Terraform

Terraform

- What is Terraform and how it works (History and advantages)?
- Terraform installation
- Terraform alternatives
- Terraform - syntax.
- Terraform Commands (init, plan, apply, state list, format, validate, destroy, -auto-approve, -targets).
- Terraform - Practical.
- Terraform .tfstate file
- Creating AWS Resources (IAM, S3 Bucket, VPC)
- Variables in Terraform.

What is Terraform and how it works (History and advantages)?

What is Terraform and how it works (History and advantages)?

- Terraform is an Infrastructure as a Code (IAAC) tool developed by Mitchel Hashimoto in 2014.
- It was developed in “Go” language. We use HCL (HashiCorp Configuration Language) to create the infra.
- Terraform Simplified multi-cloud deployments with a single tool.

Why Terraform (advantages)?

Why Terraform (advantages)?

- Terraform is a tool used to make infrastructure automation, It is free (but, not open source), easy to understand and platform independent.

- It has many advantages..

    - ❖ Reusability
    - ❖ Time saving
    - ❖ Automation
    - ❖ Avoiding mistakes
    - ❖ Multi Cloud Support

Terraform installation

Terraform installation

Windows

- Go to the Terraform website: https://developer.hashicorp.com/terraform/downloads
- Download the Windows zip for your system (usually 64-bit).
- Unzip the contents into a folder (e.g., C:\terraform)
- Add that folder to your system PATH:
- Search “Environment Variables” > Open "Edit the system environment variables"
- Click Environment Variables
- Under System variables > find “Path” > click Edit
- Click New > paste C:\terraform
- Open command prompt → terraform –v (Response: Terraform <<version>>)

Amazon Linux

- Go to the Terraform website: https://developer.hashicorp.com/terraform/downloads
- https://developer.hashicorp.com/terraform/install

  Go to Linux → package manager.
- Copy the package according to your Linux distribution.
- Go to your instance, paste and run the commands.

Terraform - syntax.

Terraform - syntax.

Syntax:

```
provider “<<provider>>" {
region = “<<region id>>"
}

resource “<<resource type>>" “<<Resource Name (Logical Name)>>"
    {
    <<Arguments (Configuration Settings)>>
    }
```

Example:

```
provider "aws" {
    region = "ap-south-2"
}

resource "aws_instance" "ccitinst"
    {
        ami = "ami-0a19d948cf5ce40b2"
        instance_type = "t3.micro"
    }
```

Terraform – Resource components.

In Terraform, a **resource** is a fundamental building block that manages AWS infrastructure components like S3 buckets, EC2 instances, VPCs, and more.

Key Components of a Terraform Resource

1. Resource Type

The type of AWS resource being created.
Example: "aws_s3_bucket", "aws_instance", "aws_vpc".

2. Resource Name (Logical name)

A unique identifier for referencing the resource within Terraform.
Example: "my_bucket“

3. Arguments

Define properties and settings for the resource.

Terraform alternatives

Terraform alternatives

Terraform Commands.

Terraform Commands.

1. terraform init
2. terraform plan
3. terraform apply -auto-approve
4. terraform state list
5. terraform fmt
6. terraform validate
7. terraform destroy -auto-approve
8. terraform destroy --auto-approve -target="aws_instance.one[1]"

Terraform Commands.

Initialize (terraform init):

Downloads provider plugins (e.g., AWS, Azure, GCP).
Sets up the backend for storing Terraform state. it will store information of plugins in .terrafrom folder

Plan (terraform plan):

Shows what Terraform will create, update, or delete before applying changes.
Helps avoid accidental modifications.

Apply (terraform apply):

Executes the planned changes and provisions the infrastructure.

`-auto-approve`
This option allows Terraform to apply changes without requiring manual confirmation.

Terraform Commands.

terraform state list

- The “terraform state list” command displays all managed resources in the Terraform state file.
- It helps you identify the resources Terraform is tracking.

Destroy (terraform destroy):

Destroys all resources defined in the configuration.

`-auto-approve`
This option allows Terraform to destroy the resources without requiring manual confirmation.

terraform Validate (Validate Syntax & Configuration)

- Checks if the Terraform configuration is syntactically correct and valid.
- Detects errors like missing arguments, incorrect types, or invalid providers.

Terraform Format – Terraform fmt

- Automatically formats Terraform configuration files (.tf and .tfvars) to follow best practices.
- Ensures consistent indentation, spacing, and style. It will be applied to all files in current folder.

`--recursive`

- [illegible]

```
provider "aws" {
region = "us-east-1"
}

resource "aws_instance" "one" {
count = 5
ami = "ami-03eb6185d756497f8"
instance_type = "t2.micro"
}
```

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "one" {
  count         = 5
  ami           = "ami-03eb6185d756497f8"
  instance_type = "t2.micro"
}
```

Terraform Commands.

terraform Validate (Validate Syntax & Configuration)

- Checks if the Terraform configuration is syntactically correct and valid.
- Detects errors like missing arguments, incorrect types, or invalid providers.

terraform target

The -target flag allows you to apply or destroy specific resources instead of the entire infrastructure. This is useful when you want to update or destroy only one resource without affecting others.

Apply a specific resource:

```
terraform apply -target=aws_instance.example
```

- This updates only aws_instance.example without affecting other resources.

```
terraform destroy -target=aws_instance.example
```

- This deletes only the aws_s3_bucket.my_bucket resource.

When to Use -target?

✅ When testing changes for a single resource.
✅ When debugging issues with a specific resource.
🚫 Not recommended for regular use, as it may lead to dependency issues.

Terraform - Practical.

Terraform “.tfstate” file

Terraform ".tfstate" file

- The Terraform state file (terraform.tfstate) is a JSON file that stores information about the infrastructure Terraform manages.

Purpose of .tfstate

1. Tracks the current state of resources deployed using Terraform.
2. Maps Terraform configurations to real-world resources in AWS, Azure, etc.
3. Speeds up Terraform operations by avoiding unnecessary changes.

Key Contents of .tfstate

1. Resource Details: Stores attributes like instance ID, AMI, security groups, etc.
2. Dependencies: Tracks relationships between resources.
3. Metadata: Contains information about the Terraform version used.

Terraform ".tfstate" file

- The Terraform state file (terraform.tfstate) is a JSON file that stores information about the infrastructure Terraform manages.

Purpose of .tfstate

1. Tracks the current state of resources deployed using Terraform.
2. Maps Terraform configurations to real-world resources in AWS, Azure, etc.
3. Speeds up Terraform operations by avoiding unnecessary changes.

Key Contents of .tfstate

1. Resource Details: Stores attributes like instance ID, AMI, security groups, etc.
2. Dependencies: Tracks relationships between resources.
3. Metadata: Contains information about the Terraform version used.

Terraform ".tfstate" file

Important Notes

1. **Do NOT manually edit .tfstate** unless necessary.
2. **Should be stored securely** (contains sensitive data).
3. **For teams, use remote state storage** (e.g., S3, Terraform Cloud) to avoid conflicts.

json


```
{
  "resources": [
    {
      "type": "aws_instance",
      "name": "ccitinst",
      "instances": [
        {
          "attributes": {
            "ami": "ami-0a19d948cf5ce40b2",
            "instance_type": "t3.micro",
            "id": "i-1234567890abcdef"
          }
        }
      ]
    }
  ]
}
```

Terraform Comments

Terraform Comments

Terraform supports comments to improve readability and document your Infrastructure as Code (IaC).

There are two types of comments in Terraform:

1. Single-Line Comments (# or //)

2. Multi-Line Comments (/* */)

These comments are ignored by Terraform and do not affect execution.

Terraform Comments

Terraform supports comments to improve readability and document your Infrastructure as Code (IaC).

1. Single-Line Comments (# or //)

Used for brief comments on a single line.

```
# This is a single-line comment

// Another way to write a single-line comment

resource "aws_instance" "example" {
  ami           = "ami-12345678"  # AMI ID for EC2 instance
  instance_type = "t2.micro"
}
```

Terraform Comments

Terraform supports comments to improve readability and document your Infrastructure as Code (IaC).

2. Multi-Line Comments (/* ... */)

Used when you need to comment out multiple lines of code.

```
/* This is a multi-line comment. It can span multiple
lines and is useful for large descriptions. */

resource "aws_s3_bucket" "example" {
 bucket = "my-terraform-bucket"
}
```

Variables in Terraform.

Terraform

- Variables in Terraform.
- Terraform ".tfvars" files
- Terraform Output block
- Terraform Taint and replace
- Terraform Import

- Creating AWS Resources (EC2, EBS and RDS)

Variables in Terraform.

Terraform variables let you **customize** your infrastructure without hardcoding values. They make your code **reusable and flexible**.

Types of Variables

String → Text values

```
variable "instance_type"
{
  type    = string
  default = "t2.micro"
}
```

Number → Numeric values

```
variable "instance_count"
{
  type    = number
  default = 2
}
```

Boolean → True/False values

```
variable
"enable_monitoring" {
  type    = bool
  default = true
}
```

Variables in Terraform.

Terraform variables let you **customize** your infrastructure without hardcoding values. They make your code **reusable and flexible**.

Types of Variables

List → Ordered collection of values

```
variable "instance_types" {
  type    = list(string)
  default = ["t2.micro", "t3.micro"]
}
```

Map → Key-value pairs

```
variable "ami_ids" {
  type = map(string)
  default = {
    "us-east-1" = "ami-12345"
    "ap-south-1" = "ami-67890"
  }
}
```

Terraform “.tfvars” files

Using .tfvars Files

- Terraform allows you to use multiple .tfvars files to manage different environments like dev, staging, and production without changing the main configuration files.

Command:

```
terraform apply -var-file="prod.tfvars"
```

When to Use .tfvars files?

✅ Helps manage multiple environments easily.

✅ Prevents accidental changes in production.

✅ Improves Terraform configuration modularity.

Terraform Output block

Terraform Output block

- In Terraform, outputs allow you to display or pass values from your infrastructure deployment.
- Outputs are useful for retrieving details like VPC IDs, instance public IPs, S3 bucket names, and more after applying your Terraform configuration.

How to Define Outputs

```
output "vpc_id" {
      description = "The ID of the created VPC"
      value = aws_vpc.my_vpc.id
}
output "s3_bucket_name" {
      description = "The name of the created S3
      bucket" value = aws_s3_bucket.my_bucket.id
}
```

Terraform Taint and replace

Terraform Taint

- It allows for you to manually mark a resource for recreation.
- in real time sometimes resources fails to create so to recreate them we use taint
- in new version we use -replace option
- TO TAINT: terraform taint aws_instance.one[0]
- TO UNTAINT: terraform untaint aws_instance.one[0]
    - terraform state list
    - terraform taint aws_instance.one[0]
    - terraform apply --auto-approve
    - TERRAFORM REPLACE: terraform apply --auto-approve -replace="aws_instance.one[0]"

Terraform Import

Terraform Import

- When we create resource manually terraform wont track that resource.
- Import command can used to import the resource which is created manually.
- can only import one resource at a time (FROM CLI).
- It can import both code to config file and state file.
- FOR STATE FILE: terraform import aws_instance.one (not preferable).
- FOR STATEFILE & CODE:
    - terraform plan -generate-config-out=manual.tf
    - terraform apply -auto-approve

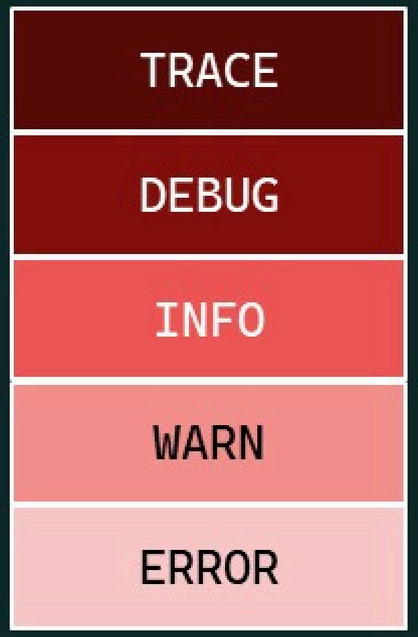
Terraform

- Terraform Debugging
- Alias & providers
- Terraform Parallelism.
- Terraform Dependency.

Terraform Debugging

Terraform Debugging

- Debugging helps understand issues (like misconfigurations or dependency errors can occur) and fix them efficiently.
- You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs, with TRACE being the most verbose.



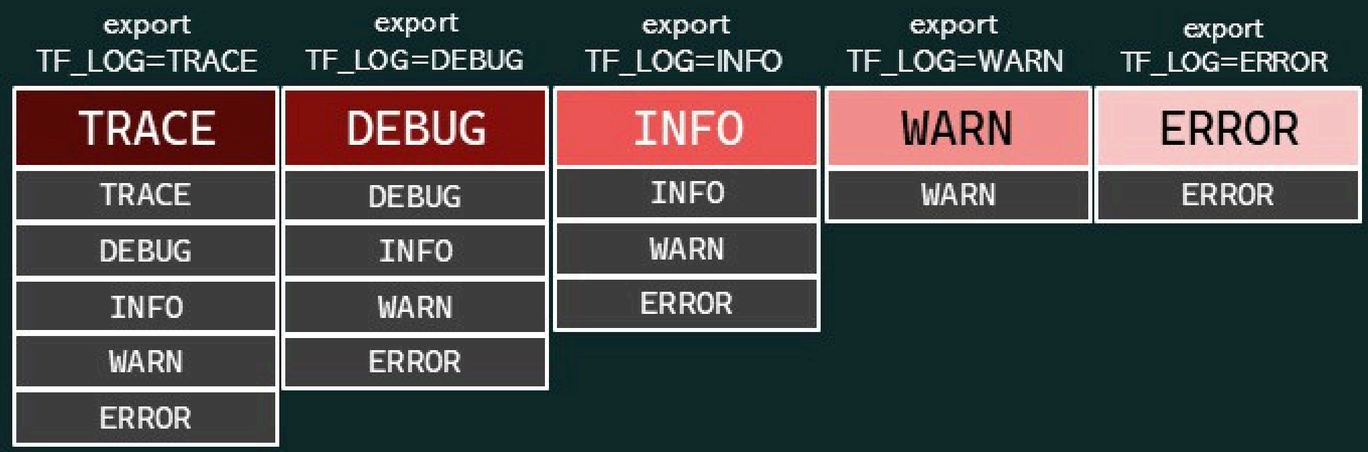
Terraform Debugging

To set LOGs:

(Linux/macOS) export TF_LOG=TRACE / (Windows) $env:TF_LOG="DEBUG"

(Linux/macOS) export TF_LOG_PATH="logs.txt“/ (Windows) $env:TF_LOG_PATH="log.txt"

terraform apply



To Unset LOGs:

(Linux/macOS) unset TF_LOG / (Windows) Remove-Item Env:TF_LOG

(Linux/macOS) unset TF_LOG_PATH / (Windows) Remove-Item Env:TF_LOG_PATH

Alias and Providers

Terraform

Alias and Providers

- In Terraform, providers are responsible for managing and interacting with external services (like AWS, Azure, GCP, etc.).
- Aliases allow you to define multiple configurations of the same provider within a single Terraform configuration.

What is a Provider?

- A provider is a plugin that enables Terraform to interact with an external service.

```
provider "aws"
    {
        region = “ap-south-2"
    }
```

- Here, Terraform downloads the AWS provider and sets the region to ap-south-2.

Why Use an Alias?

- The alias argument allows multiple configurations of the same provider. This is useful when:

1. You can deploy resources in multiple AWS regions.

2. You can deploy in multiple AWS accounts.

3. You can deploy to different service providers.

Terraform

1. You need to deploy resources in multiple AWS regions.

We can write the script to create resources in different AWS accounts (here different account profiles should set).

```
provider "aws" {
  region  = "ap-south-2"
  profile = "default"
}

provider "aws" {
  alias   = "prod"
  region  = "ap-south-1"
  profile = "production"
}

resource "aws_s3_bucket" "dev_bucket" {
  provider = aws
  bucket   = "ccit-dev-bucket"
}

resource "aws_s3_bucket" "prod_bucket" {
  provider = aws.prod
  bucket   = "ccit-prod-bucket"
}
```

Terraform

2. You need to use multiple AWS accounts.

We can write the script to create resources in different AWS accounts (here different account profiles should set).

```
provider "aws" {
  region = "ap-south-2"
}

provider "aws" {
  alias  = "mumbai"
  region = "ap-south-1"
}

resource "aws_s3_bucket" "ccithyd" {
  provider = aws
  bucket   = "ccithydbucket123"
}

resource "aws_s3_bucket" "ccitmumbai" {
  provider = aws.mumbai
  bucket   = "ccitmumaibucket123"
}
```

AWS Configure - Add default profiles
vim ~/.aws/credentials
Add different profile

Terraform

3. You want different credentials for different services.

We can create resources in different clouds (AWS, Azure, GCP etc..)

```
# AWS Provider
provider "aws" {
  region = "us-east-1"
}

# Azure Provider
provider "azurerm" {
  features {}
}

# AWS S3 Bucket
resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-terraform-s3-bucket"
  acl    = "private"

  tags = {
    Name        = "MyS3Bucket"
    Environment = "Dev"
  }
}
```

```
# Azure Resource Group
resource "azurerm_resource_group" "rg" {
  name     = "MyResourceGroup"
  location = "East US"
}

# Azure Storage Account
resource "azurerm_storage_account" "storage" {
  name                     = "mystorageaccount123"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}

# Azure Blob Storage Container
resource "azurerm_storage_container" "container" {
  name                  = "mycontainer"
  storage_account_name  = azurerm_storage_account.storage.name
  container_access_type = "private"
}
```

Terraform workspaces

Terraform Workspaces

- Terraform workspaces allow you to maintain multiple environments (e.g., development, testing, production)
- using the same Terraform configuration but with different states.
- Each workspace has its own state file (terraform.tfstate.d folder)
- resources created/managed in one workspace do not affect other workspaces.
- the default workspace in Terraform is default

Terraform Workspaces

- terraform workspace list : to list the workspaces.
- terraform workspace new <<work space name>> : to create workspace.
- terraform workspace show : to show current workspace.
- terraform workspace select <<work space name>> : to switch to <<work space name>> workspace.
- terraform workspace delete <<work space name>> : to delete <<work space name>> workspace.

Note

- We need to empty the workspace before delete.
- We can’t delete current workspace, we can switch and delete.
- We can’t delete default workspace.

Parallelism

Terraform

Parallelism

- Terraform executes resource creation, updates, and deletions in parallel whenever possible to speed up deployment.
- However, it also respects dependencies between resources, ensuring that dependent resources are processed in the correct order.
- it will execute all the resources at a time. by default parallelism limit is 10.
- terraform apply -auto-approve -parallelism=1

NOTE: IT WILL APPLICALBLE FOR BOTH APPLY & DESTROY.

Dependency

Terraform

Dependency

The depends_on argument in Terraform is used to explicitly define dependencies between resources. This ensures that one resource is created, updated, or destroyed only after another resource has been properly handled.

- We use “depends_on” keyword to implement explicit dependency.

Why Use depends_on?

Terraform automatically determines dependencies between resources based on references.

However, in some cases, dependencies are not directly referenced, and Terraform may try to create resources in parallel.

✅ A resource does not have a direct reference but must still wait for another resource.
✅ There are implicit dependencies, like provisioning **network resources before instances**.
✅ A resource needs to be updated **only after** another has changed.

Terraform

Example Use Cases

1 Creating an Instance Only After a Security Group is Ready

```
resource "aws_security_group" "sg" {
  name        = "example-sg"
  description = "Allow web traffic"
}

resource "aws_instance" "web" {
  ami           = "ami-09e23b3de35f110f6"
  instance_type = "t3.micro"

  depends_on = [aws_security_group.sg]
}
```

- Here, Terraform does **not** see a direct reference to aws_security_group.sg in aws_instance.web, so depends_on ensures Terraform creates the security group **before** launching the instance.

2 Ensuring a Load Balancer is Created Before Auto Scaling

```
resource "aws_lb" "my_lb" {
  name               = "my-load-balancer"
  internal           = false
  load_balancer_type = "application"
}
resource "aws_autoscaling_group" "asg" {
  desired_capacity = 2
  min_size         = 2
  max_size         = 5
   depends_on = [aws_lb.my_lb]
}
```

- Even though the ASG doesn't directly reference the load balancer, it must be created first.

Terraform

⚠ When Not to Use depends_on

- ❌ If Terraform already understands the dependency from resource references, you don't need depends_on.
- ❌ Using depends_on incorrectly can make Terraform slower by forcing unnecessary sequential execution
- ❌ If Terraform already understands the dependency from resource references, you don't need depends_on.

Why Use depends_on?

- depends_on forces Terraform to respect a specific dependency
- Use it only when Terraform does not automatically detect dependencies.
- It's useful for networking, load balancers, and security groups.
- Avoid unnecessary use to keep Terraform plans efficient.

Terraform

- State file management (Securing, locking)
- Terraform lifecycle (Create before destroy, Prevent destroy, Ignore changes)
- Terraform Modules

State file management (Securing, locking)

Simple Terraform structure

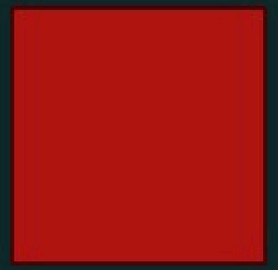
```
2. root@ip-172-31-1-147:~/CCIT
provider "aws" {
  region = "ap-south-2"
}

# Create IAM user
resource "aws_iam_user" "ccit_user" {
  name = "CCITDev1"
}

# Create S3 bucket with a dynamic name
resource "aws_s3_bucket" "ccit_bucket" {
  bucket = "ccits3jan"
}

# VPC
resource "aws_vpc" "CCITVPC" {
  cidr_block           = "10.0.0.0/22"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = { Name = "CCITVPC" }
}
```

CCIT Folder

- Cloudinfra.tf
- terraform.tfstate
- terraform.tfstate.backup

State file management (Securing, locking)

terraform.tfstate file

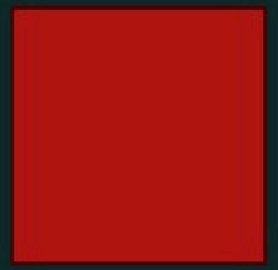


- Terraform Stores the infrastructure information on state file. it will automatically refresh when we run plan, apply & destroy.
- In Terraform, a backend is a configuration that determines how and where Terraform stores its state file & how it manages operations like apply, plan, and destroy.
- By default, Terraform uses a local backend. It stores the state file as terraform.tfstate in the local folder in JSON format.
- If we delete any resource, it stores information in terraform.tfstate.backup.

Terraform State File Locking.

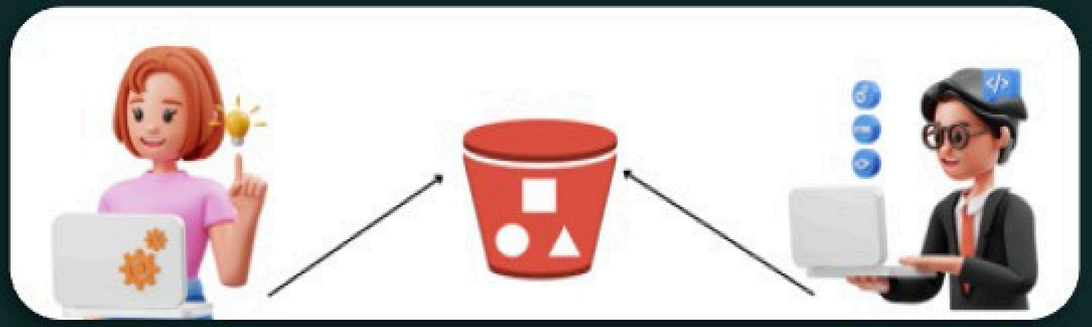
Terraform State File Locking.

terraform.tfstate file



- In Real time once we complete our work, we need to lock state file.
- It ensures that only one operation can be executed at a time.
- Once you lock state file you can't modify the infrastructure anymore.
- When two people working on state file at a time it will be locked automatically.
- Unfortunately, if two people run "apply" at same time, it can lead to unpredictable results, such as creating duplicate resources or destroying the wrong infrastructure.
- Not all Terraform backends support locking.

Why locking happened?

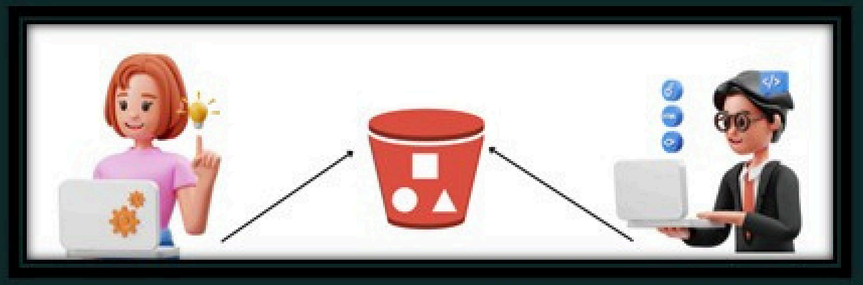


- when 2 developers work on the same project with same state file then the locking will be happened.
- if state file is locked only first operation execute and second operation waits
- to remove state lock use: terraform force-unlock <LOCK-ID>
- after adding DynamoDB run: terraform init -reconfigure

Backend	Supports Locking	Notes
Amazon S3 + DynamoDB	Yes	Requires DynamoDB table for locking.
Google Cloud Storage	Yes	Native support for state locking.
Azure Blob Storage	Yes	Native support for state locking.
HashiCorp Consul	Yes	Uses Consul's locking mechanism.
Terraform Cloud	Yes	Automatic locking with Terraform Cloud.
Alibaba Cloud OSS	Yes	Supports remote state locking.
Etcd	Yes	Distributed locking via etcd.
Local Backend	No	No support for locking (single-user only).
Git (as backend)	No	Not recommended, no locking support.

S3 backend setup for state file

S3 backend setup for state file



- Terraform used local backend to manage state file.
- But in that local backend only one person can be able to access it.
- in Real time it's often necessary to have a centralized, consistent, and secure storage mechanism for the state file
- Amazon S3 is a popular choice for this, and when combined with DynamoDB for locking, it ensures safe, consistent operations.

Advantages

- Global Access
- Team Collaboration
- Secure and Scalable
- Centralized State Management
- State File Versioning
- Disaster Recovery

S3 and DynamoDB configuration.

```
terraform {
  backend "s3" {
    bucket = "rahamterrabucketforstatefile"
    key    = "prod/terraform.tfstate"
    region = "us-east-1"
    dynamodb_table = "mytableforlock"
  }
}
```

- Add that block to existing code and run terraform init -upgrade
- DynamoDB -- > create table -- > Partition key: LockID -- > create
- now after apply state file will go to s3 bucket.
- dev-1 type destroy and dev-2 type apply now state file locked.
- you can check lock-id in new items of table.
- once destroy done for dev-1 state file will be unlocked and dev-2 can work.

Backup file location/ status

Backend Type	Where is tfstate stored?	How is backup handled?
Local Backend	Local directory	terraform.tfstate.backup
S3 Backend	AWS S3	Use **S3 Versioning**
Terraform Cloud	Terraform Cloud	Terraform manages backups

Simple Terraform structure

```
2. root@ip-172-31-1-147:~/CCIT
provider "aws" {
  region = "ap-south-2"
}

# Create IAM user
resource "aws_iam_user" "ccit_user" {
  name = "CCITDev1"
}

# Create S3 bucket with a dynamic name
resource "aws_s3_bucket" "ccit_bucket" {
  bucket = "ccits3jan"
}

# VPC
resource "aws_vpc" "CCITVPC" {
  cidr_block           = "10.0.0.0/22"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = { Name = "CCITVPC" }
}
```

CCIT Folder

- Cloudinfra.tf
- terraform.tfstate
- terraform.tfstate.backup

Terraform Modules

Simple Terraform structure

2. root@ip-172-31-1-147:~/CCIT

```hcl
provider "aws" {
  region = "ap-south-2"
}

# Create IAM user
resource "aws_iam_user" "ccit_user" {
  name = "CCITDev1"
}

# Create S3 bucket with a dynamic name
resource "aws_s3_bucket" "ccit_bucket" {
  bucket = "ccits3jan"
}

# VPC
resource "aws_vpc" "CCITVPC" {
  cidr_block           = "10.0.0.0/22"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = { Name = "CCITVPC" }
}
```

```hcl
provider "aws" {
  region = "ap-south-2"
}

# Create IAM user
module "iam_user" {
  source    = "./modules/iam"
}

# Create S3 bucket
module "s3_bucket" {
  source      = "./modules/s3"
}

# Create VPC
module "vpc" {
  source     = "./modules/vpc"
}

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.19.0"
}
```

```hcl
resource "aws_iam_user" "user" {
  name = "CCITDev1"
}
```

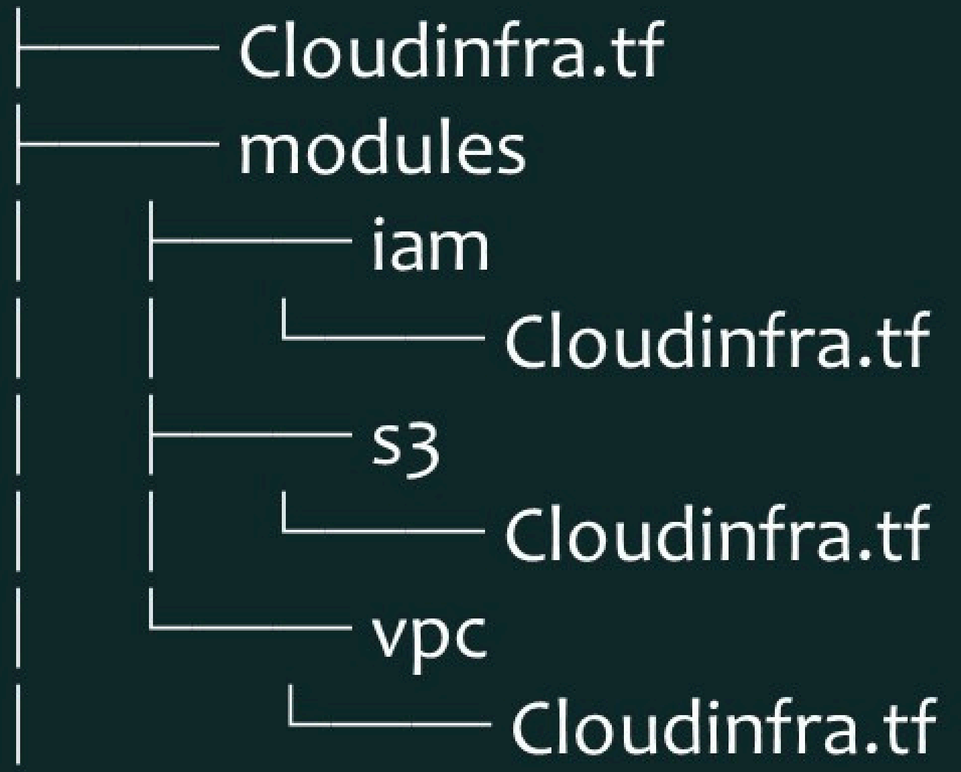
```hcl
resource "aws_s3_bucket" "bucket" {
  bucket = "ccits3jan"
}
```

```hcl
resource "aws_vpc" "this" {
  cidr_block           = "10.0.0.0/22"
  enable_dns_support   = true
  enable_dns_hostnames = true

  tags = {
    Name = "CCITVPC"
  }
}
```

Modules in Terraform

- It divides the code into folder structure.
- Modules are group of multiple resources that are used together.
- This makes your code easier to read and reusable across your organization.
- we can publish modules for others to use.
- each module will be having sperate plugin
- modules plugins will be store on .terraform/modules/

Modules in Terraform




```hcl
provider "aws" {
  region = "ap-south-2"
}

# Create IAM user
module "iam_user" {
  source    = "./modules/iam"
}

# Create S3 bucket
module "s3_bucket" {
  source      = "./modules/s3"
}

# Create VPC
module "vpc" {
  source     = "./modules/vpc"
}

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.19.0"
}
```

```hcl
resource "aws_iam_user" "user" {
  name = "CCITDev1"
}
```

```hcl
resource "aws_s3_bucket" "bucket" {
  bucket = "ccits3jan"
}
```

```hcl
resource "aws_vpc" "this" {
  cidr_block           = "10.0.0.0/22"
  enable_dns_support   = true
  enable_dns_hostnames = true

  tags = {
    Name = "CCITVPC"
  }
}
```

Types Modules

```
provider "aws" {
  region = "ap-south-2"
}

# Create IAM user
module "iam_user" {
  source    = "./modules/iam"
}

# Create S3 bucket
module "s3_bucket" {
  source      = "./modules/s3"
}

# Create VPC
module "vpc" {
  source     = "./modules/vpc"
}

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.19.0"
}
```

```
resource "aws_iam_user" "user" {
  name = "CCITDev1"
}
```

```
resource "aws_s3_bucket" "bucket" {
  bucket = "ccits3jan"
}
```

```
resource "aws_vpc" "this" {
  cidr_block           = "10.0.0.0/22"
  enable_dns_support   = true
  enable_dns_hostnames = true

  tags = {
    Name = "CCITVPC"
  }
}
```

Root Module: This is the main directory where Terraform commands are run. All Terraform configurations belong to the root module.

Child Modules: These modules are called by other modules.

Module Sources

- Modules can be sourced from a number of different locations, including both local and remote sources.
- we can get modules form Terraform Registry, HTTP URLs, S3 buckets & Local.

```
provider "aws" {
region = "us-east-1"
}

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.13.0"
}
```

Terraform life cycle

Terraform life cycle

Terraform Lifecycle: Managing Resource Behavior 🚀

Terraform provides a **lifecycle** block to control how resources are created, updated, and deleted. It helps prevent accidental destruction, force resource recreation, and ignore specific changes.

Meta-Argument	Description
create_before_destroy	Ensures a new resource is created **before** the old one is deleted.
prevent_destroy	Prevents Terraform from accidentally deleting a resource.
ignore_changes	Ignores specific attribute changes to avoid unnecessary updates.

Terraform

Terraform life cycle

create_before_destroy

✅ **Ensures a new resource is created before deleting the old one.**
🔹 **Use case:** When replacing a resource **without downtime**.

```
resource "aws_instance" "example"
{
     ami = "ami-12345678"
     instance_type = "t2.micro"

     lifecycle
          {
               create_before_destroy = true
          }
}
```

Behavior:

1. Terraform creates the new instance.
2. Once created, Terraform destroys the old one.
3. Prevents downtime since there is no gap between deletion and recreation.

Terraform

Terraform life cycle

prevent_destroy

- ✅ Prevents accidental deletion of critical resources.
- 🔹 **Use case:** Protecting **databases, S3 buckets, or production VMs**.

```
resource "aws_instance" "example"
{
     ami = "ami-12345678"
     instance_type = "t2.micro“

     lifecycle
          {
               prevent_destroy = true
          }
}
```

Behavior:

1. Terraform doesn't allow to destroy the instance

Terraform

Terraform life cycle

ignore_changes

✅ Ignores changes to specific attributes.

🔹 **Use case:** When an attribute changes outside Terraform (e.g., manual AWS console updates).

```
resource "aws_instance" "example"
{
ami = "ami-12345678"
instance_type = "t2.micro“

lifecycle
     {
       ignore_changes = [tags]
     }
tags = {
    Name = "Example-EC2"
  }
}
```

Behavior:

1. Terraform refuses the changes mentioned.

Terraform

- For each loop in Terraform
- Terraform locals
- Dynamic block
- Built-In functions
- Blocks, args & expressions
- Advantages & disadvantages

For each loop in Terraform

For each loop in Terraform

- The "for_each" in Terraform is used to loop over a map or a set to create multiple resources dynamically.
- For_each syntax:

```
resource "resource_type" "name" {
  for_each = <map or set>

  attribute = each.value   # Access the value
  another_attribute = each.key  # Access the key (only for maps)
}
```

List in Terraform

Terraform

List in Terraform

- A list in Terraform is an ordered collection of elements where each item has a specific position (index). Duplicates are allowed.

  - Syntax

```
variable "my_list" {
  type    = list(string)
  default = ["bucket-1", "bucket-2", "bucket-3"]
}
```

Set in Terraform

- A set in Terraform is an unordered collection of unique elements. Duplicates are not allowed.

  - Syntax

```
variable "my_set" {
  type    = set(string)
  default = ["bucket-1", "bucket-2", "bucket-3"]
}
```

Difference between List and Set

Terraform

Key Differences Between Lists and Sets

Feature	List ✅	Set ✅
Maintains order	✅ Yes	❌ No
Allows duplicates	✅ Yes	❌ No
Uses index (e.g., var.my_list[0])	✅ Yes	❌ No
Used with count	✅ Yes	❌ No
Used with for_each	❌ No	✅ Yes

Map

Terraform

Map

Used to pass both key and value to variables.

KEY-VALUE can also called as object or Dictionary.

```
variable "variable_name" {
  type    = map(string)
  default = {
    key1 = "value1"
    key2 = "value2"
  }
}
```

Terraform locals

Terraform

Terraform locals:

In Terraform, locals are used to define local values within a module. These local values help simplify configurations by assigning names to expressions, making them easier to reference throughout the configuration.

Why Use locals?

1. **Improve Readability** – Instead of repeating complex expressions, you can define them once and use them multiple times.
2. **Avoid Repetition** – Helps in maintaining the DRY (Don't Repeat Yourself) principle.
3. **Enhance Maintainability** – If a value needs to be updated, you only change it in one place.

```
# Define Local Variables
locals {
  bucket_name   = “ccitmarch25bucket"
  instance_type = "t3.micro"
  ami_id        = "ami-0150f9c33a6958e02"
}
```

```
# Create S3 Bucket
resource "aws_s3_bucket" "my_bucket" {
  bucket = local.bucket_name
  acl    = "private"

  tags = {
    Name = "MyS3Bucket"
  }
}
```

```
# Launch EC2 Instance
resource "aws_instance" "my_ec2" {
  ami           = local.ami_id
  instance_type = local.instance_type

  tags = {
    Name = "MyEC2Instance"
  }
}
```

Terraform

Terraform locals:

Feature	variable (input variable)	local (local value)
Purpose	Accept **external input** (from user, CLI, tfvars)	Define **internal reusable values**
Can be overridden?	✅ Yes — via CLI, tfvars, etc.	❌ No — fixed at definition time
Use case	Configure environment, region, name, etc.	Simplify expressions, avoid duplication
Where used	Module input	Module internal only
Change from outside?	✅ Yes	❌ No

🧠 Summary:

Use variable when:	Use local when:
You expect input from outside	You want to clean up internal logic
Config needs to be flexible	Computed values, reused expressions
Used across modules	Used only inside one module/file

Dynamic block

Dynamic block

Dynamic blocks in Terraform are a powerful feature that allow for the dynamic generation of nested blocks within resources, data sources, providers, or provisioners. Here's a concise overview:

1. **Purpose:** Dynamic blocks enable the creation of multiple nested configurations based on variable input, reducing code repetition and adhering to the DRY (Don't Repeat Yourself) principle

2. **Structure:** A dynamic block consists of a for_each argument to iterate over a collection and a content block that defines the structure to be dynamically generated.

3. **Common Use Cases:** Dynamic blocks are particularly useful for resources that require multiple nested blocks, such as defining multiple ingress rules in a security group or configuring multiple subnets within a VPC.

4. **Advantages:** They enhance code maintainability by reducing redundancy and allow for more flexible and scalable configurations.

5. **Limitations:** Overusing dynamic blocks can make configurations harder to read and maintain. It's essential to balance their use to avoid unnecessary complexity.

Terraform Built-in functions

Terraform Built-in functions

Terraform provides a variety of **built-in functions** to manipulate and transform values. These functions are mainly used in expressions and variables.

1. String Functions 🔤
2. Numeric Functions 🔢
3. Collection Functions (Lists & Maps) 📦
4. IP & CIDR Functions 🌐
5. Type Conversion Functions 🔄
6. Encoding & Hashing Functions 🔑
7. File & Path Functions 📁

Terraform

1 String Functions 🔤

Function	Description	Example
upper()	Converts a string to uppercase	upper("terraform") → "TERRAFORM"
lower()	Converts a string to lowercase	lower("TERRAFORM") → "terraform"
length()	Returns length of a string	length("Terraform") → 9
substr()	Extracts part of a string	substr("Terraform", 0, 4) → "Terr"
replace()	Replaces substrings	replace("hello world", "world", "Terraform") → "hello Terraform"
trimspace()	Removes spaces from start and end	trimspace(" Terraform ") → "Terraform"
split()	Splits a string into a list	split(",", "a,b,c") → ["a", "b", "c"]
join()	Joins a list into a string	join("-", ["a", "b", "c"]) → "a-b-c"

2 Numeric Functions 🔢

Function	Description	Example
abs()	Absolute value	abs(-10) → 10
ceil()	Rounds up	ceil(4.3) → 5
floor()	Rounds down	floor(4.9) → 4
max()	Returns the largest number	max(10, 20, 30) → 30
min()	Returns the smallest number	min(10, 20, 30) → 10
pow()	Exponentiation	pow(2, 3) → 8
log()	Logarithm	log(100, 10) → 2

Terraform

3 Collection Functions (Lists & Maps) 📦

Function	Description	Example
length()	Returns number of elements	length(["a", "b", "c"]) → 3
element()	Gets an element by index	element(["a", "b", "c"], 1) → "b"
concat()	Combines lists	concat(["a"], ["b", "c"]) → ["a", "b", "c"]
contains()	Checks if a list has an item	contains(["a", "b"], "b") → true
flatten()	Flattens nested lists	flatten([["a", "b"], ["c"]]) → ["a", "b", "c"]
keys()	Returns keys of a map	keys({name="John", age=30}) → ["name", "age"]
values()	Returns values of a map	values({name="John", age=30}) → ["John", 30]
merge()	Merges multiple maps	merge({a=1},{b=2}) → {a=1, b=2}

4 IP & CIDR Functions 🌐

Function	Description	Example
cidrsubnet()	Creates a subnet from a CIDR	cidrsubnet("192.168.1.0/24", 2, 1) → "192.168.1.64/26"
cidrhost()	Gets an IP from a CIDR	cidrhost("192.168.1.0/24", 5) → "192.168.1.5"
cidrnetmask()	Gets the netmask length	cidrnetmask("192.168.1.0/24") → "255.255.255.0"

5 Type Conversion Functions 🔄

Function	Description	Example
tostring()	Converts to string	tostring(123) → "123"
tonumber()	Converts to number	tonumber("42") → 42
tobool()	Converts to boolean	tobool("true") → true

Terraform

6 Encoding & Hashing Functions 🔑

Function	Description	Example
base64encode()	Encodes to Base64	base64encode("hello") → "aGVsbG8="
base64decode()	Decodes Base64	base64decode("aGVsbG8=") → "hello"
md5()	Generates MD5 hash	md5("hello") → "5d41402abc4b2a76b9719d911017c592"
sha256()	Generates SHA-256 hash	sha256("hello") → "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"

7 File & Path Functions 📁

Function	Description	Example
file()	Reads file content	file("config.json")
dirname()	Gets the directory name	dirname("/usr/local/bin") → "/usr/local"
basename()	Gets the filename	basename("/usr/local/bin/script.sh") → "script.sh"
path.cwd	Current working directory	"${path.cwd}"

Blocks, args & expressions

Blocks

In Terraform, configurations are written using **blocks**, **arguments**, and **expressions**.

1 Blocks (Structural Units)

A block is a container for configuration settings.

Each block has:

1. A block type (e.g., resource, provider, variable)
2. A label (optional, depends on the block type)
3. Arguments inside {} brackets

Block Type	Purpose	Usually Covered In
resource	Creates infrastructure	✅ Main list
provider	Configures cloud/API access	✅ Main list
module	Reuses other Terraform code	✅ Main list
variable	Accepts input values	✅ Main list
locals	Defines reusable internal values	✅ Main list
output	Displays/export values after apply	✅ Main list
data	Fetches existing cloud resources	✅ Main list
terraform	Sets backend, version, providers	🛠 Setup/meta config

Terraform

Blocks

In Terraform, configurations are written using **blocks**, **arguments**, and **expressions**.

1 Blocks (Structural Units)

A block is a container for configuration settings.

Each block has:

1. A block type (e.g., resource, provider, variable)
2. A label (optional, depends on the block type)
3. Arguments inside {} brackets

Provider Block

```
provider "aws" {
  region = "ap-south-2"
}
```

Local Block

```
locals {
  bucket_name   = “ccitmarch25bucket"
  instance_type = "t3.micro"
  ami_id        = "ami-0150f9c33a6958e02"
}
```

Resource block

```
resource "aws_s3_bucket" "my_bucket" {
  bucket = “ccitmar25bucket”}
```

Variable block

```
variable "instance_type" {
  default = "t2.micro"
}
```

2 Arguments (Key-Value Pairs Inside Blocks)

An argument is a key-value pair inside a block that defines setting

👉 Arguments can accept:

- ✅ **String** ("t3.micro")
- ✅ **Number** (10)
- ✅ **Boolean** (true / false)
- ✅ **List** (["dev", "test"])
- ✅ **Map** ({ name = "server", env = "dev" })

```
resource "aws_instance" “inst" {
  ami           = "ami-0c55b1"  # ami is an argument
  instance_type = "t3.micro"    # ins_type is an argument
}
```

Terraform

Expressions

Expressions compute values dynamically based on variables, functions, or resource attributes.

Types of Expressions

- Variables Expressions
- Function Expressions
- Resource Attributes
- Conditional Expressions

Variable expressions

```
variable "inst_type" {
  default = "t2.micro"
}

resource "aws_instance" “inst" {
  instance_type = var. inst_type  # Using a variable in an expression
}
```

Function Expressions

```
locals {
  upper_name = upper("terraform")  # Converts to uppercase
}
```

Resource Attributes

```
output "public_ip" {
  value = aws_instance.example.public_ip  # Fetching EC2 public IP dynamically
}
```

Conditional Expressions

```
locals {
  instance_type = var.is_production ? "t3.large" : "t2.micro"
}
```

Advantages & disadvantages

Terraform

✅ Advantages of Terraform

1 Multi-Cloud Support

- Works with AWS, Azure, GCP, Kubernetes, etc.
- Allows managing resources across multiple clouds.

2 Declarative Language (HCL - HashiCorp Configuration Language)

- Defines what the infrastructure should look like rather than how to achieve it.
- Easier to read and manage.

3 Modular & Reusable Code

- Supports modules to create reusable components.
- Example: A single VPC module can be used across different projects.

4 Supports Infrastructure Automation

Integrates with CI/CD pipelines (Jenkins, GitHub Actions, etc.).

5 Open-Source & Extensible

- Free to use, with Terraform Cloud offering extra features.
- Can be extended with providers & plugin

Terraform

❌ Disadvantages of Terraform

1 State File Management Issues

- The .tfstate file contains sensitive information (e.g., secrets, credentials).
- Needs remote storage (S3, Terraform Cloud) for security and collaboration.

2 Learning Curve

- HCL syntax is different from traditional scripting languages.
- Troubleshooting Terraform errors can be complex.

3 Longer Execution Time for Large Deployments

- Terraform re-evaluates dependencies before each apply, which can be slow for large infrastructures.
- Performance may be impacted when managing thousands of resources.

5 Destroy Command is Dangerous

- Running terraform destroy removes all infrastructure.
- Needs extra safety measures to avoid accidental deletions.

Terraform

- Terraform CLI
- Providers block
- Local version changing
- Provisioners (File, Local-exec, Remote exec)
- Variable precedence

Terraform CLI

Terraform

Initialization and Setup

Command	Description
terraform init	Initialize Terraform in a directory
terraform version	Show Terraform version
terraform providers	List providers used in the configuration

Workspaces

Command	Description
terraform workspace list	List all workspaces
terraform workspace show	Show current workspace
terraform workspace new	Create a new
terraform workspace select	Switch to a different
terraform workspace delete	Delete a workspace

Core Workflow

Command	Description
terraform plan	Show execution plan
terraform apply	Apply changes
terraform destroy	Destroy managed infrastructure
terraform validate	Validate configuration syntax and structure
terraform fmt	Format configuration files
terraform show	Show current or planned state
terraform output	Show output variables

Resource Lifecycle

Command	Description
terraform taint	Mark a resource for recreation
terraform untaint	Remove taint from a resource
terraform import	Import existing infrastructure into Terraform
terraform refresh	Update the state file with real-world resources

Terraform

🧪 Debugging and Logs

Command	Description
terraform console	Interactive console for Terraform expressions
terraform debug	Start a debug session (advanced use)
TF_LOG=DEBUG (env var)	Enable debug logging for troubleshooting

📦 Modules & Providers

Command	Description
terraform providers	Show required and installed providers
terraform providers lock	Generate dependency lock file
terraform providers mirror	Download provider plugins to a local directory
terraform get	Download modules (for older workflows)

Providers block

Terraform

Providers block

- By default, provider plugins in Terraform change version every few weeks.
- When we run the init command, it downloads the latest plugins always.
- Some code will not work with old plugins, so we need to update them.
- To get the latest provider plugins: https://registry.terraform.io/browse/providers.
- When you add a new provider, “terraform init” is a must.
- terraform providers: Lists the providers required to run the code.
- To create infrastructure on any cloud using Terraform, all we need is a provider.

Types:

- OFFICIAL: Managed by Terraform.
- PARTNER: Managed by a third-party company.
- COMMUNITY: Managed by individuals.

Local version changing

Terraform

Local version changing

- When you add/ update the provider, terraform init (-upgrade) is a must.
- terraform providers command gives you the required providers information.
- cat .terraform.lock.hcl : Here you can get the initiated providers information.

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "5.91.0"
    }
  }
}
```

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "≤5.91.0"
    }
  }
}
```

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "≥5.91.0"
    }
  }
}
```

Terraform

Update the providers

- When you add/ update the provider, terraform init (-upgrade) is a must.
- terraform providers command gives you the required providers information.
- cat .terraform.lock.hcl : Here you can get the initiated providers information.

Operator	Description
= 1.8.0	Only version **1.8.0** is allowed
!= 1.8.0	Any version **except** 1.8.0 is allowed
> 1.8.0	Any version **greater than** 1.8.0
< 1.8.0	Any version **less than** 1.8.0
>= 1.8.0	**1.8.0 or newer**
<= 1.8.0	**1.8.0 or older**
~> 1.8.0	**1.8.x versions only** (e.g., 1.8.1, 1.8.5, but not 1.9.0)
~> 1.8	**1.x versions only** (e.g., 1.8.1, 1.9.0, but not 2.0.0)

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">=5.67.0"
    }

    azurerm = {
      source  = "hashicorp/azurerm"
      version = ">=4.24.0"
    }

    google = {
      source  = "hashicorp/google"
      version = ">=6.26.0"
    }
  }
}
```

Terraform

Terraform block and Provider Version Constraints

You can define version constraints for providers using the required_providers block.

It usually includes details such as required providers, backend configuration, and version constraints.

- terraform -v
- terraform -version
- terraform --version

```hcl
terraform {
  required_version = ">= 1.8.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.91.0"  # Allows 5.91.x versions but not 5.92.0
    }
  }
}
```

Setting	Description
required_version = ">= 1.8.0"	Ensures Terraform is version 1.8.0 or newer
required_providers	Specifies required providers (e.g., AWS)
source = "hashicorp/aws"	Fetches the AWS provider from HashiCorp's registry
version = "~> 5.91.0"	Allows 5.91.x versions but prevents upgrading to 5.92.0+

Provisioners (File, Local-exec, Remote exec)

🔧 What Are Provisioners?

Provisioners are used to execute scripts or commands on a local or remote machine after the resource is created.

They're often used to:

- Copy files
- Run installation commands
- Trigger configuration scripts

⚠️ Note: Provisioners are not the best practice for production — try to use cloud-init or user_data instead. But they're great for learning and small tasks.

Terraform

1 File Provisioner

- ✅ Transfers files from your local machine to the remote instance.
- ✅ Works only for resources with SSH access (like EC2).

📁 ccit.txt must exist in the same directory.

Example: Upload a Script to an EC2 Instance

```
provider "aws" {`}

resource "aws_instance" "example" {
  ami           = "ami-0af13678ce46c8851"
  instance_type = "t2.micro"
  key_name      = "my-key"

  provisioner "file" {
    source      = "hello.txt"
    destination = "/home/ec2-user/ccit.txt"
  }

  connection {
    type        = "ssh"
    user        = "ec2-user"
    private_key = file("~/.ssh/my-key.pem")
    host        = self.public_ip
  }
}
```

Terraform

2 Local-Exec Provisioner

- ✅ Runs a command on your local machine (where Terraform is executed).
- ✅ Useful for tasks like sending notifications or running local scripts.

Example: Run a Local Script After resource Creation

```
provider "aws" {
  region = "ap-south-2"
}
resource "aws_s3_bucket" "bucket" {
  bucket = "ccits3bucketmarch"
}
resource "null_resource" "upload_file" {
  depends_on = [aws_s3_bucket.bucket]
  provisioner "local-exec" {
    command = "aws s3 cp myfile.txt s3://ccits3bucketmarch/"
  }
}
```

3 Remote-Exec Provisioner

- ✅ Runs **commands on the remote machine** after SSH access.
- ✅ Used for configuration tasks like installing packages.

Example: Install Apache on EC2

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  connection {
    type        = "ssh"
    user        = "ec2-user"
    private_key = file("~/.ssh/id_rsa")
    host        = self.public_ip
  }

  provisioner "remote-exec" {
    inline = [
      "sudo yum update -y",
      "sudo yum install -y httpd",
      "sudo systemctl start httpd"
    ]
  }
}
```

Variable precedence

Terraform

VARIABLE PRECEDENCE:

- Defines the Level Priority for Variables in Terraform
- Terraform will pick the variables in the following order:

  - CLI
  - terraform.tfvars
  - Environment Variable
  - variable.tf

Environment Variable Example:

- export TF_VAR_instance_type="t2.medium"
- $env:TF_VAR_instance_type = "t2.medium"

Terraform

- HCP Cloud introduction and features
- GitHub Communication
- Build infra with HCP

HCP Cloud introduction and features

HCP Cloud - Introduction

- HCP means HashiCorp Cloud Platform
- It is a managed platform to automate cloud infrastructure.
- It provides privacy, security, and isolation.
- It supports multiple providers like AWS, Azure, and Google Cloud.
- It offers a suite of open-source tools for managing infrastructure, including Terraform, Vault, Consul, and Nomad.
- We can use different code repositories for a project.
- We can use variable sets to apply the same variables to all the workspaces.

Terraform

Main products

- Terraform: to create infrastructure.
- Vault: to manage the secrets and protecting sensitive data.
- Nomad: to schedule, deploy and manage applications.
- Consul: to secure service-to-service communication and networking.
- Packer: to create images for launching servers.

Account Creation

- Go to Google & type: HCP Cloud account Sign-In/ Sign-Up
- Email & Password
- Verify the email → Create ORG
- Click on Terraform.
- Continue with HCP account
- Create A GitHub account.
- create repo → name → add new file → write terraform code → commit

HCP Cloud signup

Sign up | HashiCorp Cloud Platf

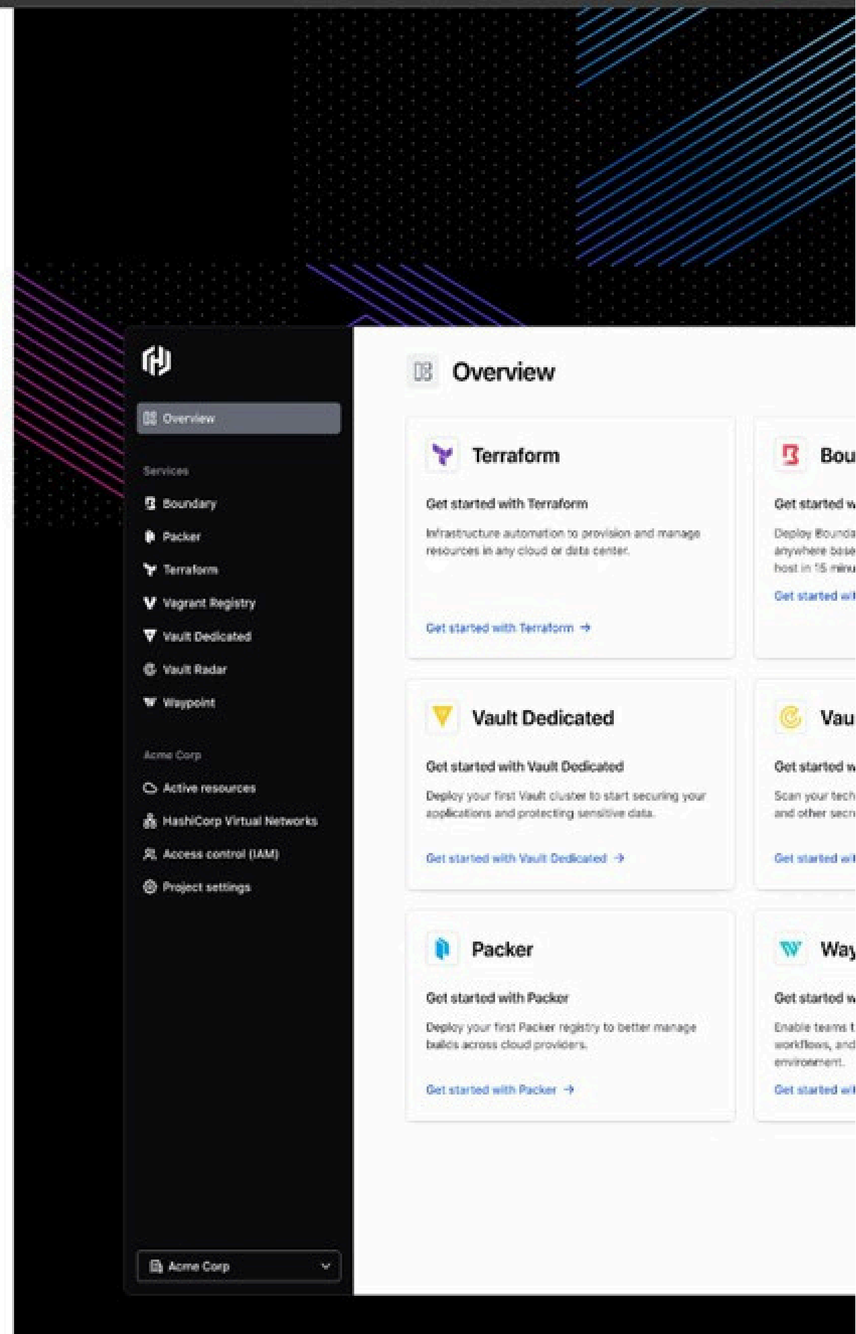
portal.cloud.hashicorp.com/sign-up

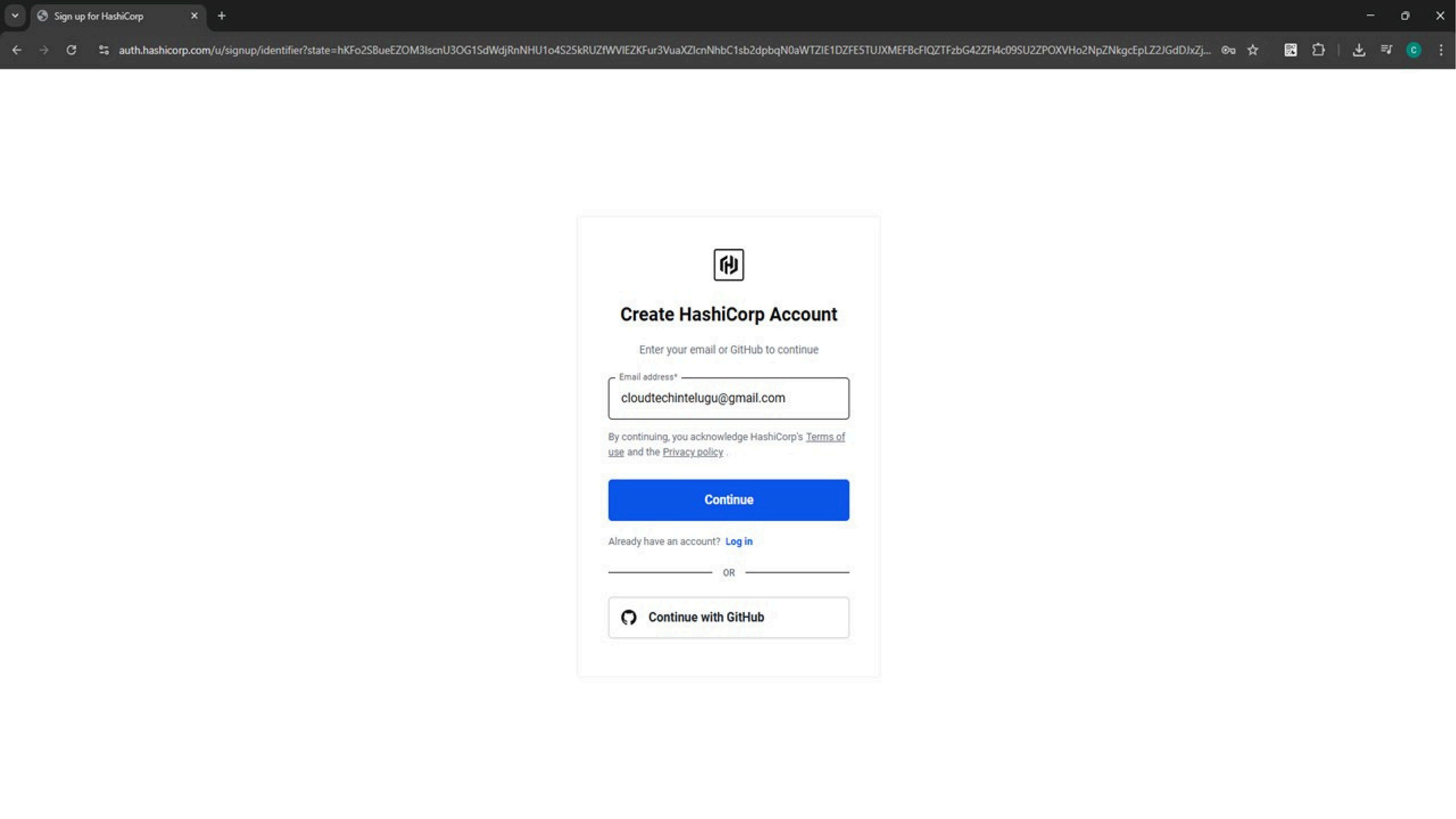
HashiCorp Cloud Platform

Terraform Vagrant Packer Waypoint Vault Boundary

Get started in minutes with our cloud products

Create an account | Sign in




Sign up for HashiCorp
auth.hashicorp.com/u/signup/identifier?state=hKFo2SBueEZOM3IscnU3OG1SdWdjRnNHU1o4S25kRUZfWVlEZKFur3VuaXZIcnNhbC1sb2dpbqN0aWTZIE1DZFE5TUJXMEFBcFlQZTFzbG42ZFI4c09SU2ZPOXVHo2NpZNkgcEpLZ2JGdDJxZj...
Create HashiCorp Account
Enter your email or GitHub to continue
Email address*
cloudtechintelugu@gmail.com
By continuing, you acknowledge HashiCorp's Terms of use and the Privacy policy
Continue
Already have an account? Log in
OR
Continue with GitHub

WhatsApp | Inbox (14) - cloudtechintelugu@g

https://mail.google.com/mail/u/1/#inbox

Gmail

Search mail

Compose

Inbox 14
Starred
Snoozed
Sent
Drafts
More

Labels

Upgrade

1–24 of 24

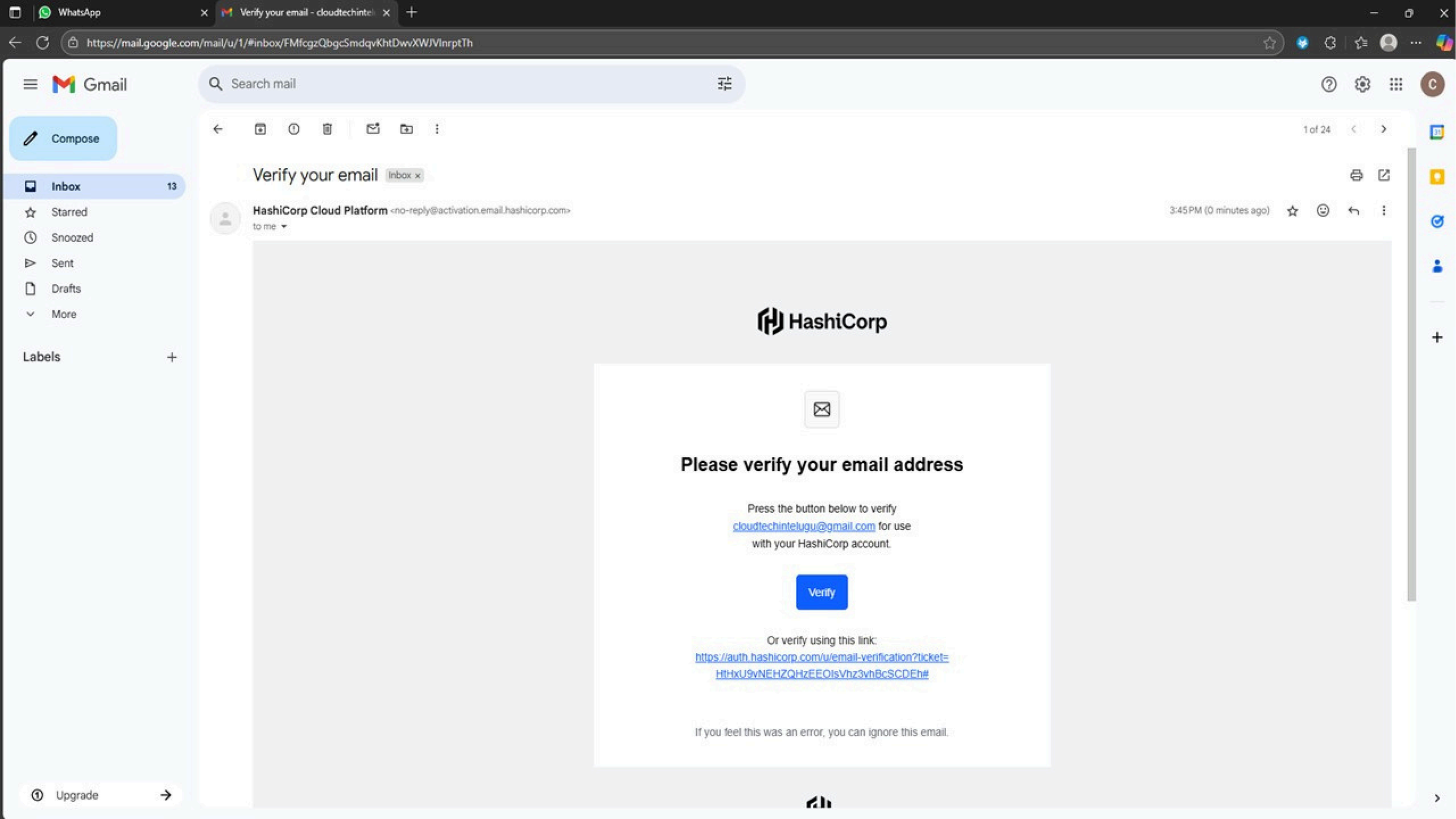
Get additional protection against phishing
Turn on Enhanced Safe Browsing to get additional protection against dangerous emails
Continue No thanks

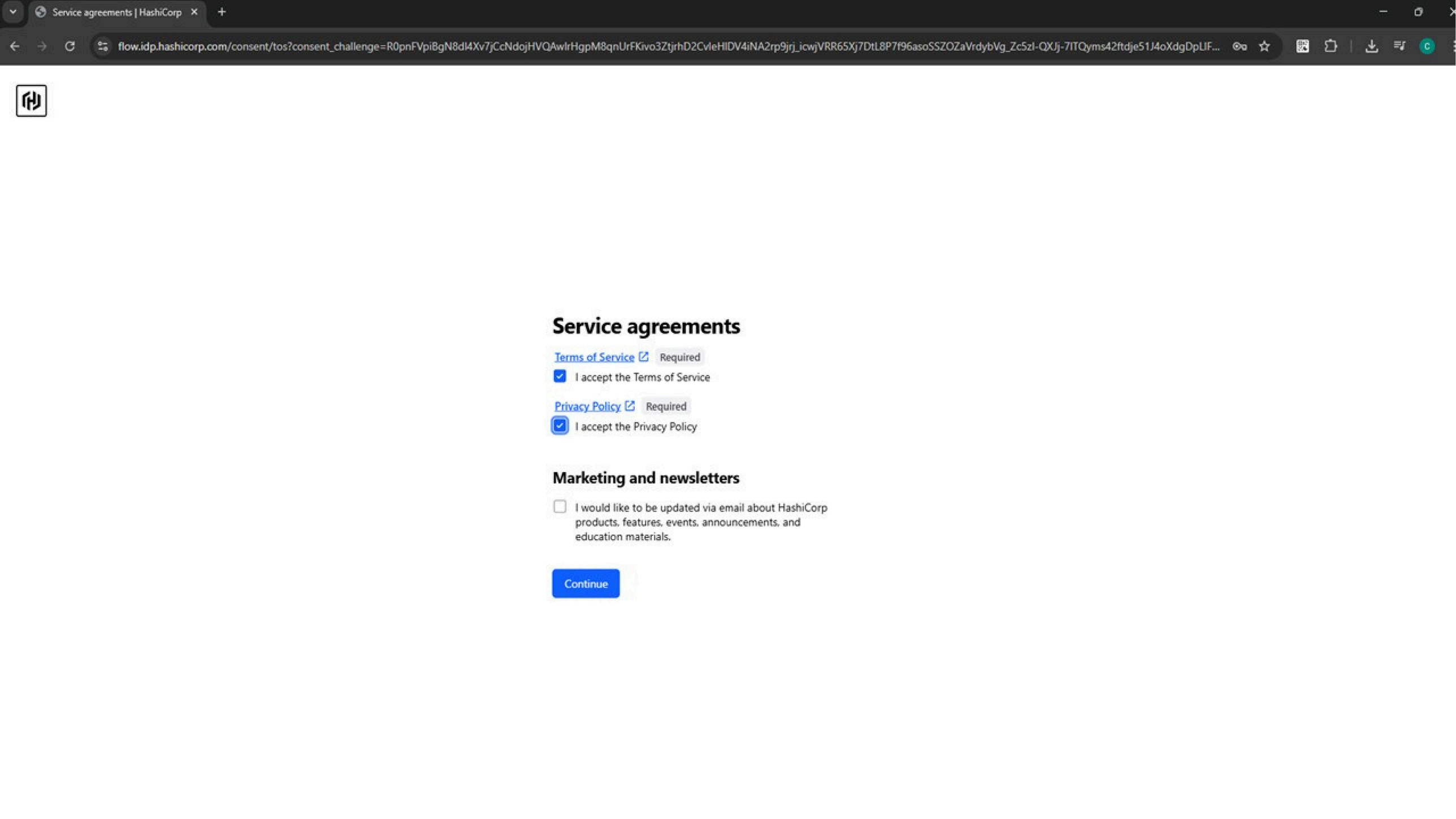
Primary | Promotions 3 new — Zoom — Expand your reach | Social

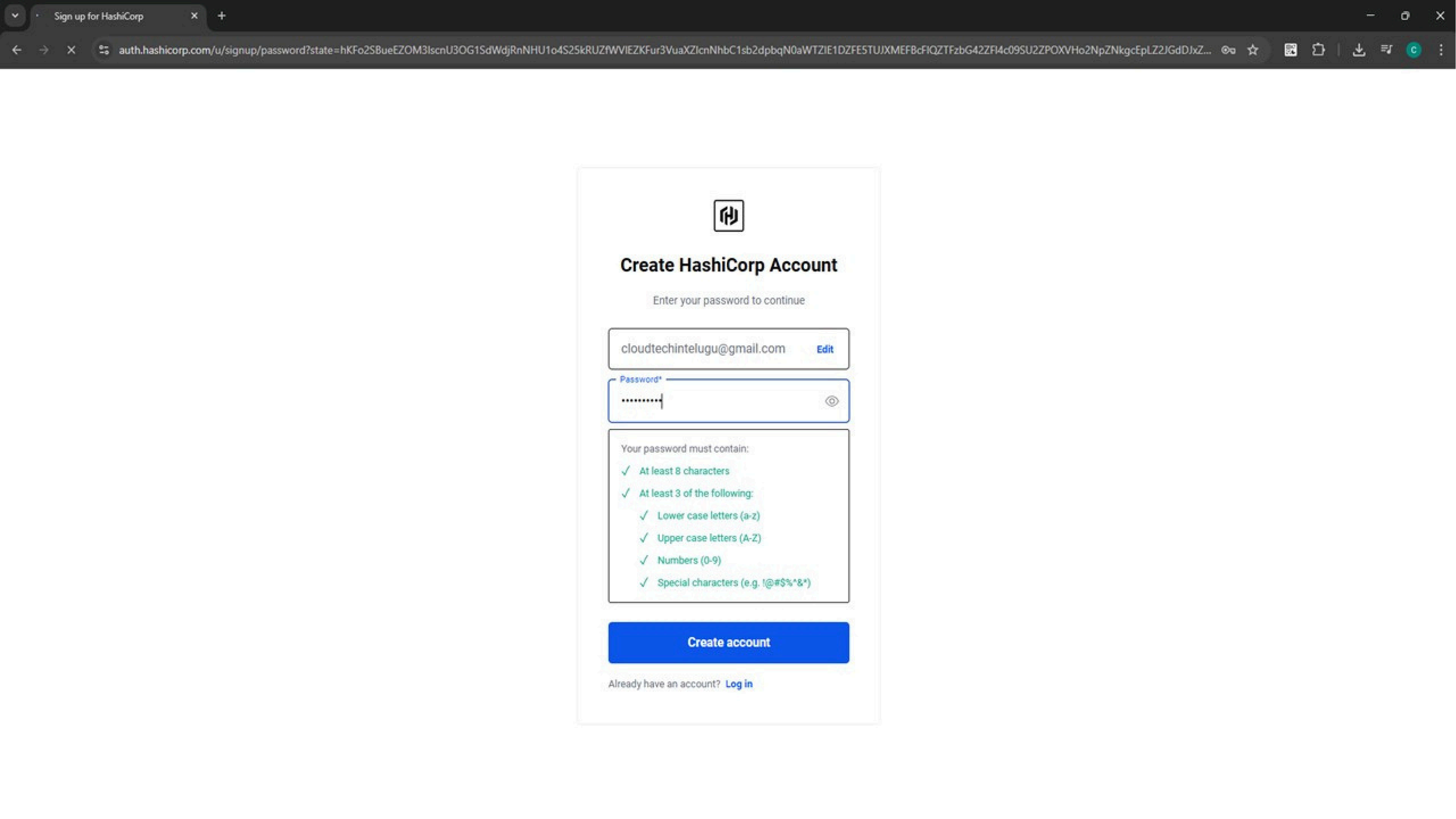
Get started with Gmail

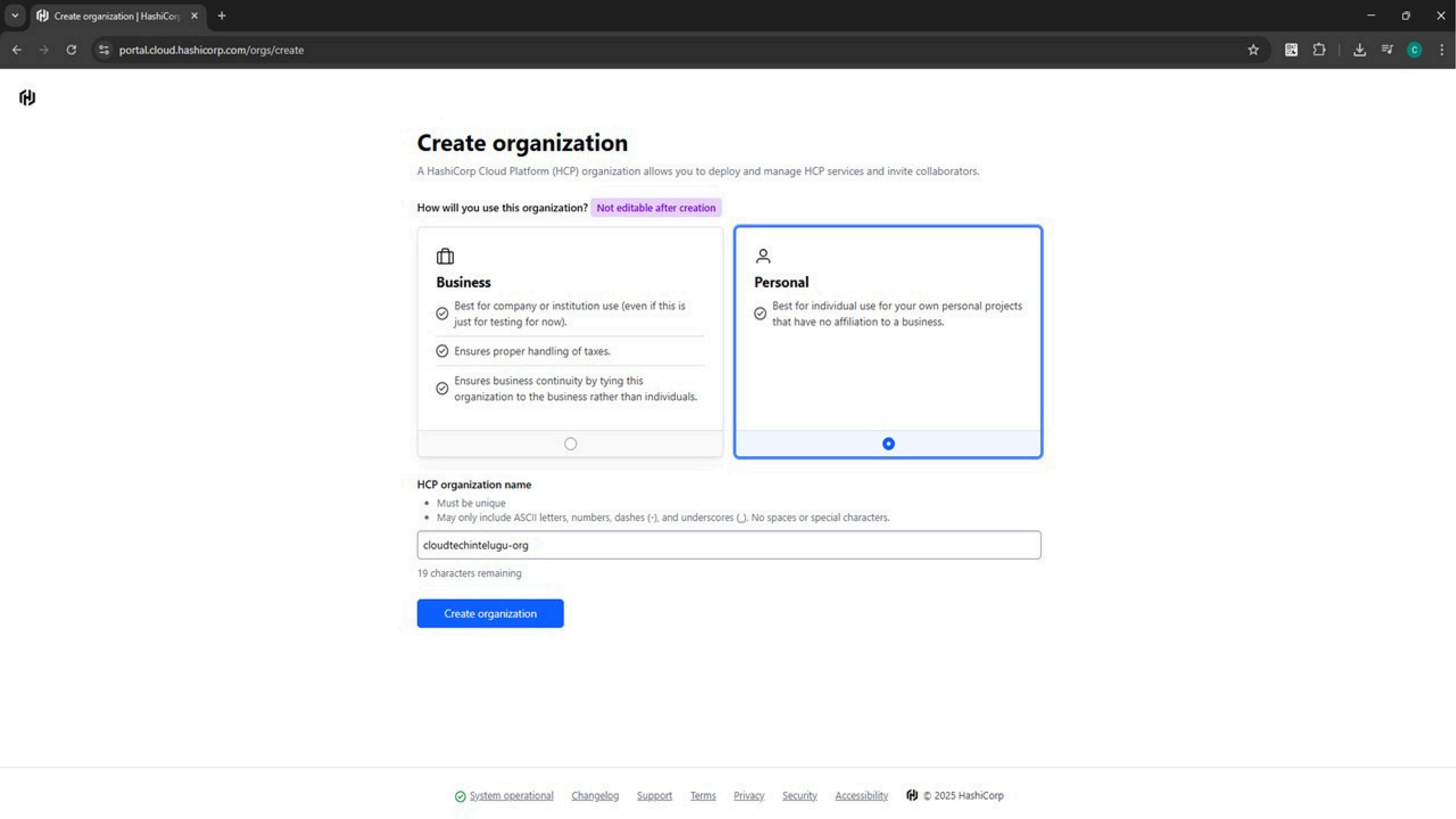
Customize your inbox
Change profile image
Import contacts and mail
Get Gmail for mobile

HashiCorp Cloud Pla.	**Verify your email** - Please verify your email address Press the button below to verify cloudtechintelugu@gmail.com for use with your HashiCorp account. Verify Or verify using this link: htt...	3:45 PM
HashiCorp Cloud Pla.	**Invitation to enroll in Multifactor Authentication** - Multifactor Authentication is now available MFA is an optional feature that will increase the security of your account. Enroll in Multifact...	3:45 PM
Amazon Web Services	**Amazon Web Services GST Invoice Available** - Greetings from Amazon Web Services, We are writing to notify you that your AWS Account has GST Invoice(s) available. They can be down...	Aug 2
GitHub	**[GitHub] A fine-grained personal access token has been added to your account** - @cloudtechintelugu, a personal access token was created on your account. Hey cloudtechintelugu! ...	Jul 31
GitHub	**[GitHub] A first-party GitHub OAuth application has been added to your account** - Hey cloudtechintelugu! A first-party GitHub OAuth application (Git Credential Manager) with gist, r...	Jul 31
GitHub 3	**[GitHub] A fine-grained personal access token has been added to your account** - @cloudtechintelugu, a personal access token was created on your account. Hey cloudtechintelugu! ...	Jul 26
GitHub 2	🚀 Your GitHub launch code - Here's your GitHub launch code! Continue signing up for GitHub by entering the code below: 12527076 Open GitHub Not able to enter the code? Paste the foll...	Jul 26
Zoom	**New Invoice Notification** - Thanks for your business! Please see the attached copy of your invoice. Please refer to the payment instructions on the attached invoice. Zoom Logo Header In... 8a129b6a9836d...	Jul 24
Zoom	**Welcome to Zoom!** - INV314735435_...	Jul 24
Zoom Communications.	**Payment Processing** - Your payment is processing. Zoom Logo Header Payment processing Dear Cloud Computing In Telugu, Your recent payment of ₹1623.68 Indian Rupee for Account N...	Jul 24
Zoom Communications	**Payment Processed** - Your payment has been successfully processed and applied to your account. Zoom Logo Header Payment processed Dear Cloud Computing In Telugu, Thank you fo...	Jul 24


WhatsApp
Verify your email - cloudtechintel
https://mail.google.com/mail/u/1/#inbox/FMfcgzQbgcSmdqvKhtDwvXWJVlnrptTh
Gmail
Search mail
Compose
Inbox 13
Starred
Snoozed
Sent
Drafts
More
Labels
Upgrade
1 of 24
Verify your email
Inbox ×
HashiCorp Cloud Platform <no-reply@activation.email.hashicorp.com>
to me
3:45 PM (0 minutes ago)
HashiCorp
Please verify your email address
Press the button below to verify
cloudtechintelugu@gmail.com for use
with your HashiCorp account.
Verify
Or verify using this link:
https://auth.hashicorp.com/u/email-verification?ticket=
HtHxU9vNEHZQHzEEOIsVhz3vhBcSCDEh#
If you feel this was an error, you can ignore this email.


Service agreements | HashiCorp
flow.idp.hashicorp.com/consent/tos?consent_challenge=R0pnFVpiBgN8dl4Xv7jCcNdojHVQAwlrHgpM8qnUrFKivo3ZtjrhD2CvleHIDV4iNA2rp9jrj_icwjVRR65Xj7DtL8P7f96asoSSZOZaVrdybVg_Zc5zl-QXJj-7ITQyms42ftdje51J4oXdgDpLIF...
Service agreements
Terms of Service
Required
I accept the Terms of Service
Privacy Policy
Required
I accept the Privacy Policy
Marketing and newsletters
I would like to be updated via email about HashiCorp products, features, events, announcements, and education materials.
Continue


Sign up for HashiCorp
auth.hashicorp.com/u/signup/password?state=hKFo2SBueEZOM3IscnU3OG1SdWdjRnNHU1o4S25kRUZfWVlEZKFur3VuaXZIcnNhbC1sb2dpbqN0aWTZIE1DZFE5TUJXMEFBcFlQZTFzbG42ZFl4c09SU2ZPOXVHo2NpZNkgcEpLZ2JGdDJxZ...
Create HashiCorp Account
Enter your password to continue
cloudtechintelugu@gmail.com
Edit
Password*
Your password must contain:
At least 8 characters
At least 3 of the following:
Lower case letters (a-z)
Upper case letters (A-Z)
Numbers (0-9)
Special characters (e.g. !@#$%^&*)
Create account
Already have an account? Log in


Create organization | HashiCorp
portal.cloud.hashicorp.com/orgs/create
Create organization
A HashiCorp Cloud Platform (HCP) organization allows you to deploy and manage HCP services and invite collaborators.
How will you use this organization?
Not editable after creation
Business
Best for company or institution use (even if this is just for testing for now).
Ensures proper handling of taxes.
Ensures business continuity by tying this organization to the business rather than individuals.
Personal
Best for individual use for your own personal projects that have no affiliation to a business.
HCP organization name
Must be unique
May only include ASCII letters, numbers, dashes (-), and underscores (_). No spaces or special characters.
cloudtechintelugu-org
19 characters remaining
Create organization
System operational
Changelog
Support
Terms
Privacy
Security
Accessibility
© 2025 HashiCorp

Dashboard | HashiCorp Cloud

portal.cloud.hashicorp.com/orgs/b361d4b4-d67b-4111-8eaa-cfd054fe4b70/projects/59525d48-67c6-4713-b7a2-da7d92d1f8de

HashiConf 2025

Tickets are on sale now for HashiConf, HashiCorp's global cloud conference. Join us in San Francisco 9/24-26. Register today

default-project

Services

- Boundary
- Consul
- Packer
- Terraform
- Vagrant Registry
- Vault Dedicated
- Vault Radar Trial
- Waypoint

default-project

- Active resources
- HashiCorp Virtual Networks
- Access control (IAM)
- Project settings

default-project

cloudtechintelugu-org / Projects / default-project

default-project

Project dashboard

An overview of all resources in the project. Learn more

Manage project

Project status	Last resource seen
✓ Active	—

View project settings →

Active resources

No active resources

There are 0 active resources inside this project.

View active resources →

Billing summary

To view billing information across projects, go to the organization.

View organization billing summary →

Boundary

Get started with Boundary

Deploy Boundary to access any system from anywhere based on user identity. Secure your first host in 15 minutes.

Get started with Boundary →

Consul

Get started with Consul

Deploy your first Consul cluster and start building your service mesh.

Get started with Consul →

Packer

Get started with Packer

Deploy your first Packer registry to better manage builds across cloud providers.

Get started with Packer →

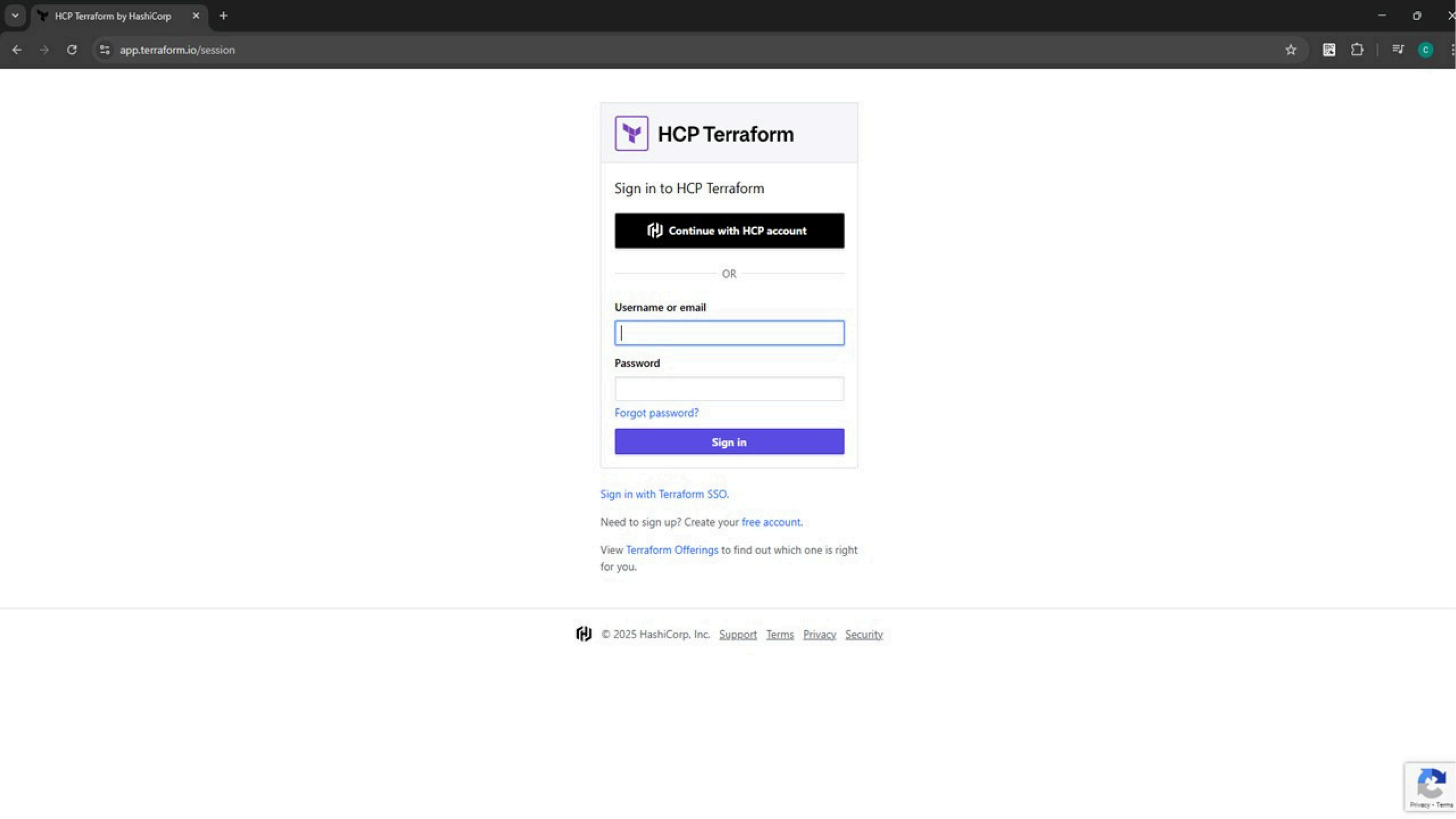
Terraform

Get started with Terraform

Infrastructure automation to provision and manage resources in any cloud or data center.

Get started with Terraform →

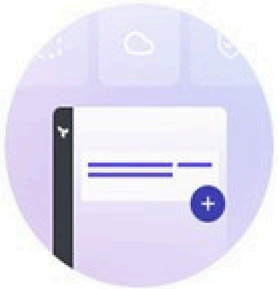
HCP Terraform signup


HCP Terraform by HashiCorp
app.terraform.io/session
HCP Terraform
Sign in to HCP Terraform
Continue with HCP account
OR
Username or email
Password
Forgot password?
Sign in
Sign in with Terraform SSO.
Need to sign up? Create your free account.
View Terraform Offerings to find out which one is right for you.
© 2025 HashiCorp, Inc. Support Terms Privacy Security

Organizations

Organizations

Terraform organizations let you manage organizations, projects, and teams.



Create your first organization

+ Create organization

New Organization | HCP Terraform

app.terraform.io/app/organizations/new

Organizations / New

Create a new organization

Organizations are privately shared spaces for teams to collaborate on infrastructure. Learn more about organizations in HCP Terraform.

How will you use this organization? Not editable after creation

Business

- Best for company or institution use (even if this is just for testing for now).
- Ensures proper handling of taxes.
- Ensures business continuity by tying this organization to the business rather than individuals.

Personal

- Best for individual use for your own personal projects that have no affiliation to a business.

Terraform organization name

- Must be unique.
- May contain valid characters including ASCII letters, numbers, spaces, as well as dashes (-), and underscores (_).

ccitorg

Email address

The organization email is used for any future notifications, such as billing alerts, and the organization avatar, via gravatar.com.

ccitnov2024@gmail.com

Create organization | Cancel

Terraform

Cost Estimation policy:

- we can estimate the cost of infra before we create it.
- By default, this feature is disable, we need to enable.
- Click on terraform logo -- > Organization -- > settings -- > Cost Estimation -- > enable

Sentinel Policy

- By this policy we can write our own conditions.
- It will check the condition of resource before it created.
- Resource will be created/ updated only when the condition is satisfied.
- EX: TAGS, VERIFIED AMIS, SG etc…

Terraform

Terraform cloud features:

1. Workspaces
2. Projects
3. Runs
4. Variables and Variable Sets
5. Policies and Policy Sets
6. Run Tasks
7. Single Sign-On (SSO)
8. Remote State

TERRAFORM CLOUD OFFERS:

Free	Standard	Plus	Enterprise
UP TO	STARTING AT		
500 resources	**$0.00014**	**Custom**	**Custom**
per month	per hour per resource		
Cloud	Cloud	Cloud	Self-managed
Get started with all capabilities needed for infrastructure as code provisioning.	For professional individuals or teams adopting infrastructure as code provisioning.	For enterprises standardizing and managing infrastructure automation and lifecycle, with scalable runs.	For enterprises with special security, compliance, and additional operational requirements.
No credit card required	Enterprise support included	Enterprise support included	Enterprise support included
Start for free	Start for free	Contact sales	Contact sales
	First 500 resources per month are free		

Terraform

Secure Secrets in the Terraform Code:

- **Tip1:** Never store secrets in plain text
- **Tip2:** Mark Variables as sensitive
- **Tip3.:** Use Environment variables
- **Tip4:** Secret Store (e.g.: Vault, AWS Secrets manager)
- **Note:** Even after marked a value as sensitive in tf file, they are stored within the Terraform state file.

Thank you