**Terraform State Management – Handling Changes (Azure Cloud)**

**"Infrastructure as Code (IaC) sounds great, but how does Terraform keep track of my deployments?"**

Terraform **state management** is the backbone of your infrastructure automation. Without it, Terraform wouldn’t know:

✔ **What’s already deployed** in Azure

✔ **What’s changed** since the last deployment

✔ **What needs to be updated, deleted, or created**

This is where **Terraform State** comes in!

In this newsletter, we will cover:

✅ **What is Terraform State? Why does it matter?**

✅ **Types of Terraform State – Local vs. Remote**

✅ **Handling Changes – Terraform Plan, Apply, Destroy**

✅ **Step-by-Step: Setting Up Remote State in Azure**

✅ **Best Practices for Secure & Scalable State Management**

**🔹 What is Terraform State? Why Does It Matter?**

Terraform **state** is a JSON file (terraform.tfstate) that tracks the current state of infrastructure managed by Terraform.

It acts as a **source of truth** for Terraform, storing information about resources that have been created, updated, or deleted.

**Why Does Terraform State Matter?**

1. Tracks Infrastructure Changes
2. Enables Efficient Plan and Apply Operations
3. Supports Dependencies Between Resources
4. Facilitates Collaboration
5. Handles Sensitive Information

**Where is Terraform State Stored?**

* **Local State:** Stored on the local machine (terraform.tfstate).
* **Remote State:** Stored in remote backends like AWS S3, Azure Blob, Google Cloud Storage, Terraform Cloud, etc., for better collaboration and security.

**Best Practices for Managing Terraform State**

✅ Use Remote State Storage – Prevents loss and enables collaboration.

✅ Enable State Locking – Avoids conflicts in concurrent updates (e.g., AWS DynamoDB with S3 backend).

✅ Encrypt State Files – Prevents exposure of sensitive data.

✅ Use terraform refresh Carefully – Updates the state file but can overwrite unintended changes.

✅ Restrict State File Access – Limit access to prevent unauthorized modifications.

**Types of Terraform State – Local vs. Remote**

When using Terraform in **Azure Cloud**, managing state files efficiently is crucial for infrastructure automation.

Terraform stores the state of managed resources in a **state file (**terraform.tfstate), which can be stored **locally** or **remotely**.

**1️⃣ Local Terraform State in Azure**

**What is Local State?**

By default, Terraform saves the **state file locally** in the working directory where Terraform is executed.

In Azure projects, this means the terraform.tfstate file is stored on the developer’s local machine.

**How Local State Works?**

When you run terraform apply, Terraform:

* Creates or updates Azure resources.
* Generates a terraform.tfstate file in the same directory.
* Reads this file in future Terraform operations to track changes.

**Where is the Local State File Stored?**

Terraform stores the state file in the root of the Terraform configuration directory:

/my-terraform-project/

├── main.tf

├── variables.tf

├── outputs.tf

├── terraform.tfstate ← Local state file

You can specify a different file name manually:

terraform apply -state=my\_custom\_state.tfstate

**Challenges of Using Local State in Azure Projects**

1. **Not Shareable Across Teams** In Azure DevOps or team environments, developers must share the latest state file manually, which can lead to conflicts.
2. **Security Risks** The state file may contain sensitive data such as Azure storage access keys, database passwords, or VM IPs.
3. **No State Locking** If two engineers apply Terraform changes at the same time, the state file may get corrupted.
4. **No Disaster Recovery** If the local machine is lost or formatted, the Terraform state is gone unless manually backed up.

**When to Use Local State in Azure?**

* Personal projects or **small-scale Azure deployments**.
* Proof-of-concept (PoC) or **testing environments** where state file loss is not critical.
* Quick **development iterations** without team collaboration.

**2️⃣ Remote Terraform State in Azure**

**What is Remote State?**

Remote state stores Terraform’s **state file in an Azure Storage Account**, allowing multiple team members to access and update infrastructure safely.

**How Remote State Works in Azure?**

* Terraform saves the terraform.tfstate file in **Azure Blob Storage** instead of the local machine.
* Terraform automatically **locks the state** using Azure Storage's concurrency features, preventing simultaneous updates.
* Developers or CI/CD pipelines retrieve the latest state from Azure before applying changes.

**How to Configure Remote State in Azure?**

**Step 1: Create an Azure Storage Account**

Terraform needs an Azure Storage Account and a Blob Container to store the state file. You can create these using Azure CLI:

az storage account create \

--name mystorageaccount \

--resource-group myResourceGroup \

--location eastus \

--sku Standard\_LRS

az storage container create \

--name terraform-state \

--account-name mystorageaccount

**Step 2: Define Remote Backend in Terraform**

Modify the Terraform configuration to use **Azure Storage as the backend**:

terraform {

backend "azurerm" {

resource\_group\_name = "myResourceGroup"

storage\_account\_name = "mystorageaccount"

container\_name = "terraform-state"

key = "prod.terraform.tfstate"

}

}

**Step 3: Initialize Terraform**

Run the following command to initialize Terraform and migrate the state file to Azure:

terraform init

Terraform will now use Azure Storage to manage the state file instead of storing it locally.

**Advantages of Using Remote State in Azure**

1. **Collaboration in Teams**
2. **State Locking**
3. **Disaster Recovery & Backup**
4. **Security & Compliance**

**Challenges of Using Remote State in Azure**

1. **Requires Initial Setup** You must create an Azure Storage Account and configure Terraform.
2. **Performance Latency** Retrieving state from Azure Storage can be slower than local storage, but it's negligible for most use cases.
3. **Additional Costs** Azure Storage incurs costs for storage and data transfer, although minimal.

**When to Use Remote State in Azure?**

* **Production environments** where multiple engineers work on infrastructure.
* **CI/CD pipelines in Azure DevOps** that need to access Terraform state.
* **Enterprise-scale Azure deployments** that require state security and consistency.
* **Disaster recovery planning** where losing state files is not an option.

**🔹 Handling Changes – Terraform Plan, Apply, Destroy**

Terraform follows a **3-step process** when making changes:

✔ **Step 1: Plan** (terraform plan)

* Compares the **state file** with your Terraform code.
* Shows what changes will be made.
* Safe way to preview before applying.

✔ **Step 2: Apply** (terraform apply)

* Executes the planned changes.
* Updates the Terraform state file.

✔ **Step 3: Destroy** (terraform destroy)

* Deletes all resources **AND removes them from the state file**.

**Step-by-Step Guide: Storing Terraform State in Azure Storage**

Storing Terraform state in **Azure Storage** allows for **team collaboration, security, disaster recovery, and state locking**.

Follow this guide to set up **Azure Blob Storage as a remote backend** for Terraform.

**Step 1: Create an Azure Storage Account**

Terraform needs an **Azure Storage Account** to store the terraform.tfstate file.

**1.1 Log in to Azure**

Ensure you are logged into your Azure account using Azure CLI:

az login

If you have multiple subscriptions, set the correct one:

az account set --subscription "SUBSCRIPTION\_ID"

**1.2 Create a Resource Group**

A **resource group** is needed to hold the storage account.

az group create --name myResourceGroup --location eastus

**1.3 Create a Storage Account**

Azure Storage accounts must have **globally unique names**.

az storage account create \

--name mystorageaccount123 \

--resource-group myResourceGroup \

--location eastus \

--sku Standard\_LRS \

--kind StorageV2

* Standard\_LRS (Locally Redundant Storage) ensures cost-effective storage.
* StorageV2 is recommended for better performance.

**Step 2: Create an Azure Storage Container**

Terraform state files are stored in a **container** within the storage account.

az storage container create \

--name terraform-state \

--account-name mystorageaccount123

**Step 3: Retrieve the Storage Access Key or Use Managed Identity**

Terraform needs authentication to access Azure Storage. You can either:

* **Use Storage Account Key (Basic Method)**
* **Use Managed Identity (Recommended for security)**

**3.1 Get the Storage Account Key (Basic Method)**

Retrieve the storage account key and store it securely:

az storage account keys list \

--resource-group myResourceGroup \

--account-name mystorageaccount123 \

--query '[0].value' \

--output tsv

Export the key as an environment variable:

export ARM\_ACCESS\_KEY="YOUR\_STORAGE\_ACCOUNT\_KEY"

**⚠️ WARNING:** Never hardcode the key in Terraform files. Instead, use environment variables.

**3.2 Use Managed Identity (Recommended for Security)**

For better security, use **Azure Managed Identity** and Role-Based Access Control (RBAC).

Assign the **Storage Blob Data Contributor** role to Terraform:

az role assignment create \

--assignee <your-service-principal-id> \

--role "Storage Blob Data Contributor" \

--scope "/subscriptions/YOUR\_SUBSCRIPTION\_ID/resourceGroups/myResourceGroup/providers/Microsoft.Storage/storageAccounts/mystorageaccount123"

This method eliminates the need to store access keys in Terraform code.

**Step 4: Configure Terraform to Use Remote State**

Modify the **Terraform configuration** to use Azure Storage as the backend.

**4.1 Update main.tf to Use Remote State**

Add the following **backend configuration** inside terraform {}:

terraform {

backend "azurerm" {

resource\_group\_name = "myResourceGroup"

storage\_account\_name = "mystorageaccount123"

container\_name = "terraform-state"

key = "prod.terraform.tfstate"

}

}

* key = "prod.terraform.tfstate" → This is the **file name** inside Azure Storage.

**Step 5: Initialize Terraform and Migrate State to Azure**

**5.1 Initialize Terraform**

Run terraform init to initialize the backend and migrate state:

terraform init

Terraform will prompt:

Do you want to migrate your existing state to the new backend? [yes]

Enter yes to proceed.

**5.2 Verify the State File in Azure**

Check if terraform.tfstate is successfully stored in Azure:

az storage blob list \

--container-name terraform-state \

--account-name mystorageaccount123 \

--output table

You should see:

Name Blob Type Length Content Type

---------------------- ----------- ------ -------------------

prod.terraform.tfstate BlockBlob 1234 application/json

**Step 6: Test Terraform with Remote State**

**6.1 Apply Terraform Configuration**

Deploy infrastructure using:

terraform apply -auto-approve

Terraform will:

* Create Azure resources
* Update the **state file in Azure Storage** instead of locally

**6.2 Destroy Resources (Optional)**

To clean up, run:

terraform destroy -auto-approve

**Step 7: Secure Terraform State in Azure**

**7.1 Enable Storage Account Access Control**

Instead of allowing **account keys**, use **Azure RBAC** with Managed Identities.

**7.2 Enable State File Versioning**

Enable **Soft Delete** for Terraform state files:

az storage blob service-properties update \

--account-name mystorageaccount123 \

--delete-retention true \

--delete-retention-days 7

This ensures accidental deletions can be **recovered within 7 days**.

**7.3 Encrypt State File**

Azure automatically encrypts blobs, but for additional security, enable **customer-managed keys (CMK)** if required.

**Final Thoughts: Why Store Terraform State in Azure?**

✅ **Collaboration** – Teams can share the state file securely.

✅ **State Locking** – Prevents multiple users from modifying infrastructure at the same time.

✅ **Security & Compliance** – RBAC and encryption protect sensitive data.

✅ **Disaster Recovery** – No risk of losing the Terraform state.

**🔹 Real-World Use Cases of Terraform State in Azure**

Terraform state is critical in managing infrastructure efficiently in **Azure Cloud**.

It helps maintain the desired state, track changes, and ensure consistency across deployments.

Below are some **real-world use cases** where Terraform state plays a vital role in Azure environments.

**1. Multi-Environment Infrastructure Management (Dev, Staging, Prod)**

**Scenario**

A company manages different environments (**Dev, Staging, and Production**) in Azure. Each environment must have a separate state file to avoid conflicts.

**How Terraform State Helps**

* Stores the state separately for each environment in **Azure Storage**.
* Prevents accidental modifications of the production environment.
* Allows different teams to work on separate environments without interference.

**Implementation**

Use separate state files in **Azure Blob Storage**:

terraform {

backend "azurerm" {

resource\_group\_name = "terraform-backend-rg"

storage\_account\_name = "terraformstorage123"

container\_name = "terraform-state"

key = "prod/terraform.tfstate"

}

}

This ensures production infrastructure changes do not affect other environments.

**2. Shared State for Teams Working on the Same Infrastructure**

**Scenario**

A DevOps team manages a large Azure infrastructure with multiple team members applying Terraform configurations. Without a shared state, each team member might overwrite previous changes.

**How Terraform State Helps**

* **State Locking**: Prevents multiple users from running terraform apply simultaneously.
* **Consistency**: Ensures all changes are tracked in a central location.
* **Version Control**: Teams can track who made infrastructure changes.

**Implementation**

Use **Azure Storage with State Locking** (via Terraform Enterprise or third-party locking mechanisms like Consul).

terraform {

backend "azurerm" {

resource\_group\_name = "terraform-backend-rg"

storage\_account\_name = "terraformstorage123"

container\_name = "terraform-state"

key = "shared/terraform.tfstate"

}

}

🔹 **Azure Storage does not support native locking**, so use **Terraform Cloud** for **enhanced collaboration and state locking**.

**3. Automating CI/CD Pipeline Deployments in Azure DevOps**

**Scenario**

An organization wants to **automate infrastructure deployments** using Terraform in an **Azure DevOps pipeline**.

**How Terraform State Helps**

* Allows the CI/CD pipeline to **track infrastructure changes**.
* Ensures that deployments are **incremental and idempotent**.
* Helps in **rollback and disaster recovery** if a deployment fails.

**Implementation**

1. **Store Terraform State in Azure Storage**
2. **Configure Azure DevOps Pipeline to Access the State**
3. **Run Terraform Commands in the Pipeline**

trigger:

- main

pool:

vmImage: ubuntu-latest

steps:

- task: TerraformTaskV2@2

displayName: 'Terraform Init'

inputs:

backendType: 'azurerm'

backendAzureRmSubscription: 'My-Azure-Subscription'

backendResourceGroupName: 'terraform-backend-rg'

backendStorageAccountName: 'terraformstorage123'

backendContainerName: 'terraform-state'

backendKey: 'cicd/terraform.tfstate'

- task: TerraformTaskV2@2

displayName: 'Terraform Apply'

inputs:

command: 'apply'

environmentServiceNameAzureRM: 'My-Azure-Subscription'

This automates Terraform state management in **Azure DevOps pipelines**.

**4. Disaster Recovery & Infrastructure Backup**

**Scenario**

A company wants to ensure that **infrastructure can be recovered** in case of accidental deletion or failures.

**How Terraform State Helps**

* **Backup & Restore**: Stores historical state files for rollback.
* **Disaster Recovery**: If a region fails, infrastructure can be restored from the last known state.
* **Soft Delete**: Prevents accidental deletions of state files.

**Implementation**

Enable **Azure Blob Soft Delete** to retain Terraform state files:

az storage blob service-properties update \

--account-name terraformstorage123 \

--delete-retention true \

--delete-retention-days 7

If an infrastructure failure occurs, reapply the last known Terraform state:

terraform apply

This ensures quick **infrastructure restoration**.

**5. Managing Large-Scale Microservices Deployments on Azure Kubernetes Service (AKS)**

**Scenario**

A company manages **100+ microservices** deployed on Azure Kubernetes Service (AKS). They need to track changes to the Kubernetes infrastructure dynamically.

**How Terraform State Helps**

* **Tracks AKS Cluster Changes**: Keeps a record of node pools, networking, and security settings.
* **Modularization**: Breaks large AKS deployments into **multiple Terraform state files**.
* **Prevents Configuration Drift**: Ensures Kubernetes infrastructure is always in the desired state.

**Implementation**

Use separate Terraform states for different AKS components:

terraform {

backend "azurerm" {

resource\_group\_name = "terraform-backend-rg"

storage\_account\_name = "terraformstorage123"

container\_name = "terraform-state"

key = "aks-cluster/terraform.tfstate"

}

}

This helps manage complex **Kubernetes workloads** in Azure.

**6. Managing Azure Networking (VNet, Peering, NSG, Private Endpoints)**

**Scenario**

A security team wants to maintain strict **networking policies** across **multiple Azure Virtual Networks (VNets)**.

**How Terraform State Helps**

* **Tracks Subnet Configurations**: Ensures IP ranges and security groups are consistent.
* **Enforces Security Compliance**: Prevents accidental misconfigurations in NSGs.
* **Manages VNet Peering**: Automates secure **inter-region** and **cross-region** connectivity.

**Implementation**

Use separate Terraform state files for **networking configurations**:

terraform {

backend "azurerm" {

resource\_group\_name = "terraform-backend-rg"

storage\_account\_name = "terraformstorage123"

container\_name = "terraform-state"

key = "networking/terraform.tfstate"

}

}

This ensures **centralized networking management** in **Azure**.

**7. Implementing Role-Based Access Control (RBAC) for Azure Resources**

**Scenario**

A company needs to enforce **RBAC policies** across all Azure resources to prevent unauthorized access.

**How Terraform State Helps**

* **Tracks User & Group Permissions**: Ensures consistent access management.
* **Prevents Privilege Escalation**: Enforces least-privilege access.
* **Audits Changes**: Provides a history of permission modifications.

**Implementation**

Define RBAC roles in Terraform and store state in Azure:

resource "azurerm\_role\_assignment" "example" {

scope = "/subscriptions/YOUR\_SUBSCRIPTION\_ID"

role\_definition\_name = "Reader"

principal\_id = "USER\_OR\_GROUP\_ID"

}

terraform {

backend "azurerm" {

resource\_group\_name = "terraform-backend-rg"

storage\_account\_name = "terraformstorage123"

container\_name = "terraform-state"

key = "rbac/terraform.tfstate"

}

}

This ensures that **RBAC changes are properly tracked**.

Terraform state is essential for **managing Azure infrastructure at scale**.

It helps **track changes, prevent misconfigurations, enable team collaboration, and ensure security compliance**.

**Key Benefits:**

✅ **Multi-environment management** (Dev, Staging, Prod)

✅ **Shared state for team collaboration**

✅ **CI/CD automation with Azure DevOps**

✅ **Disaster recovery and state backup**

✅ **Large-scale AKS microservices management**

✅ **Networking & security policy enforcement**

✅ **RBAC tracking and auditing**

**🔹 Best Practices for Secure & Scalable Terraform State Management**

✔ **Always use Remote State Storage** (Azure Storage, AWS S3, etc.)

✔ **Enable State Locking** to prevent simultaneous changes

✔ **Encrypt Terraform State** (Use Azure SSE for security)

✔ **Backup Terraform State Files** to avoid accidental loss

✔ **Use Terraform Workspaces** for managing multiple environments

**🔹 Key Takeaways**

✔ **Terraform State** keeps track of infrastructure deployments

✔ **Local State** is useful for testing but unsafe for teams

✔ **Remote State** in Azure Storage is the best practice for production

✔ **Terraform Plan & Apply** ensures controlled infrastructure updates

✔ **Best Practices** include encryption, state locking, and backups

By mastering **Terraform State Management**, you can **handle changes safely** and **avoid infrastructure disasters!**