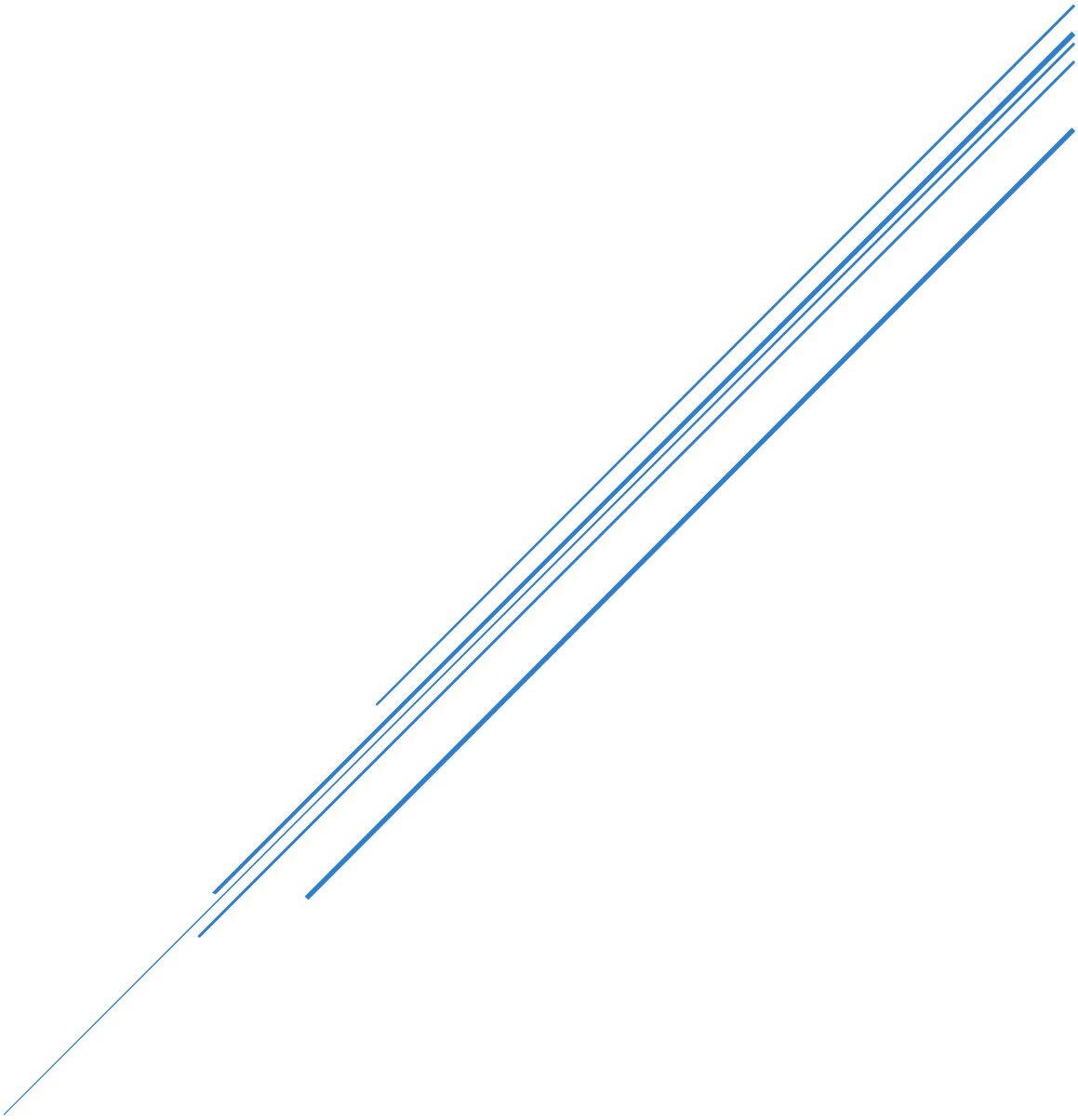
TERRAFORM MORE TOPICS 2

AWS IAC



Contents

State Management

In Terraform, state is a critical concept that helps manage your infrastructure. Here's a breakdown of Terraform state and its key aspects:

1. What is Terraform State?

Terraform uses state to map real-world resources to your Terraform configuration. It serves as a record of:

- Resource Metadata: Information about the resources managed by Terraform.
- Dependencies: Relationships between resources.
- Current State: How resources are configured currently.

State is stored in a file called terraform.tfstate.

2. Why is State Important?

- Resource Tracking: Tracks resources created by Terraform, so it knows how to manage changes.
- Performance: Allows Terraform to determine what needs to be updated without querying the entire infrastructure.
- Collaboration: Centralized state allows teams to work together without conflicts.
- Drift Detection: Identifies when real-world resources differ from the configuration.

3. Terraform State Files

- Local State: By default, Terraform stores the state file locally (terraform.tfstate).
- Remote State: For collaboration, you can store the state file in remote backends like AWS S3, Azure Blob Storage, or HashiCorp Consul.

4. State Management Commands

View State:

```
terraform show
```

List Resources:

```
terraform state list
```

Inspect a Resource:

```
terraform state show <resource_name>
```

Move Resources:

```
terraform state mv <source> <destination>
```

Remove Resources:

```
terraform state rm <resource_name>
```

Import Resources:

```
terraform import <resource_name> <id>
```

5. Remote State Configuration
To use remote state, you configure a backend in your terraform block. For example, using AWS S3:

```
terraform {
  backend "s3" {
    bucket       = "my-terraform-state"
    key          = "prod/terraform.tfstate"
    region       = "us-west-2"
    encrypt      = true
    dynamodb_table = "terraform-lock"
  }
}
```

This configuration:

- Stores the state file in an S3 bucket.
- Uses DynamoDB for state locking to prevent race conditions.

6. State Locking

- Ensures that only one operation can modify the state at a time.
- Built into remote backends like S3 (with DynamoDB), Azure, and Google Cloud Storage.

Scenario 1:
Example of configuring and using Terraform with an S3 backend without DynamoDB. This setup demonstrates how to store your Terraform state file in S3.

1. Prerequisites
Before starting, ensure the following:

- An AWS account is set up.
- AWS CLI is installed and configured with your credentials.
- An S3 bucket exists (or you can create one during the process).

2. Create the S3 Bucket
Use the AWS CLI to create an S3 bucket if you don't already have one. Replace <your-bucket-name> with your preferred bucket name and <your-region> with your AWS region.

```
aws s3api create-bucket --bucket <your-bucket-name> --region <your-region> --create-bucket-configuration LocationConstraint=<your-region>
```

3. Configure Terraform for Remote State
In your Terraform project, set up the backend configuration in the terraform block in the main.tf file:

```
terraform {
  backend "s3" {
    bucket       = "your-bucket-name"
    key          = "path/to/terraform.tfstate" # Customize the key (path) for your state file
    region       = "your-region"
    encrypt      = true # Encrypt the state file at rest
  }
}
```

```
provider "aws" {
  region = "your-region"
}

resource "aws_s3_bucket" "example" {
  bucket = "example-terraform-bucket"
  acl    = "private"
}
```

4. Initialize Terraform
Run the following command to initialize Terraform and configure the S3 backend:
terraform init

Terraform will:

- Detect the backend configuration.
- Ask for confirmation to migrate any existing local state to the S3 backend.

5. Apply Terraform Configuration
Run the following commands to apply the Terraform configuration:

Plan the changes:

```
terraform plan
```

Apply the changes:

```
terraform apply
```

6. Verify the State File in S3
After applying, you can verify that the terraform.tfstate file is stored in the S3 bucket:

- Navigate to your S3 bucket in the AWS Management Console.
- Go to the specified key path (path/to/terraform.tfstate) and confirm the presence of the state file.

7. Updating or Destroying Resources
To update the configuration:

- Modify your main.tf file.
- Run terraform plan and terraform apply to apply the changes.

To destroy the resources:

- terraform destroy

8. Notes

- Versioning: Enable versioning in your S3 bucket to keep a history of state file changes.
- Bucket Policies: Apply bucket policies to restrict access to the state file.
- No Locking: Without DynamoDB, there's no locking mechanism. Be cautious if multiple users might modify the state simultaneously.

Scenario 2:
You want to manage an AWS S3 bucket using Terraform and store the state remotely in an S3 bucket with DynamoDB for locking.

1. Prerequisites
- Terraform Installed: Ensure Terraform is installed on your machine.
- AWS CLI Configured: You should have AWS credentials set up.
- S3 Bucket and DynamoDB Table: Create these beforehand for remote state storage.

2. Create Remote State Backend

Terraform Configuration
Create a **backend.tf** file to configure the remote backend:

```
terraform {
  backend "s3" {
    bucket       = "my-terraform-state"      # S3 bucket for state storage
    key          = "dev/terraform.tfstate"   # Path to the state file in S3
    region       = "us-west-2"               # AWS region
    encrypt      = true                      # Enable encryption
    dynamodb_table = "terraform-lock"        # DynamoDB table for locking
  }
}
```

Setup Commands
Initialize Terraform to configure the backend:

```
terraform init
```

3. Write Terraform Configuration
Create a main.tf file to define the resources:

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

resource "aws_s3_bucket" "example" {
  bucket = "my-example-bucket"
  acl    = "private"

  tags = {
    Name       = "ExampleBucket"
    Environment = "Dev"
  }
}

resource "aws_s3_bucket_public_access_block" "example" {
  bucket = aws_s3_bucket.example.id

  block_public_acls      = true
  block_public_policy    = true
  ignore_public_acls     = true
```

```
  restrict_public_buckets = true
}
```

4. Initialize and Apply Terraform

```
terraform init
terraform plan
terraform apply
```

Confirm the changes when prompted. Terraform will:

- Create an S3 bucket.
- Store the state remotely in the configured S3 bucket.

5. Managing Terraform State

View State

```
terraform show
```

List Resources

```
terraform state list
```

Inspect a Resource

```
terraform state show aws_s3_bucket.example
```

6. Import an Existing Resource

If an S3 bucket already exists, import it into Terraform state:

Create an Empty Resource in main.tf:

```
resource "aws_s3_bucket" "example" {
  bucket = "existing-bucket-name"
}
```

Import the Resource:

```
terraform import aws_s3_bucket.example existing-bucket-name
```

7. Move Resources Between State Files

Suppose you want to move the S3 bucket resource to a separate state file for better organization.

Create a New State File:

```
terraform state pull > new-state.tfstate
```

Move the Resource:

```
terraform state mv aws_s3_bucket.example new_state.tfstate
```

8. Troubleshooting Common Issues

State Drift

If resources are modified outside Terraform, detect drift with:

```
terraform plan
```

State Locking Error
If Terraform reports the state is locked, manually unlock it:

```
terraform force-unlock <LOCK_ID>
```

Recover a Lost State

1. Recover State from Remote Backend (if applicable)
If you were using a remote backend like S3, check if versioning or backups were enabled.

Steps for S3:

- Log in to the AWS Management Console.
- Navigate to the S3 bucket where the state file was stored.
- Check if versioning is enabled:
- If enabled, restore a previous version of terraform.tfstate.
- Download the state file and re-upload it if necessary.

2. Reconstruct State Using terraform import
If the state file is completely lost and cannot be recovered, you can rebuild it by importing resources into a new state file.

Steps:
Initialize a New State: Create a minimal main.tf file representing your existing infrastructure. For example, if you had an S3 bucket:

```
provider "aws" {
  region = "your-region"
}

resource "aws_s3_bucket" "example" {
  bucket = "existing-bucket-name"
}
```

Import the Resource: Use the terraform import command to add the resource to the state.

```
terraform import aws_s3_bucket.example existing-bucket-name
```

Repeat for All Resources: Repeat the terraform import command for all resources. You can find resource IDs in your cloud provider's console.

Run terraform plan: Verify that the configuration matches the actual infrastructure. Resolve discrepancies by updating the configuration.

3. Recover State from Local Backup

If you had local backups:

- Locate the backup state file, typically saved as terraform.tfstate.backup.
- Replace the lost terraform.tfstate with the backup.

Security Scanning using tfsec and Checkov

Both TFSec and Checkov are excellent tools for analyzing Terraform code to identify potential security issues.

1. Prerequisites
   - Install Terraform on your Windows machine. Download Terraform
   - Install Python 3.x if using Checkov (Checkov requires Python). Download Python

2. Install TFSec

TFSec is a static analysis tool for Terraform code.

a) Download the latest Windows binary (tfsec-windows-amd64.exe).
https://github.com/aquasecurity/tfsec/releases

b) Rename the file to **tfsec.exe** and place it in a directory that's in your PATH (e.g., C:\Windows\System32).

c) Verify Installation

```
tfsec --version
```

3. Install Checkov

Checkov is another security tool that supports Terraform, CloudFormation, Kubernetes, and more.

Install via pip

```
pip install checkov
```

Verify Installation

```
checkov --version
```

4. Analyze Terraform Code

Using TFSec

Navigate to the directory containing your Terraform configuration files:

```
cd path\to\terraform\project
```

Run TFSec:

```
tfsec
```

Review the output for any security issues.

Using Checkov

Navigate to the directory containing your Terraform configuration files:

```
cd path\to\terraform\project
```

Run Checkov:

```
checkov --directory .
```

Review the report for identified issues.

Terraform Version Control with Git / Github

Managing Terraform configurations with Git and GitHub helps you collaborate, track changes, and maintain a history of your infrastructure as code (IaC). Here's a guide to using Git/GitHub for Terraform version control.

1. Set Up Your Terraform Project

Organize Terraform Files Create a directory for your Terraform project and organize it with the following structure:

```
terraform-project/
├── main.tf        # Core configuration
├── variables.tf   # Input variables
├── outputs.tf     # Outputs
├── backend.tf     # Remote state configuration (optional)
├── .gitignore     # Files to ignore in Git
```

Initialize Terraform:

```
terraform init
```

2. Initialize a Git Repository

Navigate to your Terraform project directory:

```
cd terraform-project
git init
```

Add a .gitignore file to exclude sensitive or unnecessary files:

```
echo ".terraform/" >> .gitignore
echo "terraform.tfstate" >> .gitignore
echo "terraform.tfstate.backup" >> .gitignore
echo ".terraform.lock.hcl" >> .gitignore
```

Stage and commit your files:

```
git add .
git commit -m "Initial commit"
```

***Quick review of Git Commands**

Basic Commands

Command	Description
git init	Initialize a new Git repository in the current directory.
git clone <repo-url>	Clone an existing repository to your local machine.
git status	Show the current state of the working directory and staging area.
git add <file>	Stage changes for the next commit (use . to add all changes).
git commit -m "message"	Commit the staged changes with a message.
git log	Show commit history.

Branching and Merging

Command	Description
git branch	List all branches.
git branch <branch-name>	Create a new branch.
git checkout <branch-name>	Switch to a different branch.
git checkout -b <branch-name>	Create and switch to a new branch.

git merge <branch-name>	Merge a branch into the current branch.
git branch -d <branch-name>	Delete a branch.

Staging and Changes

Command	Description
git diff	Show changes in unstaged files.
git diff --staged	Show changes in staged files.
git reset <file>	Unstage a file (move from staged to unstaged).
git checkout <file>	Revert changes in the working directory to the last commit.

Remote Repositories

Command	Description
git remote add origin <url>	Add a remote repository.
git push origin <branch>	Push changes to a remote repository.
git pull origin <branch>	Fetch and merge changes from a remote repository.
git fetch	Fetch changes from a remote repository without merging.

Undoing Changes

Command	Description
git reset --soft HEAD~1	Undo the last commit but keep changes staged.
git reset --mixed HEAD~1	Undo the last commit and unstage changes.
git reset --hard HEAD~1	Undo the last commit and discard changes.
git revert <commit-hash>	Create a new commit that reverses a specific commit.

Viewing and Comparing

Command	Description
git log --oneline	Show commit history in a compact format.
git show <commit-hash>	Show details of a specific commit.
git blame <file>	Show which commit last modified each line of a file.

Cleaning Up

Command	Description
git clean -f	Remove untracked files.
git clean -fd	Remove untracked files and directories.

3. Push to GitHub

a) Create a new repository on GitHub.
b) Link your local repository to the GitHub repository:

```
git remote add origin https://github.com/<your-username>/<repository-name>.git
```

c) Push your code to GitHub:

```
git branch -M main
git push -u origin main
```

4. Collaborate with GitHub

Branching Strategy:

Use branches to work on new features or updates without affecting the main branch.

```
git checkout -b feature/add-s3-bucket
```

Pull Requests:
Create pull requests on GitHub for code review and collaboration.

Commit Best Practices:
Write meaningful commit messages to describe your changes.

5. Manage Terraform State with Git

Never store Terraform state files (terraform.tfstate, .tfstate.backup) in Git. These files may contain sensitive data. Use a remote backend for state management.

Example using S3:

```
terraform {
  backend "s3" {
    bucket       = "my-terraform-state"
    key          = "prod/terraform.tfstate"
    region       = "us-west-2"
    encrypt      = true
  }
}
```

6. Automate with GitHub Actions (CI/CD)

a) Create a GitHub Workflow Add a .github/workflows/terraform.yml file to automate Terraform tasks like plan and apply.

```
name: Terraform

on:
  push:
    branches:
      - main

jobs:
  terraform:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.5.6

      - name: Terraform Init
```

```
      run: terraform init

    - name: Terraform Plan
      run: terraform plan

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

b) Secure Secrets: Store sensitive information (e.g., AWS credentials) in GitHub Secrets:
   a. Go to your GitHub repository > Settings > Secrets and Variables > Actions.
   b. Add secrets like AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY.
   c. Modify your workflow to use these secrets:

```
- name: Setup AWS Credentials
  uses: aws-actions/configure-aws-credentials@v2
  with:
    aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
    aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    aws-region: us-west-2
```

Terraform Cloud

Setup:

1. Create a terraform cloud account https://app.terraform.io/
2. Sign-in and create organization
3. Create workspace
4. In your local machine (with Terraform installed and where Terraform Configurations will be created), run:

```
terraform  login
```

5. Create an API Access Token (note of the token, it will be required later)
6. Back to your cmd prompt, enter your API Access Token.
7. You can now create your project and add sections in your terraform config that connects that project to your cloud like the following:

   data.tf

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

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }
}
```

   main.tf

```
terraform {

  cloud {
    organization = "jxxxxxxh-training"
```

```
  workspaces {
    name = "jxxxxxxh-workspace"
  }
 }
}

provider "aws" {
 region = "us-east-1"
 #  shared_credentials_files = ["/Users/core360/.aws/credentials"]
 #  profile                  = "default"
 access_key = "AxxxxxxxxZE5GYJ7"
 secret_key = "fUxxxxxxxxxjBaTOpSZELxBQZTNo"
}

resource "aws_instance" "testserver2" {
 ami           = data.aws_ami.amznlinux.id
 instance_type = "t2.micro"
 tags = {
  Name = "testserver2"
 }

}
```

8. Terraform will store the token in plain text in the following file for use by subsequent commands:

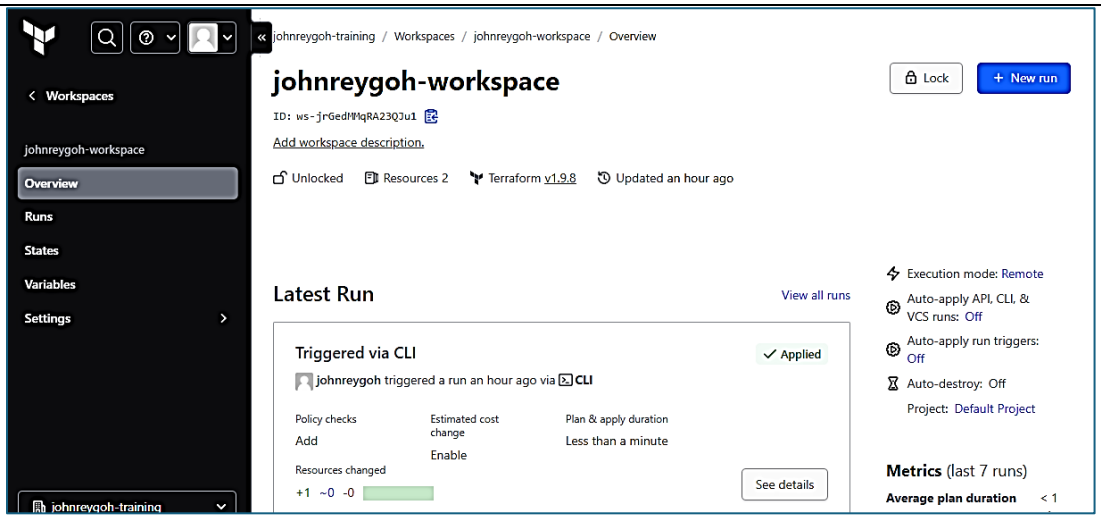
```
C:\Users\<username>\AppData\Roaming\terraform.d\credentials.tfrc.json
```

9. If you used s3 / local state prior to using terraform cloud, it should be migrated:
    a. If using ***backend*** block in ***terraform*** block, you have to remove it since you can only use either backend or cloud (but not both)
    b. Re-initialize the project

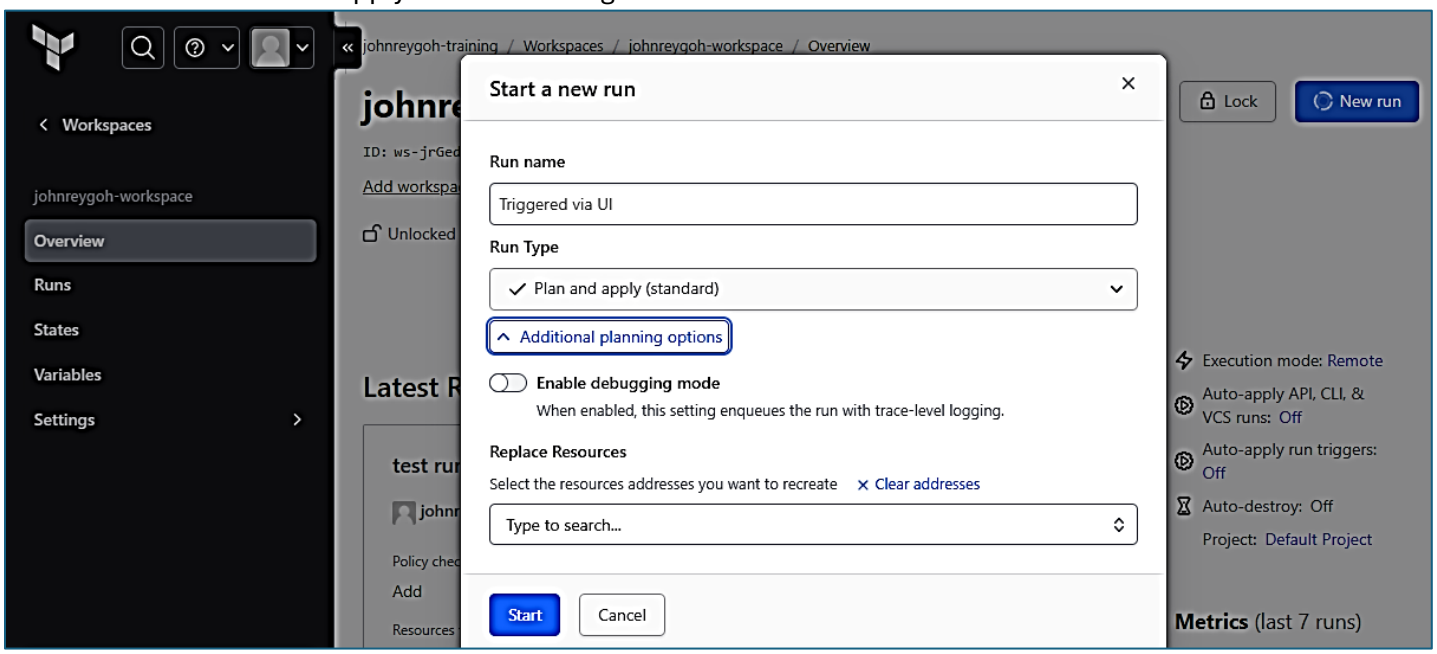
```
\> terraform init
```

10. Apply and check the terraform cloud status in your workflow

```
\> terraform apply
```



11. You can re-run / re-apply terraform config from the cloud ui



12. You can also execute resource removal (terraform destroy) from the cloud ui