Terraform Modules

Use case: There are 10 teams in your organizations using terraform to create and manage EC2 instance

DRY Principle: In software engineering don’t repeat yourself (DRY) is a principle of software development

Challenges:

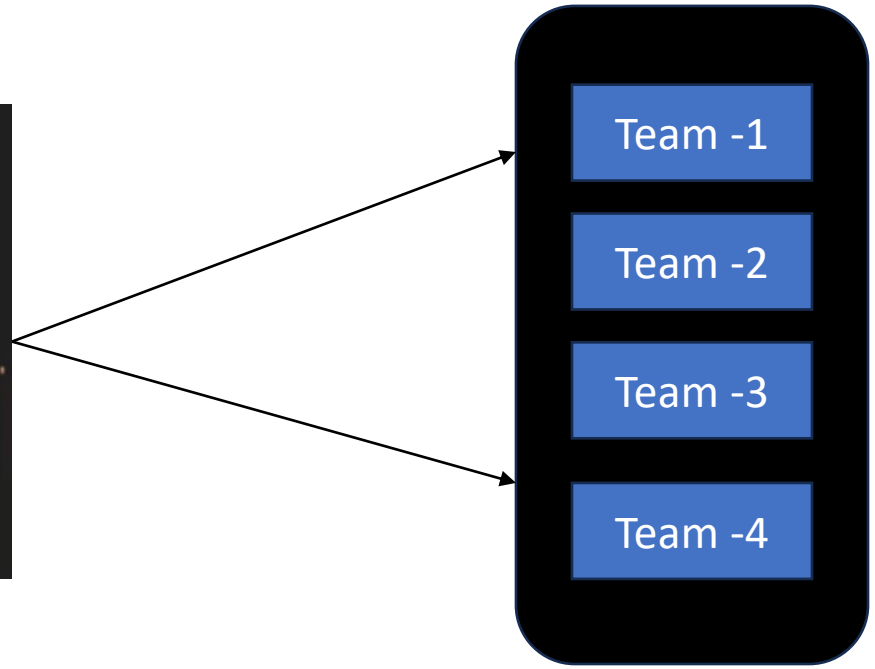
1. Repetition of Code
2. One change in AWS provider option will require to make changes in multiple blocks
3. Lack of standardization
4. Unproductive

Recommended Approach

Solution: Terraform modules allows us define standard configuration in a central location to be re-used across multiple projects

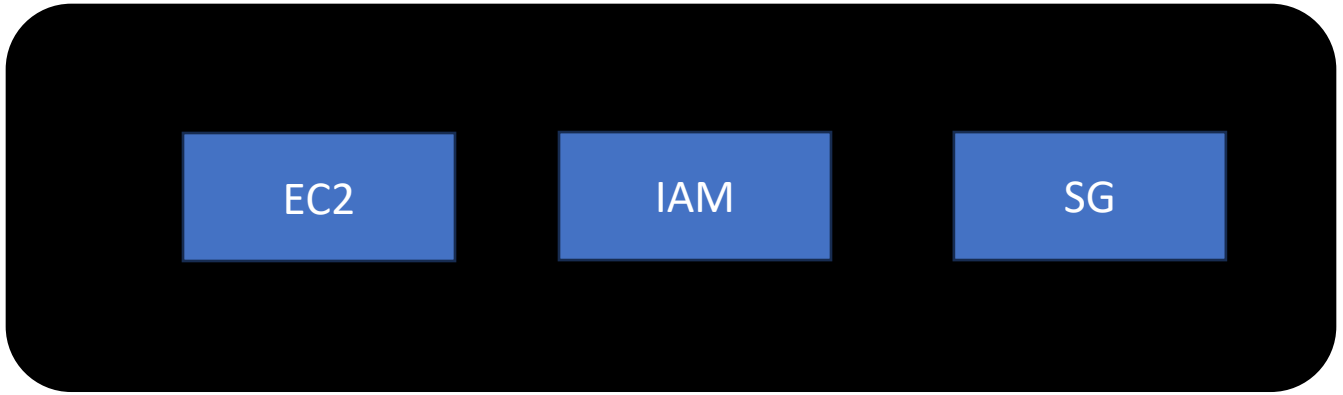
```
resource "aws_instance" "web" {
  ami           = "ami-1234"
  instance_type = "t3.micro"
  key_name      = "user1"
  monitoring    = true
  vpc_security_group_ids = ["sg-12345678"]
  associate_public_ip_address = true
  instance_initiated_shutdown_behavior = "stop"
  ebs_optimized  = true
  source_dest_check = false
  hibernation = true
  disable_api_termination = true
}
```

Terraform Module



Understanding the base structure

1. A base “modules” folder
2. A sub folder containing name for each modules that are available
3. Each sub folder will contain actual module terraform code that other projects can refer from



Modules folder

Calling a module

1. In order to reference to a module , you need to make a use of module block

2. The module block must contain source argument that cantains location to the reference to the referenced module

```
module "ec2" {
  source = "github.com/<username>/<repo name>"
}
```

```
module "ec2" {
  source = "../../modules/ec2"
}
```

Module

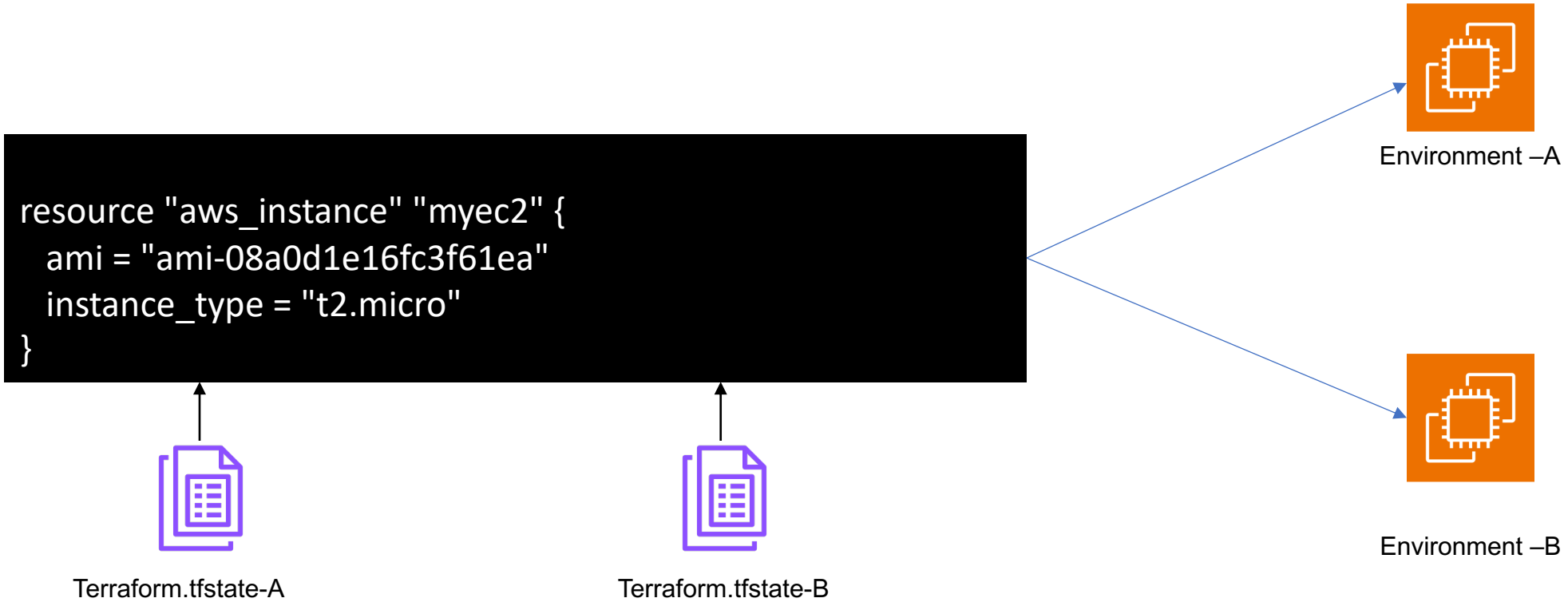
```
provider "aws" {
  region = var.region
}

resource "aws_instance" "myec2" {
    ami = var.ami
    instance_type = var.instance_type
}


variable "ami" {}
variable "instance_type" {}
variable "region" {}
```

Calling a module

```
module "ec2" {
  source = "../../modules/ec2"
  instance_type = "t2.large"
  ami = "ami-123"
  region = "ap-south-1"
}
```

Terraform workspace

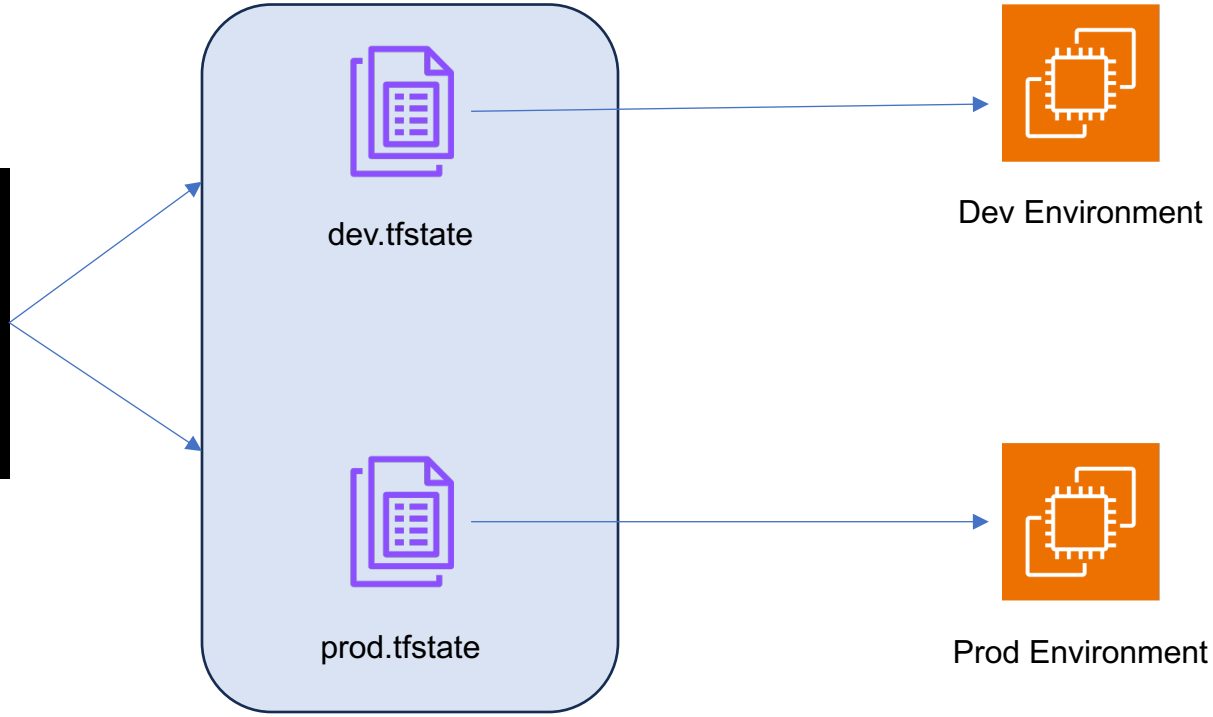
Infrastructure is tied to the underlying Terraform configuration and a state file.



Terraform workspace

Terraform workspace allows us to manage multiple environments with same set of code

```
resource "aws_instance" "myec2" {
  ami = "ami-08a0d1e16fc3f61ea"
  instance_type = "t2.micro"
}
```



Terraform workspace commands

```
terraform workspace
terraform workspace show
terraform workspace new dev
terraform workspace new prod
terraform workspace list
terraform workspace select dev
```

Base code

```
resource "aws_instance" "myec2" {
  ami = "ami-08a0d1e16fc3f61ea"
  instance_type = "t2.micro"
}
```

Final code

```
locals {
  instance_type = {
    default = "t2.nano"
    dev     = "t2.micro"
    prod    = "m5.large"
  }
}

resource "aws_instance" "myec2" {
  ami           = "ami-08a0d1e16fc3f61ea"
  instance_type = local.instance_type[terraform.workspace]
}
```

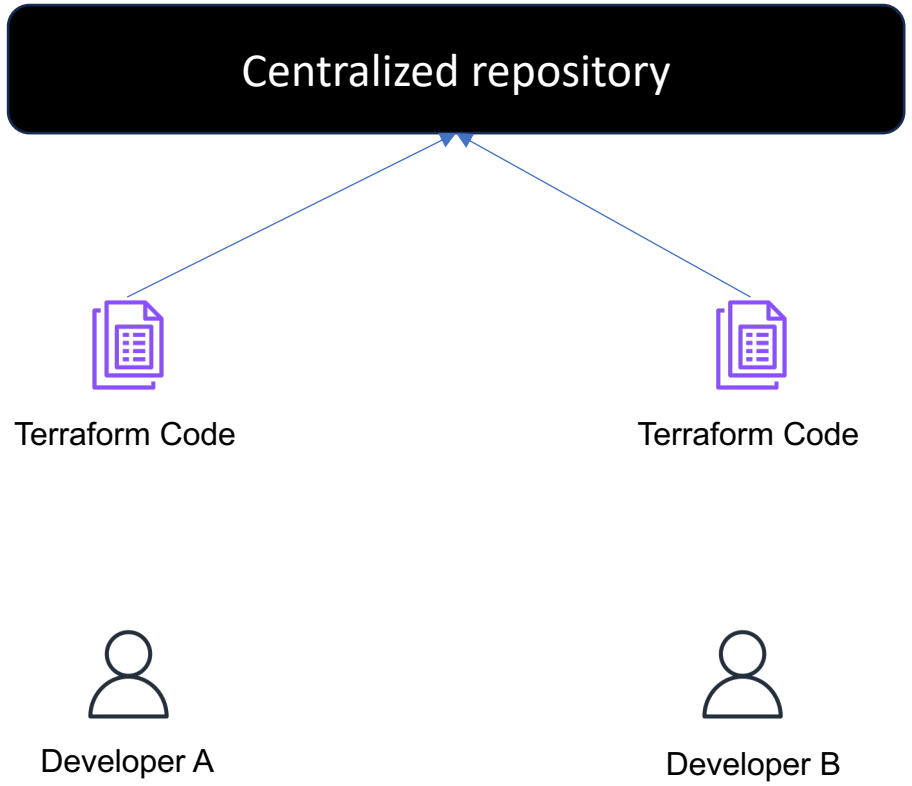
Remote State management

Challenges:

1. Local changes may not be persistent
2. Lack of collaboration

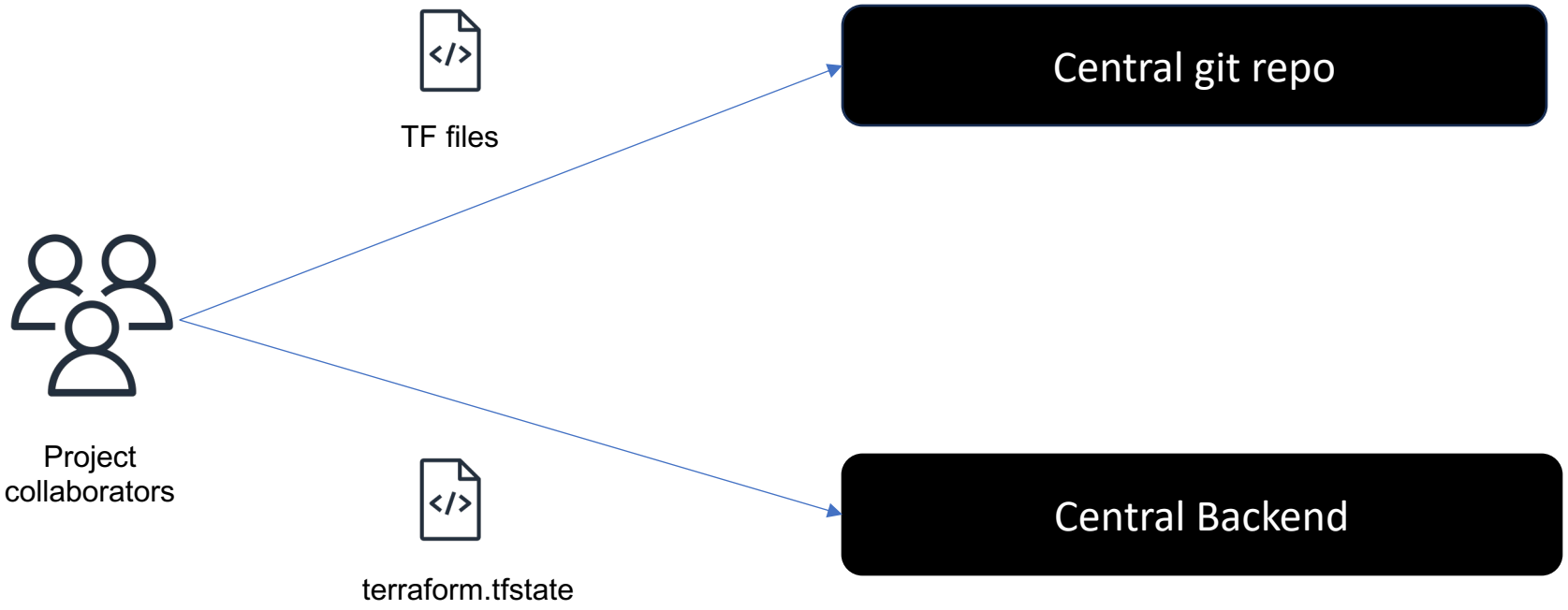
Solution:

1. Centralized management
2. Keep your credential outside of code
3. Ignore unwanted files using gitignore



Backends supported for Terraform

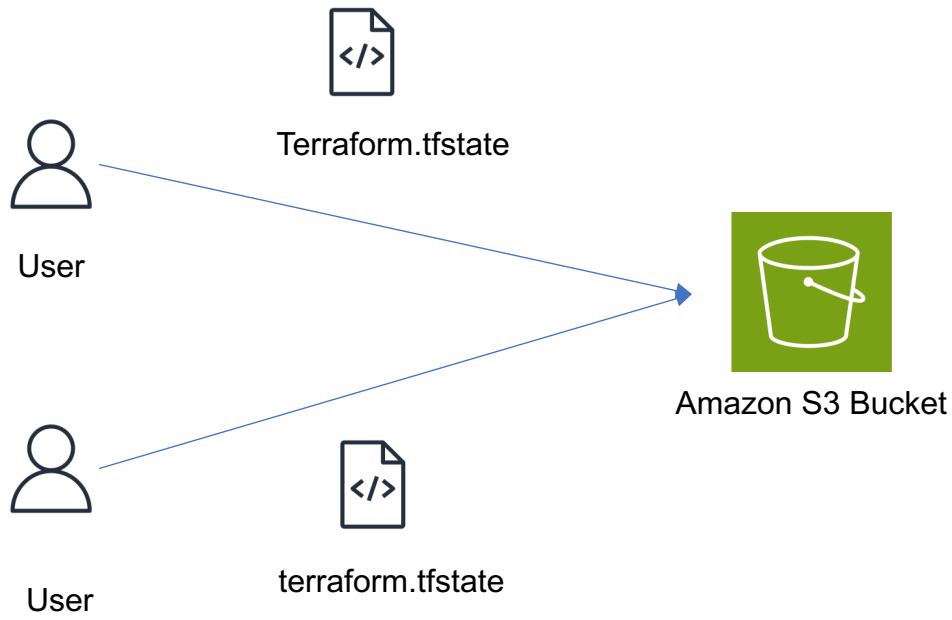
1. S3
2. HTTP
3. Consul
4. Azurerm



S3 Backends Configuration

Challenges:

1. User permission management
2. Conflict management



Backend.tf

```
terraform {
  backend "s3" {
    bucket = ”hst-terraform-backend"
    key    = "network/terraform.tfstate"
    region = "us-east-1"
  }
}
```

providers.tf

```
provider "aws" {
  region     = "us-west-2"
}
```

Eip.tf

```
resource "aws_eip" "lb"
{
  domain = "vpc"
}
```

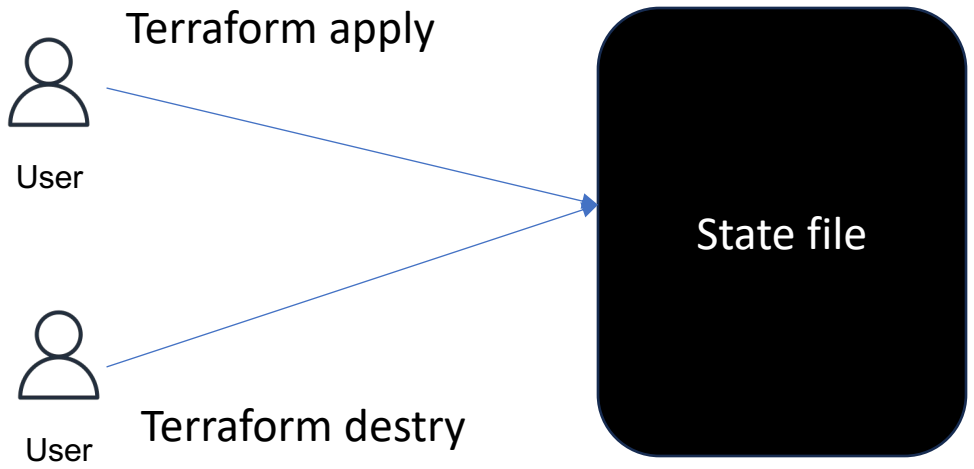
State File Locking

1. While performing write operations, terraform would lock the file

2. If others also try the same time , it can corrupt your state file

3. Force-unlock command for manual unlock

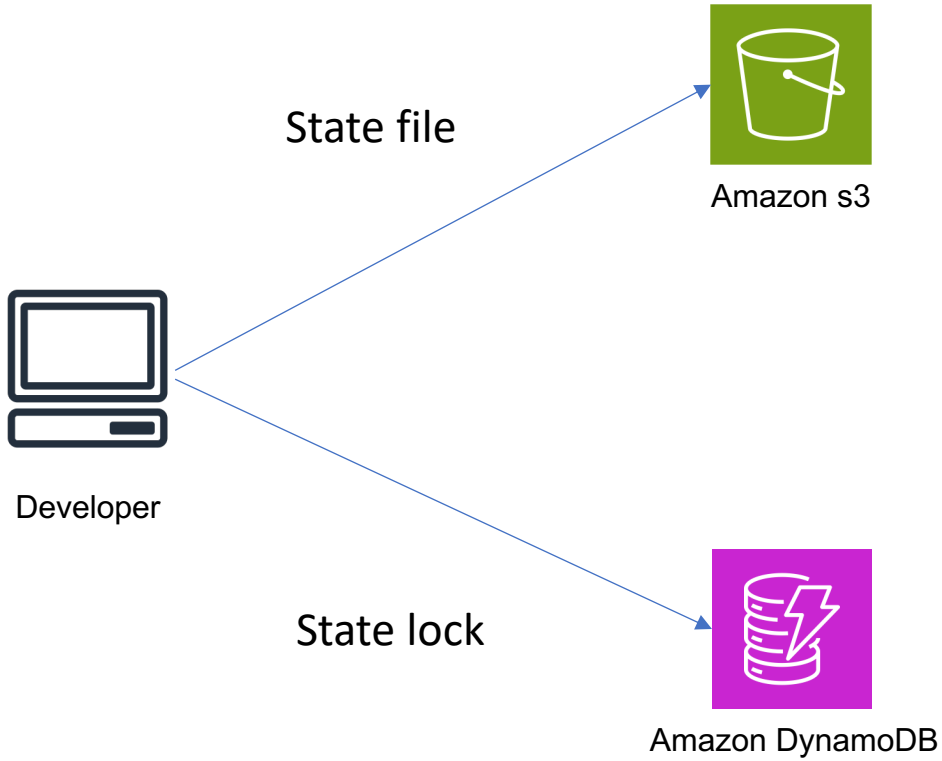
4. Force-unlock should be used to unlock your lock if automatic unlocking fails



State Locking in S3 backend

1. S3 doesn’t support State Locking functionality

2. Leverage DynamoDB table to achieve state locking



State management

1. List the Resources Managed through Terraform

```
terraform state pull
```

2. Show Attributes of Resource

```
terraform state show aws_security_group.prod
```

3. Pull the State file From Remote Backend

```
terraform state pull
```

Terraform Best Practices

- **Use a Consistent Versioning Approach**
- **Organize and Structure your Configuration**
- **Implement State Management Best Practices**
- **Follow the Principle of Least Privilege**
- **Automate and Integrate Terraform Workflows**
- **Keep your Configurations DRY (Don't Repeat Yourself)**
- **Use Modules for Reusability**:
- **Document and Test your Configurations**

Recommended workflow