10

Azure Infra with Terraform:

MUST-KNOW PROBLEMS

Shruthi Chikkela
learnwithshruthi

1. Resource Already Exists Error

Problem:

You run `terraform apply` and get:

```
Error: A resource with the ID already exists -
ResourceGroupName already exists.
```

Why It Happens:

Terraform tries to **create a resource that already exists in Azure**, but it doesn't know it was created manually or outside its configuration.

Scenario:

You manually created a resource group called `dev-rg` in Azure Portal, but later included the same in your Terraform script.

```
resource "azurerm_resource_group" "example" {
  name     = "dev-rg"
  location = "East US"
}
```

Terraform thinks it needs to create this, but Azure says it already exists, causing a conflict.

Solution:

Use `terraform import` to bring the existing resource into Terraform state:

Steps:

1. Identify the Azure resource ID:
   `/subscriptions/<sub_id>/resourceGroups/dev-rg`
2. Run:
   ```
   terraform import azurerm_resource_group.example /
   subscriptions/xxxx/resourceGroups/dev-rg
   ```
3. Run `terraform plan` to confirm no changes are required.

Avoid hardcoded names if reusability or automation is the goal. Use:

```
name = "rg-${var.env}"
```

```
}
```

2. Terraform State File Conflicts

Problem:

Two users run `terraform apply` at the same time → corrupted or conflicting `terraform.tfstate`.

Why It Happens:

The state file is **not locked**, so multiple users can write to it concurrently, which breaks consistency.

Scenario:

You and a teammate both run `terraform apply` on a shared project using local state. One apply succeeds, the other fails or creates conflicts.

Solution: Use Remote State with Locking

Steps:

- Create an Azure Storage Account container for state:
  - RG: `tfstate-rg`
  - Storage Account: `tfstateacct`
  - Container: `tfstate`

Configure backend in your Terraform:

```
terraform {
  backend "azurerm" {
    resource_group_name  = "tfstate-rg"
    storage_account_name = "tfstateacct"
    container_name       = "tfstate"
    key                  = "terraform.tfstate"
  }
}
```

- Run:
  ```
  terraform init
  ```

Now, Terraform locks the state while one user applies, preventing parallel edits.

3. Resource Dependency Timing Issues

Problem:

Resources like VMs fail to deploy because dependent resources (like subnets or NSGs) aren't ready.

Why It Happens:

Terraform executes in parallel unless dependencies are **explicit**.

Scenario:

You create a VM and subnet, but Terraform deploys them at the same time.

The VM fails because the subnet isn’t ready yet.

```
resource "azurerm_subnet" "subnet" { ... }

resource "azurerm_linux_virtual_machine" "vm" {
  network_interface_ids = [azurerm_network_interface.nic.id]
}
```

Solution: Ensure Proper Dependencies

Ways to Handle It:

- Use `depends_on`:

```
resource "azurerm_network_interface" "nic" {

  depends_on = [azurerm_subnet.subnet]
  ...
}
```

- Reference values (implicit dependency):
  `subnet_id = azurerm_subnet.subnet.id`

Terraform now understands it must create the subnet before using its `id`.

4. Provider Authentication Failures

Problem:

You run `terraform plan` and get:

```
Error: unable to authenticate to Azure
```

Why It Happens:

Terraform doesn't have credentials for Azure. This can happen in CI/CD or new CLI environments.

Scenario:

You're using Terraform on a new machine or DevOps pipeline and forget to log in or set credentials.

Solution:

Local Dev:

1. Run:
   ```
   az login
   ```

2. Terraform uses your active CLI session automatically.

Service Principal (for automation):

```
export ARM_CLIENT_ID="xxxx"
export ARM_CLIENT_SECRET="xxxx"
export ARM_SUBSCRIPTION_ID="xxxx"
export ARM_TENANT_ID="xxxx"
```

You can also create an SP using:

```
az ad sp create-for-rbac --role="Contributor" —scopes="/
subscriptions/<sub_id>"
```

Then plug the credentials into your Terraform environment.

5. Inconsistent Naming & Tagging Across Resources

Problem:

Resources have inconsistent names and missing or mismatched tags.

Why It Happens:

Hardcoding names or manually applying tags can lead to unreadable and disorganised environments.

Scenario:

Your VM is named `vm-dev1`, your storage account is `storagedev02`, and your tags are inconsistent:

- One has `env = dev`
- Another has `environment = development`

Solution:

Use Naming Standards:

```
variable "env" {
  default = "dev"
}
locals {
  name_prefix = "app-${var.env}"
}
resource "azurerm_storage_account" "example" {
  name = "${local.name_prefix}stg"
  ...
}
```

Use Tag Blocks:

```
locals {
  common_tags = {
    environment = var.env
    owner       = "team-infra"
  }
}

resource "azurerm_virtual_machine" "vm" {
  ...
  tags = local.common_tags
}
```

This ensures naming consistency across all resource types.

6. Destroying Production Resources by Mistake

Problem:

Someone accidentally runs `terraform destroy` or applies wrong changes in the **production** environment.

Why It Happens:

- The same state file or workspace is used across environments.
- Lack of proper environment separation or safeguards.

Scenario:

You meant to destroy a **dev** environment but pointed to the **production** backend accidentally.

Solution: Isolate environments using Workspaces, Separate State, and Permissions

Method 1: Workspaces

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

In Terraform:

```
resource "azurerm_resource_group" "rg" {
  name     = "rg-${terraform.workspace}"
  location = "East US"
}
```

Each workspace has a separate state and isolates environments like `dev`, `staging`, `prod`.

Method 2: Separate Backends

Use different `backend` blocks for each environment (e.g., `dev.tfbackend`, `prod.tfbackend`).

```
terraform init -backend-config="prod.tfbackend"
```

Method 3: RBAC and Approvals

- Restrict `terraform destroy` permission using Azure RBAC.
- Use **approval gates** in CI/CD pipelines to prevent direct applies to production.

7. Configuration Drift Between Terraform and Azure Portal

Problem:

Terraform thinks everything is fine, but someone changed something manually in Azure Portal.

Why It Happens:

Terraform only tracks what's in the **state file**, not what's actually deployed—unless you run a **refresh or plan**.

Scenario:

Terraform says VM is `Standard_B1s`, but someone manually changed it to `Standard_B2ms`. You won't know unless you check manually or run a plan.

Solution:

Detect Drift:

```
terraform plan
```

If the real Azure config doesn't match Terraform's state, it will show a diff.

Refresh Local State:

```
terraform refresh
```

This updates your `.tfstate` file to reflect the current state of real infrastructure.

Best Practice:

- Avoid manual changes in Azure once Terraform manages a resource.
- Add “noPortalEdits” policies using **Azure Policy**.

8. Incorrect Module Usage and Errors

Problem:

You use a custom or public module, but the resource fails due to wrong or missing input variables.

Why It Happens:

Modules are reusable, but you must pass **all required inputs** correctly.

Scenario:

You use a VNet module, but forget to define `address_space` or `subnet_prefixes`.

```
module "vnet" {
  source = "./modules/network"
  name   = "my-vnet"
  # missing critical variables
}
```

Terraform throws:

```
Error: Missing required argument
```

Solution:

Read module documentation:

Understand required `variables.tf`, outputs, and how the module is structured.

Validate early:

```
terraform validate
```

Use default values where appropriate:

In module `variables.tf`:

```
variable "address_space" {
  type    = list(string)
  default = ["10.0.0.0/16"]
}
```

Test modules in isolation first:

Before reusing in prod, test with dummy values and run:

```
terraform plan -out=tfplan
```

9. Terraform and Provider Version Mismatch

Problem:

Using outdated providers with newer Terraform versions can cause compatibility errors.

Why It Happens:

Terraform and providers evolve separately. A feature you're using might not be supported in the version declared or installed.

Scenario:

Your config uses:

```
lifecycle {
  ignore_changes = [os_disk]
}
```

But your `azurerm` provider doesn't support this behavior for that resource.

Solution:

Lock Compatible Versions:

```
terraform {
  required_version = ">= 1.3.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
  }
}
```

Upgrade Provider:

```
terraform init -upgrade
```

Check Release Notes:

Use the Terraform Registry to confirm new features and deprecations.

10. Slow Apply Times Due to Sequential Execution

Problem:

Terraform apply takes too long, especially when creating many similar resources.

Why It Happens:

Resources are deployed one after another when not designed for **parallelism**.

Scenario:

You create 10 Linux VMs using `count`, and they deploy **sequentially**.

```
resource "azurerm_linux_virtual_machine" "vm" {
  count = 10
  name  = "vm-${count.index}"
  ...
}
```

Solution: Use parallelism-friendly patterns

Option 1: Use `for_each` for independent resources

```
variable "vm_names" {
  default = ["vm1", "vm2", "vm3"]
}

resource "azurerm_linux_virtual_machine" "vm" {
  for_each = toset(var.vm_names)
  name     = each.key
  ...
}
```

This allows parallel execution because Terraform can evaluate each independently.

Option 2: Use `-parallelism` flag

```
terraform apply -parallelism=10
```

(Default is 10; you can increase for faster execution—test for resource limits!)

Issue	Strategy
Resource conflict	Use `import` and avoid hardcoding
State conflict	Use remote backends with locking
Drift	Use `plan`, `refresh`, and avoid portal edits
Module bugs	Validate inputs, test in isolation
Apply performance	Use `for_each`, tweak parallelism

For any RealTime Handson Projects

And for more tips like this

+ Follow

Shruthi Chikkela

Like & ReShare

👍 🔁

in learnwithshruthi