Roll No: | | | | | | | | | | | |

PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE
REACH GREATER HEIGHTS

Presidency University Act, 2013 of the Karnataka Act No. 41 of 2013 | Established under Section 2(f) of UGC Act, 1956
Approved by AICTE, New Delhi | Approved By BCI
Bengaluru

Mid Term Examinations - October 2026

Date: 05/10/2026 **Time:** 11:45 AM - 01:15 PM

Course Code: CDV3403 **Course Name:** Infrastructure as Code (IaC) with Terraform And Ansible

Semester: Seventh Semester **Max. Marks:** 50 **Weightage:** 50%

CO - Levels	CO1	CO2
Marks	46	44

PART A: Answer Following Questions. 10 M

Q.N.	Questions	M	CO	BT
1	Infrastructure as Code (IaC) is the practice of managing and provisioning computing infrastructure using machine-readable configuration files or code instead of manual processes. It enables infrastructure to be created and managed in a consistent and repeatable manner.	2	CO1	BT1
2	Two benefits of IaC are: Automation: Reduces manual effort in infrastructure provisioning. Consistency: Ensures infrastructure is configured uniformly across environments. (or any 2)	2	CO1	BT1
3	Two challenges of IaC are: Security risks: Configuration files may contain sensitive information such as credentials. Complexity: Managing large and complex infrastructure code can be difficult. (or any 2)	2	CO1	BT1
4	A Terraform provider is a plugin that allows Terraform to interact with an external platform or service such as AWS, Azure, Google Cloud, or Kubernetes. It provides the resources and APIs required to manage that platform.	2	CO2	BT1
5	A Terraform resource represents an infrastructure component that Terraform can create, modify, or delete. Example resource "aws_instance" "web" { ami = "ami-123456" instance_type = "t2.micro" }	2	CO2	BT1

PART B : Answer Any 1 Following Questions. 10 M

Q.N.	Questions	M	CO	BT

ANSWER KEY

6	Traditional model Development → Handover → Operations → Production Problems Development and operations worked in separate silos. Deployment was largely manual. Releases became slower as infrastructure grew. Configuration drift resulted in inconsistent servers. DevOps model Development ↔ Operations Key idea Development and operations collaborate closely. Infrastructure and deployment activities increasingly become software-based. Automation becomes a core principle.	10	CO1	BT2
7	Benifits 1. Automation Reduces repetitive manual tasks. 2. Speed Infrastructure can be deployed rapidly. 3. Consistency Same configuration can be repeatedly applied. 4. Documentation Infrastructure configuration becomes readable code. 5. Version Control Infrastructure changes can be tracked. 6. Validation Code can be reviewed and tested before deployment. 7. Reusability Infrastructure components can be packaged into reusable modules. 8. Self-Service Teams can initiate deployments without depending entirely on administrators. Challenges 1. Learning Curve IaC tools require knowledge of new languages, workflows and concepts. 2. Tool Selection IaC tools overlap in functionality, making it difficult to select the most appropriate tool. 3. State Management Terraform requires careful management of infrastructure state. 4. Secrets Management Terraform state can contain sensitive information and must be protected. 5. Configuration Drift Manual or mutable changes can make infrastructure inconsistent. 6. Provider Differences AWS, Azure and GCP expose different infrastructure capabilities, so IaC does not provide completely identical infrastructure across clouds. 7. Testing Difficulty Infrastructure testing can require actual resources, making testing slower and more expensive. 8. Infrastructure Complexity Large environments can result in complex configurations and dependencies. 9. State Locking Concurrent Terraform operations can cause conflicts or state corruption without proper locking. 10. Team Collaboration Shared infrastructure requires proper workflows, remote state management and controlled changes.	10	CO1	BT2

ANSWER KEY

PART C: Answer Any 1 Following Questions. 10 M				
Q.N.	**Questions**	**M**	**CO**	**BT**

8	Category 1)Ad-hoc scripting 2)Configuration management 3)Server templating 4)Orchestration 4)Provisioning Purpose 1)Automate individual tasks 2)Configure existing servers 3)Create reusable images 4)Manage applications/containers 5)Create infrastructure resources Examples 1)Bash, Python 2)Ansible, Chef, Puppet 3)Packer, Docker 4)Kubernetes 5)Terraform, CloudFormation	10	CO1	BT2
9	Provisioning Creates infrastructure such as: VMs Networks Subnets Databases Load balancers Storage Example: Terraform Terraform Configuration ↓ Terraform CLI ↓ Provider ↓ API ↓ Cloud / Infrastructure Example main.tf ↓ Terraform ↓ AWS Provider ↓ AWS API ↓ EC2 / VPC / S3 / etc. Explaination of the same	10	CO1	BT2

PART D: Answer Any 1 Following Questions. 10 M				
Q.N.	**Questions**	**M**	**CO**	**BT**

ANSWER KEY

10	A **Terraform provider** is a plugin that enables Terraform to interact with a particular cloud platform, service, or technology. It contains the API information and functionality required to create, modify, and delete resources. Examples of providers include: • AWS • Microsoft Azure • Google Cloud • Kubernetes • GitHub **Provider Configuration Example** The following configuration uses the **AWS provider**: terraform { required_providers { aws = { source = "hashicorp/aws" version = "~> 5.0" } } } provider "aws" { region = "us-east-1" } A **Terraform resource** represents an infrastructure component that Terraform can create, configure, modify, or delete. Examples of resources include: • Virtual machines • Storage buckets • Virtual networks • Databases • Security groups A resource is generally defined using the following syntax: resource "aws_instance" "web_server" { ami = "ami-0abcdef1234567890" instance_type = "t2.micro" tags = { Name = "WebServer" } }	10	CO2	BT2

ANSWER KEY

11	Terraform provides **input variables, local values, and output values** to make configurations reusable, organised, and easier to manage. They serve different purposes within a Terraform configuration. **1. Input Variables** **Input variables** are parameters that allow users to provide values to a Terraform configuration without modifying the configuration code. They are useful when the same configuration needs to be used with different values, such as different regions, instance types, or environment names. variable "region" { description = "AWS region" type = string default = "us-east-1" } variable "instance_type" { description = "EC2 instance type" type = string default = "t2.micro" } The variables can then be used in the configuration: provider "aws" { region = var.region } resource "aws_instance" "web_server" { ami = "ami-0abcdef1234567890" instance_type = var.instance_type } **Local values**, or **locals**, are named values defined inside a Terraform configuration. They are used to store **calculated values or expressions that are reused within the configuration**. Unlike input variables, local values are normally not supplied by the user. **Example** locals { environment = "production" server_name = "web-${local.environment}" } The local values can be used as: resource "aws_instance" "web_server" { ami = "ami-0abcdef1234567890" instance_type = "t2.micro" tags = { Name = local.server_name Environment = local.environment } } **Output values** are used to display or expose information about resources after Terraform has created or updated them. They are useful for obtaining important information such as an instance ID, IP address, or DNS name. **Example** output "instance_id"{ description = "ID of the EC2 instance" value = aws_instance.web_server.id } output "public_ip" { description = "Public IP address of the EC2 instance" value = aws_instance.web_server.public_ip } **Input variables** make Terraform configurations flexible by accepting external values, **local values** simplify configurations by storing reusable expressions, and **output values** expose useful information after infrastructure is provisioned. Together, they improve the **reusability, maintainability, and organisation** of Terraform configurations.	10	CO2	BT2

ANSWER KEY

PTO (Please Turn Over)

PART E: Answer Any 1 Following Questions. 10 M				
Q.N.	**Questions**	**M**	**CO**	**BT**
12	Terraform input variables allow values to be passed to a configuration without changing the Terraform code. Terraform provides several methods for assigning values to variables. The commonly used methods are **default values, command-line arguments, variable definition files, and environment variables**. **1. Using Default Values** A default value can be specified directly in the `variable` block. Terraform uses this value when no other value is provided. variable "instance_type" { type = string default = "t2.micro" } The variable can be used as: resource "aws_instance" "web" { instance_type = var.instance_type } If no value is supplied, Terraform uses t2.micro. **2. Using Command-Line Arguments** A variable value can be passed using the -var option with Terraform commands. **Example:** variable "instance_type" { type = string } The value can be supplied during execution: terraform apply -var="instance_type=t2.small" Here, t2.small is assigned to var.instance_type **3. Using .tfvars Files** Terraform allows variable values to be stored in a separate **.tfvars file**. **variables.tf:** variable "region" { type = string } variable "instance_type" { type = string } **terraform.tfvars:** region = "us-east-1" instance_type = "t2.micro" Terraform automatically loads values from a file named terraform.tfvars. A custom variable file can also be used: terraform apply -var-file="production.tfvars" For example, production.tfvars could contain: region = "us-west-2" instance_type = "t2.medium" **4. Using Environment Variables** Terraform variables can be assigned through environment variables using the prefix **TF_VAR_** followed by the variable name. **Variable declaration:** variable "region" { type = string } export TF_VAR_region="us-east-1" Then Terraform automatically assigns the value to: var.region This method is particularly useful in **CI/CD pipelines**, where values can be supplied without modifying Terraform configuration files.	10	CO2	BT2

ANSWER KEY

A **Terraform module** is a collection of Terraform configuration files that are grouped together to manage related infrastructure resources. Modules help to **organise, reuse, and simplify** Terraform configurations.
A module can contain resources, variables, outputs, and other Terraform configuration elements.

1. Root Module

The **root module** is the main Terraform configuration from which Terraform commands are executed. It consists of the `.tf` files present in the working directory.
For example:

```
project/
├── main.tf
├── variables.tf
└── outputs.tf
```

Here, the project directory represents the root module.
A root module can directly define resources:

```
resource "aws_instance" "web" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"
}
```

The root module can also call one or more child modules.

2. Child Module

A **child module** is a module that is called by another module, usually the root module. It contains reusable Terraform configurations for a specific infrastructure component.
For example:

```
project/
├── main.tf
└── modules/
    └── web_server/
        ├── main.tf
        ├── variables.tf
        └── outputs.tf
```

The web_server directory represents a child module.
The root module can call it using:

```
module "web_server" {
  source        = "./modules/web_server"
  instance_type = "t2.micro"
}
```

13 | The source argument specifies the location of the child module. | 10 | CO2 | BT2

3. Inputs

Inputs are values passed from the calling module to a child module. They are normally defined using **input variables**.
For example, the child module can define:

```
variable "instance_type" {
  type = string
}
```

The root module passes a value to the child module:

```
module "web_server" {
  source        = "./modules/web_server"
  instance_type = "t2.micro"
}
```

Here:
• instance_type in the child module is an input variable.
• "t2.micro" is the value supplied by the root module.
• The child module can access the value using var.instance_type.
Inputs make a module **customisable and reusable**.

4. Outputs

Outputs allow a child module to expose useful information to the calling module. For example, a child module may create an EC2 instance and return its public IP address.
Inside the child module:

```
output "public_ip" {
  value = aws_instance.web.public_ip
}
```

The root module can access this output using:

```
output "web_server_ip" {
  value = module.web_server.public_ip
}
```

Thus, outputs provide a way for information to flow **from a child module back to the calling module**.

ANSWER KEY