Devops Shack

THE ULTIMATE TERRAFORM INTERVIEW PLAYBOOK

BY DEVOPS SHACK

DevOps Shack

The Ultimate Terraform Interview Playbook

Table of Contents

- How do you use a backend configuration in Terraform?

8. **Lifecycle Management**
   - How does Terraform manage resource lifecycles?
   - What is the purpose of the terraform taint command?
9. **Dynamic Blocks and Conditional Logic**
   - What are Terraform dynamic blocks, and how are they used?
   - How does Terraform support conditional resource creation?
10. **Remote State Management**
    - How do you manage remote state in Terraform?
    - How does Terraform state file locking work in remote backends?
11. **Formatting and Debugging**
    - What is terraform fmt, and why is it important?
    - How do you debug errors in Terraform?
12. **Zero-Downtime Deployments**
    - How does Terraform handle zero-downtime deployments?
13. **Provisioners**
    - Explain the difference between local-exec and remote-exec provisioners.
14. **Shared Modules**
    - How do you manage shared modules in Terraform?
15. **Terraform Cloud**
    - What is Terraform Cloud, and how does it differ from Terraform CLI?
16. **Resource Management**
    - What are Terraform backends, and why are they important?
    - What is the purpose of the terraform output command?

- What are resource taints in Terraform, and how do you manually taint a resource?

17. **Version Constraints**
    - How does Terraform manage provider and configuration version constraints?
18. **Secrets Management**
    - How can you securely manage secrets in Terraform?
19. **Interactive Console**
    - What is the purpose of the terraform console command?
20. **Limitations and Best Practices**
    - What are the limitations of Terraform?
    - How can you ensure best practices while working with Terraform?

Introduction

Terraform, developed by HashiCorp, is one of the most popular Infrastructure as Code (IaC) tools, enabling developers and operations teams to define, provision, and manage infrastructure efficiently. With its declarative configuration language (HCL) and multi-cloud compatibility, Terraform has become a go-to tool for automating infrastructure management. This document compiles 50 Terraform interview questions and answers, covering fundamental concepts, advanced features, and practical use cases. It serves as a comprehensive guide for professionals preparing for Terraform interviews or looking to strengthen their understanding of the tool.

1. What is Terraform and what are its main features? Answer:

Terraform is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp. It allows you to define, provision, and manage infrastructure across various cloud providers and services using a declarative configuration language known as HashiCorp Configuration Language (HCL).

Main Features:

- **Infrastructure as Code (IaC):** Manage infrastructure using code, enabling version control, reuse, and sharing.
- **Provider Agnostic:** Supports multiple providers like AWS, Azure, GCP, and others, allowing for a consistent workflow.
- **Execution Plans:** Generates and shows execution plans before applying changes, helping you understand what Terraform will do.
- **Resource Graph:** Builds a graph of all resources and their dependencies, optimizing resource creation and modification.
- **Change Automation:** Automates complex changesets to your infrastructure with minimal human interaction.

2. Can you explain the difference between Terraform and other configuration management tools like Ansible, Puppet, or Chef?

Answer:

- **Purpose:**
  - **Terraform:** Primarily an infrastructure provisioning tool. It focuses on creating, updating, and versioning infrastructure safely and efficiently.
  - **Ansible/Puppet/Chef:** Primarily configuration management tools. They are used to install and manage software on existing servers.
- **Approach:**
  - **Terraform:** Declarative. You describe the desired state, and Terraform figures out how to achieve it.
  - **Ansible/Puppet/Chef:** Can be both declarative and procedural, depending on how you write your configurations or playbooks.
- **Infrastructure Lifecycle:**
  - **Terraform:** Manages the entire lifecycle of infrastructure, including creation, scaling, and destruction.
  - **Ansible/Puppet/Chef:** Manages the software and settings on already provisioned infrastructure.

3. What is state in Terraform, and why is it

important? Answer:

- **Terraform State:** A persistent data store that maps Terraform configurations to real-world resources. It's typically stored in a file named terraform.tfstate.
- **Importance:**
  - **Mapping:** Keeps track of resource IDs and metadata, enabling Terraform to manage resources effectively.
  - **Planning and Execution:** Allows Terraform to generate accurate execution plans by knowing the current state of resources.

- **Collaboration:** When stored remotely (e.g., in AWS S3 or Terraform Cloud), it enables team collaboration by sharing the state.

4. How do you manage multiple environments (e.g., development, staging, production) in Terraform?

Answer:

- **Workspaces:**
  - Use Terraform workspaces to maintain separate state files within the same configuration for different environments.
  - Example:

terraform workspace new development terraform

workspace select development

- **Directory Structure:**
  - Organize configurations into separate directories for each environment, each with its own state.
  - Example:

```
├── environments
   ├── dev
   ├── staging
   └── prod
```

- **Variable Files:**
  - Use different .tfvars files for each environment to parameterize configurations.
  - Example:

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

- **Backend Configuration:**

- Use different backend configurations to store state files separately for each environment.

5. What is a Terraform provider, and how do you use it? Answer:

- **Terraform Provider:**
  - A plugin that interacts with APIs of cloud platforms and services (e.g., AWS, Azure, Google Cloud).
  - Providers define resources and data sources for a service.
- **Usage:**
  - **Declaration:**

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

  - **Version Pinning:**

```
provider "aws" {
 version = "~> 3.0"
 region  = "us-west-2"
}
```

  - **Multiple Providers:**
    - You can configure multiple providers to manage resources across different platforms.

6. Explain the difference between Terraform modules and resources. Answer:

- **Resources:**
  - Basic building blocks in Terraform.

- Represent infrastructure objects like virtual networks, compute instances, or databases.
- Example:

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

- **Modules:**
  - Containers for multiple resources that are used together.
  - Promote code reuse and organization.
  - Can be shared and versioned.
  - Example of using a module:

```
module "vpc" {
 source = "terraform-aws-modules/vpc/aws"
 version = "2.77.0"

 name = "my-vpc"
 cidr = "10.0.0.0/16"
}
```

7. How can you import existing infrastructure into Terraform? Answer:

- **Step 1: Write Resource Configuration**
  - Define the resource in your .tf files without any parameters that Terraform can't infer.

```
resource "aws_instance" "existing" {
 # Configuration will be populated after import
```

```
}
```

- **Step 2: Run Import Command**
  - Use terraform import to map the existing resource to the Terraform resource.

terraform import aws_instance.existing i-0abcdef1234567890

- **Step 3: Refresh and Update Configuration**
  - Run terraform plan to see differences and update the configuration to match the actual settings.

8. What are Terraform variables, and how do you use them? Answer:

- **Terraform Variables:**
  - **Input Variables:** Parameters for Terraform modules, making configurations flexible and reusable.

```
variable "instance_type" {
 type        = string
 default     = "t2.micro"
 description = "EC2 instance type"
}
```

  - **Usage:**

```
resource "aws_instance" "web" {
 ami           = "ami-0c55b159cbfafe1f0"
 instance_type = var.instance_type
}
```

- **Setting Variables:**

- **Environment Variables:** export TF_VAR_instance_type="t2.small"
- **Command-Line Flags:** terraform apply
  - var="instance_type=t2.small"
- **Variable Files:** Create .tfvars files and pass them with -var-file flag.

- **Output Variables:**
  - Used to expose values to the user or other

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

9. How do you handle secrets or sensitive data in Terraform? Answer:

- **Sensitive Variables:**
  - Mark variables as sensitive to prevent them from being displayed in logs.

```
variable "db_password" {
  type      = string
  sensitive = true
}
```

- **Avoid Hardcoding:**
  - Do not store secrets in code or version control.
  - Use environment variables or prompt for input.
- **Use Vault or Secret Management Services:**
  - Integrate with tools like HashiCorp Vault to fetch secrets at runtime.
- **Secure State Storage:**
  - Use encrypted remote backends to store state files securely.

- **Example of Fetching a Secret from**

```
data "vault_generic_secret" "db_password" {
  path = "secret/data/db_password"
}

resource "aws_db_instance" "example" {
  password = data.vault_generic_secret.db_password.data["password"]
  # Other configurations
}
```

10. What is the purpose of the terraform init

command? Answer:

- **terraform init:**
    - Initializes a Terraform working directory by downloading and installing the necessary providers and modules.
    - **Functions:**
        - **Plugin Installation:** Downloads provider plugins required for the configuration.
        - **Backend Initialization:** Sets up the backend for state storage.
        - **Module Installation:** Downloads modules from sources like GitHub or the Terraform Registry.
    - **When to Run:**
        - First time setting up a configuration.
        - After adding or changing providers or modules.
        - After cloning a repository containing Terraform configurations.

How does Terraform handle resource dependencies?

Answer:

- **Implicit Dependencies:**
  - Terraform automatically determines resource dependencies by analyzing references in configurations.

```
resource "aws_instance" "example" {
 ami           = "ami-0c55b159cbfafe1f0"
 instance_type = "t2.micro"
 subnet_id     = aws_subnet.example.id
}

resource "aws_subnet" "example" {
 vpc_id = aws_vpc.example.id
 cidr_block = "10.0.1.0/24"
}
```

Here, the aws_instance depends on aws_subnet because of the subnet_id reference.

- **Explicit Dependencies:**
  - Use depends_on when a dependency isn't detected

automatically.

```
resource "null_resource" "example" {
 depends_on = [aws_instance.example]
}
```

12. How do you manage remote state in

Terraform? Answer:

Remote state is used to share the state file among team members and secure it.

- **Example Using AWS S3:**

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

- **Features:**
  - **Storage:** Stores the state in a remote backend like S3, Azure Blob, or Terraform Cloud.
  - **Locking:** Prevents concurrent changes using mechanisms like DynamoDB tables.

13. What are Terraform data sources, and how are they used? Answer:

- **Data Sources:**
  - Allow you to fetch existing information or resources from a provider.
- **Example:**

```
data "aws_ami" "example" {
  most_recent = true
  owners      = ["self"]

  filter {
    name   = "name"
```

```
  values = ["my-ami-*"]
 }
}

resource "aws_instance" "example" {
 ami           = data.aws_ami.example.id
 instance_type = "t2.micro"
}
```

14. What is the terraform apply command, and how does it differ from terraform plan?

Answer:

- **terraform plan:**
  - Shows the changes Terraform will make to your infrastructure without actually applying them.
  - Use for review and approval.
- **terraform apply:**
  - Executes the changes proposed in the plan, creating, modifying, or destroying resources as necessary.

15. What is the difference between count and for_each in Terraform? Answer:

- **count:**
  - Creates multiple resources by a specified number.

```
resource "aws_instance" "example" {
 count         = 3
 instance_type = "t2.micro"
```

```
  ami           = "ami-0c55b159cbfafe1f0"
}
```

- Accessed using count.index.

- **for_each:**
  - Creates resources based on a map or a set.

```
resource "aws_instance" "example" {
  for_each = {
    server1 = "t2.micro"
    server2 = "t2.small"
  }

  instance_type = each.value
  ami           = "ami-0c55b159cbfafe1f0"
}
```

- Accessed using each.key and each.value.

16. How do you debug errors in Terraform? Answer:

- **Debugging Steps:**
  - **Enable Debug Logs:** Set the TF_LOG environment variable.

```
export TF_LOG=DEBUG
terraform apply
```

  - **Log Output File:** Redirect logs to a file for detailed review. `export TF_LOG_PATH="terraform.log"`

- **Validate Configurations:** Use terraform validate to check for syntax errors.
- **Plan Execution:** Run terraform plan to identify issues in execution plans.

17. Explain the difference between local-exec and remote-exec provisioners. Answer:

- **local-exec:**
  - Executes commands on the machine running Terraform.

```
resource "null_resource" "example" {
  provisioner "local-exec" {
    command = "echo 'Hello, World!'"
  }
}
```

- **remote-exec:**
  - Executes commands on a remote resource (e.g., an EC2 instance).

```
resource "aws_instance" "example" {
  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx"
    ]
  }
}
```

18. What is a null_resource in Terraform, and when would you use it? Answer:

- **null_resource:**
  - A resource that doesn't directly manage infrastructure but allows running provisioners and triggers.
- **Example:**

```
resource "null_resource" "example" {
  provisioner "local-exec" {
    command = "echo 'Triggered by change in variables!'"
  }

  triggers = {
    variable = var.example_variable
  }
}
```

- **Use Cases:**
  - Execute local commands or scripts based on conditions.
  - Handle non-infrastructure workflows.

19. What is terraform fmt, and why is it important? Answer:

- **terraform fmt:**
  - Formats Terraform configuration files to ensure consistent style.
  - Run it in the directory containing .tf files: `terraform fmt`
- **Importance:**
  - Improves readability and standardizes configuration files.

20. What is the purpose of the terraform taint command?

Answer:

- **terraform taint:**
    - Marks a resource as needing to be destroyed and recreated during the next terraform apply.
- **Example:**

```
terraform taint aws_instance.example
```

- **Use Case:**
    - When a resource is in an inconsistent state or needs to be updated due to external changes.

What is the difference between terraform destroy and terraform apply -destroy?

Answer:

- **terraform destroy:**
    - Deletes all the resources defined in the current state file.

```
terraform destroy
```

- **terraform apply -destroy:**
    - Combines terraform plan and terraform destroy into one command, showing a plan before destruction.

```
terraform apply -destroy
```

22. How do you roll back changes in Terraform if something goes wrong? Answer:

- **Options for Rollback:**
    - **State Restoration:** Restore a previous state backup if state file corruption occurs.

```
cp terraform.tfstate.backup terraform.tfstate
```

- **Revert Code Changes:** Revert to a previous commit in version control and reapply:

```
git checkout <commit-id>
terraform apply
```

- **Manual Correction:** Edit configurations and use terraform plan to apply corrective changes.

23. What are Terraform modules, and how do you create a reusable module? Answer:

- **Terraform Modules:**
  - A way to encapsulate resources for reuse.
- **Steps to Create a Module:**
  - **Structure:**

```
├── main.tf
├── variables.tf
├── outputs.tf
```

  - **Module Definition:**

```
# main.tf
resource "aws_instance" "example" {
  ami           = var.ami
  instance_type = var.instance_type
}

# variables.tf
variable "ami" {}
variable "instance_type" {
  default = "t2.micro"
```

```
}

# outputs.tf
output "instance_id" {
  value = aws_instance.example.id
}
```

- **Use the Module:**

```
module "example_instance" {
  source        = "./path/to/module"
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.small"
}
```

24. What is drift detection in Terraform, and how do you handle drift? Answer:

- **Drift Detection:**
  - Drift occurs when resources are modified outside Terraform, causing the actual state to differ from the state file.
- **How to Handle Drift:**
  - Run terraform plan to detect changes.
  - Apply the plan to reconcile the drift: `terraform apply`

25. What is the purpose of the terraform state command? Answer:

- **Purpose:**

```
}
```

- Manage Terraform's state file.

- **Common Commands:**
    - **List Resources:**

```
terraform state list
```

    - **Show Resource Details:**

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

    - **Move Resources:**

```
terraform state mv old_resource new_resource
```

    - **Remove Resources:**

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

26. How does Terraform handle versioning for configurations and providers? Answer:

- **Provider Versioning:**
    - Use the required_providers block to specify provider versions.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}
```

- **Terraform Versioning:**
    - Specify the required Terraform version in the configuration.

```
terraform {
  required_version = ">= 1.1.0"
```

```
}
```

27. Can Terraform handle circular dependencies? Answer:

- **Circular Dependencies:**
  - Terraform detects and reports circular dependencies during a plan or apply operation.
- **Solution:**
  - Refactor configurations to remove circular dependencies.
  - Use depends_on to explicitly define dependencies.

28. What is the purpose of terraform workspace, and when would you use it? Answer:

- **Purpose:**
  - Manage multiple instances of state files for different environments or teams.
- **Commands:**
  - **List Workspaces:**

```
terraform workspace list
```

  - **Create a Workspace:**

```
terraform workspace new development
```

  - **Switch Workspaces:**

```
terraform workspace select production
```

29. How do you use a backend configuration in Terraform? Answer:

- **Backend Configuration:**

- Defines where Terraform stores state.

- **Example Using AWS S3:**

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

- **Initializing Backend:**

```
terraform init
```

30. How can you secure sensitive outputs in Terraform? Answer:

- **Sensitive Outputs:**
  - Mark outputs as sensitive to hide them in the Terraform CLI output.

```
output "db_password" {
  value     = aws_secretsmanager_secret.example.secret_string
  sensitive = true
}
```

- **Best Practices:**
  - Store secrets in a secure vault like HashiCorp Vault.
  - Use remote state with encryption for storing state files.

31. What is the purpose of the terraform output command? Answer:

- **Purpose:**

- Displays the values of outputs defined in the configuration after a successful apply.

- **Usage:**

```
terraform output
```

- **Access Specific Output:**

```
terraform output instance_ip
```

- **Sensitive Outputs:**
  - If marked sensitive, outputs won't be displayed in plain text.

32. How does Terraform handle concurrent operations in a team environment?

Answer:

- **State Locking:**
  - Terraform uses state locking to prevent simultaneous changes to the state file.
  - Remote backends like S3 with DynamoDB for locking ensure safe operations.
- **Handling Lock Issues:**

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

- **Terraform Cloud:**
  - Offers remote state management with built-in locking and collaboration features.

33. What are Terraform dynamic blocks, and how are they used? Answer:

- **Dynamic Blocks:**
  - Allow programmatic generation of nested blocks.
- **Example:**

```
resource "aws_security_group" "example" {
  name        = "example"

  dynamic "ingress" {
    for_each = var.ingress_rules
    content {
      from_port   = ingress.value.from_port
      to_port     = ingress.value.to_port
      protocol    = ingress.value.protocol
      cidr_blocks = ingress.value.cidr_blocks
    }
  }
}
```

34. What is the purpose of the terraform refresh command? Answer:

- **Purpose:**
  - Updates the state file with the latest real-world infrastructure data without applying any changes.
- **Usage:**

```
terraform refresh
```

- **Common Use Case:**
  - To detect and update drift between the configuration and actual infrastructure.

35. What are the limitations of Terraform? Answer:

- **State Management:**
  - The state file must be managed carefully to avoid conflicts or corruption.
- **No Native Rollback:**
  - Terraform does not have built-in rollback functionality.
- **Lack of Procedural Logic:**
  - Terraform is declarative and does not support complex procedural logic.
- **Limited Provider Support:**
  - New or niche providers may not be supported.

36. How do you handle Terraform state file locking in a remote backend? Answer:

- **Using AWS S3 with DynamoDB Lock Table:**
  - Add DynamoDB as a locking mechanism for the state file.

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

- **Why?**
  - Prevents concurrent state modifications in team environments.

37. What are Terraform local values, and how are they used?

Answer:

- **Local Values:**
  - Temporary variables to simplify complex expressions.
- **Example:**

```
locals {
 instance_count = length(var.instance_types)
 ami_id        = "ami-0c55b159cbfafe1f0"
}

resource "aws_instance" "example" {
 count         = local.instance_count
 ami           = local.ami_id
 instance_type = var.instance_types[count.index]
}
```

38. How does Terraform support conditional resource creation? Answer:

- **Using count:**
  - Set count to 0 to skip creating a resource.

```
resource "aws_instance" "example" {
 count         = var.create_instance ? 1 : 0
 ami           = "ami-0c55b159cbfafe1f0"
 instance_type = "t2.micro"
}
```

- **Using for_each:**
  - Create resources based on conditions in a map or list.

39. What is the purpose of the terraform console command? Answer:

- **Purpose:**
  - Opens an interactive shell to experiment with Terraform expressions.
- **Usage Example:**

```
terraform console
> length([1, 2, 3])
3
```

- **Use Case:**
  - Debugging and testing expressions without applying changes.

40. What are resource taints in Terraform, and how do you manually taint a resource?

Answer:

- **Resource Tainting:**
  - Marks a resource for destruction and recreation during the next apply.
- **Command:**

```
terraform taint <resource_name>
```

- **Removing Taint:**

```
terraform untaint <resource_name>
```

- **Use Case:**
  - Forcefully replace resources when updates are not supported.

41. What are Terraform backends, and why are they important? Answer:

- **Backends:**
  - Define where Terraform stores the state file (local or remote).
  - Support features like remote state storage, locking, and collaboration.
- **Example: S3 Backend**

```
terraform {
 backend "s3" {
  bucket = "my-terraform-state"
  key    = "path/to/statefile"
  region = "us-west-2"
 }
}
```

- **Importance:**
  - Enable secure, shared state management.
  - Prevent state corruption in team environments.

42. What is the purpose of the terraform import command? Answer:

- **Purpose:**
  - Imports existing infrastructure into Terraform's state file.
- **Steps:**

1. Write the resource block in your configuration.
2. Run the import command:

```
terraform import aws_instance.example i-0abcdef1234567890
```

- **Limitations:**
  - Only updates the state file, not the configuration.

43. How does Terraform manage resource

Answer:

- **Resource Lifecycle Meta-Arguments:**
  - create_before_destroy: Ensures a new resource is created before deleting the old one.
  - prevent_destroy: Prevents accidental resource deletion.
  - ignore_changes: Ignores changes to specific attributes.
- **Example:**

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

  lifecycle {
    create_before_destroy = true
    prevent_destroy       = true
    ignore_changes        = [tags]
  }
}
```

44. How can you manage secrets securely in

Terraform? Answer:

- **Options:**
  1. **Environment Variables:**

```
export TF_VAR_db_password="secure_password"
```

  2. **Secret Management Tools:** Use HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault.

```
data "vault_generic_secret" "example" {
```

```
 path = "secret/data/db_password"
}
```

3. **Sensitive Variables:**

```
variable "db_password" {
 type      = string
 sensitive = true
}
```

45. What is the difference between terraform validate and terraform plan? Answer:

- **terraform validate:**
  - Checks the syntax and correctness of the configuration files.
  - Does not interact with providers or the state.
- **terraform plan:**
  - Simulates infrastructure changes by interacting with the provider and state.
  - Generates an execution plan.

46. Can you explain Terraform provider version constraints? Answer:

- **Purpose:**
  - Ensures the correct version of a provider is used.
- **Examples:**
  - Exact Version:

```
provider "aws" {
 version = "= 3.5.0"
}
```

- Range:

```
provider "aws" {
 version = "~> 3.0"
}
```

- **Why?**
  - Prevents breaking changes when updating providers.

47. How does Terraform handle zero-downtime deployments? Answer:

- **Strategies:**
  - Use create_before_destroy in the lifecycle block.

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

 lifecycle {
  create_before_destroy = true
 }
}
```

  - Leverage external tools like Terraform Enterprise or third-party orchestration for canary deployments.

48. What is the difference between provider and provisioner in Terraform? Answer:

- **Provider:**
  - Integrates with APIs to manage infrastructure (e.g., AWS, Azure, GCP).

- Example:

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

- **Provisioner:**
  - Executes scripts or commands on resources during creation or destruction.
  - Example:

```
provisioner "remote-exec"
{
 inline = ["sudo apt-get update"]
}
```

49. How do you manage shared modules in Terraform? Answer:

- **Options:**
  - **Module Registry:** Use Terraform Module

Registry. module "vpc" {

source = "terraform-aws-modules/vpc/aws"

```
 version = "2.77.0"
}
```

  - **Git Repository:** Source modules from Git.

```
module "example" {
 source = "git::https://github.com/user/repo.git//module-path"
}
```

  - **Local Directory:** Store modules

locally. module "local_module" {

- Example:

```
provider "aws" {
 region = "us-west-2"
}
 source = "./path/to/module"
```

```
}
```

50. What is Terraform Cloud, and how does it differ from Terraform CLI? Answer:

- **Terraform Cloud:**
  - A managed service for Terraform with state management, team collaboration, and remote execution.
  - Features include VCS integration, notifications, and workspaces.
- **Terraform CLI:**
  - Runs Terraform commands locally.
  - Requires manual management of state files and collaboration tools.

Conclusion

Mastering Terraform is crucial for professionals aiming to excel in cloud and DevOps roles. This collection of interview questions provides a well-rounded view of Terraform's capabilities, from basic configurations and state management to advanced features like dynamic blocks, modules, and zero-downtime deployments. By familiarizing yourself with these topics, you will not only enhance your Terraform expertise but also be better prepared to tackle real-world infrastructure challenges. Terraform’s growing adoption across industries underscores the importance of honing these skills for a successful career in DevOps and cloud engineering.