yes generate Terraform for EC2 instead of Fargate

Below is a **clean Terraform example for AWS EC2 (instead of ECS Fargate)** that deploys your Dockerized app on an EC2 instance using **Amazon Linux 2 + Docker**, pulling the image from **ECR**.

This setup includes:

- VPC (optional minimal example)
- Security Group
- EC2 instance with Docker installed
- User-data script that pulls your image from ECR and runs the container
- ECR repository
- IAM role + instance profile for EC2 to access ECR

📁 Terraform structure

```css
terraform/
|
├── main.tf
├── variables.tf
└── outputs.tf
```

🏗️ Terraform: EC2 Deployment with Docker + ECR

`main.tf`

```hcl
provider "aws" {
  region = var.aws_region
}

# --------------------------
#  ECR Repository
# --------------------------
resource "aws_ecr_repository" "app_repo" {
  name = var.app_name
}

# --------------------------
#  VPC + Subnet (simple setup)
# --------------------------
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "public" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.1.0/24"
  availability_zone = "${var.aws_region}a"

  map_public_ip_on_launch = true
}

resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id
}

resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id
}

resource "aws_route" "internet_access" {
  route_table_id         = aws_route_table.public.id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.igw.id
}

resource "aws_route_table_association" "public_assoc" {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}

# --------------------------
# Security Group
# --------------------------
resource "aws_security_group" "app_sg" {
  vpc_id = aws_vpc.main.id
  name   = "${var.app_name}-sg"

  ingress {
    description = "Allow HTTP"
    from_port   = 8080
    to_port     = 8080
    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"]
  }
}

# --------------------------
# IAM Role (EC2 → ECR access)
# --------------------------
resource "aws_iam_role" "ec2_role" {
  name = "${var.app_name}-ec2-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Principal = { Service = "ec2.amazonaws.com" }
    }]
  })
}

resource "aws_iam_role_policy_attachment" "ecr_readonly" {
  role       = aws_iam_role.ec2_role.name
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
}

resource "aws_iam_role_policy_attachment" "ssm_managed" {
  role       = aws_iam_role.ec2_role.name
  policy_arn = "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore"
}

resource "aws_iam_instance_profile" "ec2_profile" {
  name = "${var.app_name}-instance-profile"
  role = aws_iam_role.ec2_role.name
}

# --------------------------
# EC2 instance running Docker container
# --------------------------
data "aws_ami" "amazon_linux" {
  most_recent = true

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }

  owners = ["137112412989"] # Amazon
}

resource "aws_instance" "app_ec2" {
  ami                         = data.aws_ami.amazon_linux.id
  instance_type               = "t3.micro"
  subnet_id                   = aws_subnet.public.id
  security_groups             = [aws_security_group.app_sg.id]
  iam_instance_profile        = aws_iam_instance_profile.ec2_profile.name
  associate_public_ip_address = true

  user_data = base64encode(templatefile("${path.module}/userdata.tpl", {
    region      = var.aws_region
    account_id  = var.aws_account_id
    repo_name   = var.app_name
    app_port    = 8080
  }))

  tags = {
    Name = "${var.app_name}-ec2"
  }
}
```

📝 User Data Script Template

Create this file: `userdata.tpl`

```bash
#!/bin/bash

# Install Docker
amazon-linux-extras install docker -y
systemctl start docker
systemctl enable docker

# ECR login
aws ecr get-login-password --region ${region} | \
docker login --username AWS --password-stdin ${account_id}.dkr.ecr.${region}.amazonaws.com

# Pull latest image
docker pull ${account_id}.dkr.ecr.${region}.amazonaws.com/${repo_name}:latest

# Stop existing container if any
docker rm -f app || true

# Run the container
docker run -d \
  --name app \
  -p ${app_port}:${app_port} \
  ${account_id}.dkr.ecr.${region}.amazonaws.com/${repo_name}:latest
```

⚙️ variables.tf

```h
variable "aws_region" {
  default = "us-east-1"
}

variable "aws_account_id" {
  description = "Your AWS account ID"
}

variable "app_name" {
  default = "sample-ci-cd-app"
}
```

📤 outputs.tf

```hcl
output "ec2_public_ip" {
  value = aws_instance.app_ec2.public_ip
}

output "app_url" {
  value = "http://${aws_instance.app_ec2.public_ip}:8080"
}
```

✔️ How Jenkins fits in

Your Jenkins `Jenkinsfile` from earlier stays mostly the same:

1. Builds Docker image
2. Pushes to ECR
3. Runs `terraform apply` to update EC2
4. The EC2 instance pulls the new image via user-data or SSM

If you want **zero-downtime rolling updates**, I can generate:

- An Auto Scaling Group + Launch Template
- ALB (Application Load Balancer)
- Rolling deployment strategy