc.main.id

ingress {

description = "SSH"

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"] # restrict to your IP for production

}

ingress {

description = "Express"

from\_port = 3000

to\_port = 3000

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

description = "Flask"

from\_port = 5000

to\_port = 5000

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

egress {

from\_port = 0; to\_port = 0; protocol = "-1"; cidr\_blocks = ["0.0.0.0/0"]

}

}

resource "aws\_instance" "app" {

ami = data.aws\_ami.ubuntu.id

instance\_type = var.instance\_type

subnet\_id = aws\_subnet.public.id

security\_groups = [aws\_security\_group.ec2\_sg.id]

key\_name = var.ssh\_key\_name

user\_data = file("${path.module}/user\_data\_single.sh")

tags = { Name = "single-ec2-app" }

}

data "aws\_ami" "ubuntu" {

most\_recent = true

owners = ["099720109477"] # Ubuntu

filter { name = "name"; values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-\*"] }

}

output "public\_ip" {

value = aws\_instance.app.public\_ip

}

#### **user\_data\_single.sh (make executable)**

This will run at boot: install Node, Python, git, clone repo, create venv, install dependencies, start apps with systemd.

#!/bin/bash

set -e

# variables

REPO\_URL="https://github.com/YOUR\_USERNAME/YOUR\_REPO.git"

APP\_DIR="/home/ubuntu/app"

FRONT\_DIR="$APP\_DIR/frontend"

BACK\_DIR="$APP\_DIR/backend"

# update & install

apt-get update

DEBIAN\_FRONTEND=noninteractive apt-get install -y git python3-venv python3-pip nodejs npm curl

# create user ubuntu if not exists (AMI likely has it)

cd /home/ubuntu

git clone $REPO\_URL app || (cd app && git pull)

# backend setup

cd $BACK\_DIR

python3 -m venv venv

source venv/bin/activate

pip install --upgrade pip

pip install -r requirements.txt

# create systemd service for backend

cat >/etc/systemd/system/backend.service <<'EOF'

[Unit]

Description=Flask backend

After=network.target

[Service]

User=ubuntu

WorkingDirectory=/home/ubuntu/app/backend

Environment="PATH=/home/ubuntu/app/backend/venv/bin"

ExecStart=/home/ubuntu/app/backend/venv/bin/gunicorn --bind 0.0.0.0:5000 app:app

Restart=always

[Install]

WantedBy=multi-user.target

EOF

systemctl daemon-reload

systemctl enable backend.service

systemctl start backend.service

# frontend setup

cd $FRONT\_DIR

npm install --production

# create systemd service for frontend

cat >/etc/systemd/system/frontend.service <<'EOF'

[Unit]

Description=Express frontend

After=network.target

[Service]

User=ubuntu

WorkingDirectory=/home/ubuntu/app/frontend

ExecStart=/usr/bin/node /home/ubuntu/app/frontend/index.js

Restart=always

[Install]

WantedBy=multi-user.target

EOF

systemctl daemon-reload

systemctl enable frontend.service

systemctl start frontend.service

# end of script

**Notes**:

* gunicorn should be added to backend requirements.txt. If not, you can run python app.py directly — but gunicorn is recommended.
* For production, secure SSH and lock down SG ingress.

### **How to run**

terraform init

terraform plan -var 'ssh\_key\_name=your-key' -var 'github\_repo=https://github.com/...'

terraform apply -var 'ssh\_key\_name=your-key' -auto-approve

After apply, terraform output public\_ip shows the EC2 public IP. Visit:

* http://PUBLIC\_IP:3000 → frontend
* http://PUBLIC\_IP:5000/health → backend health

## **PART 2 — Flask and Express on separate EC2 instances**

### **Goal**

Two EC2 instances: one runs Flask (port 5000), one runs Express (port 3000). Terraform will create VPC/subnet, two instances, and security groups that allow frontend ↔ backend communication and public access.

### **Files (minimum):**

You can reuse provider.tf and backend from Part1; change main.tf to provision two instances and SG rules.

#### **variables.tf**

Same as Part1 but maybe instance\_count not needed.

#### **main.tf (key excerpts)**

# (VPC, subnet, IGW, route table similar to Part1)

resource "aws\_security\_group" "frontend\_sg" {

name = "frontend-sg"

vpc\_id = aws\_vpc.main.id

ingress {

from\_port = 22; to\_port=22; protocol="tcp"; cidr\_blocks=["0.0.0.0/0"]

}

ingress {

from\_port = 3000; to\_port=3000; protocol="tcp"; cidr\_blocks=["0.0.0.0/0"]

}

egress { from\_port=0; to\_port=0; protocol="-1"; cidr\_blocks=["0.0.0.0/0"] }

}

resource "aws\_security\_group" "backend\_sg" {

name = "backend-sg"

vpc\_id = aws\_vpc.main.id

# allow only ssh from your ip or frontend SG for 5000

ingress {

from\_port = 22; to\_port = 22; protocol="tcp"; cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

description = "Allow frontend to access flask"

from\_port = 5000; to\_port = 5000; protocol="tcp";

security\_groups = [aws\_security\_group.frontend\_sg.id] # allow frontend SG

}

# Optionally allow public access as well

ingress {

from\_port = 5000; to\_port = 5000; protocol="tcp"; cidr\_blocks = ["0.0.0.0/0"]

}

egress { from\_port=0; to\_port=0; protocol="-1"; cidr\_blocks=["0.0.0.0/0"] }

}

# EC2-frontend

resource "aws\_instance" "frontend" {

ami = data.aws\_ami.ubuntu.id

instance\_type = var.instance\_type

subnet\_id = aws\_subnet.public.id

key\_name = var.ssh\_key\_name

vpc\_security\_group\_ids = [aws\_security\_group.frontend\_sg.id]

user\_data = file("${path.module}/user\_data\_frontend.sh")

tags = { Name = "express-frontend" }

}

# EC2-backend

resource "aws\_instance" "backend" {

ami = data.aws\_ami.ubuntu.id

instance\_type = var.instance\_type

subnet\_id = aws\_subnet.public.id

key\_name = var.ssh\_key\_name

vpc\_security\_group\_ids = [aws\_security\_group.backend\_sg.id]

user\_data = file("${path.module}/user\_data\_backend.sh")

tags = { Name = "flask-backend" }

}

output "frontend\_public\_ip" { value = aws\_instance.frontend.public\_ip }

output "backend\_public\_ip" { value = aws\_instance.backend.public\_ip }

#### **user\_data\_frontend.sh**

Same install steps but set BACKEND\_URL env to backend private IP or public IP. Using private IP is best if both in same VPC; but since frontend will run in its own instance and browser is client-side, you often need backend public access or use the frontend server as proxy.

Simple approach: expose backend publicly on 5000 (or place in same VPC and configure Nginx reverse proxy). Example user data:

#!/bin/bash

set -e

REPO="https://github.com/YOUR\_USERNAME/YOUR\_REPO.git"

cd /home/ubuntu

apt-get update

apt-get install -y git nodejs npm

git clone $REPO app || (cd app && git pull)

cd app/frontend

npm install --production

# set BACKEND\_URL to backend public IP (Terraform can template it using templatefile if desired)

# For demo, use backend public ip placeholder replaced by terraform template:

BACKEND\_URL="http://${backend\_ip}:5000"

sed -i "s|BACKEND\_URL\_PLACEHOLDER|$BACKEND\_URL|g" public/index.html || true

cat >/etc/systemd/system/frontend.service <<'EOF'

[Unit]

Description=Express frontend

After=network.target

[Service]

User=ubuntu

WorkingDirectory=/home/ubuntu/app/frontend

ExecStart=/usr/bin/node /home/ubuntu/app/frontend/index.js

Restart=always

[Install]

WantedBy=multi-user.target

EOF

systemctl daemon-reload

systemctl enable frontend

systemctl start frontend

Use Terraform templatefile() to inject backend IP into user\_data\_frontend.sh or generate the file via the template\_file data source.

#### **user\_data\_backend.sh**

Same as Part1 backend script to set up Python, venv, gunicorn, systemd.

### **How to run**

terraform init

terraform apply -var 'ssh\_key\_name=your-key' -auto-approve

Outputs show public IPs for both instances. Test:

* http://FRONTEND\_PUBLIC\_IP:3000 → frontend UI (should call backend)
* http://BACKEND\_PUBLIC\_IP:5000/health → backend

**Notes & best practices**

* Prefer to restrict backend SG to allow only frontend SG (security group referencing).
* Use Terraform templatefile() to render user\_data\_frontend.sh substituting backend IP: e.g.,

user\_data = templatefile("${path.module}/user\_data\_frontend.tpl", { backend\_ip = aws\_instance.backend.private\_ip })

* For production, put backend in private subnet and use NAT/publicly accessible ALB in front.

## **PART 3 — Dockerized apps using ECR, ECS (Fargate), VPC, ALB managed by Terraform**

### **Goal**

* Build Docker images for Express and Flask, push to ECR.
* Use Terraform to create VPC, ECS cluster, Task Definitions, Services (Fargate), and an ALB to route traffic.

This part is longer — I’ll give the essential Terraform resources + shell commands to build/push images.

### **A — Terraform structure (high-level)**

* provider.tf (AWS provider + S3 state backend)
* vpc.tf (VPC, subnets, IGW, route tables)
* ecr.tf (create two ECR repos)
* iam.tf (ecs task execution role)
* ecs.tf (cluster, task defs, services)
* alb.tf (ALB, target groups, listeners)
* variables.tf + outputs.tf

I'll provide minimal examples for each.

#### **ecr.tf**

resource "aws\_ecr\_repository" "frontend" {

name = "cloud-frontend"

image\_tag\_mutability = "MUTABLE"

}

resource "aws\_ecr\_repository" "backend" {

name = "cloud-backend"

image\_tag\_mutability = "MUTABLE"

}

#### **iam.tf**

resource "aws\_iam\_role" "ecs\_task\_execution" {

name = "ecsTaskExecutionRole"

assume\_role\_policy = data.aws\_iam\_policy\_document.ecs\_task\_assume\_role.json

}

data "aws\_iam\_policy\_document" "ecs\_task\_assume\_role" {

statement {

actions = ["sts:AssumeRole"]

principals { type = "Service" , identifiers = ["ecs-tasks.amazonaws.com"] }

}

}

resource "aws\_iam\_role\_policy\_attachment" "ecs\_exec\_attach" {

role = aws\_iam\_role.ecs\_task\_execution.name

policy\_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"

}

#### **vpc.tf (simplified)**

module "vpc" {

source = "terraform-aws-modules/vpc/aws"

name = "ecs-vpc"

cidr = "10.0.0.0/16"

azs = slice(data.aws\_availability\_zones.available.names,0,2)

public\_subnets = ["10.0.1.0/24","10.0.2.0/24"]

private\_subnets = ["10.0.3.0/24","10.0.4.0/24"]

enable\_nat\_gateway = true

}

(Use terraform-aws-modules for convenience — add required provider and module source.)

#### **alb.tf**

resource "aws\_lb" "app\_alb" {

name = "app-alb"

internal = false

load\_balancer\_type = "application"

subnets = module.vpc.public\_subnets

security\_groups = [aws\_security\_group.alb\_sg.id]

}

resource "aws\_lb\_target\_group" "frontend\_tg" {

name = "frontend-tg"

port = 3000

protocol = "HTTP"

vpc\_id = module.vpc.vpc\_id

target\_type = "ip"

health\_check { path = "/health" ; matcher = "200" ; interval = 30 }

}

resource "aws\_lb\_target\_group" "backend\_tg" {

name = "backend-tg"

port = 5000

protocol = "HTTP"

vpc\_id = module.vpc.vpc\_id

target\_type = "ip"

health\_check { path = "/health" ; matcher = "200" ; interval = 30 }

}

resource "aws\_lb\_listener" "http" {

load\_balancer\_arn = aws\_lb.app\_alb.arn

port = 80

default\_action { type = "forward"; target\_group\_arn = aws\_lb\_target\_group.frontend\_tg.arn }

}

# Add rule for /submit or /api to forward to backend\_tg

resource "aws\_lb\_listener\_rule" "backend\_rule" {

listener\_arn = aws\_lb\_listener.http.arn

priority = 100

action { type = "forward"; target\_group\_arn = aws\_lb\_target\_group.backend\_tg.arn }

condition { field = "path-pattern"; values = ["/submit\*","/api/\*"] }

}

#### **ecs.tf (task definitions + services)**

resource "aws\_ecs\_cluster" "cluster" { name = "app-cluster" }

# frontend task definition

resource "aws\_ecs\_task\_definition" "frontend" {

family = "frontend-task"

network\_mode = "awsvpc"

requires\_compatibilities = ["FARGATE"]

cpu = "256"; memory = "512"

execution\_role\_arn = aws\_iam\_role.ecs\_task\_execution.arn

container\_definitions = jsonencode([

{

name = "frontend",

image = "${aws\_ecr\_repository.frontend.repository\_url}:latest",

portMappings = [{ containerPort = 3000, protocol = "tcp" }],

essential = true,

healthCheck = { command = ["CMD-SHELL","curl -f http://localhost:3000/health || exit 1"], interval = 30, timeout = 5, retries = 3 }

}

])

}

resource "aws\_ecs\_task\_definition" "backend" {

family = "backend-task"

network\_mode = "awsvpc"

requires\_compatibilities = ["FARGATE"]

cpu = "256"; memory = "512"

execution\_role\_arn = aws\_iam\_role.ecs\_task\_execution.arn

container\_definitions = jsonencode([

{

name = "backend",

image = "${aws\_ecr\_repository.backend.repository\_url}:latest",

portMappings = [{ containerPort = 5000, protocol = "tcp" }],

essential = true,

healthCheck = { command = ["CMD-SHELL","curl -f http://localhost:5000/health || exit 1"], interval = 30, timeout = 5, retries = 3 }

}

])

}

resource "aws\_ecs\_service" "frontend\_service" {

name = "frontend-service"

cluster = aws\_ecs\_cluster.cluster.id

task\_definition = aws\_ecs\_task\_definition.frontend.arn

desired\_count = 2

launch\_type = "FARGATE"

network\_configuration {

subnets = module.vpc.private\_subnets

security\_groups = [aws\_security\_group.ecs\_sg.id]

assign\_public\_ip = false

}

load\_balancer {

target\_group\_arn = aws\_lb\_target\_group.frontend\_tg.arn

container\_name = "frontend"

container\_port = 3000

}

depends\_on = [aws\_lb\_listener.http]

}

resource "aws\_ecs\_service" "backend\_service" {

name = "backend-service"

cluster = aws\_ecs\_cluster.cluster.id

task\_definition = aws\_ecs\_task\_definition.backend.arn

desired\_count = 2

launch\_type = "FARGATE"

network\_configuration {

subnets = module.vpc.private\_subnets

security\_groups = [aws\_security\_group.ecs\_sg.id]

assign\_public\_ip = false

}

load\_balancer {

target\_group\_arn = aws\_lb\_target\_group.backend\_tg.arn

container\_name = "backend"

container\_port = 5000

}

}

#### **Security groups (alb\_sg and ecs\_sg)**

* alb\_sg: allow inbound 80 from 0.0.0.0/0, allow outbound to ecs\_sg
* ecs\_sg: allow inbound from alb\_sg for ports 3000 & 5000, allow outbound to internet (for pulling images)

### **B — Build & Push Docker images to ECR (commands)**

After creating ECR repos in Terraform (or console), run locally:

AWS\_REGION=us-east-1

ACCOUNT\_ID=YOUR\_AWS\_ACCOUNT\_ID

# login

aws ecr get-login-password --region $AWS\_REGION | docker login --username AWS --password-stdin $ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com

# build frontend

docker build -t cloud-frontend:latest ./frontend

docker tag cloud-frontend:latest $ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/cloud-frontend:latest

docker push $ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/cloud-frontend:latest

# build backend

docker build -t cloud-backend:latest ./backend

docker tag cloud-backend:latest $ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/cloud-backend:latest

docker push $ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/cloud-backend:latest

If using CI (GitHub Actions), configure workflow to build & push automatically on commit.

### **C — Terraform apply**

After you push images to ECR and update task definitions (Terraform references repo URL with :latest tag), run:

terraform init

terraform plan -out plan.tfplan

terraform apply "plan.tfplan"

After apply completes, get ALB DNS from Terraform outputs or console — open it in browser; it should reach frontend. If you used path-based routing for /submit or /api, frontend can call relative paths and ALB will forward to backend.

### **D — Logging & Monitoring**

* Ensure container logging to CloudWatch by adding logConfiguration in container definitions (awslogs driver). Example:

"logConfiguration": {

"logDriver": "awslogs",

"options": {

"awslogs-group": "/ecs/frontend",

"awslogs-region": var.aws\_region,

"awslogs-stream-prefix": "frontend"

}

}

* Add CloudWatch alarms for unhealthy tasks.

## **Additional required files & tips**

### **outputs.tf (example for Part3)**

output "alb\_dns" {

value = aws\_lb.app\_alb.dns\_name

}

### **README.md (deliverable)**

Include:

* Terraform prerequisites
* How to configure backend (S3 bucket & DynamoDB table)
* How to run terraform init && terraform apply
* Where to find outputs (public IP / ALB DNS)
* Docker build & push commands
* Note about replacing placeholders

### **S3 backend & DynamoDB**

Create S3 bucket and DynamoDB table before terraform init or use separate Terraform to create them (requires local state for first run). Example CLI:

aws s3 mb s3://YOUR\_TERRAFORM\_STATE\_BUCKET --region us-east-1

aws dynamodb create-table --table-name terraform-locks \

--attribute-definitions AttributeName=LockID,AttributeType=S \

--key-schema AttributeName=LockID,KeyType=HASH \

--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \

--region us-east-1

## **Common debugging tips**

* If ECS tasks fail: check Task Events & CloudWatch logs (enable logs in task def).
* If ALB health check fails: confirm container listens on target port and path /health returns 200.
* If EC2 user data didn’t run: check instance system logs in AWS console.
* If Terraform plan complains about IAM: ensure your IAM user has required permissions.

## **Deliverables Checklist (what to hand in)**

* main.tf, variables.tf, outputs.tf, provider.tf, and any other .tf files for each part separated into directories:  
  + part1/ (single EC2)
  + part2/ (separate EC2s)
  + part3/ (ECR + ECS + VPC + ALB)
* user\_data\_\*.sh scripts used for EC2 provisioning.
* README.md explaining how to run terraform and how to test.
* Dockerfiles and instructions to build & push to ECR.
* Terraform S3 backend setup instructions or create\_backend.sh CLI snippet.
* Optional: GitHub Actions workflow to build & push images and run terraform apply (with secure secrets).