🔥 Hands-On Guide: Automating Infrastructure & Configuration with Terraform, Ansible, and CI/CD

If you're managing infrastructure at scale, you already know: **manual deployments don't scale**.

For me, the goal was simple:
✅ **Use Terraform** for infrastructure provisioning
✅ **Use Ansible** for server configuration
✅ Tie it all together in a **CI/CD pipeline** so it runs automatically on every push

Here's exactly how I set it up—**hands-on, end-to-end.**

🏗️1️⃣ Infrastructure as Code with Terraform

I started by defining my infrastructure in Terraform.

👉 **Folder structure:**

css
CopyEdit
```
infra/
  ├── main.tf
  ├── variables.tf
  ├── outputs.tf
```

✅ `main.tf` (provision an AWS EC2 instance)

hcl
CopyEdit
```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"  # Replace with valid AMI
  instance_type = "t2.micro"

  tags = {
    Name = "WebServer"
  }
```

```
}
```

✅ `outputs.tf` (expose instance public IP)

hcl
CopyEdit

```
output "public_ip" {
  value = aws_instance.web.public_ip
}
```

✅ Deploy:

bash
CopyEdit

```
terraform init
terraform plan
terraform apply -auto-approve
```

⚠️ Important: Configure your AWS credentials either via `~/.aws/credentials`, environment variables, or an IAM role if running inside AWS.

After deployment, you can grab the public IP:

bash
CopyEdit

```
terraform output public_ip
```

✅ Now I had an EC2 instance up and running.

🔧2️⃣ Configuration Management with Ansible

Next, I needed to install Nginx on this instance.

👉 **Folder structure:**

arduino
CopyEdit

```
config/
  ├── inventory.ini
  ├── site.yml
```

✅ `inventory.ini` (static for now, we'll make it dynamic later)

ini
CopyEdit

```
[web]
192.168.1.12 ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/id_rsa
```

(replace `192.168.1.12` with the EC2 public IP)

✅ `site.yml` (simple playbook to install Nginx)

yaml
CopyEdit

```
- hosts: web
  become: yes
  tasks:
    - name: Update apt
      apt:
        update_cache: yes

    - name: Install Nginx
      apt:
        name: nginx
        state: present
```

Run manually:

bash
CopyEdit

```
ansible-playbook -i inventory.ini site.yml
```

At this point, infrastructure and configuration were handled separately.

But I wanted **zero manual steps**.

🔄3️⃣ Automating with GitHub Actions

Here's where it gets interesting.

My goal:
→ Push code → GitHub Actions triggers →

→ Terraform provisions infra →
→ Ansible configures server

✅ .github/workflows/deploy.yml

yaml
CopyEdit

```
name: Infra + Config Deployment

on:
  push:
    branches: [ main ]

jobs:
  terraform:
    runs-on: ubuntu-latest
    outputs:
      public_ip: ${{ steps.tf_output.outputs.public_ip }}
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v3

      - name: Terraform Init
        run: terraform init

      - name: Terraform Plan
        run: terraform plan

      - name: Terraform Apply
        run: terraform apply -auto-approve

      - name: Get Terraform Output
        id: tf_output
        run: |
          echo "public_ip=$(terraform output -raw public_ip)" >>
$GITHUB_OUTPUT

  ansible:
    runs-on: ubuntu-latest
    needs: terraform
```

```
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Install Ansible
        run: sudo apt update && sudo apt install ansible -y

      - name: Generate Dynamic Inventory
        run: |
          echo "[web]" > inventory.ini
          echo "${{ needs.terraform.outputs.public_ip }}
ansible_user=ubuntu ansible_ssh_private_key_file=~/.ssh/id_rsa" >>
inventory.ini

      - name: Run Ansible Playbook
        run: ansible-playbook -i inventory.ini site.yml
```

✅ Here's what happens:

1. Terraform provisions the infrastructure
2. Captures the new EC2 public IP
3. Passes it into Ansible as a **dynamic inventory**
4. Ansible configures the new server automatically

🎯 **No hardcoded IPs. No manual steps. No waiting.**

📝 Key Learnings:

✔️ **Terraform outputs are critical** → expose necessary values for downstream tools
✔️ **Dynamic inventories make Ansible + Terraform integration smooth**
✔️ You don't need crazy plugins—just shell + output piping works well

This approach saved hours of manual effort and improved consistency across environments.

🙌 Final Thoughts

Combining **Terraform (infra provisioning)** and **Ansible (configuration management)** inside a **CI/CD pipeline** brings the best of both worlds:

- Code-driven infrastructure
- Automated, repeatable configurations
- Continuous deployments with every commit

It’s a pattern I recommend for any team looking to scale cloud operations without scaling human effort.