

## Experiment

**Aim :** To construct change and destroy AWS resources using terraform

### Theory:

Infrastructure as Code (IaC) is a modern DevOps practice that involves managing and provisioning computing infrastructure through machine-readable configuration files, rather than through physical hardware configuration or interactive configuration tools. Terraform, developed by HashiCorp, is one of the most widely used IaC tools that enables users to define both low-level (e.g., EC2, networking) and high-level (e.g., DNS, load balancers) components of infrastructure.

Terraform allows users to write configuration files using HashiCorp Configuration Language (HCL) or JSON. These files describe the desired state of the infrastructure. When Terraform is run, it compares the current state of infrastructure with the configuration, and determines the actions needed to align the infrastructure with the declared configuration.

### Key Features of Terraform:

- **Declarative Syntax:** Users declare *what* resources are needed, and Terraform figures out *how* to create them.
- **Execution Plan:** Before making changes, Terraform shows a preview (using `terraform plan`) of what it will do.
- **State Management:** Terraform keeps track of infrastructure using a state file, which records what resources it manages.
- **Provisioning and Tearing Down:** With `terraform apply`, resources are created or modified. With `terraform destroy`, they are safely deleted.
- **Idempotency:** Running the same configuration multiple times results in the same infrastructure state.
- **Cloud Provider Support:** Terraform supports multiple providers like AWS, Azure, GCP, Kubernetes, etc., enabling multi-cloud strategies.

### How Terraform Works with AWS:

To use Terraform with AWS, the user must configure credentials—usually through access keys generated by creating an IAM user with programmatic access. Once configured, users write `.tf` files that specify AWS services like EC2 instances, VPCs, security groups, and more.

The typical workflow includes:

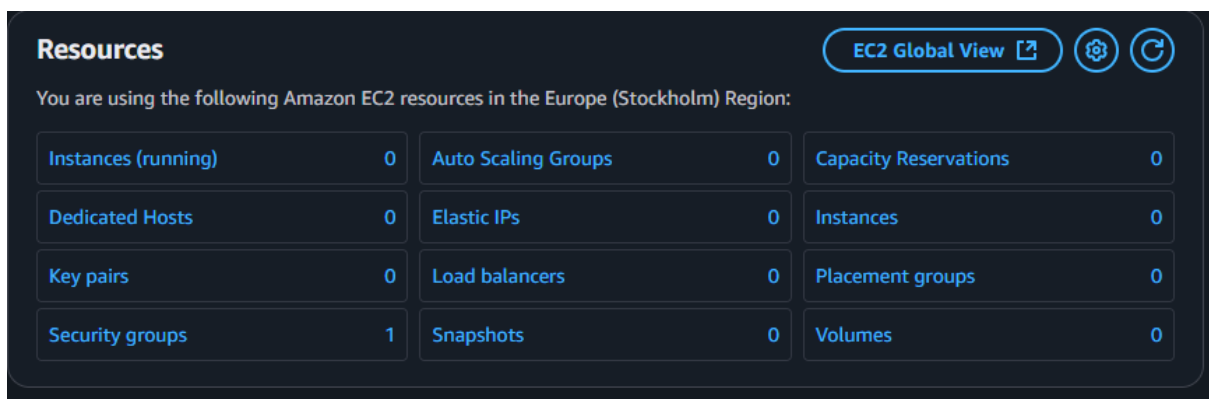
1. **Initialization** (`terraform init`): Prepares the working directory and downloads necessary provider plugins.
2. **Planning** (`terraform plan`): Shows the actions Terraform will take to achieve the desired state.
3. **Applying** (`terraform apply`): Applies the changes and provisions the resources.
4. **Destroying** (`terraform destroy`): Removes all infrastructure managed by Terraform when it's no longer needed.

## Advantages of Using Terraform:

- Automation reduces the risk of human error.
- Reusability of configuration files improves productivity.
- Infrastructure changes are version-controlled like code.
- Enables consistent environments across development, staging, and production.
- Useful in CI/CD pipelines for automated infrastructure deployment.

In this experiment, Terraform is used to automate the deployment and removal of an EC2 instance in AWS. This showcases Terraform's ability to manage cloud infrastructure reliably and efficiently.

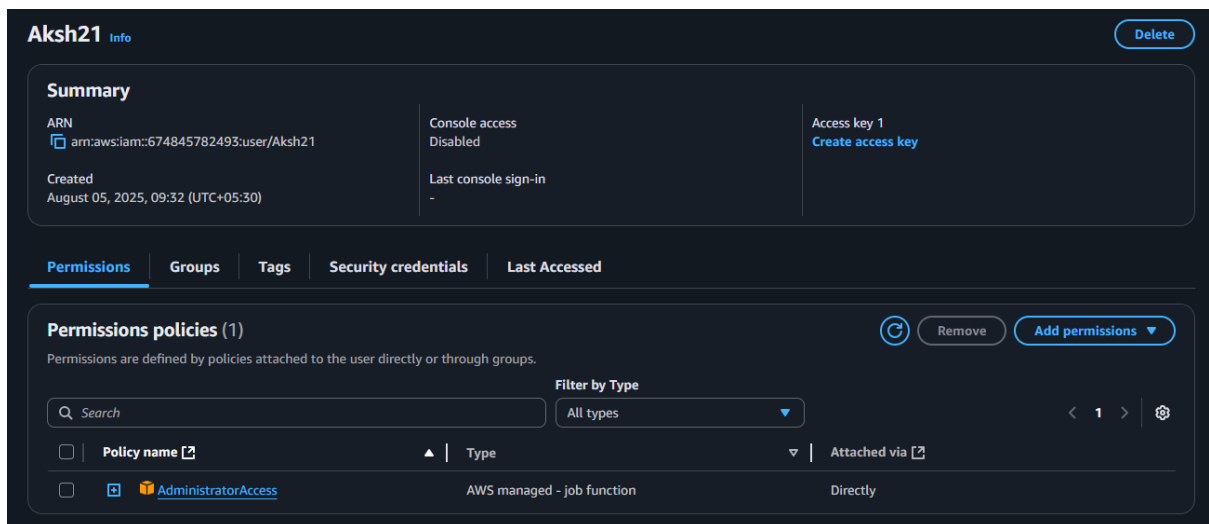
### Step 1 : Check that no instance is running on EC2



The screenshot shows the 'Resources' page in the AWS Management Console for the Europe (Stockholm) Region. It displays a grid of resource counts for various EC2-related services. The counts are as follows:

Resource Type	Count
Instances (running)	0
Auto Scaling Groups	0
Capacity Reservations	0
Dedicated Hosts	0
Elastic IPs	0
Instances	0
Key pairs	0
Load balancers	0
Placement groups	0
Security groups	1
Snapshots	0
Volumes	0

### Step 2 : Create IAM user with programmatic password and administrator access



The screenshot shows the 'Aksh21' user page in the AWS IAM console. The user is an IAM user with the ARN 'arn:aws:iam::674845782493:user/Aksh21'. The console access is disabled, and the last console sign-in is '-'. There is one access key attached, labeled 'Access key 1'. The 'Permissions' tab is selected, showing that the user has one policy attached: 'AdministratorAccess'. The policy is AWS managed and attached directly.

Policy name	Type	Attached via
AdministratorAccess	AWS managed - job function	Directly


### Step 3 : Create access key and secret key command line interface


### Access key

If you lose or forget your secret access key, you cannot retrieve it. Instead, create a new access key and make the old key inactive.

Access key

Secret access key

 AKIAZ2H7ROHOSD4AI2HJ

 \*\*\*\*\* [Show](#)

### Step 4 : Write a terraform program

```
AWSexp.tf
1  provider "aws" {
2      access_key = "AKIAZ2H7ROHOSD4AI2HJ"
3      secret_key = "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"
4      region = "us-east-1"
5  }
6
7  resource "aws_instance" "terra_xie" {
8      ami = "ami-0b59aaac1a4f1a3d1"
9      instance_type = "t3.micro"
10 }
```

### Step 5 : Initialize the terraform by command `terraform init`

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\XIE21>terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v6.7.0...
- Installed hashicorp/aws v6.7.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.

C:\XIE21>
```

### Step 6 : Run the command terraform plan

```
C:\XIE21>terraform plan

Planning failed. Terraform encountered an error while generating this plan.

Error: invalid AWS Region: us-north-1

with provider["registry.terraform.io/hashicorp/aws"],
on AWSexp.tf line 1, in provider "aws":
  1: provider "aws" {
```

**Step 7 :** Check the instance on EC2 before the apply command

**Resources**
[EC2 Global View](#)

You are using the following Amazon EC2 resources in the Europe (Stockholm) Region:

Instances (running)	0	Auto Scaling Groups	0	Capacity Reservations	0
Dedicated Hosts	0	Elastic IPs	0	Instances	0
Key pairs	0	Load balancers	0	Placement groups	0
Security groups	1	Snapshots	0	Volumes	0

**Step 8 :** Run the command terraform apply

```
C:\XIE21>terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the
following symbols:
+ create

Terraform will perform the following actions:

# aws_instance.terra_xie will be created
+ resource "aws_instance" "terra_xie" {
+   ami               = "ami-0c9fb5d338f1eec43"
+   arn               = (known after apply)
+   associate_public_ip_address = (known after apply)
+   availability_zone = (known after apply)

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes



aws_instance.terra_xie: Creating...
aws_instance.terra_xie: Still creating... [00m10s elapsed]
aws_instance.terra_xie: Creation complete after 16s [id=i-06954e48d7c3d531d]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

**Step 9:** Check terraform create instance on EC2

Resources

EC2 Global View



You are using the following Amazon EC2 resources in the United States (N. Virginia) Region:

Instances (running)	1	Auto Scaling Groups	0	Capacity Reservations	0
Dedicated Hosts	0	Elastic IPs	0	Instances	1
Key pairs	0	Load balancers	0	Placement groups	0
Security groups	1	Snapshots	0	Volumes	1

## Step 10: Now Destroy the instance from command prompt by writing the command terraform destroy

```
C:\XIE21>terraform destroy
aws_instance.terra_xie: Refreshing state... [id=i-06954e48d7c3d531d]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
- destroy

Terraform will perform the following actions:

# aws_instance.terra_xie will be destroyed
- resource "aws_instance" "terra_xie" {
  - ami                  = "ami-0c9fb5d338f1eec43" -> null
  - arn                  = "arn:aws:ec2:us-east-1:674845782493:instance/i-06954e48d7c3d531d" -> null
  - associate_public_ip_address = true -> null
  - availability_zone      = "us-east-1a" -> null
  - disable_api_stop       = false -> null
  - disable_api_termination = false -> null
  - ebs_optimized          = false -> null
  - get_password_data       = false -> null
  - hibernation            = false -> null
  - id                    = "i-06954e48d7c3d531d" -> null
  - instance_initiated_shutdown_behavior = "stop" -> null
  - instance_state         = "running" -> null
  - instance_type          = "t3.micro" -> null
```

**Instances** Info
 Connect
Instance state ▾
Actions ▾
Launch instances ▾

All states ▾

Instance state = running X
Clear filters
< 1 > ⚙️

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
No matching instances found							

**Conclusion:** In this experiment, we leveraged Terraform to create, update, and tear down AWS infrastructure. Using a declarative approach, we successfully deployed an EC2 instance and observed how Terraform applies changes in a controlled and incremental way. When the resources were no longer required, they were cleanly removed through automation. This exercise highlighted Terraform's effectiveness in simplifying and managing cloud infrastructure workflows.