Terrafrom Notes

🌍 What is Terraform?

Terraform is an **Infrastructure as Code (IaC)** tool developed by **HashiCorp** that allows you to **define, provision, and manage cloud infrastructure** using a declarative configuration language.

🚀 Key Features of Terraform

✅ **Declarative Syntax** – You define **what** infrastructure you want, and Terraform figures out **how** to create it.

✅ **Multi-Cloud Support** – Works with **AWS, Azure, Google Cloud, Kubernetes, and more**.

✅ **State Management** – Tracks infrastructure changes via a **state file** **(`terraform.tfstate`).**

✅ **Plan & Apply Workflow** – You can preview changes before applying them (`terraform plan → terraform apply`).

✅ **Reusable Modules** – Create reusable templates for infrastructure.

🛠️ How Does Terraform Work?

1️⃣ Write Configuration

- Define resources (e.g., EC2, S3, RDS) in `.tf` files using HashiCorp Configuration Language (HCL).

```
resource "aws_instance" "example" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"
}
```

2️⃣ Initialize Terraform

```
terraform init
```

- Downloads provider plugins (e.g., AWS, Azure).

3️⃣ Plan Infrastructure Changes

```
terraform plan
```

- Shows what changes will be made.

4️⃣ Apply the Configuration

```
terraform apply
```

- Creates or updates resources.

5️⃣ Destroy Infrastructure (if needed)

```
terraform destroy
```

- Deletes all provisioned resources.

📌 Terraform vs Other IaC Tools

Feature	Terraform	CloudFormation (AWS)	Ansible
Cloud Agnostic	✅ Yes	❌ AWS Only	✅ Yes
Declarative	✅ Yes	✅ Yes	❌ No (Imperative)
State Management	✅ Yes	✅ Yes	❌ No
Agentless	✅ Yes	✅ Yes	✅ Yes

📦 Where is Terraform Used?

🔷 **Infrastructure provisioning** (EC2, VPC, S3, RDS, etc.)

🔷 **Multi-cloud deployments** (AWS, Azure, GCP)

🔷 **Automating Kubernetes clusters**

🔷 **CI/CD Pipelines** (GitHub Actions, Jenkins)

🔷 **Security & Compliance automation**

🚀 Summary

- Terraform is an **IaC tool** to **automate infrastructure deployment**.
- It uses a **declarative approach** to define resources.
- Supports **multiple cloud providers** like AWS, Azure, GCP.
- Uses a **state file** to track changes.
- Ideal for **scalable, repeatable, and automated cloud provisioning**.

🔥 Important Concepts in Terraform

Terraform is built on several core concepts that help manage cloud infrastructure efficiently. Below are the **most important concepts** you should know.

1️⃣ Providers

Providers are plugins that let Terraform interact with different cloud services (AWS, Azure, GCP, Kubernetes, etc.).

👉 **Example: AWS Provider**

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

✅ **Popular Providers**: AWS, Azure, Google Cloud, Kubernetes, Helm, etc.

2️⃣ Resources

A **resource** is any infrastructure component you create with Terraform (e.g., EC2 instance, S3 bucket, VPC).

👉 **Example: Creating an EC2 Instance**

```
resource "aws_instance" "my_server" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"
}
```

✅ **Resources define actual cloud infrastructure.**

3️⃣ Variables

Terraform **variables** allow reusability and flexibility.

👉 **Example: Declaring Variables**

```
variable "instance_type" {
  default = "t2.micro"
}
```

👉 **Using the Variable in a Resource**

```
resource "aws_instance" "my_server" {
  ami           = "ami-0abcdef1234567890"
  instance_type = var.instance_type
}
```

✅ **Variables help avoid hardcoding values.**

4️⃣ Outputs

Outputs display useful information after Terraform applies changes (e.g., public IP of an EC2 instance).

👉 **Example: Getting the Public IP of an Instance**

```
output "instance_public_ip" {
  value = aws_instance.my_server.public_ip
```

✅ **Outputs are useful for debugging and automation.**

5️⃣ State Management (`terraform.tfstate`)

Terraform stores infrastructure information in a **state file** (`terraform.tfstate`). This helps Terraform track what is already created.

👉 **Important Commands:**

```
terraform show         # View the state file details
terraform state list   # List all managed resources
```

✅ **State file is crucial for tracking resources.**

6️⃣ Terraform Modules

Modules allow you to **reuse Terraform code** across projects.

👉 **Example: Creating an EC2 Module**

- **Module Folder:** `modules/ec2/main.tf`

```
resource "aws_instance" "my_server" {
  ami           = var.ami
  instance_type = var.instance_type
}
```

- **Calling the Module in** `main.tf`

```
module "ec2_instance" {
  source        = "./modules/ec2"
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"
}
```

✅ **Modules improve reusability and maintainability.**

7️⃣ Terraform Lifecycle (`create_before_destroy` , `prevent_destroy`)

Terraform allows **customizing how resources are created or destroyed**.

👉 **Example: Preventing Accidental Deletion**

```
resource "aws_instance" "my_server" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"

  lifecycle {
    prevent_destroy = true  # Prevents accidental deletion
  }
}
```

✅ **Lifecycle rules help manage infrastructure changes.**

8️⃣ Provisioners (Executing Scripts on Resources)

Provisioners allow running scripts inside created resources (e.g., installing packages on an EC2 instance).

👉 **Example: Running a Shell Script After EC2 Creation**

```
resource "aws_instance" "my_server" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"

  provisioner "remote-exec" {
    inline = [
      "sudo apt update -y",
      "sudo apt install nginx -y"
    ]
  }
}
```

✅ **Provisioners execute custom scripts on resources.**

9️⃣ Remote Backends (Storing State Remotely)

By default, Terraform stores the state file locally, but it's better to use a **remote backend** (S3, Azure Blob, etc.).

👉 **Example: Storing State in an S3 Bucket**

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state-bucket"
    key    = "terraform.tfstate"
    region = "us-east-1"
  }
}
```

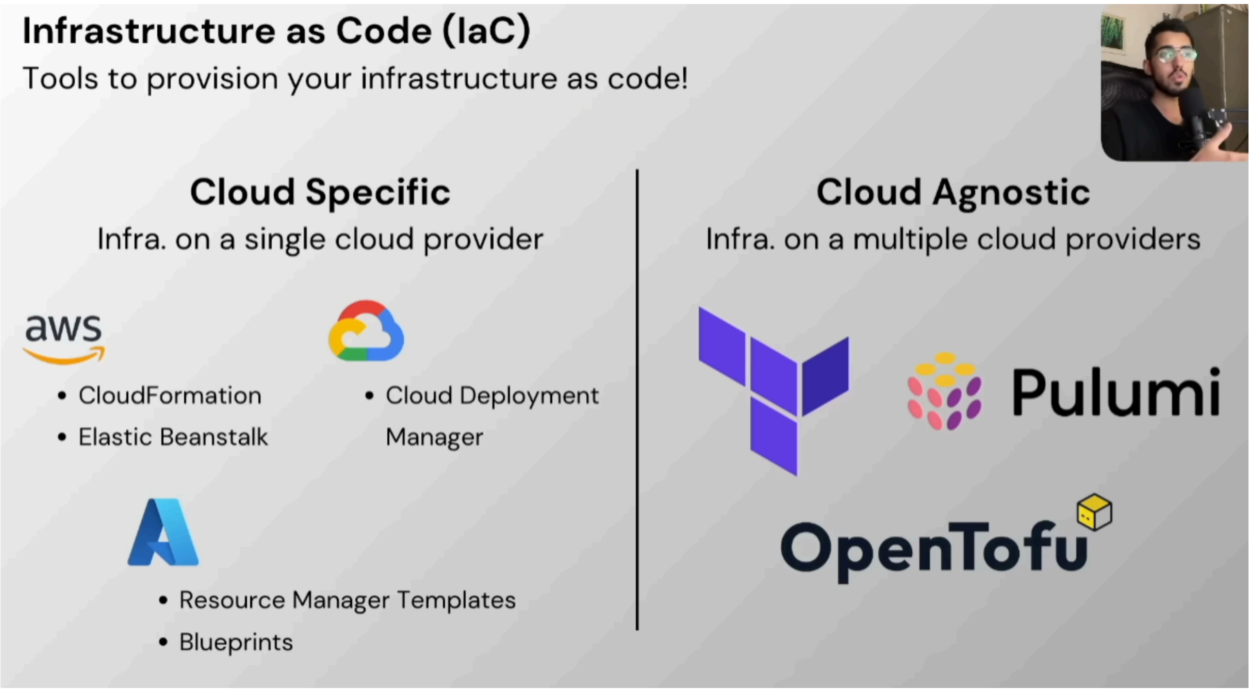
✅ **Remote backends enable team collaboration.**

🔟 Terraform Commands You Must Know

Command	Description
`terraform init`	Initializes the working directory
`terraform plan`	Shows what Terraform will change
`terraform apply`	Applies the changes
`terraform destroy`	Destroys all managed resources
`terraform validate`	Validates syntax errors in `.tf` files
`terraform fmt`	Formats Terraform code
`terraform output`	Displays output values
`terraform show`	Shows the Terraform state file details
`terraform state list`	Lists all managed resources

🚀 Summary

1. **Providers** – Connect Terraform to cloud services.
2. **Resources** – Define cloud infrastructure (EC2, S3, etc.).
3. **Variables** – Make Terraform configurations flexible.
4. **Outputs** – Display important information.
5. **State Management** – Tracks infrastructure changes.
6. **Modules** – Reusable Terraform configurations.
7. **Lifecycle Rules** – Control resource creation and deletion.
8. **Provisioners** – Run scripts inside created instances.
9. **Remote Backends** – Store Terraform state remotely.
10. **Commands** – Essential Terraform CLI commands.



Infrastructure as Code (IaC)

Tools to provision your infrastructure as code!

Cloud Specific

Infra. on a single cloud provider

- CloudFormation
- Elastic Beanstalk

- Cloud Deployment Manager

- Resource Manager Templates
- Blueprints

Cloud Agnostic

Infra. on a multiple cloud providers

🚨 Terraform Directory Structure – The Right Way! 🚨

A well-structured Terraform directory ensures scalability, reusability, and efficient infrastructure management. below is the best practices to follow

1️⃣ Environments – Separate Configs for Dev, Staging & Prod

Managing multiple environments? Here’s how to structure them:

📂 Development/

📂 Staging/

📂 Production/

Each contains:

✅

main.tf – Defines cloud resources.

✅

variables.tf – Declares variables without values.

✅

outputs.tf – Stores Terraform outputs for dependencies.

✅ terraform.tfvars – Provides values for variables.

🔷 Why?

Isolates Dev, Staging, and Production setups.

Avoids accidental production changes.

Makes configurations modular & reusable.

2️⃣ Modules – Reusable Infrastructure Components

Instead of repeating code, Terraform Modules help reuse configurations.

📌 VPC Module – Handles Virtual Private Cloud creation.

📌 EC2 Module – Manages EC2 instances efficiently.

🔷 Why?

🚀 Eliminates duplicate code – Define once, use everywhere!

🔄 Ensures consistency across environments.

⚙️ Faster deployment – Just call the module!

3️⃣ Scripts – Automate Terraform Workflows

Automation is key in DevOps & IaC. These scripts help:

⚙️

init.sh – Initializes Terraform.

🛑

teardown.sh – Destroys infrastructure to save costs.

🔷 Why?

Saves time by automating Terraform operations.

Reduces manual errors while setting up infrastructure.

4️⃣ Core Terraform Files – The Brains of Your Infrastructure

These files are the foundation of your Terraform project:

✅

provider.tf – Specifies the cloud provider (AWS, Azure, GCP).

✅

backend.tf – Defines state management (e.g., AWS S3, Terraform Cloud).

🔷 Why?

Keeps Terraform state secure instead of local files.

Prevents conflicts in team environments.

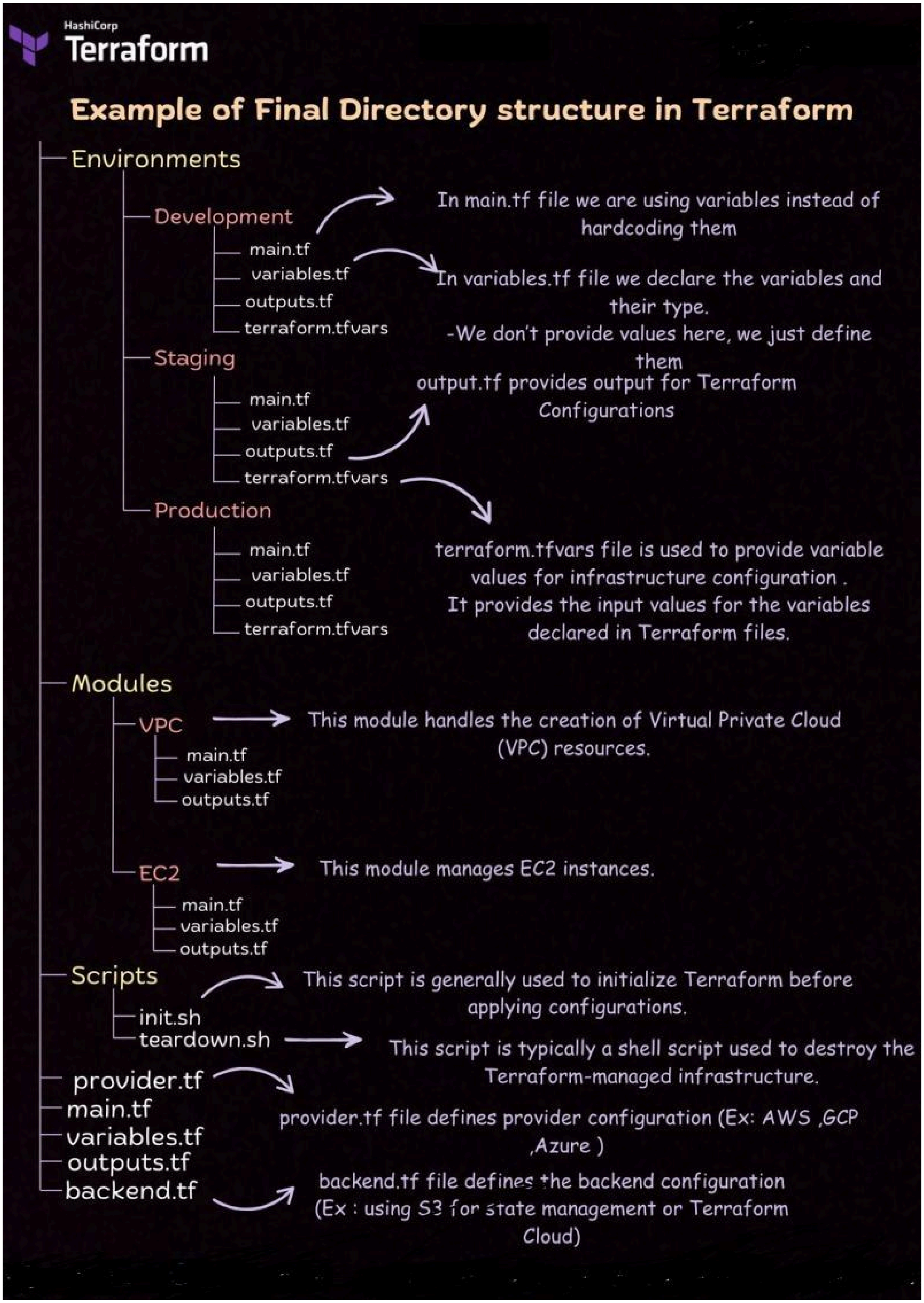
🔍 Why This Directory Structure Matters?

✅ Organized, modular, and scalable Terraform projects.

✅ Prevents accidental changes in production.

✅ Reusable infrastructure with Terraform Modules.

✅ Automated setup & cleanup with scripts.



Example of Final Directory structure in Terraform

```
├─ Environments
│   ├─ Development
│   │     ├─ main.tf
│   │     ├─ variables.tf
│   │     ├─ outputs.tf
│   │     └─ terraform.tfvars
│   ├─ Staging
│   │     ├─ main.tf
│   │     ├─ variables.tf
│   │     ├─ outputs.tf
│   │     └─ terraform.tfvars
│   └─ Production
│         ├─ main.tf
│         ├─ variables.tf
│         ├─ outputs.tf
│         └─ terraform.tfvars
├─ Modules
│   ├─ VPC
│   │     ├─ main.tf
│   │     ├─ variables.tf
│   │     └─ outputs.tf
│   └─ EC2
│         ├─ main.tf
│         ├─ variables.tf
│         └─ outputs.tf
├─ Scripts
│   ├─ init.sh
│   └─ teardown.sh
├─ provider.tf
├─ main.tf
├─ variables.tf
├─ outputs.tf
└─ backend.tf
```

In main.tf file we are using variables instead of hardcoding them

In variables.tf file we declare the variables and their type.
-We don't provide values here, we just define them

output.tf provides output for Terraform Configurations

terraform.tfvars file is used to provide variable values for infrastructure configuration .
It provides the input values for the variables declared in Terraform files.

VPC → This module handles the creation of Virtual Private Cloud (VPC) resources.

EC2 → This module manages EC2 instances.

init.sh: This script is generally used to initialize Terraform before applying configurations.

teardown.sh → This script is typically a shell script used to destroy the Terraform-managed infrastructure.

provider.tf file defines provider configuration (Ex: AWS ,GCP ,Azure)

backend.tf file defines the backend configuration (Ex : using S3 for state management or Terraform Cloud)

🚀 How MNCs Use Terraform in Their Codebase

Large enterprises and MNCs use **Terraform** to manage their cloud infrastructure in a structured and scalable way. Below are key practices that top companies follow to integrate Terraform into their workflows.

1️⃣ Infrastructure-as-Code (IaC) with GitOps

How MNCs Implement It:

- Store Terraform code in **GitHub, GitLab, Bitbucket, or AWS CodeCommit**.
- Every change to infrastructure goes through **code reviews** and is applied only after approval.

Example: GitOps Workflow

1. **Developer makes changes in a feature branch.**
2. **Pull Request (PR) is created and reviewed.**
3. **CI/CD pipeline runs `terraform plan` to verify changes.**
4. **If approved, `terraform apply` is executed.**

✅ Ensures **auditing**, **approval process**, and **collaboration** before making changes.

2️⃣ Modularized Codebase for Scalability

How MNCs Implement It:

- **Break down Terraform configurations into reusable modules.**
- Different teams use the same **module repository** to maintain consistency across multiple cloud environments.

Example: Modular Terraform Code Structure

```
terraform-repo/
|── modules/
│   ├── vpc/
│   ├── ec2/
│   ├── rds/
|── environments/
│   ├── dev/
│   ├── staging/
│   ├── prod/
|── main.tf
|── variables.tf
|── outputs.tf
|── terraform.tfvars
```

✅ Helps in managing **large infrastructure with reusable components**.

3️⃣ Using Remote State Storage & State Locking

How MNCs Implement It:

- Store Terraform state in **AWS S3, GCS, Azure Blob Storage, or Terraform Cloud**.
- Use **state locking** to prevent multiple users from modifying the same infrastructure at the same time.

Example: Storing Terraform State in S3

```
terraform {
  backend "s3" {
    bucket         = "mnc-terraform-state"
    key            = "prod/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "terraform-lock"
  }
}
```

✅ Prevents state conflicts when multiple engineers work on Terraform simultaneously.

4️⃣ Managing Multiple Environments Efficiently

How MNCs Implement It:

- Use **Terraform Workspaces** or separate folders for **Dev, Staging, and Production**.
- Each environment has different configurations and access control.

Example: Using Workspaces

```
terraform workspace new dev
terraform workspace new staging
terraform workspace new prod
terraform workspace select prod
terraform apply
```

✅ Ensures **isolation** between different environments.

5️⃣ CI/CD Pipelines for Automated Infrastructure Deployment

How MNCs Implement It:

- Use **GitHub Actions, Jenkins, GitLab CI/CD, or AWS CodePipeline** to automate Terraform deployment.
- Pipelines **run Terraform commands** (init, plan, validate, apply) automatically on every code change.

Example: GitHub Actions Pipeline for Terraform

```
name: Terraform Deployment

on:
  push:
    branches:
      - main

jobs:
  terraform:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v2

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v1

      - name: Terraform Init
        run: terraform init

      - name: Terraform Plan
        run: terraform plan -out=tfplan

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

✅ Automates **infrastructure provisioning and updates** in a controlled manner.

6️⃣ Implementing Role-Based Access Control (RBAC)

How MNCs Implement It:

- **Restrict Terraform access** using **IAM roles, service accounts, and policy-based permissions**.
- Only approved users can execute `terraform apply`.

Example: IAM Role for Terraform in AWS

```
{
  "Effect": "Allow",
  "Action": [
    "ec2:*",
    "s3:*",
    "iam:ListRoles"
  ],
  "Resource": "*"
}
```

✅ Ensures **only authorized teams** can make infrastructure changes.

7️⃣ Using Terraform Sentinel for Policy Enforcement

How MNCs Implement It:

- Use **Terraform Sentinel** (HashiCorp Enterprise) to enforce policies like:
    - No public-facing S3 buckets
    - No hardcoded access keys
    - Only approved AWS instance types are allowed

Example: Sentinel Policy (Deny Public S3 Buckets)

```
import "tfplan"

policy "no-public-s3" {
  enforcement_level = "hard-mandatory"

  condition = tfplan.resource_changes.aws_s3_bucket.exists and
              tfplan.resource_changes.aws_s3_bucket.public_read
}
```

✅ Ensures **security compliance** before applying infrastructure changes.

8️⃣ Terraform Cloud & Enterprise for Team Collaboration

How MNCs Implement It:

- Use **Terraform Cloud or Terraform Enterprise** for:
  - **State management** (instead of using S3 or GCS)
  - **Policy checks (Sentinel)**
  - **Team access control & approvals**

✅ Provides a **centralized control plane** for Terraform in large organizations.

9️⃣ Handling Secrets Securely with Vault & AWS Secrets Manager

How MNCs Implement It:

- Instead of **storing secrets** (like database passwords) in Terraform, use **HashiCorp Vault or AWS Secrets Manager**.

Example: Fetching Secrets from AWS Secrets Manager

```
data "aws_secretsmanager_secret" "db_password" {
  name = "prod/db_password"
}

resource "aws_db_instance" "database" {
  engine         = "mysql"
  instance_class = "db.t3.medium"
  password       = data.aws_secretsmanager_secret.db_password.value
}
```

✅ Keeps secrets **secure and encrypted**.

🔟 Cost Optimization & Governance with Terraform

How MNCs Implement It:

- **Tagging Policies**: Ensure all AWS resources have cost allocation tags.
- **Scheduled Resource Management**: Auto-scale EC2 instances based on usage.
- **Cost Monitoring**: Integrate Terraform with AWS Cost Explorer.

Example: Tagging EC2 Resources

```
resource "aws_instance" "web_server" {
  ami           = "ami-12345678"
  instance_type = "t3.micro"

  tags = {
    Environment = "Production"
    Owner       = "DevOps"
  }
}
```

✅ Helps **track resource costs** and **optimize cloud expenses**.

🚀 Summary: How MNCs Use Terraform in Their Codebase

Best Practice	Implementation
Infrastructure as Code (IaC)	All cloud infra is managed through Git & Terraform
Remote State Management	Store state in **S3, Azure Blob, GCS, or Terraform Cloud**
Reusable Modules	Standardized infrastructure modules for VPC, EC2, RDS, etc.
Multi-Environment Management	Separate Workspaces for **Dev, Staging, and Prod**
CI/CD Automation	**Jenkins, GitHub Actions, GitLab CI/CD** for auto-deployment
RBAC & IAM Policies	Restrict Terraform execution to approved users

Policy as Code (Sentinel)	Ensure **security compliance** before deployment
Secrets Management	Use **Vault / AWS Secrets Manager** for sensitive data
Cost Optimization	Enforce tagging and cost policies using Terraform

🎯 Final Thoughts

MNCs use **Terraform at scale** to manage thousands of cloud resources while ensuring **security, cost optimization, and automation**.

📌 Advanced Terraform Concepts You Should Know

🔷 1. Terraform DRY Principle (Don't Repeat Yourself)

To avoid writing repetitive code, **use Terraform modules** and **loops**.

Example: Creating Multiple EC2 Instances with `count`

```
resource "aws_instance" "web" {
  count         = 3
  ami           = "ami-12345678"
  instance_type = "t3.micro"
  tags = {
    Name = "web-${count.index}"
  }
}
```

✅ Creates **3 instances dynamically** without copy-pasting the same code.

🔷 2. Terraform `for_each` vs `count`

- `count` is useful for creating a **fixed number** of resources.
- `for_each` is better for **dynamically creating** resources from a map or list.

Example: Creating Resources with `for_each`

```
variable "users" {
  type = map
  default = {
    "user1" = "Developer"
    "user2" = "Admin"
  }
}

resource "aws_iam_user" "users" {
  for_each = var.users
  name     = each.key
  tags = {
    Role = each.value
  }
}
```

✅ Creates IAM users dynamically **based on a map**.

🔷 3. Terraform `depends_on` for Managing Resource Dependencies

- Terraform **automatically** determines dependencies, but in some cases, you need to **manually define them**.

Example: Ensuring an RDS Instance is Created After a Security Group

```
resource "aws_security_group" "rds_sg" {
  name = "rds-sg"
}

resource "aws_db_instance" "database" {
  depends_on     = [aws_security_group.rds_sg]
  engine         = "mysql"
  instance_class = "db.t3.micro"
  allocated_storage = 20
}
```

✅ Ensures the security group is created **before** the database instance.

🔷 4. Terraform Data Sources for Dynamic Configuration

- **Data sources** fetch **existing** infrastructure details instead of creating new ones.

Example: Fetching the Latest AMI ID for an EC2 Instance

```
data "aws_ami" "latest_amazon_linux" {
  most_recent = true
  owners      = ["amazon"]

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

resource "aws_instance" "web" {
  ami           = data.aws_ami.latest_amazon_linux.id
  instance_type = "t3.micro"
}
```

✅ Always picks the **latest Amazon Linux AMI** automatically.

🔷 5. Terraform Import: Managing Existing Infrastructure

Terraform can **import existing cloud resources** into state management **without recreating them**.

Example: Import an Existing AWS S3 Bucket

```
terraform import aws_s3_bucket.my_bucket my-existing-bucket-nam
```

✅ Useful when **migrating manual infrastructure** into Terraform.

🔷 6. Terraform Taint: Forcing Resource Recreation

If you need to **recreate a specific resource** without changing its configuration, use `terraform taint`.

Example: Marking an EC2 Instance for Recreation

```
terraform taint aws_instance.my_instance
terraform apply
```

✅ Terraform **deletes & recreates** the resource.

🔷 7. Terraform Workspaces for Multi-Tenant Deployments

- Workspaces allow you to manage **multiple environments** (e.g., **dev, staging, production**) using a **single configuration**.

Example: Creating Workspaces

```
terraform workspace new dev
terraform workspace select dev
terraform apply
```

✅ No need to **duplicate Terraform files** for different environments.

🔷 8. Terraform Debugging & Logging

If you get unexpected errors, enable **detailed logs** using the `TF_LOG` environment variable.

Example: Enable Debug Logging

```
export TF_LOG=DEBUG
terraform apply
```

✅ Helps **debug Terraform issues** efficiently.

🔷 9. Terraform Best Practices for Security & Compliance

✔ **Use `terraform validate` to check syntax before applying changes.**

✔ **Use `terraform fmt` to format code for consistency.**

✔ **Always use `terraform plan` before `terraform apply`.**

✔ **Store secrets in AWS Secrets Manager or HashiCorp Vault (never in `.tf` files).**

✔ **Enable AWS IAM policies to restrict who can run Terraform commands.**

🔷 10. Advanced Terraform Cloud & Enterprise Features

- Terraform Cloud provides a **managed Terraform execution environment**.
- You can **set policies using Sentinel** to **enforce compliance rules** (e.g., no public S3 buckets).

🚀 Summary: Next Steps for Mastering Terraform

🔥 **Concept**	🚀 **Why It's Important**
Modules & DRY Code	Avoids duplication & improves reusability
`count` **vs** `for_each`	Dynamic infrastructure creation
`depends_on`	Controls execution order of resources
Data Sources	Fetches existing infra details
Import & Taint	Manage existing infra & force recreation
Workspaces	Multi-environment management

Debugging with Logs	Helps troubleshoot Terraform issues
Security Best Practices	Prevents secrets exposure & enforces compliance