**Cloud Native Tools vs. Terraform**

*(DevOps Series)*

**1. Introduction to Terraform**

* Terraform is an **Infrastructure as Code (IaC)** tool by HashiCorp.
* Before jumping into Terraform, we must understand **Cloud Native Tools** and their limitations.

**2. Cloud Native Tools**

Each cloud provider has its own IaC tool:

* **AWS** → **CloudFormation (CFT)**
* **Azure** → **ARM Templates**
* **GCP** → **Deployment Manager**

**Disadvantages of Cloud Native Tools**

1. **Single File Configuration**
   * All configurations (VPC, Security Groups, S3, etc.) are dumped into **one file (JSON/YAML)**.
   * If the infrastructure is large, debugging becomes difficult due to **indentation errors** or **wrong function calls**.
2. **Complex Syntax**
   * Learning **JSON/YAML** is complex.
   * JSON is **not human-readable** compared to YAML.
3. **No Resource Importing**
   * **AWS CloudFormation** allows importing resources, but **Azure ARM Templates** do not.
4. **Cloud Lock-in**
   * **CloudFormation** works only in **AWS**, **ARM Templates** only in **Azure**.
   * Terraform is **multi-cloud** (AWS, Azure, GCP, etc.) via **provider blocks**.
5. **No Modules Concept**
   * **Modules** (reusable code) are **not available** in Cloud Native Tools.
   * In Terraform, modules help **avoid code repetition**.
6. **No Workspaces**
   * **Workspaces** (managing multiple environments) are **not available** in Cloud Native Tools.
7. **No Dry Run (Plan) Feature**
   * AWS recently added **dry run**, but Azure still lacks it.

**3. Why Terraform?**

✅ **Multi-cloud support** (AWS, Azure, GCP, etc.)  
✅ **Modular code** (reusability)  
✅ **Workspaces** (multiple environments)  
✅ **Better debugging & readability** (HCL language)  
✅ **State Management** (Terraform State)

**4. HashiCorp Tools**

* **Terraform** → Infrastructure Automation
* **Packer** → Image Automation
* **Consul** → Service Discovery & Cluster Management
* **Vault** → Secrets & Password Management
* **Nomad** → Container Orchestration

**5. Terraform Basics**

**Provider Block**

* Tells Terraform **which cloud** to deploy in (AWS, Azure, etc.).

hcl

provider "aws" {

region = "us-east-1"

}

**Resource Block**

* Used to **create infrastructure** (VPC, EC2, etc.).

hcl

resource "aws\_vpc" "vpc\_terraform" {

cidr\_block = "10.0.0.0/16"

enable\_dns\_hostnames = true

tags = {

Name = "vpc-terraform"

}

}

**Terraform Commands**

| **Command** | **Description** |
| --- | --- |
| terraform init | Initializes Terraform & downloads plugins |
| terraform plan | Shows what will be created (dry run) |
| terraform apply | Applies changes (creates infrastructure) |
| terraform destroy | Deletes all created resources |

**6. Data Sources**

* Used to **fetch existing resources** (manually created).
* Example: Attaching an **Internet Gateway** to a **manually created VPC**.

hcl

data "aws\_vpc" "selected" {

id = "vpc-12345678" *# Manually created VPC*

}

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

vpc\_id = data.aws\_vpc.selected.id

tags = {

Name = "igw-terraform"

}

}

**7. Remote State (S3 Backend)**

* Stores **Terraform state** in **AWS S3** (instead of local).
* Helps in **team collaboration** and **state locking**.

**Example:**

hcl

terraform {

backend "s3" {

bucket = "my-terraform-state-bucket"

key = "base-infra.tfstate"

region = "us-east-1"

}

}

**Use Case:**

* **Project 1:** Base Infrastructure (VPC, Subnets, etc.) → State stored in S3.
* **Project 2:** EC2 Instance → Uses **remote state** from Project 1 to deploy.

**8. Key Takeaways**

✔ **Terraform > Cloud Native Tools** (multi-cloud, modules, workspaces).  
✔ **Data Sources** fetch existing resources.  
✔ **Remote State (S3)** helps in team collaboration.  
✔ Always **destroy** resources after practice to avoid AWS costs.