MCQ

1. A user runs `terraform init` on their RHEL-based server, and per the output, two provider plugins are downloaded:

```
$ terraform init
 Initializing the backend...
 Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.44.0...
- Downloading plugin for provider "random" (hashicorp/random) 2.2.1...
Terraform has been successfully initialized!
```

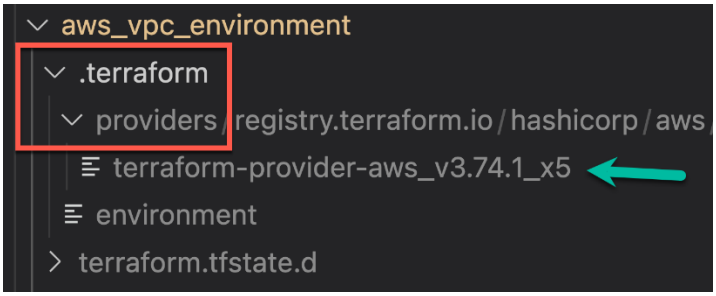
Where are these plugins downloaded and stored on the server?

   a. **`/etc/terraform/plugins`**
   b. **The `.terraform.plugins` directory in the current working directory**
   c. **The `.terraform.d` directory in the current working directory**
   d. **The `.terraform/providers` directory in the current working directory**

Explanation

By default, `terraform init` downloads plugins into a subdirectory of the working directory, `.terraform/providers` so that each working directory is self-contained.

See the example below, where I ran a terraform init and you can see the resulting directory (highlighted in the red box) and then the actual provider that was downloaded (highlighted by the green arrow)



2. You are performing a code review of a colleague's Terraform code and see the following code. Where is this module stored?

```
module "vault-aws-tgw" {
  source  = "terraform-vault-aws-tgw/hcp"
  version = "1.0.0"
  client_id      = "4djlsn29sdnjk2btk"
  hvn_id         = "a4c9357ead4de"
  route_table_id = "rtb-a221958bc5892eade331"
}
```

   a. **the Terraform public module registry**
   b. in a Terraform Cloud private module registry
   c. a local code repository on your network

d. in a local file under a directory named terraform/vault-aws-tgw/hcp

Explanation

You can use the Terraform Public Module Registry by referencing the modules you want to use in your Terraform code and including them as part of your configuration.

To reference a module from the Terraform Public Module Registry, you can use the **module** block in your Terraform code. For example, if you want to use a VPC module from the registry, you would add the following code to your Terraform configuration:

```
module "vpc" {
  source = "terraform-aws-modules/vpc"
  version = "2.34.0"
   # Add any required variables and configuration here
}
```

The **source** attribute specifies the module source, which is the repository on the Terraform Public Module Registry. The **version** attribute specifies the version of the module you want to use.

You can also pass values for variables in the module by them within the **module** block. For example:

```
module "vpc" {
  source = "terraform-aws-modules/vpc"
  version = "2.34.0"
  name = "my-vpc"
  cidr = "10.0.0.0/16"
  azs  = ["us-west-2a", "us-west-2b", "us-west-2c"]
}
```

Once you've specified the module in your Terraform code, you can use it as you would any other resource. For example, you could reference the VPC ID created by the VPC module with the following code:

```
output "vpc_id" {
  value = module.vpc.vpc_id
}
```

3. In the example below, the depends_on argument creates what type of dependency?

```
resource "aws_instance" "example" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
  depends_on = [aws_s3_bucket.company_data]
}
```

a. implicit dependency
b. explicit dependency
c. internal dependency
d. non-dependency resource

Explanation

Explicit dependencies in Terraform are dependencies that are explicitly declared in the Terraform configuration. These dependencies are used to control the order in which Terraform creates, updates, and destroys resources.

In Terraform, you can declare explicit dependencies using the **depends_on** argument in a resource block. The **depends_on** argument takes a list of resource names and specifies that the resource block in which it is declared depends on those resources.

For example, consider a scenario where you have a virtual machine (VM) that depends on a virtual network (VNET) and a subnet. You can declare these dependencies using the **depends_on** argument as follows:

```
resource "azurerm_virtual_network" "vnet" {
  name                = "example-vnet"
  address_space       = ["10.0.0.0/16"]
}
resource "azurerm_subnet" "subnet" {
  name                 = "example-subnet"
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefix       = "10.0.1.0/24"
}
resource "azurerm_network_interface" "nic" {
  name                = "example-nic"
  location            = azurerm_virtual_network.vnet.location
  subnet_id           = azurerm_subnet.subnet.id
  depends_on = [
    azurerm_subnet.subnet,
    azurerm_virtual_network.vnet
  ]
}
```

In this example, the **azurerm_network_interface** resource depends on both the **azurerm_subnet** and the **azurerm_virtual_network** resources, so Terraform will create those resources first, and then create the **azurerm_network_interface** resource.

By declaring explicit dependencies, you can ensure that Terraform creates resources in the correct order, so that dependent resources are available before other resources that depend on them. This helps prevent errors or unexpected behavior when creating or modifying infrastructure, and makes it easier to manage and understand the relationship between resources.

Overall, the use of explicit dependencies is a critical aspect of Terraform, as it helps ensure that resources are created and managed in the correct order and makes it easier to manage and understand the relationship between resources.

4. Why might a user opt to include the following snippet in their configuration file?

```
terraform {
  required_version = ">= 1.3.8"
}
```

a. **The user wants to specify the minimum version of Terraform that is required to run the configuration**
b. The user wants to ensure that the application being deployed is a minimum version of 1.3.8
c. this ensures that all Terraform providers are above a certain version to match the application being deployed
d. versions before Terraform 1.3.8 were not approved by HashiCorp to be used in production

Explanation

The **required_version** parameter in a **terraform** block is used to specify the minimum version of Terraform that is required to run the configuration. This parameter is optional, but it can be useful for ensuring that a Terraform configuration is only run with a version of Terraform that is known to be compatible.

For example, if your Terraform configuration uses features that were introduced in Terraform 1.3.8, you could include the following **terraform** block in your configuration to ensure that Terraform 1.3.8 or later is used:

```
terraform {
  required_version = ">= 1.3.8"
}
```

When you run Terraform, it will check the version of Terraform that is being used against the **required_version** parameter and it will raise an error if the version is lower than the required version.

This can be especially useful in larger organizations or projects where multiple people are working on the same Terraform code, as it helps to ensure that everyone is using the same version of Terraform and reduces the risk of encountering unexpected behavior or bugs due to differences in Terraform versions.

5. A user creates three workspaces from the command line: `prod`, `dev`, and `test`. Which of the following commands will the user run to switch to the `dev` workspace?

a. `terraform workspace switch dev`
b. `terraform workspace dev`
c. `terraform workspace -switch dev`
d. **`terraform workspace select dev`**

Explanation

The command used to switch to the dev workspace in Terraform is **`terraform workspace select dev`**.

Terraform workspaces allow you to manage multiple sets of infrastructure resources that share the same configuration. To switch to a specific workspace in Terraform, you use the **`terraform workspace select`** command followed by the name of the workspace you want to switch to. In this case, the name of the workspace is "dev".

After running this command, Terraform will switch to the **dev** workspace, and all subsequent Terraform commands will apply to the resources in that workspace. If the **dev** workspace does not yet exist, Terraform will **NOT** create it for you.

Here's an example of using the **terraform workspace select** command to switch to the dev workspace:

```
$ terraform workspace select dev
Switched to workspace "dev".
```

6. You are adding a new variable to your configuration. Which of the following is **NOT** a valid variable type in Terraform?

a. string
b. map
c. bool
d. number
e. float

Explanation

The Terraform language uses the following types for its values: string, number, bool, list (or tuple), map (or object). There are no other supported variable types in Terraform, therefore, **float** is incorrect in this question.

Don't forget that variable types are included in a variable block, but they are NOT required since Terraform interprets the type from a default value or value provided by other means (ENV, CLI flag, etc)

```
variable "practice-exam" {
  description = "bryan's terraform associate practice exams"
  type        = string
  default     = "highly-rated"
}
```

7. Which of the following is a ***valid*** variable name in Terraform?

a. module
b. 245
c. count
d. invalid

Explanation

In Terraform, variable names must follow a set of naming conventions to be considered valid. Here are some examples of invalid variable names:

Names that start with a number: **1_invalid_variable_name**

Names that contain spaces or special characters (other than underscores): **invalid variable name**

Names that contain only numbers: **12345**

Names that are the same as Terraform reserved words, such as **var**, **module**, **data**, **count**, etc.

It is recommended to use only lowercase letters, numbers, and underscores in variable names and to start variable names with a lowercase letter to ensure they are valid. Additionally, variable names should be descriptive and meaningful to help make your Terraform code more readable and maintainable.

8. Oscar is modifying his Terraform configuration file but isn't 100% sure it's correct. He is afraid that changes made could negatively affect production workloads. How can Oscar validate the changes that will be made without impacting existing workloads?

a. run `terraform apply` using a *local-exec provisioner* so the configuration won't impact existing workloads
b. run a `terraform validate` to ensure the changes won't impact the production workloads
c. run a `terraform plan` and validate the changes that will be made
d. run `terraform refresh` to compare his existing configuration file against the current one

Explanation

The `terraform plan` command is used to create an execution plan. Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.

This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or the state.

9. After many years of using Terraform Open Source (OSS), you decide to migrate to Terraform Cloud. After the initial configuration, you create a workspace and migrate your existing state and configuration. What Terraform version would the new workspace be configured to use after the migration?

a. the same Terraform version that was used to perform the migration
b. whatever version is defined in the `terraform` block
c. the most recent version of Terraform available
d. the latest major release of Terraform

Explanation

When you create a new workspace, Terraform Cloud automatically selects the most recent version of Terraform available. **If you migrate an existing project from the CLI to Terraform Cloud, Terraform Cloud configures the workspace to use the same version as the Terraform binary you used when migrating**. Terraform Cloud lets you change the version a workspace uses on the workspace's settings page to control how and when your projects use newer versions of Terraform.

It's worth noting that Terraform Cloud also provides the ability to upgrade your Terraform version in a controlled manner. This allows you to upgrade your Terraform version in a safe and predictable way, without affecting your existing infrastructure or state.

10. What environment variable can be set to enable detailed logging for Terraform?

a. `TF_LOG`
b. `TF_DEBUG`
c. `TF_INFO`
d. `TF_TRACE`

Explanation

Terraform has detailed logs that can be enabled by setting the `TF_LOG` environment variable to any value. This will cause detailed logs to appear on stderr.

You can set `TF_LOG` to one of the log levels `TRACE`, `DEBUG`, `INFO`, `WARN` or `ERROR` to change the verbosity of the logs. `TRACE` is the most verbose and it is the default if `TF_LOG` is set to something other than a log level name.

11. You are using Terraform to deploy some cloud resources and have developed the following code. However, you receive an error when trying to provision the resource. Which of the following answer fixes the syntax of the Terraform code?

```
resource "aws_security_group" "vault_elb" {
    name       = "${var.name_prefix}-vault-elb"
    description = Vault ELB
    vpc_id     = var.vpc_id
  }
```

a.
```
resource "aws_security_group" "vault_elb" {
    name       = "${var.name_prefix}-vault-elb"
    description = [Vault ELB]
    vpc_id     = var.vpc_id
 }
```

b.
```
resource "aws_security_group" "vault_elb" {
    name       = "${var.name_prefix}-vault-elb"
    description = "Vault ELB"
    vpc_id     = var.vpc_id
}
```

c.
```
resource "aws_security_group" "vault_elb" {
    name       = "${var.name_prefix}-vault-elb"
    description = var_Vault ELB
    vpc_id     = var.vpc_id
}
```

d.
```
resource "aws_security_group" "vault_elb" {
    name       = "${var.name_prefix}-vault-elb"
    description = "${Vault ELB}"
    vpc_id     = var.vpc_id
}
```

Explanation

When assigning a value to an argument in Terraform, there are a few requirements that must be met:

Data type: The value must be of the correct data type for the argument. Terraform supports several data types, including **strings**, numbers, booleans, lists, and maps.

Value constraints: Some arguments may have specific value constraints that must be met. For example, an argument may only accept values within a certain range or values from a specific set of values.

When assigning a value to an argument expecting a string, it must be enclosed in quotes ("...") unless it is being generated programmatically.

12. Which code snippet would allow you to retrieve information about existing resources and use that information within your Terraform configuration?

a. module "deploy-servers" {

```
  source = "./app-cluster"
  servers = 5
}
```

b.
```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

c.
```
provider "google" {
  project = "acme-app"
  region  = "us-central1"
}
```

d.
```
data "aws_ami" "aws_instance" {
  most_recent = true
  owners = ["self"]
  tags = {
     Name   = "app-server"
    Tested = "true"
  }
}
```

e.
```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

Explanation

In Terraform, `data` blocks are used to retrieve data from external sources, such as APIs or databases, and make that data available to your Terraform configuration. With `data` blocks, you can use information from external sources to drive your infrastructure as code, making it more dynamic and flexible.

For example, you can use a `data` block to retrieve a list of Amazon Machine Images (AMIs) from AWS, and then use that data to select the appropriate AMI for a virtual machine you are provisioning:

```
data "aws_ami" "example" {
  most_recent = true
   filter {
    name   = "name"
    values = ["amzn2-ami-hvm-2.0.*-x86_64-gp2"]
  }
   filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}
resource "aws_instance" "example" {
```

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

In this example, the **data** block retrieves the most recent Amazon Linux 2 HVM AMI, and the **aws_instance** resource uses the selected AMI to create a virtual machine.

Data blocks can be used to retrieve information from a wide range of sources, such as databases, APIs, and cloud providers. This information can then be used to conditionally create, update, or delete resources, making your Terraform configurations more flexible and adaptable to changing requirements.

13. You are writing Terraform to deploy resources, and have included provider blocks as shown below:

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

When you validate the Terraform configuration, you get the following error:

```
Error: Duplicate provider configuration
on main.tf line 5:
  5: provider "aws" {
A default provider configuration for "aws" was already given at
main.tf:1,1-15. If multiple configurations are required, set the xxxx
argument for alternative configurations.
```

What additional configuration is needed to use multiple provider blocks of the same type, but refer to unique configurations, such as a cloud region, namespace, or other desired configuration?

a. version
b. alias
c. multi
d. label

Explanation

An alias meta-argument is used when using the same provider with different configurations for different resources. This feature allows you to include multiple provider blocks that refer to different configurations. In this example, you would need something like this:

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

When writing Terraform code to deploy resources, the resources that you want to deploy to the "**west**" region would need to specify the alias within the resource block. This instructs Terraform to use the configuration specified in that provider block. So in this case, the resource would be deployed to "**us-west-2**" region and not the "**us-east-1**" region. this configuration is common when using multiple cloud regions or namespaces in applications like Consul, Vault, or Nomad.

14. You and a colleague are working on updating some Terraform configurations within your organization. You need to follow a new naming standard for the local name within your resource blocks. However, you don't want Terraform to replace the object after changing your configuration files. As an example, you want to change `data-bucket` to now be `prod-encrypted-data-s3-bucket` in the following resource block:

```
resource "aws_s3_bucket" "data-bucket" {
  bucket = "corp-production-data-bucket"
   tags = {
    Name        = "corp-production-data-bucket"
    Environment = "prod"
  }
}
```

After updating the resource block, what command would you run to update the local name while ensuring Terraform does not replace the existing resource?

a. `terraform apply -refresh-only`
b. `terraform apply -replace aws_s3_bucket.data-bucket`
c. `terraform state rm aws_s3_bucket.data-bucket`
d. `terraform state mv aws_s3_bucket.data-bucket aws_s3_bucket.prod-encrypted-data-s3-bucket`

Explanation

You can use `terraform state mv` when you wish to retain an existing remote object but track it as a different resource instance address in Terraform, such as if you have renamed a resource block or you have moved it into a different module in your configuration.

In this case, Terraform would not touch the actual resource that is deployed, but it would simply attach the existing object to the new address in Terraform.

`terraform apply - replace` - this would cause Terraform to replace the resource

`terraform apply - refresh-only` - this command is used to reconcile any changes to the real-world resources and update state to reflect those changes. It would not help us solve our problem

`terraform state rm` - This command is used to remove a resource from state entirely while leaving the real-world resource intact. The bucket would still exist but it wouldn't help us with renaming our resource in our configuration file.

15. Henry has been working on automating his Azure infrastructure for a new application using Terraform. His application runs successfully, but he has added a new resource to create a DNS record using the new Infoblox provider. He has added the new resource but gets an error when he runs a `terraform plan`. What should Henry do first before running a `plan` and `apply`?

a. **since a new provider has been introduced, terraform init needs to be run to download the Infoblox plugin**
b. Henry should run a terraform plan -refresh=true to update the state for the new DNS resource
c. you can't mix resources from different providers within the same configuration file, so Henry should create a module for the DNS resource and reference it from the main configuration
d. the Azure plugin doesn't support Infoblox directly, so Henry needs to put the DNS resource in another configuration file

Explanation

In this scenario, Henry has introduced a new provider. Therefore, Terraform needs to download the plugin to support the new resource he has added. Running **terraform init** will download the Infoblox plugin. Once that is complete, a `plan` and `apply` can be executed as needed.

You would need to rerun **terraform init** after modifying your code for the following reasons:

Adding a new provider: If you've added a new provider to your code, you'll need to run **terraform init** to download the provider's binary and configure it.

Updating the provider configuration: If you've updated the configuration of an existing provider, you'll need to run **terraform init** to apply the changes.

Updating the version of a provider: If you've updated the version of a provider, you'll need to run **terraform init** to download the updated version of the provider's binary.

Adding or removing a module: If you've added or removed a module from your code, you'll need to run **terraform init** to download the required modules and dependencies.

In short, **terraform init** is used to initialize a Terraform working directory, and you'll need to rerun it whenever you make changes to your code that affect the providers, modules, or versions you're using.

16. What Terraform command can be used to remove the lock on the state for the current configuration?

a. `terraform unlock`
b. `terraform state-unlock`
c. Removing the lock on a state file is not possible
d. `terraform force-unlock`

Explanation

The **terraform force-unlock** command can be used to remove the lock on the Terraform state for the current configuration. Another option is to use the "terraform state rm" command followed by the "terraform state push" command to forcibly overwrite the state on the remote backend, effectively removing the lock. It's important to note that these commands should be used with caution, as they can potentially cause conflicts and data loss if not used properly.

Be very careful forcing an unlock, as it could cause data corruption and problems with your state file.

17. Understanding how indexes work is essential when working with different variable types and resource blocks that use `count` or `for_each`. Therefore, what is the output value of the following code snippet?

```
variable "candy_list" {
  type = list(string)
  default = ["snickers", "kitkat", "reeces", "m&ms"]
}
output "give_me_candy" {
  value = "${lookup(var.candy_list, 2)}"
}
```

a. m&ms
b. reeces
c. snickers
d. kitkat

Explanation

In this example, the **candy_list** variable is a list of strings, and the **output** block retrieves the third element in the list (at index 2) and outputs it as the value of **give_me_candy**.
Remember that an index starts at [0], and then counts up. Therefore, the following represents the index value as shown in the variable above:
[0] = snickers
[1] = kitkat
[2] = reeces
[3] = m&ms

18. While Terraform is generally written using the HashiCorp Configuration Language (HCL). What other syntax can Terraform be expressed in?

a. YAML
b. TypeScript
c. XML
d. JSON

Explanation

Terraform can be expressed using two syntaxes: **HashiCorp Configuration Language (HCL)**, which is the primary syntax for Terraform, and **JSON**.
The *HCL syntax is designed to be human-readable and easy to write*, and it provides many features designed explicitly for Terraform, such as interpolation, variables, and modules.
The ***JSON syntax is a machine-readable alternative*** to HCL, and it is typically used when importing existing infrastructure into Terraform or when integrating Terraform with other tools that expect data in JSON format.
While Terraform will automatically detect the syntax of a file based on its extension, you can also specify the syntax explicitly by including a **terraform** stanza in the file, as follows:

```
# HCL syntax Example
# terraform { }
# JSON syntax Exmample
{
  "terraform": {}
}
```

Note that while JSON is supported as a syntax, it is not recommended to use it for writing Terraform configurations from scratch, as the HCL syntax is more user-friendly and provides better support for Terraform's specific features.

19. In the following code snippet, the `block type` is identified by which string?

```
resource "aws_instance" "db" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

a. `aws_instance`
b. `db`
c. `resource`
d. `instance_type`

Explanation

In Terraform, resource blocks define the resources you want to create, update, or manage as part of your infrastructure. Other type of block types in Terraform include `provider`, `terraform`, `output`, `data`, and `resource`.

The format of a resource block configuration is as follows:

```
resource "TYPE" "NAME" {
  [CONFIGURATION_KEY = CONFIGURATION_VALUE]
  ...
}
```

where:

`TYPE` is the type of resource you want to create, such as an AWS EC2 instance, an Azure storage account, or a Google Cloud Platform compute instance.

`NAME` is a unique identifier for the resource, which you can use in other parts of your Terraform configuration to refer to this resource.

`CONFIGURATION_KEY` is a key that corresponds to a specific attribute of the resource type.

`CONFIGURATION_VALUE` is the value for the attribute specified by **`CONFIGURATION_KEY`**.

For example, here is a simple resource block that creates an Amazon Web Services (AWS) EC2 instance:

```
resource "aws_instance" "example" {
  ami           = "ami-0323c3dd2da7fb37d"
  instance_type = "t2.micro"
}
```

In this example, the **`resource`** block creates an EC2 instance with the specified Amazon Machine Image (AMI) and instance type.

It is important to note that each resource type will have its own set of required and optional attributes, and you must specify the required attributes for each resource type in your Terraform configuration. Some common attributes for AWS EC2 instances include the AMI ID, instance type, and security group.

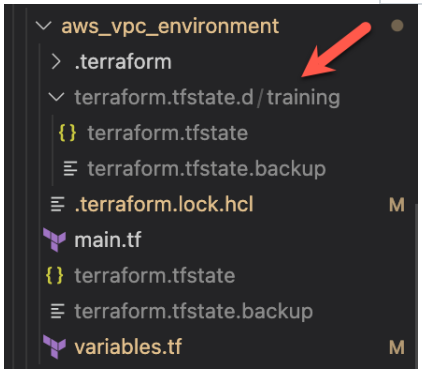
By defining resources in Terraform, you can manage your infrastructure as code and track changes to your infrastructure over time, making it easier to version control, automate, and collaborate on your infrastructure.

20. Where does Terraform Open Source (OSS) store the *local* state for workspaces?

a. a file called `terraform.tfstate.backup`
b. a file called `terraform.tfstate`
c. directory called `terraform.workspaces.tfstate`
d. directory called `terraform.tfstate.d/<workspace name>`

Explanation

Terraform Open Source (OSS) stores the local state for workspaces in a file on disk. For local state, Terraform stores the workspace states in a directory called `terraform.tfstate.d/<workspace_name>`. Here's a screenshot of a Terraform run that was created using a workspace called `training`. You can see that Terraform created the `terraform.tfstate.d` directory, and then a directory with the namespace name underneath it. Under each directory, you'll find the state file, which is name `terraform.tfstate`



21. In order to reduce the time it takes to provision resources, Terraform uses parallelism. By default, how many resources will Terraform provision concurrently during a `terraform apply`?

a. 50
b. 5
c. 10
d. 20

Explanation

By default, Terraform will provision resources concurrently with a **maximum of 10 concurrent resource operations**. This setting is controlled by the **`parallelism`** configuration option in Terraform, which can be set globally in the Terraform configuration file or on a per-module basis.

The **`parallelism`** setting determines the number of resource operations that Terraform will run in parallel, so increasing the **`parallelism`** setting will result in Terraform provisioning resources more quickly, but can also increase the risk of rate-limiting or other errors from the API.

You can adjust the **`parallelism`** setting in your Terraform configuration file by adding the following code:

```
terraform {
  parallelism = 20
}
```

This setting sets the maximum number of concurrent resource operations to 10. You can adjust this number to meet your specific needs and constraints.

22. A "backend" in Terraform determines how state is loaded and how an operation such as `apply` is executed. Which of the following is **not** a supported backend type?

a. `consul`
b. `s3`
c. `local`
d. `github`

Explanation

GitHub is not a supported backend type.

The Terraform backend is responsible for storing the state of your Terraform infrastructure and ensuring that state is consistent across all team members. Terraform state is used to store information about the resources that Terraform has created, and is used by Terraform to determine what actions are necessary when you run Terraform commands like **`apply`** or **`plan`**.

Terraform provides several backend options, including:

`local` backend: The default backend, which stores Terraform state on the local filesystem. This backend is suitable for small, single-user deployments, but can become a bottleneck as the size of your infrastructure grows or as multiple users start managing the infrastructure.

`remote` backend: This backend stores Terraform state in a remote location, such as an S3 bucket, a Consul server, or a Terraform Enterprise instance. The remote backend allows multiple users to share the same state and reduces the risk of state corruption due to disk failures or other issues.

`consul` backend: This backend stores Terraform state in a Consul cluster. Consul provides a highly available and durable storage solution for Terraform state, and also provides features like locking and versioning that are important for collaboration.

`s3` backend: This backend stores Terraform state in an S3 bucket. S3 provides a highly available and durable storage solution for Terraform state, and is a popular option for storing Terraform state for large infrastructure deployments.

When choosing a backend, you should consider the needs of your infrastructure, including the size of your deployment, the number of users who will be managing the infrastructure, and the level of collaboration that will be required. It's also important to consider the cost and performance characteristics of each backend, as some backends may be more expensive or may require more setup and maintenance than others.

23. From the code below, identify the ***implicit*** dependency:

```
resource "aws_eip" "public_ip" {
   vpc = true
   instance = aws_instance.web_server.id
}
resource "aws_instance" "web_server" {
  ami          = "ami-2757f631"
  instance_type = "t2.micro"
  depends_on = [aws_s3_bucket.company_data]
}
```

a. The AMI used for the EC2 instance
b. The EC2 instance labeled web_server
c. The EIP with an id of ami-2757f631
d. The S3 bucket labeled company_data

Explanation

Implicit dependencies are **not** explicitly declared in the configuration but are automatically detected by Terraform based on the relationships between resources. Implicit dependencies allow Terraform to automatically determine the correct order in which resources should be created, updated, or deleted, ensuring that resources are created in the right order, and dependencies are satisfied.

For example, if you have a resource that depends on another resource, Terraform will automatically detect this relationship and create the dependent resource after the resource it depends on has been created. This allows Terraform to manage complex infrastructure deployments in an efficient and predictable way.

The EC2 instance labeled **web_server** is the ***implicit*** dependency as the **aws_eip** cannot be created until the **aws_instance** labeled **web_server** has been provisioned and the **id** is available.

Note that **aws_s3_bucket.company_data** is an **explicit** dependency for the **aws_instance.web_server**

24. Which Terraform command will check and report errors within modules, attribute names, and value types to ensure they are syntactically valid and internally consistent?

a. terraform format
b. terraform fmt
c. terraform validate
d. terraform show

Explanation

The **terraform validate** command is used to check and report errors within modules, attribute names, and value types to ensure they are syntactically valid and internally consistent. This command performs basic validation of the Terraform configuration files in the current directory, checking for issues such as missing required attributes, invalid attribute values, and incorrect structure of the Terraform code.

For example, if you run **terraform validate** and there are syntax errors in your Terraform code, Terraform will display an error message indicating the line number and description of the issue. If no errors are found, the command will return with no output.

It's recommended to run **terraform validate** before running **terraform apply**, to ensure that your Terraform code is valid and will not produce unexpected results.

25. What is the correct syntax for defining a ***list of strings*** for a variable in Terraform?

a.

```
variable "public_subnet_cidr_blocks" {
 type = <removed>
 default = [
  "10.0.1.0/24",
```

```
  "10.0.1.0/24",
  "10.0.1.0/24",
  "10.0.1.0/24",
 ]
}
```

b.

```
variable "resource_tags" {
 description = "Tags to set for all resources"
 type       = <removed>
 default    = {
  project    = "exam-prep",
  environment = "prod"
  instructor  = "krausen"
 }
}
```

c.

```
variable "public_subnets" {
 description = "The number of public subnets for VPC"
 type       = <removed>
 default    = 2
}
```

d.

```
variable "aws_region" {
 description = "AWS region"
 type       = <removed>
 default    = "us-west-1"
}
```

Explanation

In Terraform, you can use a list of strings variable to store multiple string values and reference those values in your Terraform configuration. Here's how you can use a list of strings variable in Terraform:

Define the variable: To define a list of strings variable in Terraform, you need to specify the **type** as **list(string)**. Here's an example:

```
variable "example_list" {
 type = list(string)
}
```

Assign values to the variable: You can assign values to a list of strings variable in your Terraform configuration, for example:

```
variable "example_list" {
 type = list(string)
 default = ["string1", "string2", "string3"]
}
```

In this example, the **example_list** variable is defined as a list of strings and its default value is set to a list of three strings.

26. In Terraform, most resource dependencies are handled automatically. Which of the following statements describes best how Terraform resource dependencies are handled?

a. The Terraform binary contains a built-in reference map of all defined Terraform resource dependencies. Updates to this dependency map are reflected in Terraform versions. To ensure you are working with the latest resource dependency map you much be running the latest version of Terraform.

b. Terraform analyzes any expressions within a resource block to find references to other objects and treats those references as implicit ordering requirements when creating, updating, or destroying resources.

c. Resource dependencies are handled automatically by the `depends_on` meta_argument, which is set to true by default.

d. Resource dependencies are identified and maintained in a file called `resource.dependencies`. Each terraform provider is required to maintain a list of all resource dependencies for the provider and it's included with the plugin during initialization when `terraform init` is executed. The file is located in the `terraform.d` folder.

Explanation

Terraform resource dependencies control how resources are created, updated, and destroyed. When Terraform creates or modifies resources, it must be aware of any dependencies that exist between those resources. By declaring these dependencies, Terraform can ensure that resources are created in the correct order so that dependent resources are available before other resources that depend on them.

To declare a resource dependency, you can use the **`depends_on`** argument in a resource block. The **`depends_on`** argument takes a list of resource names and specifies that the resource block in which it is declared depends on those resources.

27. When multiple arguments with single-line values appear on consecutive lines at the same nesting level, HashiCorp recommends that you:

a. place all arguments using a variable at the top

```
ami = var.aws_ami
instance_type = var.instance_size
subnet_id = "subnet-0bb1c79de3EXAMPLE"
tags = {
  Name = "HelloWorld"
}
```

b. place a space in between each line

```
type = "A"
 ttl = "300"
 zone_id = aws_route53_zone.primary.zone_id
```

c. align the equals signs

```
ami           = "abc123"
instance_type = "t2.micro"
```

d. put arguments in alphabetical order

```
name = "www.example.com"
```

```
records = [aws_eip.lb.public_ip]
type = "A"
ttl = "300"
zone_id = aws_route53_zone.primary.zone_id
```

Explanation

HashiCorp style conventions suggest that you align the equals sign for consecutive arguments for easing readability for configurations:

```
ami           = "abc123"
instance_type = "t2.micro"
subnet_id     = "subnet-a6b9cc2d59cc"
```

Notice how the equal (=) signs are aligned, even though the arguments are of different lengths.

28. Emma is a Terraform expert, and she has automated *all the things* with Terraform. A virtual machine was provisioned during a recent deployment, but a local script did not work correctly. As a result, the virtual machine needs to be destroyed and recreated. How can Emma quickly have Terraform recreate ***the one resource*** without having to destroy everything that was created?

a. use `terraform apply -replace=aws_instance.web` to mark the virtual machine for replacement

b. use `terraform refresh` to refresh the state and make Terraform aware of the error

c. use `terraform import` to import the error so Terraform is aware of the problem

d. use `terraform state rm aws_instance.web` to remove the resource from the state file, which will cause Terraform to recreate the instance again

Explanation

The **`terraform apply -replace`** command manually marks a Terraform-managed resource for replacement, forcing it to be destroyed and recreated on the **`apply`** execution.

You could also use **`terraform destroy -target <virtual machine>`** and destroy only the virtual machine and then run a terraform apply again.

This command replaces **`terraform taint`**, which was the command that would be used up until 0.15.x. You may still see **`terraform taint`** on the actual exam until it is updated.

29. What do the declarations, such as **`name`**, **`cidr`**, and **`azs`**, in the following Terraform code represent and what purpose do they serve?

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.21.0"
  name = var.vpc_name
  cidr = var.vpc_cidr
  azs            = var.vpc_azs
  private_subnets = var.vpc_private_subnets
  public_subnets  = var.vpc_public_subnets
  enable_nat_gateway = var.vpc_enable_nat_gateway
  tags = var.vpc_tags
}
```

a. these are where the variable declarations are created so Terraform is aware of these variables within the calling module
b. these are the outputs that the child module will return
c. these are variables that are passed into the child module likely used for resource creation
d. the value of these variables will be obtained from values created within the child module

Explanation

To pass values to a Terraform module when calling the module in your code, you use input variables. Input variables are a way to pass values into a Terraform module from the calling code. They allow the module to be flexible and reusable, as the same module can be used with different input values in different contexts.

In this example, the values for the **name**, **cidr**, and **azs** inputs are passed to the module as values of variables. The variables are defined in the calling code in the calling module using the **variable** block.

To pass the values to the module, you can specify them in a number of ways, such as:

Using command-line flags when running Terraform

Storing the values in a Terraform .tfvars file and passing that file to Terraform when running it

Using environment variables

30. What tasks can the terraform state command and its subcommands be utilized for in Terraform?

a. create a new state file
b. refresh the existing state
c. modify the current state, such as removing items
d. There is no such command

Explanation

The **terraform state** command and its subcommands can be used for various tasks related to the Terraform state. Some of the tasks that can be performed using the **terraform state** command are:

Inspecting the Terraform state: The **terraform state show** subcommand can be used to display the current state of a Terraform configuration. This can be useful for verifying the current state of resources managed by Terraform.

Updating the Terraform state: The **terraform state mv** and **terraform state rm** subcommands can be used to move and remove resources from the Terraform state, respectively.

Pulling and pushing the Terraform state: The **terraform state pull** and **terraform state push** subcommands can be used to retrieve and upload the Terraform state from and to a remote backend, respectively. This is useful when multiple users or systems are working with the same Terraform configuration.

Importing resources into Terraform: The **terraform state import** subcommand can be used to import existing resources into the Terraform state. This allows Terraform to manage resources that were created outside of Terraform.

By using the **terraform state** command and its subcommands, users can manage and manipulate the Terraform state in various ways, helping to ensure that their Terraform configurations are in the desired state.

31. Sara has her entire application automated using Terraform, but she needs to start automating more infrastructure components, such as creating a new subnet, DNS record, and load balancer. Sara wants to create these new resources within modules to easily reuse the code later. However, Sara is having problems getting the subnet_id from the *subnet* module to pass to the *load balancer* module.

modules/subnet.tf:

```
resource "aws_subnet" "bryan" {
  vpc_id     = aws_vpc.krausen.id
  cidr_block = "10.0.1.0/24"
   tags = {
    Name = "Krausen Subnet"
  }
}
```

What could fix this problem?

a. publish the module to a Terraform registry first
b. references to resources that are created within a module cannot be used within other modules
c. move the *subnet* and *load balancer* resource into the main configuration file so they can easily be referenced
d. add an output block that references the *subnet* module and retrieves the value using module.subnet.subnet_id in the load balancer module

Explanation

Modules also have output values, which are defined within the module with the output keyword. You can access them by referring to module.<MODULE NAME>.<OUTPUT NAME>. Like input variables, module outputs are listed under the outputs tab in the Terraform registry.

Module outputs are usually either passed to other parts of your configuration, or defined as outputs in your root module.

32. Which of the following variable declarations is going to result in an error?

a.

```
variable "example" {
  description = "This is a test"
  type        = map
  default     = {"one" = 1, "two" = 2, "Three" = "3"}
}
```

b.

```
variable "example" {
  description = "This is a variable description"
  type        = list(string)
  default     = {}
```

}

c.

```
variable "example" {
  type = object({})
}
```

d.

```
variable "example" {}
```

Explanation

This variable declaration for a type `list` is incorrect because a list expects square brackets `[ ]` and ***not*** curly braces. All of the others are correct variable declarations.

From the official HashiCorp documentation found here:

Lists/tuples are represented by a pair of square brackets containing a comma-separated sequence of values, like `["a", 15, true]`.

33. What feature of Terraform Cloud allows you to publish and maintain a set of custom modules which can be used within your organization?

a. custom VCS integration
b. Terraform registry
c. remote runs
d. private module registry

Explanation

You can use modules from a private registry, like the one provided by Terraform Cloud. Private registry modules have source strings of the form `<HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>`. This is the same format as the public registry, but with an added hostname prefix.

34. A user has created three workspaces using the command line - `prod`, `dev`, and `test`. The user wants to create a fourth workspace named `stage`. Which command will the user execute to accomplish this task?

a. `terraform workspace -create stage`
b. `terraform workspace new stage`
c. `terraform workspace -new stage`
d. `terraform workspace create stage`

Explanation

The user can execute the following command to create a fourth workspace named **stage**:

$ terraform workspace new stage

This command will create a new Terraform workspace named **stage**. The user can then switch to the new workspace using the **terraform workspace select** command and use it to manage resources in the new environment.

35. When writing Terraform code, how many spaces between each nesting level does HashiCorp recommends that you use?

a. 5
b. 1
c. 2

d. 4

Explanation

HashiCorp, the creator of Terraform, recommends using **two spaces** for indentation when writing Terraform code. This is a convention that helps to improve readability and consistency across Terraform configurations.

For example, when defining a resource in Terraform, you would use two spaces to indent each level of the resource definition, as in the following example:

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

  tags = {
    Name = "example-instance"
  }
}
```

While this is the recommended convention, ***it is not a strict requirement*** and Terraform will still function correctly even if you use a different number of spaces or a different type of indentation. However, using two spaces for indentation is a widely adopted convention in the Terraform community and is recommended by HashiCorp to improve the readability and maintainability of your Terraform configurations.

36. What Terraform command will launch the Interactive console to evaluate and experiment with expressions?

a. terraform console

b. terraform cli

c. terraform cmdline

d. terraform get

Explanation

The **terraform console** command in Terraform is a command-line interface (CLI) tool that allows you to interactively evaluate expressions in Terraform. The **terraform console** command opens a REPL (Read-Eval-Print Loop) environment, where you can type Terraform expressions and see the results immediately. This can be useful for testing and debugging Terraform configurations and understanding how Terraform evaluates expressions.

Here are a few examples of how the **terraform console** command can be helpful:

Testing expressions: You can use the **terraform console** command to test Terraform expressions and see the results immediately. For example, you can test arithmetic operations, string concatenation, and other Terraform expressions to ensure that they are evaluated correctly.

Debugging configurations: If you have a complex Terraform configuration and you're not sure why it's not working as expected, you can use the **terraform console** command to debug the configuration by testing expressions and variables to see their values.

Understanding Terraform behavior: If you're new to Terraform and you want to understand how it evaluates expressions and variables, you can use the **terraform console** command to explore Terraform's behavior and see how different expressions are evaluated.

To use the **terraform console** command, simply type **terraform console** in your terminal, and Terraform will open a REPL environment. You can then type Terraform expressions and see the results immediately. You can exit the REPL environment by typing **exit** or **quit**.
It's worth noting that the **terraform console** command operates in the context of a specific Terraform configuration, so you should run the command from within the directory that contains your Terraform configuration files.

37. Which of the following statements represents the most accurate statement about the Terraform language:

a. Terraform is an immutable, declarative, Infrastructure as Code provisioning language based on Hashicorp Configuration Language, or optionally JSON.

b. Terraform is a mutable, declarative, Infrastructure as Code configuration management language based on Hashicorp Configuration Language, or optionally JSON.

c. Terraform is an immutable, imperative, Infrastructure as Code configuration management language based on Hashicorp Configuration Language, or optionally JSON.

d. Terraform is a mutable, imperative, Infrastructure as Code provisioning language based on Hashicorp Configuration Language, or optionally YAML.

Explanation
Terraform is written in HashiCorp Configuration Language (HCL). However, Terraform also supports a syntax that is JSON compatible (https://developer.hashicorp.com/terraform/language/syntax/json).
Terraform is primarily designed on ***immutable*** infrastructure principles https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure
Terraform is also a ***declarative language***, where you simply declare the desired state, and Terraform ensures that real-world resources match the desired state as written. An imperative approach is different, where the tool uses a step-by-step workflow to create the desired state.
Terraform is not a configuration management tool

38. Harry has deployed resources on Azure for his organization using Terraform. However, he has discovered that his co-workers Ron and Ginny have manually created a few resources using the Azure console. Since it's company policy to manage production workloads using IaC, how can Harry start managing these resources in Terraform without negatively impacting the availability of the deployed resources?

a. use terraform import to import the existing resources under Terraform management

b. rewrite the Terraform configuration file to deploy new resources, run a terraform apply, and migrate users to the newly deployed resources. Manually delete the other resources created by Ron and Ginny.

c. resources created outside of Terraform cannot be managed by Terraform

d. run a terraform get to retrieve other resources that are not under Terraform management

Explanation
To manage the resources created manually by Ron and Ginny in Terraform without negatively impacting the availability of the deployed resources, Harry can follow the steps below:

Import the existing resources: Harry can use the **terraform import** command to import the existing resources into Terraform. The **terraform import** command allows you to import existing infrastructure into Terraform, creating a Terraform state file for the resources.
Modify the Terraform configuration: After importing the resources, Harry can modify the Terraform configuration to reflect the desired state of the resources. This will allow him to manage the resources using Terraform just like any other Terraform-managed resource
Test the changes: Before applying the changes, Harry can use the **terraform plan** command to preview the changes that will be made to the resources. This will allow him to verify that the changes will not negatively impact the availability of the resources.
Apply the changes: If the changes are correct, Harry can use the **terraform apply** command to apply the changes to the resources.
By following these steps, Harry can start managing the manually created resources in Terraform while ensuring that the availability of the deployed resources is not impacted.
The terraform import command is used to import existing resources into Terraform. This allows you to take resources that you've created by some other means and bring them under Terraform management.
*Note that terraform import **DOES NOT** generate configuration, it only modifies state. You'll still need to write a configuration block for the resource for which it will be mapped using the terraform import command.*

39. You are developing a new Terraform module to demonstrate features of the most popular HashiCorp products. You need to spin up an AWS instance for each tool, so you create the resource block as shown below using the for_each meta-argument.

```
resource "aws_instance" "bryan-demo" {
 # ...
 for_each = {
  "terraform": "infrastructure",
  "vault":    "security",
  "consul":   "connectivity",
  "nomad":    "scheduler",
 }
}
```

After the deployment, you view the state using the terraform state list command. What resource address would be displayed for the instance related to vault?

a. **aws_instance.bryan-demo["vault"]**
b. aws_instance.bryan-demo[1]
c. aws_instance.bryan-demo["2"]
d. aws_instance.bryan-demo.vault

Explanation
In Terraform, when you use the **for_each** argument in a resource block, Terraform generates multiple instances of that resource, each with a unique address. The address of each instance is determined by the keys of the **for_each** map, and it is used to identify and manage each instance of the resource.

For example, consider the following resource block in the question:

```
resource "aws_instance" "bryan-demo" {
 # ...
 for_each = {
  "terraform": "infrastructure",
  "vault":    "security",
  "consul":   "connectivity",
  "nomad":    "scheduler",
 }
}
```

In this example, Terraform will create four instances of the **aws_instance** resource, one for each key in the **for_each** map. The addresses of these instances will be **aws_instance.bryan-demo["terraform"]** , **aws_instance.bryan-demo["vault"]**,**aws_instance.bryan-demo["consul"]**, and **aws_instance.bryan-demo["nomad"]**.

When you reference the properties of these instances in your Terraform code, you can use the address and property reference syntax to access the properties of each instance. For example, you can access the ID of the first instance using **aws_instance.bryan-demo["vault"].id**.

Using the **for_each** argument in a resource block is a powerful way to manage multiple instances of a resource, and it provides a convenient way to reuse the same resource configuration for multiple instances with different properties.

40. Freddy and his co-worker Jason are deploying resources in GCP using Terraform for their team. After resources have been deployed, they must destroy the cloud-based resources to save on costs. However, two other team members, Michael and Chucky, are using a Cloud SQL instance for testing and request to keep it running. How can Freddy and Jason destroy all other resources without negatively impacting the database?

a. take a snapshot of the database, run a terraform destroy, and then recreate the database in the GCP console by restoring the snapshot

b. run a terraform state rm command to remove the Cloud SQL instance from Terraform management before running the terraform destroy command

c. delete the entire state file using the terraform state rm command and manually delete the other resources in GCP

d. run a terraform destroy, modify the configuration file to include only the Cloud SQL resource, and then run a terraform apply

Explanation

To destroy all Terraform-managed resources except for a single resource, you can use the **terraform state** command to remove the state for the resources you want to preserve. This effectively tells Terraform that those resources no longer exist, so it will not attempt to destroy them when you run **terraform destroy**.

Here's an example of how you could do this:

Identify the resource you want to preserve. In this example, let's assume you want to preserve a resource named **prod_db**.

Run **terraform state list** to see a list of all Terraform-managed resources.

Run **terraform state rm** for each resource you want to keep, like the **prod_db**. For example:
terraform state rm google_sql_database_instance.prod_db
terraform state rm aws_instance.another_instance
Run **terraform destroy** to destroy all remaining resources. Terraform will not attempt to destroy the resource you preserved in step 3 because Terraform no longer manages it.
Note that this approach can be dangerous and is not recommended if you have multiple Terraform workspaces or if you are using a remote state backend, as it can cause inconsistencies in your state file. In those cases, it is usually better to use a separate Terraform workspace for the resources you want to preserve or to utilize Terraform's built-in resource-targeting functionality to destroy only specific resources.
All other options would be too time-consuming or will cause an outage to the database.

41. After executing a terraform plan, you notice that a resource has a tilde (~) next to it. What does this mean?

a. Terraform can't determine how to proceed due to a problem with the state file
b. the resource will be created
c. the resource will be destroyed and recreated
d. the resource will be updated in place

Explanation
The prefix -/+ means that Terraform will destroy and recreate the resource, rather than updating it in-place. Some attributes and resources can be updated in-place and are shown with the ~ prefix.

```
    ~ root_block_device {
        ~ delete_on_termination = true -> (known after apply)
        ~ iops                  = 100 -> (known after apply)
        ~ volume_id             = "vol-0079e485d9e28a8e5" -> (known after apply)
        ~ volume_size           = 8 -> (known after apply)
        ~ volume_type           = "gp2" -> (known after apply)
      }
  }
```

42. Rick is writing a new Terraform configuration file and wishes to use modules in order to easily consume Terraform code that has already been written. Which of the modules shown below will be created first?

a.

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

b.

```
provider "aws" {
```

```
  region = "us-west-2"
}
```

c.

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.21.0"
   name = var.vpc_name
  cidr = var.vpc_cidr
  azs             = var.vpc_azs
  private_subnets = var.vpc_private_subnets
  public_subnets  = var.vpc_public_subnets
  enable_nat_gateway = var.vpc_enable_nat_gateway
  tags = var.vpc_tags
}
```

d.

```
module "ec2_instances" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "2.12.0"
   name          = "my-ec2-cluster"
  instance_count = 2
   ami                    = "ami-0c5204531f799e0c6"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [module.vpc.default_security_group_id]
  subnet_id              = module.vpc.public_subnets[0]
   tags = {
    Terraform   = "true"
    Environment = "dev"
  }
}
```

e.

module "ec2_instances"
module "vpc"

Explanation

The `VPC` module will be executed first since the `ec2_instances` module has dependencies on the VPC module. Both `vpc_security_group_ids` and `subnet_id` require outputs from the VPC module.

43. Which of the following represents a feature of Terraform Cloud that is **NOT** free to customers?

a. private module registry
b. workspace management
c. VCS integration
d. team management and governance

Explanation

Features

- Practitioner workflow
- Unified workflow management
- Visibility & optimization
- Policy & security
- Governance, risk & compliance
- Integrations & API
- Reliability & scale

44. By default, where does Terraform OSS/CLI store its state file?

a. shared directory
b. remotely using Terraform Cloud
c. Amazon S3 bucket
d. current working directory

Explanation

By default, the state file is stored in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.

45. Which of the following is considered a Terraform plugin?

a. Terraform logic
b. Terraform tooling
c. Terraform language
d. Terraform provider

Explanation

In Terraform, a plugin is a binary executable that implements a specific provider. **A provider is a plugin that allows Terraform to manage a specific cloud provider or service.**

When Terraform runs, it loads the plugins required to manage the resources specified in the configuration files. Each provider has its own plugin, and Terraform loads the plugins for the providers specified in the configuration.

The plugin is responsible for interacting with the cloud provider's API, translating Terraform configurations into API calls, and managing the state of the resources that Terraform manages. Plugins are stored in the Terraform plugin cache, a directory on the local machine that contains the binary executables for each plugin. When Terraform runs, it looks for plugins in the cache and automatically downloads any missing plugins from the Terraform Registry or a specified source.

Terraform plugins are written in Go and follow a specific plugin protocol, which defines the interactions between Terraform and the plugin. The plugin protocol allows Terraform to communicate with the plugin and provides a standard way for plugins to manage resources across different providers.

46. What is the downside to using Terraform to interact with sensitive data, such as reading secrets from Vault?

a. secrets are persisted to the state file
b. Terraform requires a unique auth method to work with Vault
c. Terraform and Vault must be running on the same version
d. Terraform and Vault must be running on the same physical host

Explanation

Interacting with Vault from Terraform causes any secrets that you read and write to be persisted in both Terraform's state file *and* in any generated plan files. For any Terraform module that reads or writes Vault secrets, these files should be treated as sensitive and protected accordingly.

47. What happens when a `terraform plan` is executed?

a. creates an execution plan and determines what changes are required to achieve the desired state in the configuration files.

b. reconciles the state Terraform knows about with the real-world infrastructure

c. applies the changes required in the target infrastructure in order to reach the desired configuration

d. the backend is initialized and the working directory is prepped

Explanation

The `terraform plan` command is used to create an execution plan. Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.

After a plan has been run, it can be executed by running a `terraform apply`

48. Which configuration block type is used to declare settings and behaviors specific to Terraform?

a. `resource` block

b. `data` block

c. `terraform` block

d. `provider` block

Explanation

In Terraform, the **`terraform`** block is used to configure Terraform settings and to specify a required version constraint for the Terraform CLI.

The **`terraform`** block is optional and is typically placed at the top of a Terraform configuration file. Here is an example of a **`terraform`** block:

```
terraform {
  required_version = ">= 0.12.0, < 0.13.0"
  backend "s3" {
    bucket = "my-terraform-state"
    key    = "terraform.tfstate"
    region = "us-west-2"
  }
}
```

In this example, the **`terraform`** block specifies that the Terraform configuration requires a version of at least 0.12.0 but less than 0.13.0. The block also contains a **`backend`** block, which configures the backend where the Terraform state is stored. In this case, the backend is an S3 bucket in the **`us-west-2`** region.

The **`terraform`** block can also be used to configure other settings such as the maximum number of concurrent operations (**`max_parallelism`**), the number of retries for failed operations (**`retryable_errors`**), and the default input values for variables (**`default`**).

By including a **terraform** block in the Terraform configuration, you can ensure that the correct version of Terraform is used and that the configuration is validated against the correct syntax and semantics for that version. This helps to ensure that the configuration will run correctly and consistently across different environments.

49. In the terraform block, which configuration would be used to identify the specific version of a provider required?

a. required-provider
b. required-version
c. required_versions
d. required_providers

Explanation

To identify a specific version of a provider in Terraform, you can use the **required_providers** configuration block. This block allows you to specify the provider's name and the version range you want to use by using Terraform's version constraints syntax. Here's an example of how to use the **required_providers** block to specify a specific version of the AWS provider:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "3.57.0"
    }
  }
}
```

In this example, we're specifying that we require version 3.57.0 of the AWS provider, which is hosted at the **hashicorp/aws** source. Note that the version constraint syntax allows you to specify a range of versions using operators such as **>=** and **<=**.

When you run **terraform init** with this configuration, Terraform will download and install the specified version of the AWS provider, and will use it for all subsequent Terraform commands for that module. If the specified version is not available, Terraform will return an error and fail to initialize the configuration.

50. In Terraform Cloud, a workspace can be mapped to how many VCS repos?

a. 2
b. 1
c. 5
d. 3

Explanation

A workspace can only be configured to a single VCS repo, however, multiple workspaces can use the same repo, if needed.

51. Which Terraform command will force a resource to be destroyed and recreated even if there are no configuration changes that would require it?

a. terraform destroy

b. terraform apply -refresh-only

c. terraform apply -replace=<address>

d. terraform fmt

Explanation

The **terraform apply -replace=<address>** command manually marks a Terraform-managed resource to be replaced, forcing it to be destroyed and recreated during the **apply**. Even if there are no configuration changes that would require a change or deletion of this resource, this command will instruct Terraform to replace it. This can come in handy if a resource has become degraded or damaged outside of Terraform.

This command replaces **terraform taint**, and it's possible you may still see **terraform taint** on the exam. Be prepared to know both of these commands.

52. What are the core Terraform workflow steps to use infrastructure as code?

a.

1) Code
2) Validate
3) Apply

b.

1) Plan
2) Apply
3) Pray

c.

1) Write
2) Plan
3) Apply

d.

1) Plan
2) Apply
3) Destroy

Explanation

The core Terraform workflow has three steps:

- Write - Author infrastructure as code.
- Plan - Preview changes before applying.
- Apply - Provision reproducible infrastructure.

This guide walks through how each of these three steps plays out in the context of working as an individual practitioner, how they evolve when a team is collaborating on infrastructure, and how Terraform Cloud enables this workflow to run smoothly for entire organizations.

53. What is the best and easiest way for Terraform to read and write secrets from HashiCorp Vault?

a. Vault provider

b. integration with a tool like Jenkins

c. API access using the AppRole auth method

d. CLI access from the same machine running Terraform

Explanation

The Vault provider allows Terraform to read from, write to, and configure

54. Which of the following best describes a Terraform provider?

a. describes an infrastructure object, such as a virtual network, compute instance, or other components
b. a container for multiple resources that are used together
c. a plugin that Terraform uses to translate the API interactions with the service or provider
d. serves as a parameter for a Terraform module that allows a module to be customized

Explanation

In Terraform, a **provider** is a plugin that enables Terraform to interact with a specific cloud or service provider, such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP). Providers are responsible for understanding the APIs and resources the target infrastructure platform provides and for translating Terraform configuration code into API calls that can create, read, update, and delete resources.

Each provider typically consists of resource types, data sources, and other settings that define the provider's capabilities within Terraform. These resources and data sources correspond to the resources that can be managed within the target infrastructure, such as virtual machines, storage accounts, or network interfaces.

To use a provider in Terraform, you must first configure it in your Terraform code by specifying the provider's name and any required configuration settings, such as access keys, secret keys, or region. Once the provider is configured, you can then use its resources and data sources in your Terraform code to define the infrastructure you want to manage.

Terraform has a large and growing ecosystem of third-party providers that support a wide range of infrastructure platforms and services, as well as an official set of core providers maintained by HashiCorp, the company behind Terraform. The availability and quality of providers is a crucial factor in the usefulness of Terraform as a tool for managing infrastructure as code.

55. What Terraform command can be used to inspect the current state file?

Example:

```
# aws_instance.example:
resource "aws_instance" "example" {
    ami                          = "ami-2757f631"
    arn                          = "arn:aws:ec2:us-east-1:130490850807:instance/i-0
    associate_public_ip_address  = true
    availability_zone            = "us-east-1c"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = false
    ebs_optimized                = false
    get_password_data            = false
    id                           = "i-0bbf06244e44211d1"
    instance_state               = "running"
    instance_type                = "t2.micro"
```

a. terraform inspect
b. terraform read
c. terraform state
d. terraform show

Explanation

The `terraform show` command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.
Machine-readable output can be generated by adding the `-json` command-line flag.
Note: When using the `-json` command-line flag, any sensitive values in Terraform state will be displayed in plain text.

56. What Terraform feature is shown in the example below?

```
resource "aws_security_group" "example" {
 name = "sg-app-web-01"
  dynamic "ingress" {
   for_each = var.service_ports
   content {
    from_port = ingress.value
    to_port   = ingress.value
    protocol  = "tcp"
   }
  }
}
```

a. data source
b. dynamic block
c. conditional expression
d. local values

Explanation
You can dynamically construct repeatable nested blocks like `ingress` using a special `dynamic` block type, which is supported inside `resource`, `data`, `provider`, and `provisioner` blocks.

57. Given the Terraform configuration below, which order will the resources be created?

```
resource "aws_instance" "web_server" {
   ami = "i-abdce12345"
   instance_type = "t2.micro"
}
resource "aws_eip" "web_server_ip" {
   vpc = true
   instance = aws_instance.web_server.id
}
```

a. resources will be created in parallel
b. no resources will be created
c. `aws_eip` will be created first
d. `aws_instance` will be created second
e. `aws_instance` will be created first
f. `aws_eip` will be created second

Explanation

The aws_instance will be created first, and then aws_eip will be created second due to the aws_eip's resource dependency of the aws_instance id

58. You have been given requirements to create a security group for a new application. Since your organization standardizes on Terraform, you want to add this new security group with the fewest number of lines of code. What feature could you use to iterate over a list of required tcp ports to add to the new security group?

a. dynamic block
b. dynamic backend
c. splat expression
d. terraform import

Explanation

A dynamic block acts much like a for expression, but produces nested blocks instead of a complex typed value. It iterates over a given complex value and generates a nested block for each element of that complex value.

You can find more information on dynamic blocks using this link.

59. Terry is using a module to deploy some EC2 instances on AWS for a new project. He is viewing the code that is calling the module for deployment, which is shown below. *Where is the value of the security group originating?*

```
module "ec2_instances" {
 source  = "terraform-aws-modules/ec2-instance/aws"
 version = "4.3.0"
 name          = "my-ec2-cluster"
 instance_count = 2
 ami                 = "ami-0c5204531f799e0c6"
 instance_type         = "t2.micro"
 vpc_security_group_ids = [module.vpc.default_security_group_id]
 subnet_id             = module.vpc.public_subnets[0]

 tags = {
  Terraform   = "true"
  Environment = "dev"
 }
```

a. from a variable likely declared in a .tfvars file being passed to another module
b. the output of another module
c. the Terraform public module registry
d. an environment variable being using during a terraform apply

Explanation

The required vpc_security_group_ids and subnet_id arguments reference resources created by the vpc module. The Terraform Registry module page contains the full list of arguments for the ec2-instance module.

60. Stephen is writing brand new code and needs to ensure it is syntactically valid and internally consistent. Stephen doesn't want to wait for Terraform to access any remote

services while making sure his code is valid. *What command can he use to accomplish this?*

a. terraform fmt
b. terraform validate
c. terraform show
d. terraform apply -refresh-only

Explanation

The terraform validate command validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider APIs, etc.

Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state. It is thus primarily useful for general verification of reusable modules, including correctness of attribute names and value types.

61. When multiple engineers start deploying infrastructure using the same state file, what is a feature of remote state storage that is critical to ensure the state does not become corrupt?

a. object storage
b. encryption
c. state locking
d. (Correct)
e. workspaces

Explanation

If supported by your backend, Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state.

State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, Terraform will not continue. You can disable state locking for most commands with the -lock flag but it is not recommended.

62. In the example below, where is the value of the DNS record's IP address originating from?

```
resource "aws_route53_record" "www" {
  zone_id = aws_route53_zone.primary.zone_id
  name    = "www.helloworld.com"
  type    = "A"
  ttl     = "300"
  records = [module.web_server.instance_ip_addr]
}
```

a. by querying the AWS EC2 API to retrieve the IP address
b. the regular expression named module.web_server
c. the output of a module named web_server
d. value of the web_server parameter from the variables.tf file

Explanation

In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>. For example, if a child module named web_server declared an output named instance_ip_addr, you could access that value as module.web_server.instance_ip_addr.

63. What does the command terraform fmt do?
    a. deletes the existing configuration file
    b. formats the state file in order to ensure the latest state of resources can be obtained
    c. rewrite Terraform configuration files to a canonical format and style
    d. updates the font of the configuration file to the official font supported by HashiCorp

Explanation

The **terraform fmt** command is a formatting tool in Terraform that helps to automatically format Terraform configuration files to follow a consistent style and make them more readable. Running **terraform fmt** will parse the configuration files in the current directory and recursively in subdirectories and rewrite them using a standard formatting style, including indentation, spacing, and line breaks. It will modify the original files in place, so it's vital to ensure that the files are backed up or committed to a version control system before running this command.
By running **terraform fmt**, it helps to ensure that the Terraform configuration files are consistent across the project and easy to read, especially when working with large and complex infrastructure codebases. Consistent code style makes it easier for multiple people to collaborate on a project and makes it easier to understand the configuration files when returning to the project after an extended period.
It's a best practice to run **terraform fmt** before committing any changes to the configuration files, to ensure that all changes have the same formatting style and are easy to read.

64. After running into issues with Terraform, you need to enable verbose logging to assist with troubleshooting the error. Which of the following values provides the **MOST** verbose logging?
    a. DEBUG
    b. ERROR
    c. TRACE
    d. INFO
    e. WARN

Explanation

In Terraform, you can enable verbose logging by using the **-debug** command line option or setting the **TF_LOG** environment variable to **DEBUG**. This will provide additional log messages to help with troubleshooting errors.
However, if you need even more detailed logging, you can set the **TF_LOG** environment variable to **TRACE**. This will provide the most verbose logging, including every step taken by Terraform during plan, apply, and destroy operations, as well as additional debugging information.
Here's an example of how to set the **TF_LOG** environment variable to **TRACE** on a Unix-based system:

```
$ export TF_LOG=TRACE
```

Note that enabling verbose logging can result in a large amount of output, so it should only be used when necessary for troubleshooting purposes. Once you have resolved the issue, you can turn off verbose logging by removing the **TF_LOG** environment variable or set it to a lower level, such as **DEBUG** or **INFO**.

65. In order to make a Terraform configuration file dynamic and/or reusable, static values should be converted to use what?

a. regular expressions
b. input variables
c. module
d. output value

Explanation

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

66. Why is it a good idea to declare the required version of a provider in a Terraform configuration file?

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "3.57.0"
    }
  }
}
```

a. providers are released on a separate schedule from Terraform itself; therefore, a newer version could introduce breaking changes
b. to match the version number of your application being deployed via Terraform
c. to remove older versions of the provider
d. to ensure that the provider version matches the version of Terraform you are using

Explanation

Declaring the required version of a provider in a Terraform configuration file is a good idea for several reasons:

Reproducibility: By specifying the exact version of a provider, you can ensure that anyone who runs your Terraform configuration will use the same version of the provider as you. This makes your infrastructure configuration more reproducible and helps avoid issues that can arise when different versions of a provider are used.

Predictability: When you specify a specific provider version, you can be confident that your infrastructure configuration will behave predictably, regardless of changes to the provider in future versions. ***This can help you avoid surprises and reduce the risk of unintended consequences.***

Compatibility: Different versions of a provider may have different APIs, resources, or behaviors, which can cause issues if you switch to a new version without realizing the differences. By specifying the required version of a provider in your Terraform configuration, you can ensure

that your configuration remains compatible with the specific version of the provider you have tested and validated.

Version locking: When you specify the required version of a provider in your Terraform configuration, you effectively lock the version of the provider to that version unless you explicitly change it. This can help prevent issues that may arise when using a different, potentially incompatible version of the provider.

In summary, specifying the required version of a provider in your Terraform configuration file helps ensure that your infrastructure configuration is more predictable, reproducible, compatible, and reduces the risk of unintended consequences or issues caused by version differences.

67. Which of the following best describes the default local backend?

a. The local backend is the directory where resources deployed by Terraform have direct access to in order to update their current state
b. The local backend is how Terraform connects to public cloud services, such as AWS, Azure, or GCP.
c. The local backend stores state on the local filesystem, locks the state using system APIs, and performs operations locally.
d. The local backend is where Terraform Enterprise stores logs to be processed by an log collector

Explanation

Information on the default local backend can be found at this link.

Example:

```
terraform {
  backend "local" {
    path = "relative/path/to/terraform.tfstate"
  }
}
```

68. Your organization has moved to AWS and has manually deployed infrastructure using the console. Recently, a decision has been made to standardize on Terraform for all deployments moving forward. What can you do to ensure that the existing resources are managed by Terraform moving forward without causing interruption to existing resources?

a. using `terraform import`, import the existing infrastructure to bring the resources under Terraform management
b. resources that are manually deployed in the AWS console cannot be imported by Terraform
c. delete the existing resources and recreate them using new a Terraform configuration so Terraform can manage them moving forward
d. submit a ticket to AWS and ask them to export the state of all existing resources and use `terraform import` to import them into the state file

Explanation

To ensure that existing resources in AWS are managed by Terraform moving forward without causing interruption to existing resources, there are a few steps that you can follow:

Create a new Terraform configuration that represents the existing resources in AWS. This can be done manually by examining the resources in the AWS console and recreating them in Terraform code, or automatically by using a tool like Terraforming or CloudMapper.
Import the existing resources into Terraform using the **terraform import** command. This command allows you to associate the existing resources in AWS with the new Terraform configuration. You will need to specify the resource type, name, and ID for each resource you want to import.
Use Terraform to manage all future changes to the infrastructure. With the existing resources now managed by Terraform, you can make changes to them through Terraform code and use the normal Terraform workflow of plan, apply, and destroy to manage the infrastructure going forward.
It's important to note that importing existing resources into Terraform can be a complex and error-prone process, especially for large and complex infrastructures. It's recommended to test the import process thoroughly in a development or staging environment before attempting to import production resources. Additionally, be sure to carefully review the Terraform code before running **terraform apply** to avoid accidentally modifying or deleting existing resources.

69. When using modules to deploy infrastructure, how would you export a value from one module to import into another module?

For example, a module dynamically deploys an application instance or virtual machine, and you need the IP address in another module to configure a related DNS record in order to reach the newly deployed application.

a. configure the pertinent provider's configuration with a list of possible IP addresses to use
b. preconfigure the IP address as a parameter in the DNS module
c. configure an output value in the application module in order to use that value for the DNS module
d. export the value using terraform export and input the value using terraform input

Explanation
Output values are like the return values of a Terraform module and have several uses such as a child module using those outputs to expose a subset of its resource attributes to a parent module.

70. You want to use terraform import to start managing infrastructure that was not originally provisioned through infrastructure as code. Before you can import the resource's current state, what must you do to prepare to manage these resources using Terraform?

a. shut down or stop using the resources being imported so no changes are inadvertently missed
b. update the Terraform configuration file to include the new resources that match the resources you want to import
c. modify the Terraform state file to add the new resources so Terraform will have a record of the resources to be managed
d. run terraform apply -refresh-only to ensure that the state file has the latest information for existing resources.

Explanation

The current implementation of Terraform import can only import resources into the state. It does not generate a configuration. Because of this, and before running `terraform import`, it is necessary to manually write a `resource` configuration block for the resource to which the imported object will be mapped.

First, add the resources to the configuration file:

```
resource "aws_instance" "example" {
  # ...instance configuration...
}
```

Then run the following command:

```
$ terraform import aws_instance.example i-abcd1234
```

71. What happens when a `terraform apply` command is executed?

a. applies the changes required in the target infrastructure in order to reach the desired configuration

b. the backend is initialized and the working directory is prepped

c. reconciles the state Terraform knows about with the real-world infrastructure

d. creates the execution plan for the deployment of resources

Explanation

The `terraform apply` command is used to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a terraform plan execution plan.

72. Which of the following allows Terraform users to apply policy as code to enforce standardized configurations for resources being deployed via infrastructure as code?

a. workspaces

b. functions

c. sentinel

d. module registry

Explanation

Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products. It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.

73. A provider alias is used for what purpose in a Terraform configuration file?

a. alias isn't used with providers, they are used with provisioners

b. to signify what resources should be deployed to a certain cloud provider

c. using the same provider with different configurations for different resources

d. (Correct)

e. to use as shorthand for resources to be deployed with the referenced provider

Explanation

To create multiple configurations for a given provider, include multiple `provider` blocks with the same provider name. For each additional non-default configuration, use the `alias` meta-argument to provide an extra name segment.

74. AutoPlants, Inc is a new startup that uses AI and robotics to grow sustainable and organic vegetables for California farmer's markets. The organization can quickly burst

into the public cloud during the busy season using Terraform to provision additional resources to process AI computations and images. Since its compute stack is proprietary and critical to the organization, it needs a solution to create and publish Terraform modules that only its engineers and architects can use. Which feature can provide this functionality?

a. Terraform Cloud Workspaces
b. Public Module Registry
c. Private Module Registry
d. HashiCorp Sentinel

Explanation

One feature that can provide this functionality is Terraform's **Private Module Registry**. This feature allows organizations to create and manage private modules that authorized users within the organization can only access. With Private Module Registry, AutoPlants, Inc can create and publish Terraform modules that are only available to its engineers and architects. This ensures that their proprietary compute stack remains secure while also streamlining the process of provisioning additional resources during the busy season.

75. Given a Terraform config that includes the following code, how would you reference the last instance that will be created?

```
resource "aws_instance" "database" {
 # ...
 for_each = {
  "vault": "value1",
  "terraform": "value2",
  "consul": "value3",
  "nomad": "value4",
 }
}
```

a. aws_instance.nomad
b. aws_instance.database[4]
c. aws_instance.database["nomad"]
d. aws_instance.database[2]

Explanation

The following specifications apply to index values on modules and resources with multiple instances:

`[N]` where `N` is a `0`-based numerical index into a resource with multiple instances specified by the `count` meta-argument. Omitting an index when addressing a resource where `count > 1` means that the address references all instances.

`["INDEX"]` where `INDEX` is an alphanumerical key index into a resource with multiple instances specified by the `for_each` meta-argument.

76. Terraform Cloud Agents are a feature that allows Terraform Cloud to communicate with private infrastructure, such as VMware hosts running on-premises. Which version of Terraform Cloud supports this feature?

a. Terraform Team and Governance

b. Terraform Cloud Free

c. Terraform Cloud for Business

Explanation

This newer feature is only available on Terraform Cloud for Business. Terraform Cloud for Business supports Terraform Cloud Agents, allowing communication with private infrastructure like VMware hosts running on-premises.

77. Given the following snippet of code, what does `servers = 4` reference?

```
module "servers" {
  source = "./modules/aws-servers"
   servers = 4
}
```

a. the value of an input variable

b. the output variable of the module

c. the number of times the module will be executed

d. `servers` is not a valid configuration for a module

Explanation

When calling a child module, values can be passed to the module to be used within the module itself.

78. What function does the `terraform init -upgrade` command perform?

a. update all previously installed plugins to the newest version that complies with the configuration's version constraints

b. upgrades the backend to the latest supported version

c. upgrades all of the referenced modules and providers to the latest version of Terraform

d. upgrades the Terraform configuration file(s) to use the referenced Terraform version

Explanation

The `-upgrade` will upgrade all previously-selected plugins to the newest version that complies with the configuration's version constraints. This will cause Terraform to ignore any selections recorded in the dependency lock file, and to take the newest available version matching the configured version constraints.

79. Based on the following code, which of the resources will be created first?

```
resource "aws_instance" "data_processing" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t2.micro"
   depends_on = [aws_s3_bucket.customer_data]
}
module "example_sqs_queue" {
  source  = "terraform-aws-modules/sqs/aws"
  version = "2.1.0"
  depends_on = [aws_s3_bucket.customer_data, aws_instance.data_processing]
}
resource "aws_s3_bucket" "customer_data" {
  acl    = "private"
}
```

```
resource "aws_eip" "ip" {
   vpc = true
   instance = aws_instance.data_processing.id
}
```

a. aws_eip.ip
b. aws_s3_bucket.customer_data
c. aws_instance.data_processing
d. example_sqs_queue

Explanation

In this example, the only resource that does not have an implicit or an explicit dependency is the aws_s3_bucket.customer_data. Every other resource defined in this configuration has a dependency on another resource.

80. Larissa is an experienced IT professional and is working to learn Terraform to manage the F5 load balancers that front-end customer-facing applications. Larissa writes great code, but her formatting seldom meets the Terraform canonical formatting and style recommended by HashiCorp. What built-in tool or command can Larissa use to easily format her code to meet the recommendations for formatting Terraform code?

a. terraform env
b. terraform fmt
c. terraform validate
d. terraform plan

Explanation

The terraform fmt command is used to rewrite Terraform configuration files to a canonical format and style. This command applies a subset of the Terraform language style conventions, along with other minor adjustments for readability.

81. Given the following snippet of code, what will the value of the "Name" tag equal after a terraform apply?

```
variable "name" {
  description = "The username assigned to the infrastructure"
  default = "data_processing"
}
variable "team" {
  description = "The team responsible for the infrastructure"
  default = "IS Team"
}
locals {
  name  = (var.name != "" ? var.name : random_id.id.hex)
  owner = var.team
  common_tags = {
    Owner = local.owner
    Name  = local.name
  }
}
```

a. a random hex value
b. data_processing
c. IS Team
d. an empty string

Explanation

The syntax of a conditional expression first names the condition. In this example, if `var.name` is not (`!=`) empty, assign the `var.name` value; else, assign the new `random_id` resource as the name value. Since `var.name` equals **data_processing**, then the value of `Name` will equal `data_processing`.

82. When a `terraform apply` is executed, where is the AWS provider retrieving credentials to create cloud resources in the code snippet below?

```
provider "aws" {
  region     = us-east-1
  access_key = data.vault_aws_access_credentials.creds.access_key
  secret_key = data.vault_aws_access_credentials.creds.secret_key
}
```

a. from a script that is executing commands against Vault
b. from the .tfvars file called `vault`
c. From a data source that is retrieving credentials from HashiCorp Vault. Vault is dynamically generating the credentials on Terraform's behalf.
d. From a variable called `vault_aws_access_credentials`

Explanation

In this case, Terraform is using a data source to gather credentials from Vault. The data block would look something like this:

```
data "vault_aws_access_credentials" "creds" {
  backend = vault_aws_secret_backend.aws.path
  role    = vault_aws_secret_backend_role.role.name
}
```

83. Teddy is using Terraform to deploy infrastructure using modules. Where is the module below stored?

```
module "monitoring_tools" {
 source = "./modules/monitoring_tools"
 cluster_hostname = module.k8s_cluster.hostname
}
```

a. a private module registry in Terraform Cloud (free)
b. locally on the instance running Terraform
c. on the Terraform public module registry
d. in a public GitLab repository

Explanation

A local path must begin with either `./` or `../` to indicate that a local path is intended, to distinguish from a module registry address.

84. Michael has deployed many resources in AWS using Terraform and can easily update or destroy resources when required by the application team. A new employee, Dwight, is

working with the application team and deployed a new EC2 instance through the AWS console. When Michael finds out, he decided he wants to manage the new EC2 instance using Terraform moving forward. He opens his terminal and types:

`$ terraform import aws_instance.web_app_42 i-b54a26b28b8acv7233`

However, Terraform returns the following error: `Error: resource address "aws_instance.web_app_42" does not exist in the configuration.`

What does Michael need to do first in order to manage the new Amazon EC2 instance with Terraform?

a. import the configuration of the EC2 instance called `web_app_42` from AWS first
b. create a configuration for the new resource in the Terraform configuration file, such as:
resource "aws_instance" "web_app_42" {
 # (resource arguments)
}
c. Terraform cannot manage resources that were provisioned manually
d. configure the appropriate tags on the Amazon EC2 resource so Terraform knows that it should manage the resource moving forward

Explanation

The `terraform import` command is used to import existing resources into Terraform. However, Terraform will not create a configuration for the imported resource. The Terraform operator must create/add a configuration for the resource that will be imported first. Once the configuration is added to the configuration file, the `terraform import` command can be executed to manage the resource using Terraform.

85. Terraform has detailed logs that can be enabled using the `TF_LOG` environment variable. Which of the following log levels is the most verbose, meaning it will log the most specific logs?

a. `ERROR`
b. `DEBUG`
c. `INFO`
d. `TRACE`

Explanation

You can set `TF_LOG` to one of the log levels `TRACE`, `DEBUG`, `INFO`, `WARN` or `ERROR` to change the verbosity of the logs. `TRACE` is the most verbose and it is the default if `TF_LOG` is set to something other than a log level name.

86. Ralphie has executed a `terraform apply` using a complex Terraform configuration file. However, a few resources failed to deploy due to incorrect variables. After the error is discovered, what happens to the resources that were successfully provisioned?

a. resources successfully deployed are marked for replacement
b. the resources that were successfully provisioned will remain as deployed
c. Terraform rolls back the configuration due to the error, therefore the resources are automatically destroyed
d. Terraform deletes the resources on the next run

Explanation

During a terraform apply, any resources that are successfully provisioned are maintained as deployed. On the other hand, resources that failed during the provisioning process, such as a provisioned, will be tainted to be recreated during the next run.

87. What feature of Terraform provides an abstraction above the upstream API and is responsible for understanding API interactions and exposing resources?

a. Terraform provisioner
b. Terraform configuration file
c. Terraform backend
d. Terraform provider

Explanation

Terraform relies on plugins called "providers" to interact with remote systems.
Terraform configurations must declare which providers they require so that Terraform can install and use them. Additionally, some providers require configuration (like endpoint URLs or cloud regions) before they can be used.

88. Which of the following is not a benefit of Terraform state?

a. determines the dependency order for deployed resources
b. increases performance by reducing the requirement to query multiple resources at once
c. provides a one-to-one mapping of the configuration to real-world resources
d. reduces the amount of outbound traffic by requiring that state is stored locally

Explanation

"Increases performance by reducing the requirement to query multiple resources at once" is not a benefit of Terraform state. While Terraform state does provide several advantages, such as determining dependency order for deployed resources, providing a one-to-one mapping of configuration to real-world resources, and reducing outbound traffic by storing state locally, it doesn't directly increase performance by reducing the need to query multiple resources simultaneously.

89. Pam just finished up a new Terraform configuration file and has successfully deployed the configuration on Azure using Terraform open-source. After confirming the configuration on Azure, Pam changes to a new workspace and then heads to lunch. When she arrives back at her desk, Pam decides to destroy the resources to save on cost. When Pam executes a terraform destroy, the output indicates there are no resources to delete. *Why can't Pam delete the newly created resources in Azure?*

$ terraform destroy
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
Terraform will perform the following actions:
Plan: 0 to add, 0 to change, 0 to destroy.

a. an Azure administrator manually deleted the resources
b. the Terraform state was deleted when she created the new workspace
c. there is no Terraform state in the current workspace she is working in
d. Terraform reached the maximum timeout while Pam was away from lunch, therefore the resources were automatically destroyed

Explanation

Workspaces isolate their state, so if Pam runs a `terraform destroy`, Terraform will not see any existing state for this configuration. Pam may use the command `terraform workspace select <name>` to choose the original workspace where the Azure resources were provisioned in order to properly destroy them in Azure.

90. Based on the code provided, how many subnets will be created in the AWS account?

variables.tf

```
variable "private_subnet_names" {
 type    = list(string)
 default = ["private_subnet_a", "private_subnet_b", "private_subnet_c"]
}
variable "vpc_cidr" {
 type    = string
 default = "10.0.0.0/16"
}
variable "public_subnet_names" {
 type    = list(string)
 default = ["public_subnet_1", "public_subnet_2"]
}
```

main.tf

```
resource "aws_subnet" "private_subnet" {
 count             = length(var.private_subnet_names)
 vpc_id            = aws_vpc.vpc.id
 cidr_block        = cidrsubnet(var.vpc_cidr, 8, count.index)
 availability_zone = data.aws_availability_zones.available.names[count.index]

 tags = {
  Name      = var.private_subnet_names[count.index]
  Terraform = "true"
 }
}
```

a. 3
b. 0
c. 1
d. 2

Explanation

The code above will create **three** subnets. The value of `count` is determined by the number of strings included in the `private_subnet_names` variable.

91. You have created a new workspace for a project and added all of your Terraform configuration files in the new directory. Before you execute a `terraform plan`, you want to validate the configuration using the `terraform validate` command. However, Terraform returns the error:

```
$ terraform validate
Error: Could not load plugin
```

What would cause this error when trying to validate the configuration?

a. the configuration is invalid
b. **the directory was not initialized**
c. the directory does not contain valid Terraform configuration files
d. the credentials for the provider are invalid

Explanation

`terraform validate` requires an initialized working directory with any referenced plugins and modules installed. If you don't initiate the directory, you will get an error stating you need to run a `terraform init`

92. Which type of configuration block assigns a name to an expression that can be used multiple times within a module without having to repeat it?

a. local
b. resources
c. provider
d. backend

Explanation

A local value assigns a name to an expression, so you can use it multiple times within a module without repeating it.

93. Variables and their default values are typically declared in a `main.tf` or `variables.tf` file. What type of file can be used to set explicit values for the current working directory that will override the default variable values?

a. .tfstate file
b. .tfvars file
c. .txt file
d. .sh file

Explanation

To set lots of variables, it is more convenient to specify their values in a *variable definitions file* (with a filename ending in either `.tfvars` or `.tfvars.json`)

94. Margaret is calling a child module to deploy infrastructure for her organization. Just as a good architect does (and suggested by HashiCorp), she specifies the module version she wants to use even though there are newer versions available. During a `terrafom init`, Terraform downloads v0.0.5 just as expected. What would happen if Margaret removed the version parameter in the module block and ran a `terraform init` again?

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"
  servers = 3
}
```

a. Terraform would use the existing module already downloaded
b. Terraform would skip the module
c. Terraform would download the latest version of the module
d. Terraform would return an error, as the `version` parameter is required

Explanation

When using modules installed from a module registry, HashiCorp recommends explicitly constraining the acceptable version numbers to avoid unexpected or unwanted changes.
The version argument accepts a version constraint string. Terraform will use the newest installed version of the module that meets the constraint; if no acceptable versions are installed, it will download the newest version that meets the constraint.
A version number that meets every applicable constraint is considered acceptable.
Terraform consults version constraints to determine whether it has acceptable versions of itself, any required provider plugins, and any required modules. For plugins and modules, it will use the newest installed version that meets the applicable constraints.
To test this, I ran a terraform init with the code as shown in the file:

```
$ terraform init
Initializing modules...
Downloading hashicorp/consul/aws 0.0.5 for consul...
- consul in .terraform\modules\consul
- consul.consul_clients in .terraform\modules\consul\modules\consul-cluster
- consul.consul_clients.iam_policies in .terraform\modules\consul\modules\consul-iam-policies
- consul.consul_clients.security_group_rules in .terraform\modules\consul\modules\consul-security-group-rules
- consul.consul_servers in .terraform\modules\consul\modules\consul-cluster
- consul.consul_servers.iam_policies in .terraform\modules\consul\modules\consul-iam-policies
- consul.consul_servers.security_group_rules in .terraform\modules\consul\modules\consul-security-group-rules
```

Then I removed the constraint from the configuration file and ran a terraform init again:

```
$ terraform init
Initializing modules...
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/template from the dependency lock file
```

Terraform did not download a newer version of the module. It reused the existing one.

95. Rigby is implementing Terraform and was given a configuration that includes the snippet below. Where is this particular module stored?

```
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

a. locally in the hashicorp/consul/aws directory
b. public Terraform registry
c. locally but a directory back from the current directory
d. a private module registry supported by your organization

Explanation

Modules on the public Terraform Registry can be referenced using a registry source address of the form <NAMESPACE>/<NAME>/<PROVIDER>, with each module's information page on the registry site including the exact address to use.

96. When running a terraform plan, how can you save the plan so it can be applied at a later time?

a. use the -out flag
b. you cannot save a plan
c. use the -save flag
d. use the -file flag

Explanation

The optional -out flag can be used to save the generated plan to a file for later execution with terraform apply, which can be useful when running Terraform in automation.

97. You are working with a cloud provider to deploy resources using Terraform. You've added the following data block to your configuration. When Terraform the data block is executed, what value is the data source returning?

```
data "aws_ami" "amzlinux2" {
 most_recent = true
 owners     = ["amazon"]
 filter {
  name   = "name"
  values = ["amzn2-ami-hvm-*-x86_64-ebs"]
 }
}
resource "aws_instance" "vault" {
 ami                    = data.aws_ami.amzlinux2.id
 instance_type          = "t3.micro"
 key_name               = "vault-key"
 vpc_security_group_ids     = var.sg
 subnet_id              = var.subnet
 associate_public_ip_address = "true"
 user_data              = file("vault.sh")
 tags = {
  Name = "vault"
 }
}
```

a. the latest AMI you have previously used for an Amazon Linux 2 image
b. all possible data of a specific Amazon Machine Image(AMI) from AWS
c. a custom AMI for Amazon Linux 2
d. the IP address of an EC2 instance running in AWS

Explanation

When you add a data block to your configuration, Terraform will retrieve all of the available data for that particular resource. It is then up to you to reference a specific attribute that can be exported from that data source. For example, if you include a data block for

the aws_ami resource, Terraform will get a ton of attributes about that AMI that you can use elsewhere in your code.
Within the block body (between { and }) are query constraints defined by the data source. Most arguments in this section depend on the data source, and indeed in this example most_recent, owners and tags are all arguments defined specifically for the aws_ami data source.

98. Which of the following best describes a "data source"?
a. maintains a list of strings to store the values of declared outputs in Terraform
b. a file that contains the current working version of Terraform
c. enables Terraform to fetch data for use elsewhere in the Terraform configuration
d. provides required data for declared variables used within the Terraform configuration

Explanation
Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration. Use of data sources allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another separate Terraform configuration.

99. Using the Terraform code below, where will the resource be provisioned?

```
provider "aws" {
  region = "us-east-1"
}
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
provider "aws" {
  alias  = "eu"
  region = "eu-west-2"
}
resource "aws_instance" "vault" {
  ami                    = data.aws_ami.amzlinux2.id
  instance_type            = "t3.micro"
  key_name                = "krausen_key"
  vpc_security_group_ids     = var.vault_sg
  subnet_id               = var.vault_subnet
  user_data               = file("vault.sh")
  tags = {
    Name = "vault"
  }
}
```

a. us-west-1
b. us-west-2
c. us-east-1

Explanation

The resource above will be created in the default region of us-east-1, since the resource does signify an alternative provider configuration. If the resource needs to be created in one of the other declared regions, it should have looked like this, where "aws" signifies the provider name and "west" signifies the alias name as such <PROVIDER NAME>.<ALIAS>:

```
resource "aws_instance" "vault" {
 provider                = aws.west
 ami                     = data.aws_ami.amzlinux2.id
 instance_type           = "t3.micro"
 key_name                = "krausen_key"
 vpc_security_group_ids  = var.vault_sg
 subnet_id               = var.vault_subnet
 user_data               = file("vault.sh")
 tags = {
  Name = "vault"
 }
}
```

100. Philip works at a payment processing company and manages the organization's VMware environment. He recently provisioned a new cluster for a production environment. To ensure everything is working as expected, Philip has been using Terraform and the VMware vSphere client to create and destroy new virtual machines. Currently, there are three virtual machines running on the new cluster, so Philip runs `terraform destroy` to remove the remaining virtual machines from the cluster. However, Terraform only removes two of the virtual machines, leaving one virtual machine still running. Why would Terraform only remove two of the three virtual machines?

a. the vSphere provider credentials are invalid, and therefore Terraform cannot reach the third virtual machine
b. Terraform can only destroy a maximum of 2 resources per `terraform destroy` execution
c. the remaining virtual machine was not created by Terraform, therefore Terraform is not aware of the virtual machine and cannot destroy it
d. the virtual machine was marked with vSphere tags to prevent it from being destroyed

Explanation

The `terraform destroy` command terminates resources defined in your Terraform configuration. This command is the reverse of `terraform apply` in that it terminates all the resources specified by the configuration. It does *not* destroy resources running elsewhere that are not described in the current configuration.

101. You have your production environment deployed with Terraform. A developer requested that you update a load balancer configuration to better work with the application. After modifying your Terraform, how can you perform a dry run to ensure no unexpected changes will be made?

a. run `terraform console` and validate the result of any expressions
b. run `terraform state list` to view the existing resources and ensure they won't conflict with the new changes

c. run `terraform plan -auto-approve` to push the changes

d. run `terraform plan` and inspect the proposed changes

Explanation

The ultimate goal of a `terraform plan` is to compare the configuration file against the current state and propose any changes needed to apply the desired configuration.

102. There are multiple ways to provide sensitive values when using Terraform. However, sensitive information provided in your configuration can be written to the state file, which is not desirable. Which method below will not result in sensitive information being written to the state file?

a. using a declared variable

b. none of the above

c. retrieving the credentials from a data source, such as HashiCorp Vault

d. using a tfvars file

Explanation

When using sensitive values in your Terraform configuration, all of the configurations mentioned above will result in the sensitive value being written to the state file. Terraform stores the state as plain text, including variable values, even if you have flagged them as `sensitive`. Terraform needs to store these values in your state so that it can tell if you have changed them since the last time you applied your configuration.

Terraform runs will receive the full text of sensitive variables and might print the value in logs and state files if the configuration pipes the value through to an output or a resource parameter. Additionally, Sentinel mocks downloaded from runs will contain the sensitive values of Terraform (but not environment) variables. Take care when writing your configurations to avoid unnecessary credential disclosure. (Environment variables can end up in log files if TF_LOG is set to TRACE.)

103. Based on the Terraform code below, what block type is used to define the VPC?

```
vpc_id = aws_vpc.main.id
...
```

a. locals block

b. resource block

c. data block

d. provider block

Explanation

Based on the Terraform code provided in the question, the VPC is defined in a resource block, meaning that there is a VPC resource being defined, such as:

```
resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}
```

If it were locals, the resource would be referred to as `local.aws_vpc`

If it were in a data block, it would be referred to as `data.aws_vpc.i.main.id`

104. You have a Terraform configuration file defining resources to deploy on VMware, yet there is no related state file. You have successfully run `terraform init` already. What happens when you run a *`terraform apply`*?

a. Since there is no state file associated with this configuration file, the defined resources will be not created on the VMware infrastructure.
b. Terraform will scan the VMware infrastructure, create a new state file, and deploy the new resources as defined in the configuration file.
c. All existing infrastructure on VMware will be deleted, and the resources defined in the configuration file will be created.
d. Terraform will produce an error since there is no state file

Explanation

If there is no state file associated with a Terraform configuration file, a terraform apply will create the resources defined in the configuration file. This is a normal workflow during the first `terraform apply` that is executed against a configuration file. This, of course, assumes that the directory has been initialized using a `terraform init`

105. Which feature of Terraform Cloud can be used to enforce fine-grained policies to enforce standardization and cost controls before resources are provisioned with Terraform?

a. remote runs
b. workspaces
c. sentinel
d. module registry

Explanation

Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products. It enables fine-grained, logic-based policy decisions and can be extended to use information from external sources.

Please Note: HashiCorp announced at HashiConf Global '22 that Open Policy Agent (OPA) is now supported in Terraform Cloud. However, this new feature will likely take a while to appear in the actual Terraform exam. I'm already working with HashiCorp to address any conflicts that might appear on the real exam.

106. You are an Infrastructure Engineer at Strategies, Inc, which is a new organization that provides marketing services to startups. All of your infrastructure is provisioned and managed by Terraform. Despite your pleas to not make changes outside of Terraform, sometimes the other engineers log into the cloud platform and make minor changes to resolve problems. What Terraform command can you use to reconcile the state with the real-world infrastructure in order to detect any drift from the last-known state?

a. `terraform apply -refresh-only`
b. `terraform state show`
c. `terraform validate`
d. `terraform graph`

Explanation

The `terraform apply -refresh-only` command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. This can be used to detect any drift from the last-known state, and to update the state file.

107. Which of the following Terraform features is NOT available in the open-source version?

a. resource graph
b. public cloud providers
c. public module registry
d. sentinel policies

Explanation

All of the options are available to open-source users except for Sentinel, which is only available in Terraform Enterprise and Terraform Cloud paid tiers.

108. Which of the following commands can be used to detect configuration drift?

a. `terraform get`
b. `terraform fmt`
c. `terraform apply -refresh-only`
d. `terraform init`

Explanation

If the state has drifted from the last time Terraform ran, `terraform plan -refresh-only` or `terraform apply -refresh-only` allows drift to be detected.

109. You have a Terraform configuration file with no defined resources. However, there is a related state file for resources that were created on AWS. What happens when you run a *`terraform apply`*?

a. Terraform will produce an error since there are no resources defined
b. Terraform will destroy all of the resources
c. Terraform will scan the AWS infrastructure and create a new configuration file based on the state file.
d. Terraform will not perform any operations.

Explanation

In this case, since there is a state file with resources, Terraform will match the desired state of no resources since the configuration file doesn't include any resources. Therefore, all resources defined in the state file will be destroyed.

110. Jeff is a DevOps Engineer for a large company and is currently managing the infrastructure for many different applications using Terraform. Recently, Jeff received a request to remove a specific VMware virtual machine from Terraform as the application team no longer needs it. Jeff opens his terminal and issues the command:

```
$ terraform state rm vsphere_virtual_machine.app1
Removed vsphere_virtual_machine.app1
Successfully removed 1 resource instance(s).
```

The next time that Jeff runs a `terraform apply`, the resource is not marked to be deleted. In fact, Terraform is stating that it is creating another identical resource.

```
.....
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
  # vsphere_virtual_machine.app1 will be created
```

What would explain this behavior?

a. **Jeff removed the resource from the *state file*, and not the *configuration file*. Therefore, Terraform is no longer aware of the virtual machine and assumes Jeff wants to create a new one since the virtual machine is still in the Terraform configuration file**
b. after running the terraform rm command, Jeff needs to run a Terraform plan first to tell Terraform of the updated configuration. A plan will instruct Terraform that the resource should be deleted upon the next `terraform apply`
c. the resource was manually deleted within the VMware infrastructure and needs to be recreated
d. the state file was not saved before the `terraform apply` was executed, therefore Terraform sees that the resource is still in the state file

Explanation

Because Jeff manually deleted the resource from the state file, Terraform was no longer aware of the virtual machine. When Jeff ran a `terraform apply`, it refreshed the state file and discovered that the configuration file declared a virtual machine but it was not in state, therefore Terraform needed to create a virtual machine so the provisioned infrastructure matched the desired configuration, which is the Terraform configuration file.

Hopefully, this isn't a tricky one but I thought it was good to test on, especially since `terraform state` commands are listed in Objective 4 of the exam. In this case, Jeff should NOT have removed the resource from the state file, but rather remove it from the configuration file and run a `terraform plan/apply`. In this scenario, Terraform would recognize that the virtual machine was no longer needed and would have destroyed it.

111. *Scenario*: You are deploying a new application and want to deploy it to multiple AWS regions within the same configuration file. Which of the following features will allow you to configure this?

a. a provider with multiple versions defined
b. one provider block that defines multiple regions
c. multiple provider blocks using an alias
d. using the default provider along with a single defined provider

Explanation

You can optionally define multiple configurations for the same provider, and select which one to use on a per-resource or per-module basis. The primary reason for this is to support multiple regions for a cloud platform; other examples include targeting multiple Docker hosts, multiple Consul hosts, etc.

112. You need Terraform to destroy and recreate a single database server that was deployed with a bunch of other resources. You don't want to modify the Terraform code. What command can be used to accomplish this task?

a. `terraform state show aws_instance.database`
b. `terraform apply -replace="aws_instance.database"`
c. `terraform state rm aws_instance.database`
d. `terraform plan -destroy="aws_instance.database"`

Explanation

When working with resources, there may be times where a particular resource didn't deploy correctly, although Terraform thinks it did. An example of this might be a script that runs on a

virtual machine in the background. The virtual came up fine, so Terraform believes it was successful, but the script didn't perform the tasks you needed it to, so you need Terraform to destroy and recreate the one resource. In this case, you can use `terraform apply -replace="<resource_id>"` to have Terraform replace this one resource on the next `terraform apply`.

This command was formally `terraform taint`, and you may or may not see `terraform taint` still on the exam. The `taint` command was deprecated in Terraform 0.15.2 and replaced with the `terraform apply -replace` command. Note that the resource is NOT immediately replaced when you run a `terraform taint`. It will only happen on the next `terraform plan/apply`.

113. Thomas has recently developed a new Terraform configuration in a new working directory and is very cost-conscious. After running a `terraform init`, how can Thomas perform a dry run to ensure Terraform will create the right resources without deploying real-world resources?

a. run `terraform output`
b. run `terraform show`
c. run `terraform plan -out=thomas`
d. run `terraform apply -refresh-only`

Explanation

To perform a dry-run of your Terraform configuration, you should run a `terraform plan`. The entire purpose of running a `terraform plan` is to validate the change(s) to your infrastructure before you apply the change. In this case, Thomas could see what resources would be created without actually deploying the resources and costing him money.

Running a `terraform apply -refresh-only` would not give you the desired output. This command is used to update the state file for existing resources deployed with Terraform. This would be useful if somebody made a change outside of Terraform, and you needed to reflect that change in the state file.

Using `terraform output` wouldn't work because this command is used to view any outputs defined in your Terraform code. You can also use `terraform output <output name>` to view more detailed information about a particular output.

Running a `terraform show` would not give you what you're looking for here. This command displays the output from a state or plan file.

114. Which of the following best describes the primary use of Infrastructure as Code (IaC)?

a. ensures that the operations team understands how to develop code
b. the ability to programmatically deploy and configure resources
c. combining disparate technologies and various tasks into a single workflow
d. ensuring that applications remain in the desired state configuration

Explanation

The primary use case for IaC is to deploy and configure resources in almost any environment in a single, unified way that also abstracts the user from the APIs.

Combing disparate technologies and tasks is really the job of a pipeline, such as GitLab CI/CD, Jenkins, or Azure DevOps. While Terraform CAN be used with multiple technologies within a

single configuration file, it's not really the ideal job for Terraform. Terraform isn't an orchestrator like the aforementioned tools would be used for.
The goal of Terraform is NOT to ensure that operations folks know how to develop code, although I'd say that is somewhat of an end result in most organizations. While they are developing Java or Golang applications, operations folks tend to use Terraform as an opportunity to learn more developer-centric workflows, like using Git or learning how to develop code in a repeatable fashion.
When deploying Terraform, it's often a one-time thing, and Terraform doesn't actively monitor applications for changes. That's the job of a configuration management tool, such as Ansible, Chef, Puppet, or SaltStack.

115. Your organization has standardized on Microsoft Azure to run its applications on PaaS, SaaS, and IaaS offerings. The deployment quickly standardized on Azure ARM to provision these resources quickly and efficiently. Which of the following is true about how the team currently deploys its infrastructure?

a. the team would not be able to develop reusable code in order to reduce the time it takes to develop code for new applications
b. the adoption of another public cloud provider will prove to be more challenging since all of its codebase is based on ARM
c. the team would not be able to quickly adapt and integrate baseline security measures in its code to help standardize application deployments
d. the team would not be able to use its existing skill set to develop code for newly announced services

Explanation
While each of the main public cloud providers has its own version of Infrastructure as Code (ARM, AWS CloudFormation, etc.), adopting the native solution can be limiting as the organization matures its cloud capabilities and offerings. If the organization only learns the native solution, what happens if they decide to use a different public cloud provider or they acquire a company that uses a different one? While the developer mindset would still apply, the skillset used to deploy ARM doesn't necessarily apply one-to-one for writing AWS CloudFormation, for example. Each of these solutions handles development and deployment differently, and the engineers have to learn a second solution.
By using Terraform, engineers and developers can focus their time on ***learning a single solution*** applicable to all the public cloud providers and other SaaS, PaaS, and IaaS offerings available on the market.
The wrong answers are all benefits of using any infrastructure as code, such as standardization, reusability, and familiarity with using a cloud provider solution.

116. You work for a retail organization that has multiple peak seasons throughout the year. During those peak seasons, your applications need to be scaled up quickly to handle the increased demand. However, the deployment of application servers is manual and new servers are only deployed when problems are reported by users. How can you reduce the effort required to deploy new resources, increase the speed of deployments, and reduce or eliminate the negative experiences of your customers?

a. Develop a manual runbook that the developers and operations teams can follow during the peak seasons to reconfigure the compute resources used to serve the primary application.
b. Rather than wait on user reports, implement a ticketing system that alerts the operations team of poor performance or customer errors. Automatically assign the tickets to the on-call team to quickly resolve.
c. Deploy new IaC code that automatically shuts down existing application servers and scales the resources down during periods of high demand.
d. Develop code that provisions new application servers programmatically. Use monitoring software to trigger a pipeline that deploys additional servers during periods of increased demand.

Explanation

In this case, automation is key. And considering that this is a course/question focused on Infrastructure as Code, developing IaC to trigger automatically based on workloads is the best answer here.

While the others sound like a great idea or an improvement in the troubleshooting process, they wouldn't resolve the errors with their customers.

117. Given the code snippet below, how would you refer to the value of `ip` for the `dev` environment if you are using a `for_each` argument?

```
variable "env" {
 type = map(any)
 default = {
  prod = {
   ip = "10.0.150.0/24"
   az = "us-east-1a"
  }
  dev  = {
   ip = "10.0.250.0/24"
   az = "us-east-1e"
  }
 }
}
```

a. `each.value.ip`
b. `each.dev.ip`
c. `var.env.dev.ip`
d. `var.env["dev.ip"]`

Explanation

Sort of testing two different things here - a complex map variable plus the for_each argument. A `for_each` argument will iterate over a map or set of strings and create a similar instance/resource for each item in the map or set. In our case, the map is the input variable and the "each" would be the higher-level map, so prod and dev. Underneath each value, there are two arguments, both `az` and `ip` that you can choose from.

The input variable that is shown in this example is essentially a map of maps.

None of the wrong answers are valid ways to reference the values provided by the input variable.

118. You have recently cloned a repo containing Terraform that you want to test in your environment. Once you customize the configuration, you run a `terraform apply` but it immediately fails. Why would the apply fail?

a. Terraform needs to obtain authentication credentials using the `terraform login` command
b. Terraform needs to initialize the directory and download the required plugins
c. you need to run a `terraform plan` before you can apply the configuration
d. you can't run Terraform code that was cloned from another users code repository

Explanation

When you're learning the basics of Terraform, one of the critical requirements of executing any Terraform code is running `terraform init` to download all of the required plugins needed for the resources that will be deployed/managed. If you don't run a `terraform init` when running code in a new directory, a `terraform apply/plan` will immediately fail since it needs to download the plugins required to run.

While the traditional Terraform workflow is `init` --> `plan` --> `apply`, running a `terraform plan` is not required to execute a `terraform plan`. It's recommended to make changes to real environments, but when I'm testing or building labs for my other courses, I rarely run a terraform plan and go straight to `terraform apply`.

As for running Terraform cloned from another repo, you can absolutely do this. Many people use existing code as a starting point for their own environment. They will clone the repo, customize it however they need and then run it. This is a perfectly acceptable practice and it prevents you from constantly reinventing the wheel.

Regarding authentication, you only need to use the `terraform login` command when you are working with Terraform Cloud. You don't need authentication credentials outside of that use case unless you are deploying resources to some platform, like AWS, Azure, VMware, etc. There are plenty of use cases for Terraform where you don't need authentication credentials at all, like using the TLS or random provider.

119. You have deployed your network architecture in AWS using Terraform. A colleague recently logged in to the AWS console and made a change manually and now you need to be sure your Terraform state reflects the new change. What command should you run to update your Terraform state?

a. `terraform get -update`
b. `terraform plan -out=refresh`
c. `terraform apply -refresh-only`
d. `terraform init -upgrade`

Explanation

Terraform includes the ability to use the command `terraform apply -refresh-only` to refresh the local state based on the changes made outside of the Terraform workflow. Terraform will use the platform's API to query information about each known/managed resource and update any changes it finds.

`terraform apply -refresh-only` replaced the deprecated command `terraform refresh`. However, you still may find `terraform refresh` in the real exam until it gets updated. Keep this in mind when taking the real exam. HashiCorp does update the exams very often, and this could very well come out at the beginning of 2022 when they overhaul the existing exam as noted on the exam page that a new version would be released in early 2022.
`terraform plan -out=refresh` just runs a terraform plan and saves a plan called refresh. I was being a little tricky here but just know that this isn't how to refresh your state file
`terraform init -upgrade` is the command to use if you want Terraform to upgrade your existing downloaded providers
`terraform get -update` is used to download and update modules that are referenced in your Terraform configuration files
Oddly enough, the `-refresh-only` option doesn't currently exist as a valid option.

120. You are using Terraform to manage resources in Azure. Due to unique requirements, you need to specify the version of the Azure provider so it remains the same until newer versions are thoroughly tested. What block would properly configure Terraform to ensure it always installs the same Azure provider version?

a.
```
provider "azurerm" {
  source = "hashicorp/azurerm"
  version = "2.90.0"
}
```
b.
```
required_providers {
  azurerm = {
    source = "hashicorp/azurerm"
    version = "2.90.0"
  }
}
```
c.
```
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "2.90.0"
    }
  }
}
```
d.
```
data "azurerm" {
  source = "hashicorp/azurerm"
  version = 2.90.0
}
```
Explanation

When you need to constrain the provider to a specific version, you would do this under the `terraform` configuration block. Within that block, you would use the `required_providers` block to set certain configurations, including the version of each provider you want to lockdown.
Note that even though you would add the provider constraint under the `terraform` block, you may still indeed have a separate `provider` block to set certain configurations, like credentials, regions, or other settings specific to the provider. Just keep in mind that each distinct block is used for different settings.
Example:
Set the version of the Azure provider:

```
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "2.90.0"
    }
  }
}
```

Configure settings for the Azure provider:

```
provider "azurerm" {
  features {}
}
```

121. You are using modules to deploy various resources in your environment. You want to provide a "friendly name" for the DNS of a new web server so you can simply click the CLI output and access the new website. Which of the following code snippets would satisfy these requirements?

a. ***Add the following code to the web module:***

```
output "website" {
  description = "Outputs the URL of the provisioned website"
  value       = module.web.public_dns
}
```

b. ***Add the following code to the web module:***

```
output "website" {
  description = "Outputs the URL of the provisioned website"
  value       = "https://${aws_instance.web.public_dns}:8080/index.html"
}
```

c. ***Add the following code to the parent module:***

```
output "website" {
  description = "Outputs the URL of the provisioned website"
  value       = "https://${module.web.public_dns}:8080/index.html"
}
```

d. ***Add the following code to the parent module:***

```
output "website" {
  description = "Outputs the URL of the provisioned website"
  value       = aws_instance.web.public_dns
}
```

Explanation

When working with outputs, you need to determine where the value will be coming from and work your way backward from there. For example, if the resource was created inside of a module, then the module will require an output block to export that value. That said, output blocks that are created in a module aren't displayed on the Terraform CLI. Therefore, you need to create an output block in the parent/calling module to output the value while referencing the output in the module. Because of this, the correct answer requires you to create an output in the parent module and reference the output value from the module.

Add the following code to the web module:

```
output "website" {
  description = "Outputs the URL of the provisioned website"
  value       = "https://${aws_instance.web.public_dns}:8080/index.html"
}
```

While this could be a way to get the proper URL, the output of a module wouldn't show up in the CLI output, therefore this is incorrect.

****

Add the following code to the parent module:

```
output "website" {
  description = "Outputs the URL of the provisioned website"
  value       = aws_instance.web.public_dns
}
```

The resource was created inside of the web module, therefore you wouldn't be able to access their attributes directly from the parent module, making this an incorrect answer.

****

Add the following code to the web module:

```
output "website" {
  description = "Outputs the URL of the provisioned website"
  value       = module.web.public_dns
}
```

Even if you could output the value from a module to the CLI, the resource ID for the module is incorrect because it is referring to another module, making this answer incorrect.

122. After running `terraform apply`, you notice some odd behavior and need to investigate. Which of the following environment variables will configure Terraform to write more detailed logs to assist with troubleshooting?

a. `LOG_CONFIG=INFO`
b. `TF_LOG_CONFIG=WARN`
c. `TF_LOGS=ERROR`
d. `TF_LOG=TRACE`

Explanation

Terraform has detailed logs which can be enabled by setting the `TF_LOG` environment variable to any value. This will cause detailed logs to appear on stderr.
You can set `TF_LOG` to one of the log levels `TRACE`, `DEBUG`, `INFO`, `WARN` or `ERROR` to change the verbosity of the logs.
None of the incorrect answers are valid environment variables that you can use to configure Terraform logs.

123. You have declared a variable named ***db_connection_string*** inside of the ***app*** module. However, when you run a `terraform apply`, you get the following error message:

Error: Reference to undeclared input variable
on main.tf line 35:
4: db_path = var.db_connection_string
An input variable with the name "db_connection_string" has not been declared. This variable can be declared with a variable "db_connection_string" {} block.
Why would you receive such an error?

a. input variables are not referenced using the `var` prefix
b. the variable should be referenced as `var.module.app.db_connection_string`
c. an output block was not created in the module, and therefore the variable cannot be referenced
d. since the variable was declared within the module, it cannot be referenced outside of the module

Explanation
When using modules, it's common practice to declare variables outside of the module and pass the value(s) to the child module when it is called by the parent/root module. However, it's perfectly acceptable to declare a variable inside of a module if you needed. Any variables declared *inside* of a module are only directly referencable within that module. You can't directly reference that variable outside of the module. You can, however, create an output in the module to export any values that might be needed outside of the module.
Output block? While an output block would allow you to get information from within the module, creating an output block still wouldn't allow you to reference the variable directly using the var.<name> nomenclature.
Referencing a Variable You can't reference a variable declared inside of a module, therefore the name shown in this incorrect answer wouldn't work. Ideally, you would create an output inside the module and reference the output rather than the variable inside of the module itself.
Using Interpolation in Terraform This incorrect answer is just plain wrong. In fact, you would reference an accessible variable using the var prefix. Interpolation is the ability to reference data or values within your Terraform code using specific formats. For variables, that format is var.<name>.

124. When initializing Terraform, you notice that Terraform's CLI output states it is downloading the modules referenced in your code. Where does Terraform cache these modules?

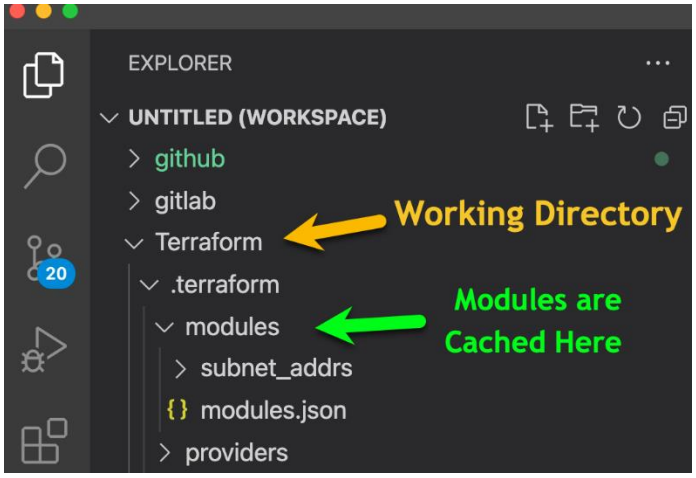
a. in the `/downloads` directory for the user running the terraform init
b. in a `/modules` directory in the current working directory

c. in the `.terraform/modules` subdirectory in the current working directory

d. in the `/tmp` directory on the machine executing Terraform

Explanation

The `.terraform` directory contains the modules and plugins used to provision your infrastructure. These files are specific to a specific instance of Terraform when provisioning infrastructure, not the configuration of the infrastructure defined in `.tf` files.



125. You have an existing resource in your public cloud that was deployed manually, but you want the ability to reference different attributes of that resource throughout your configuration without hardcoding any values. How can you accomplish this?

a. Run a `terraform state list` to find the *resource_id* of the resource you need the attributes from. Reference that *resource_id* throughout your configuration to get the exported attributes as needed.

b. Create a new `resource` block that matches the exact configuration of the existing resource. Run a terraform apply to import the resource. Use the available exported attributes of that resource throughout your configuration as needed.

c. Create a new `variable` block within your configuration. Add the *resource_id* as the default value and reference the variable using `var.<name>` throughout your configuration as needed.

d. Add a `data` block to your configuration to query the existing resource. Use the available exported attributes of that resource type as needed throughout your configuration to get the values you need.

Explanation

Anytime you need to reference a resource that is NOT part of your Terraform configuration, you need to query that resource using a data block - assuming a data source is available for that resource_type. Once you add the data block to your configuration, you will be able to export attributes from that data block using interpolation like any other resource in Terraform. For example, if you had an AWS S3 bucket, you could get information using a data block that looked like this:

```
data "aws_s3_bucket" "data_bucket" {
  bucket = "my-data-lookup-bucket-btk"
```

```
}
```
Once you add the data block, you can refer to exported attributes like this: `data.aws_s3_bucket.data_bucket.arn`

None of the wrong answers would allow you to import or query information so that Terraform can use it through interpolation.

126. By default, Terraform OSS stores its state file in what type of backend?

a. `encrypted` backend
b. `remote` backend
c. `local` backend
d. `shared` backend

Explanation

The default local backend will be used if you don't specify a backend at all in your Terraform configuration. The local backend stores state on the local filesystem, locks that state using system APIs, and performs operations locally.

Note that you can define the backend to be local by using the `backend "local" {}` block.

While `remote` is a valid backend type, `shared` or `encrypted` is not a valid backend type. Local is the default and a remote backend must be explicitly configured in your configuration file.

127. A new variable has been created using the `list` type as shown below. How would you reference `terraform` in your configuration?

```
variable "products" {
  type = list(string)
  default = [
    "vault",
    "consul",
    "terraform",
    "boundary",
    "nomad"
  ]
}
```

a. `var.list.products[2]`
b. `var.products[2]`
c. `var.default.products["terraform"]`
d. `var.products[3]`

Explanation

For a collection type such as a `list`, the provided values are referenced using an index that starts with [0]. In this case, the strings would be represented as such:

[0] = vault
[1] = consul
[2] = terraform
[3] = boundary
[4] = nomad

When referencing a variable that contains a list, you reference it almost identically to a regular variable, except you just add the index on the end of the variable. So for `terraform`, it would simply be `var.products[2]`.
`var.products[3]` = actually equals **boundary** since an index starts with 0
`var.list.products[2]` = this isn't a valid way to reference a variable in Terraform
`var.default.products["terraform"]` = this isn't a valid way to reference a list variable in Terraform. If you remove the default from the answer, it could be a valid option if the variable was of type map, though.

128. You have a module named `prod_subnet` that outputs the `subnet_id` of the subnet created by the module. How would you reference the subnet ID when using it for an input of another module?

a. `subnet = prod_subnet.subnet_id`
b. `subnet = module.prod_subnet.subnet_id`
c. `subnet = module.outputs.prod_subnet.subnet_id`
d. `subnet = prod_subnet.outputs.subnet_id`

Explanation
Using interpolation, you can reference the output of an exported value by using the following syntax: `module.<module name>.<output name>`
Don't forget that before you can reference data/values from a module, the module has to have an output declared that references the desired value(s).
None of the wrong answers are valid interpolation syntax to reference an output that originates from a module.

129. Which of the following is ***not true*** about the `terraform.tfstate` file used by Terraform?

a. it always matches the infrastructure deployed with Terraform
b. the file includes information about each resource managed by Terraform
c. the file can potentially contain sensitive values
d. it is recommended not to modify the file directly

Explanation
The one thing that cannot be guaranteed is that the `terraform.tfstate` file **ALWAYS** matches the deployed infrastructure since changes can easily be made outside of Terraform. For example, if you deploy a bunch of resources in GCP and nobody makes any changes, then yes, the `terraform.tfstate` file does match the current state of those resources. However, if an engineer makes a change in the GCP console or CLI, then the `terraform.tfstate` would **NOT** match the infrastructure deployed until you ran a `terraform apply -refresh-only` command.
This is why the only false statement in this question is: ***it always matches the infrastructure deployed with Terraform.***
The following statements are TRUE about Terraform, which makes them the incorrect choice for this question.
Terraform uses the terraform.tfstate file to store everything it needs to manage the resources it is managing. This includes a ton of information about each resource it provisions and manages.

Because of this, HashiCorp recommends that you DO NOT modify the file directly outside of using the Terraform workflow (`terraform init, plan, apply, destroy`) and `terraform state` CLI commands.
Many times, you'll need to provide sensitive values to deploy and manage resources, or Terraform may retrieve sensitive values at your request (like data blocks). In that case, these values may get saved to the state file, therefore you should limit who can access the state file to protect this sensitive data.

130. After using Terraform locally to deploy cloud resources, you have decided to move your state file to an Amazon S3 remote backend. You configure Terraform with the proper configuration as shown below. What command should be run in order to complete the state migration while copying the existing state to the new backend?

```
terraform {
  backend "s3" {
    bucket = "tf-bucket"
    key = "terraform/krausen/"
    region = "us-east-1"
  }
}
```

a. **`terraform init -migrate-state`**
b. `terraform state show`
c. `terraform apply -refresh-only`
d. `terraform plan -replace`

Explanation
Whenever a configuration's backend changes, you must run `terraform init` again to validate and configure the backend before you can perform any plans, applies, or state operations. Re-running init with an already-initialized backend will update the working directory to use the new backend settings. Either `-reconfigure` or `-migrate-state` must be supplied to update the backend configuration.
When changing backends, Terraform will give you the option to migrate your state to the new backend. This lets you adopt backends without losing any existing state.
None of the wrong answers would allow you to migrate state. They are simply other CLI commands that are commonly used with Terraform.

131. You need to use multiple resources from different providers in Terraform to accomplish a task. Which of the following can be used to configure the settings for each of the providers?

a.

```
data "consul" {
  address = "https://consul.krausen.com:8500"
  namespace = "developer"
  token = "45a3bd52-07c7-47a4-52fd-0745e0cfe967"
}
```

b.

```
data "vault" {
```

```
  address = "https://vault.krausen.com:8200"
  namespace = "developer"
}
```

c.

```
required_providers {
  consul {
    address = "https://consul.krausen.com:8500"
    namespace = "developer"
    token = "45a3bd52-07c7-47a4-52fd-0745e0cfe967"
  }
  vault {
    address = "https://vault.krausen.com:8200"
    namespace = "developer"
  }
}
```

d.

```
terraform {
  providers {
    consul {
      address = "https://consul.krausen.com:8500"
      namespace = "developer"
      token = "45a3bd52-07c7-47a4-52fd-0745e0cfe967"
    }
    vault {
      address = "https://vault.krausen.com:8200"
      namespace = "developer"
    }
  }
}
```

e.

```
provider "consul" {
  address = "https://consul.krausen.com:8500"
  namespace = "developer"
  token = "45a3bd52-07c7-47a4-52fd-0745e0cfe967"
}
```

f.

```
provider "vault" {
  address = "https://vault.krausen.com:8200"
  namespace = "developer"
}
```

Explanation

To configure each provider, you need to define a provider block and provide the configuration within that block. You would need to do this for each provider that you need to configure. For

example, if you needed to customize the `aws`, `gcp`, and `vault` provider, you'd need to create three separate provider blocks, one for each provider.

Additional Clarity: While you can configure parameters inside a `provider` block, the provider block is not needed to use Terraform successfully. The most common configurations within a provider block are **credentials** to access the platform, which should be placed in *environment variables* rather than inside a `provider` block. In my examples above, I am providing custom configurations for my needs. But, if I were using the defaults, I wouldn't need to add a `provider` block for my project to be successfully deployed.

Don't forget that configurations for a provider go inside of a `provider` block, but any provider constraints go inside of the `terraform` --> `required_providers` block.

132. Which of the following is ***true*** about working with modules?

a. modules must be published to the Terraform module registry before they can be used
b. a single module can be called many times in a single configuration file
c. a module can only contain a single resource to be deployed or managed
d. every module that is called from a parent module must output values

Explanation

Ok, so there's a lot to unpack for this question here. First, let's talk about the correct answer. Modules can be called one or more times by a parent module. The configuration file/module that calls a module is often called the `parent`, `root`, or `calling module`. The module that is called is the `child module`, or sometimes just "module". The whole point of using a module is to be able to call it one or many times to create resources without having to rewrite the same code over and over.

Where do modules live? While modules can be published to the Terraform module registry, they don't have to be. They can be simply stored locally on your machine or in a private code repository. Publishing them to the public registry, or using a private module registry, is completely optional.

Module outputs: While modules are often more valuable when they output values, they don't necessarily have to output values. They can be used to simply manage resources. If you do need values from the module, that's when you'd create outputs. Those outputs can be used just for informational purposes or they can be used as inputs for other modules. For example, you might create a subnet in a public cloud in one module and need to output the subnet ID so you can use it as an input on a second module to deploy application workloads.

Resources in Modules: Modules can be used to deploy and manage one or more resources within the module. For example, you might need to deploy multiple resources needs for a specific application or requirements.

133. If supported by your backend, Terraform will lock your state for all operations that could write state. What purpose does this serve?

a. Ensures the state file cannot be moved after the initial `terraform apply`
b. Locks colleagues from making manual changes to the managed infrastructure
c. This prevents others from acquiring the lock and potentially corrupting your state.
d. Prevents others from committing Terraform code that could override your updates.

Explanation

State locking prevents others from acquiring the lock and potentially corrupting your state. If Terraform didn't use state locking, multiple people could try to make changes to your infrastructure and corrupt the state file. At that point, Terraform would no longer understand how to manage the resources deployed since the state file wouldn't be consistent.
State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, Terraform will not continue.
Committing code? State locking wouldn't prevent somebody from committing code to your code repository.
Moving the state file!..Even after you run your first `terraform apply`, you can move the state file by modifying your state block and running a terraform init. State locking does not prevent you from moving state in the future.
Changing Infrastructure Even with state locking, somebody could make manual changes to your infrastructure using the API, CLI, or console if they wanted to. State locking doesn't prevent this.

134. You need to ensure your Terraform is easily readable and follows the HCL canonical format and style. In the current directory, you have a `main.tf` that calls modules stored in a `modules` directory. What command could you run to easily rewrite your Terraform to follow the HCL style in both the current directory and all sub-directories?

a. `terraform fmt -check`
b. `terraform fmt -list=false`
c. `terraform fmt -recursive`
d. `terraform fmt -diff`

Explanation
By default, `fmt` scans the current directory for configuration files and formats them according to the HCP canonical style and format. However, if you need it to also scan and format files in sub-directories, you can use the `-recursive` flag to instruct `terraform fmt` to also process files in subdirectories.
WRONG ANSWERS: None of the wrong answers would instruct `terraform fmt` to scan subdirectories. All of these are, however, other valid flags that you can use with `terraform fmt`.

135. Which common action does not cause Terraform to refresh its state?

a. `terraform plan`
b. `terraform state list`
c. `terraform apply`
d. `terraform destroy`

Explanation
Running a `terraform state list` does not cause Terraform to refresh its state. This command simply reads the state file but it will not modify it.
When running a `plan`, `apply`, or `destroy`, Terraform needs to refresh state to ensure that it has the latest information about the managed resources so it understands what changes should be made when applying the desired state configuration.

136. You are working on updating your infrastructure managed by Terraform. Before lunch, you update your configuration file and run a `terraform plan` to validate the

changes. While you are away, a colleague manually updates a tag on a managed resource directly in the console (UI). What will happen when you run a `terraform apply`?

a. Terraform will recognize the manual change and return an error since the Terraform state no longer matches the real-world infrastructure.
b. Before applying the new configuration, Terraform will refresh the state and recognize the manual change. It will update the resource based on the desired state as configured in the Terraform configuration. The manual change will no longer exist.
c. Terraform will update the manually changed resource back to the original configuration. It will then apply the new changes defined in the updated configuration file.
d. Terraform will destroy the manually-changed resource and recreate it to ensure the infrastructure matches the desired state.

Explanation

There's a lot to this question, but the reasoning is pretty basic. Since a resource was manually changed, it means that Terraform state is no longer accurate. However, before a `terraform plan` or `terraform apply` is executed, Terraform refreshes its state to ensure it knows the status of all its managed resources. During this process, Terraform would recognize the change, update state, and compare that to the new configuration file. Assuming the change defined in the configuration is identical to the manual change, Terraform would simply apply any changes (if any), update the state file, and complete the `terraform apply`.

Change the resource back? Terraform relies on state for everything and any changes are a result of comparing the current state of the resource to the desired configuration (the .tf files). Therefore, Terraform won't revert the resource back to the original configuration because the configuration has been updated for the new change, and that's the desired state.

Return an Error? Since Terraform performs a state refresh before executing a `plan` or `apply`, Terraform will recognize any configuration changes and then apply any changes. This will not result in an error being returned.

Will Terraform destroy my resources? It won't destroy the resource since the resource is still defined in the configuration file. The only way that TF would destroy your resource is if you actually remove that resource from your configuration file(s).

137. Steve is a developer who is deploying resources to AWS using Terraform. Steve needs to gather detailed information about an EC2 instance that he deployed earlier in the day. What command can Steve use to view this detailed information?

a. `terraform state list`
b. `terraform state show aws_instance.frontend`
c. `terraform state rm aws_instance.frontend`
d. `terraform state pull`

Explanation

All resources that are managed by Terraform are referenced in the state file, including detailed information about the resource. Terraform uses the state to map your configuration to the real-world resources that are deployed and managed on the backend platform (AWS, GCP, F5, Infoblox, etc.). You can use the `terraform state` commands to view and manipulate Terraform state if needed.

`terraform state show <resource address>` will show you a lot of details on the resource, including things like the ID, IP address, the state of the resource, and lots more.
`terraform state list` will just show you a list of the resources being managed by Terraform, but it won't show you details on each of those resources
`terraform state rm aws_instance.frontend` would remove the resource from state. This ***would not destroy the resource*** on the public cloud, but it would tell Terraform to stop managing it.
`terraform state pull` will download the state from its current location, upgrade the local copy to the latest state file version that is compatible with locally-installed Terraform, and output the raw format to stdout

138. Your organization uses IaC to provision and manage resources in a public cloud platform. A new employee has developed changes to existing code and wants to push it into production. What best practice should the new employee follow to submit the new code?

a. Execute the code locally on the developer's machine to make the changes directly to the infrastructure.
b. Submit a merge/pull request of the proposed changes. Have a team member validate the changes and approve the request.
c. Make the change directly using the cloud provider's API to ensure the changes are valid. Store the code locally and email a copy of the code to a teammate so they have an extra copy.
d. Submit the change to the change control board and wait for the approval. Commit the code directly to the main repository.

Explanation
Following best practices for code, the new changes should be submitted as a pull/merge request in the existing code repository. A teammate, or the security team, should validate the changes and approve the request, ultimately merging the new changes into the existing codebase
None of these follow best practices for managing code. Please don't do any of these things :)

139. You want to use a Terraform provisioner to execute a script on the remote machine. What block type would use to declare the provisioner?

a. `provider` block
b. `resource` block
c. `data` block
d. `terraform` block

Explanation
First, provisioners should only be used as a last resort. Keep that in mind when writing Terraform.
When using a Terraform provisioned, you would declare the provisioned inside of a resource block to determine when the provisioner should be executed. For example, if you add the provisioner to an `aws_instance` resource, the provisioner will be executed when the `aws_instance` resource is built.
Example:

```
resource "aws_instance" "web" {
  # ...
```

```
  provisioner "local-exec" {
    command = "echo The server's IP address is ${self.private_ip}"
  }
}
```

140. Which of the following is the best description of a `dynamic` block?

a. exports a value exported by a module or configuration
b. requests that Terraform read from a given data source and export the result under the given local name
c. declares a resource of a given type with a given local name
d. produces nested configuration blocks instead of a complex typed value

Explanation

A `dynamic` block acts much like a for expression, but produces nested blocks instead of a complex typed value. It iterates over a given complex value and generates a nested block for each element of that complex value. You can dynamically construct repeatable nested blocks like `setting` using a special `dynamic` block type, which is supported inside `resource`, `data`, `provider`, and `provisioner` blocks.

**** declares a resource of a given type with a given local name =*** this is the definition of a resource block

**** requests that Terraform read from a given data source and export the result under the given local name =*** this is a data block

**** exports a value exported by a module or configuration =*** this is an output block

141. Which of the following code snippets will properly configure a Terraform backend?

a.

```
provider "consul" {
  address = "consul.btk.com"
  scheme  = "https"
  path    = "terraform/"
  }
}
```

b.

```
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "btk"
  workspaces {
    name = "bryan-prod"
  }
 }
}
```

c.

```
data "terraform_remote_state" "btk" {
  backend = "etcd"
```

```
  config = {
    path      = "terraform/terraform.tfstate"
    endpoints = "http://server1:4001"
  }
}
```

d.

```
backend "s3" {
  bucket = "krausen-bucket"
  key    = "terraform/"
  region = "us-west-2"
 }
}
```

Explanation

Backends are configured with a nested `backend` block within the top-level `terraform` block.
There are some important limitations on backend configuration:

* A configuration can only provide one backend block.
* A backend block cannot refer to named values (like input variables, locals, or data source attributes).

None of the wrong answers are correct, since the state is ONLY configured inside of the terraform block.

142. What Terraform command can be used to evaluate and experiment with expressions in your configuration?

a. `terraform plan`
b. `terraform console`
c. `terraform env`
d. `terraform get`

Explanation

The `terraform console` command provides an interactive command-line console for evaluating and experimenting with expressions. This is useful for testing interpolations before using them in configurations, and for interacting with any values currently saved in state.
Example from the Terraform documentation:

```
echo 'split(",", "foo,bar,baz")' | terraform console
```

143. Which statement below is ***true*** regarding using Sentinel in Terraform Enterprise?

a. Sentinel runs before a configuration is applied, therefore potentially reducing cost for public cloud resources
b. Sentinel can extend the functionality of user permissions in Terraform Enterprise
c. Sentinel runs before each phase of the Terraform workflow, meaning a `terraform init`, `terraform plan`, and `terraform apply`
d. Sentinel policies can be developed using HCL, JSON, or YAML

Explanation

Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products. It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.

When using Sentinel policies to define and enforce policies, it (Sentinel) runs after a terraform plan, but before a terraform apply. Therefore, you can potentially reduce costs on public cloud resources by NOT deploying resources that do NOT conform to policies enforced by Sentinel. For example, without Sentinel, your dev group might deploy instances that are too large, or too many of them, by accident or just because they can. Rather than being *REACTIVE* and shutting them down after they have been deployed, which it would cost you $, you can use Sentinel to prevent those large resources from being deployed in the first place.

144. Which of the following is an advantage of using Infrastructure as Code?

a. increase the time to market for application deployment
b. elimination of security vulnerabilities in your application deployment workflow
c. simplification of using a user interface to define your infrastructure
d. standardize your deployment workflow

Explanation

IaC helps organizations standardize their deployment workflows since they can codify and automate the deployment of applications and underlying infrastructure.

increase the time to market for application deployment - nope, who wants to use a tool that would INCREASE the time to market for application deployment? We want to find tools that DECREASE it.

simplification of using a user interface to define your infrastructure - Nah, we want to move quickly, and using a user interface for any tool likely slows us down and increases our chances of human error. We want to use a CLI or API for our changes so it's quick and predictable.

elimination of security vulnerabilities in your application deployment workflow - while Terraform could help with the security of your workflow, it doesn't guarantee it since Terraform just deploys what you tell it. It's not checking for security requirements or vulnerabilities or anything like that

145. As part of a Terraform configuration, you are deploying a Linux-based server using a default image that needs to be customized based on input variables. What feature of Terraform can execute a script on the server once is has been provisioned?

a. provider
b. data resource
c. remote-exec provisioner
d. local-exec provisioner

Explanation

We can utilize Terraform provisioners to deploy a web app onto an instance we've created. In order to run these steps, Terraform needs a connection block along with our generated SSH key from the previous labs in order to authenticate into our instance. Terraform can utilize both the `local-exec` provisioner to run commands on our local workstation (that is executing Terraform) and the `remote-exec` provisioner to execute commands against a resource that has been provisioned with Terraform.

Note: Provisioners should only be used as a last resort. For most common situations there are better alternatives.

146. After deploying a new virtual machine using Terraform, you find that the local script didn't run properly. However, Terraform reports the virtual machine was

successfully created. How can you force Terraform to replace the virtual machine without impacting the rest of the managed infrastructure?

a. update the virtual machine resource and run a terraform init
b. use terraform apply -replace to tag the resource for replacement
c. run a terraform destroy and then run terraform import to pull in the other resources under Terraform management
d. execute a terraform debug command to see why the script failed to run

Explanation

Using terraform apply -replace is how you tag a resource for replacement.
The terraform taint command is deprecated. However, you may still see terraform taint on the real exam. Please note the information below and be prepared to understand both options.
For Terraform v0.15.2 and later, we recommend using the -replace option with terraform apply instead.

147. You are having trouble with executing Terraform and want to enable the *most verbose* logs. What log level should you set for the TF_LOG environment variable?

a. ERROR
b. TRACE
c. DEBUG
d. INFO

Explanation

Trace is the most verbose logging level. In order, they
go TRACE, DEBUG, INFO, WARN and ERROR

148. Your organization has multiple engineers that have permission to manage Terraform as well as administrative access to the public cloud where these resources are provisioned. If an engineer makes a change outside of Terraform, what command can you run to detect drift and update the state file?

a. terraform get
b. terraform init
c. terraform apply -refresh-only
d. terraform state list

Explanation

To instruct Terraform to refresh the state file based on the current configuration of managed resources, you can use the terraform apply -refresh-only command. If Terraform discovers drift, it will update the state file with the changes.
Note that terraform refresh used to be the correct command here, but that command is deprecated. It might show up on the exam though.

149. You need to input variables that follow a key/value type structure. What type of variable would be used for this use case?

a. use a string variable to accomplish this task
b. use a map to satisfy this requirement
c. use a list of strings for this use case

d. use an array to satisfy the requirement

Explanation

Map is the best choice for this use case because it allows you to create a key/value structure that can easily be referenced in your Terraform configuration.

- use a list of strings for this use case - nope, this is just a list of strings that wouldn't create a key/value structure. '

- use a string variable to accomplish this task - nope, this would allow you to just create a single string value

- use an array to satisfy the requirement - array isn't a valid Type constraint in Terraform

150. Both you and a colleague are responsible for maintaining resources that host multiple applications using Terraform CLI. What feature of Terraform helps ensure only a single person can update or make changes to the resources Terraform is managing?

a. local backend

b. state locking

c. version control

d. provisioners

Explanation

If supported by your backend, Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state.

State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, Terraform will not continue. You can disable state locking for most commands with the -lock flag but it is not recommended.

151. By default, where does Terraform CLI store its state?

a. in the default workspace in Terraform Cloud

b. in the .terraform directory within the current working directory

c. in the terraform.tfstate file on the local backend

d. in a temp directory on the local machine executing the Terraform configurations

Explanation

Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to map real-world resources to your configuration, keep track of metadata, and improve performance for large infrastructures.

Terraform will store its state in the terraform.tfstate file on the local backend. This is the default but you can always change it if you want.

152. You need to set the value for a Terraform input variable. Which of the following allows you to set the value using an environment variable?

a. export VAR_database=prodsql01

b. export TF_db-pass=P@ssw0rd01!

c. export TF_VARIABLE_app=eCommerce01

d. export TF_VAR_user=dbadmin01

Explanation

As a fallback for the other ways of defining variables, Terraform searches the environment of its own process for environment variables named TF_VAR_ followed by the name of a declared variable.
None of the other environment variables shown as answers will successfully set a value for a Terraform variable.

153. Terraform includes a few types of provisioners. Which provisioner type will invoke a process on the machine executing Terraform?

a. local-exec provisioner
b. remote-exec provisioner
c. None of the provisioners have the ability to invoke a process on the local machine executing Terraform
d. file provisioner

Explanation
The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.
A remote provisioner will execute a process on a ***remote*** resource, such as the resource created by Terraform.
The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource.

154. You have a number of different variables in a parent module that calls multiple child modules. Can the child modules refer to *any* of the variables declared in the parent module?

a. No, it can only refer to the variables passed to the module
b. No, child modules can never refer to any variables declared in the parent module
c. Yes, child modules can refer to any variable in a parent module

Explanation
Child modules can only access variables that are passed in the calling module block.
The resources defined in a module are encapsulated, so the calling module cannot access their attributes directly. However, the child module can declare output values to selectively export certain values to be accessed by the calling module.

155. Based on the code snippet below, where is the module that the code is referencing?

```
module "server_subnet_1" {
  source         = "./modules/web_server"
  ami            = data.aws_ami.ubuntu.id
  key_name       = aws_key_pair.generated.key_name
  user           = "ubuntu"
  private_key    = tls_private_key.generated.private_key_pem
  subnet_id      = aws_subnet.public_subnets["public_subnet_1"].id
  security_groups = [aws_security_group.vpc-ping.id, aws_security_group.vpc-web.id]
}
```

a. stored in a private module registry managed by the organization

b. in the modules subdirectory in the current working directory where Terraform is being executed

c. the same working directory where Terraform is being executed

d. stored in the Terraform public module registry

Explanation

In this example, the user created a `modules` directory and then saved the module in that new directory. Therefore, the answer is in the modules subdirectory in the current working directory where Terraform is being executed.

Anytime you have a local path as the source, the module will be sourced from the referenced directory. You could also put the path of a VCS repository here, or a reference to a private or public module registry.

For example, if you wanted to reference a public module, you could use something like this:

```
module "network" {
  source  = "Azure/network/azurerm"
  version = "3.5.0"
  # insert the 1 required variable here
}
```

156. A child module created a new subnet for some new workloads. What Terraform block type would allow you to pass the subnet ID back to the parent module?

a. terraform block

b. output block

c. resource block

d. data block

Explanation

The resources defined in a module are encapsulated, so the calling module cannot access their attributes directly. However, the child module can declare output values to selectively export certain values to be accessed by the calling module.

157. What is ***NOT*** a benefit of using Infrastructure as Code?

a. the reduction of misconfigurations that could lead to security vulnerabilities and unplanned downtime

b. reducing vulnerabilities in your publicly-facing applications

c. your infrastructure configurations can be version controlled and stored in a code repository alongside the application code

d. the ability to programmatically deploy infrastructure

Explanation

Although Infrastructure as Code (IaC) tools allow you to programmatically deploy and manage your applications, it does NOT ensure that your applications have a reduced number of vulnerabilities. This security feature is not the responsibility of IaC, and you would need to pair IaC with another tool to scan your code to identify security vulnerabilities.

All of the wrong answers in this question are actually the primary use cases of Infrastructure as Code tools.

Infrastructure as code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage

your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.

158. When working with Terraform CLI/OSS workspaces, what command can you use to display the current workspace you are working in?

a. `terraform workspace show`
b. `terraform workspace select`
c. `terraform workspace new`
d. `terraform workspace`

Explanation

The `terraform workspace show` command is used to output the current workspace.
`terraform workspace new` will create a new workspace
`terraform workspace select` will tell Terraform what workspace to change to and use
`terraform workspace` is just a container that requires additional subcommands

159. You have infrastructure deployed with Terraform. A developer recently submitted a support ticket to update a security group to permit a new port. To satisfy the ticket, you update the Terraform configuration to reflect the changes and run a `terraform plan`. However, a co-worker has since logged into the console and manually updated the security group. What will happen when you run a `terraform apply`?

a. the security group will be changed back to the original configuration
b. Terraform will detect the drift and return an error.
c. the `terraform apply` command will require you to re-run the `terraform plan` command first
d. Nothing will happen. Terraform will validate the infrastructure matches the desired state.

Explanation

A `terraform apply` will run its own state refresh and see the configuration matches the deployed infrastructure, so no changes will be made to the infrastructure.

Terraform will detect the drift and return an error **-** since terraform apply will refresh state, it will see that the configuration has changed and now meets the desired state, therefore it won't do anything.

the security group will be changed back to the original configuration **-** this won't happen because the Terraform configuration now states it should have the new port. If Terraform changed it back to the original configuration, the real-world resources would NOT match the desired state

the terraform apply command will require you to re-run the terraform plan command first **-** terraform plan is not a requirement of the terraform apply command, so this won't happen

160. Your colleague provided you with a Terraform configuration file and you're having trouble reading it because parameters and blocks are not properly aligned. What command can you run to quickly update the file configuration file to make it easier to consume?

a. `terraform init`
b. `terraform workspace`

c. terraform fmt

d. terraform state

Explanation

The terraform fmt command is used to rewrite Terraform configuration files to a canonical format and style. This command applies a subset of the Terraform language style conventions, along with other minor adjustments for readability.

Other Terraform commands that generate Terraform configuration will produce configuration files that conform to the style imposed by terraform fmt, so using this style in your own files will ensure consistency.

The canonical format may change in minor ways between Terraform versions, so after upgrading Terraform we recommend to proactively run terraform fmt on your modules along with any other changes you are making to adopt the new version.

If you want to format ALL of your .tf files, you can use terraform fmt -recursive and it'll format all files in the current and all subdirectories.

terraform init - this is used to initialize and work with the Terraform backend

terraform state - this command is for working with and viewing Terraform state

terraform workspace - this command is used to create and manage Terraform OSS workspaces

161. Using multi-cloud and provider-agnostic tools like Terraform provides which of the following benefit?

a. slower provisioning speed allows the operations team to catch mistakes before they are applied

b. can be used across major cloud providers and VM hypervisors

c. forces developers to learn Terraform alongside their current programming language

d. increased risk due to all infrastructure relying on a single tool for management

Explanation

Terraform can be used across major cloud providers and VM hypervisors, which is a huge benefit. This is made possible by the thousands of providers/plugins that are written by HashiCorp, third-party partners, or individual contributors.

162. What command can be used to display the resources that are being managed by Terraform?

a. terraform state rm

b. terraform show

c. terraform version

d. terraform output

Explanation

The terraform show command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Machine-readable output is generated by adding the -json command-line flag.

terraform state rm is used to remove resources from state

terraform output is used to get values from an output of a module or configuration

terraform version is used to display the current version of Terraform

163. The command terraform destroy is actually just an alias to which command?

a. terraform apply -replace-all
b. terraform plan - destroy
c. terraform delete
d. terraform apply -destroy

Explanation

This command is just a convenience alias for the command terraform apply -destroy
For that reason, this command accepts most of the options that terraform apply accepts, although it does not accept a plan file argument and forces the selection of the "destroy" planning mode.

164. You have declared the variable as shown below. How should you reference this variable throughout your configuration?

```
variable "aws_region" {
  type        = string
  description = "region used to deploy workloads"
  default     = "us-east-1"
  validation {
    condition     = can(regex("^us-", var.aws_region))
    error_message = "The aws_region value must be a valid region in the USA, starting with \"us-\"."
  }
}
```

a. variable.aws_region.id
b. var.aws_region
c. var.aws_region.id
d. variable.aws_region

Explanation

Input variables (commonly referenced as just 'variables') are often declared in a separate file called `variables.tf`, although this is not required. Most people will consolidate variable declaration in this file for organization and simplification of management. Each variable used in a Terraform configuration must be declared before it can be used. Variables are declared in a variable block - one block for each variable. The variable block contains the variable name, most importantly, and then often includes additional information such as the type, a description, a default value, and other options.
The value of a Terraform variable can be set multiple ways, including setting a default value, interactively passing a value when executing a terraform plan and apply, using an environment variable, or setting the value in a `.tfvars` file. Each of these different options follows a strict order of precedence that Terraform uses to set the value of a variable.
A huge benefit of using Terraform is the ability to reference other resources throughout your configuration for other functions. These might include getting certain values needed to create other resources, creating an output to export a specific value, or using data retrieved from a data block. Most of these use dot-separated paths for elements of object values.
The following represents the kinds of named values available in Terraform:

* <RESOURCE TYPE>.<NAME> represents a managed resource of the given type and name.
*** var.<NAME> is the value of the input variable of the given name.**
* local.<NAME> is the value of the local value of the given name.
* module.<MODULE NAME> is a value representing the results of a module block.
* data.<DATA TYPE>.<NAME> is an object representing a data resource of a given type and name
* Additional named values include ones for filesystem and workspace info and block-local values

165. What feature does Terraform use to map configuration to resources in the real world?

a. local variables
b. parallelism
c. resource blocks
d. state

Explanation

Terraform requires some sort of database to map Terraform config to the real world. When you have a resource `resource "aws_instance" "foo"` in your configuration, Terraform uses this map to know that instance `i-abcd1234` is represented by that resource.
For some providers like AWS, Terraform could theoretically use something like AWS tags. Early prototypes of Terraform actually had no state files and used this method. However, we quickly ran into problems. The first major issue was a simple one: not all resources support tags, and not all cloud providers support tags.
Therefore, for mapping configuration to resources in the real world, Terraform uses its own state structure.
Parallelism is the way Terraform can deploy many resources at the same time to speed up the deployment
Local variables are used to reduce repeating the same expressions or values over and over in your code
Resource blocks are the block type that deploys actual resources

166. Where is the most secure place to store credentials when using a remote backend?

a. using an input variable defined in your `variables.tf` file
b. environment variables
c. defined outside of Terraform
d. in the backend configuration block where the remote state location is defined

Explanation

Anytime you can configure these credentials outside of Terraform is your best choice. Environment variables would be the second most-secure choice here. The primary focus is to ensure your credentials are not stored in plaintext and committed to a code repository. ***NOTE:*** *You could use an encrypted file to store credentials and that encrypted file could be accessed by Terraform to read the creds.*
environment variables **-** this is the SECOND best choice here, with storing outside of Terraform using a credential file being the best choice

in the backend configuration block where the remote state location is defined **-** this is exactly what we DO NOT want to do because the creds are now stored in cleartext, which isn't desirable. Also, backend config might also get committed to a code repo.
using an input variable defined in your variables.tf file **-** this is another example of what we DO NOT want - storing the credentials in a cleartext file

167. Which of the following Terraform versions would be permitted to run the Terraform configuration based on the following code snippet?

```
terraform {
  required_version = "~> 1.0.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.1.0"
    }
  }
}
```

a. Terraform v1.1.0
b. Terraform v1.2.0
c. Terraform v1.4.9
d. Terraform v1.0.5

Explanation
When setting Terraform `required_version` or provider constraints, the `~` specifies that only the right-most version number can be incremented - therefore only v1.0.5 will satisfy the requirements.

168. You've included two different modules from the official Terraform registry in a new configuration file. When you run a `terraform init`, where does Terraform OSS download and store the modules locally?

a. in the `.terraform/modules` folder in the working directory
b. in the `/tmp` directory of the machine executing Terraform
c. Terraform stores them in memory on the machine running Terraform
d. in the same root directory where the Terraform configuration files are stored

Explanation
When plugins and modules are downloaded, they are stored under their respective directory in the `.terraform` folder within the current working directory. For example, providers/plugins are downloaded to `.terraform/providers` and modules are downloaded to the `.terraform/modules` directory.
Terraform doesn't use any of these directories to store any downloaded content when running a `terraform init`

169. Infrastructure as Code (IaC) provides many benefits to help organizations deploy application infrastructure much faster than manually clicking in the console. Which is NOT an additional benefit to IaC?

a. eliminates API communication to the target platform
b. code can easily be shared and reused
c. allows infrastructure to be versioned
d. creates self-documenting infrastructure

Explanation

Eliminating API communication to the target platform is NOT a benefit of IaC. In fact, Terraform likely increases communication with the backend platform since Terraform uses the platform's API to build and manage infrastructure.

Remember that Terraform providers/plugins are essentially the features that enable communication with the platform's API on your behalf.

All of the wrong answers are essentially direct benefits of using Terraform.

170. After hours of development, you've created a new Terraform configuration from scratch and now you want to test it. Before you can provision the resources, what is the first command that you should run?

a. `terraform import`
b. `terraform init`
c. `terraform apply`
d. `terraform validate`

Explanation

When you develop new Terraform code, and you're ready to test it out, the first thing you need to do is run `terraform init` in order to initialize the working directory and download any required providers or referenced modules. Even if you're in a directory that has some of these plugins, you should still run `terraform init` to make sure all the providers have been downloaded. You could even run a `terraform init -upgrade` to ensure you have the latest versions of the plugins that meet your requirements.

None of these commands would work if you haven't initialized the working directory yet. You would get an error similar to this:

```
| Error: Could not load plugin
|
|
| Plugin reinitialization required. Please run "terraform init".
|
| Plugins are external binaries that Terraform uses to access and manipulate
| resources. The configuration provided requires plugins which can't be located,
| don't satisfy the version constraints, or are otherwise incompatible.
|
| Terraform automatically discovers provider requirements from your
| configuration, including providers used in child modules. To see the
| requirements and constraints, run "terraform providers".
```

```
|
| failed to instantiate provider "registry.terraform.io/hashicorp/aws" to obtain schema:
| unknown provider "registry.terraform.io/hashicorp/aws"
```

171. Which of the following code snippets will ensure you're using a specific version of the AWS provider?

a.

```
terraform {
 required_version = ">= 3.0"
}
```

b.

```
provider "aws" {
 region = "us-east-1"
 required_provider ">= 3.0"
}
```

c.

```
terraform {
 required_providers {
  aws = ">= 3.0"
 }
}
```

d.

```
provider "aws" {
 region = "us-east-2"
 required_version ">= 3.0"
}
```

Explanation

To specify the version of Terraform provider that is required, you need to use the `required_providers` block parameter under the `terraform` block. HashiCorp recommends that you explicitly set the version of both Terraform and the required providers/plugins to avoid issues when upgrading to the latest versions.

None of these wrong answers are valid configuration blocks for Terraform to set the specific version of Terraform.

172. Beyond storing state, what capability can an enhanced storage backend, such as the remote backend, provide your organization?

a. provides versioning capabilities on your state file in the event it becomes corrupted
b. allow multiple people to execute operations on the state file at the same time
c. replicate your state to a secondary location for backup
d. execute your Terraform on infrastructure either locally or in Terraform Cloud

Explanation

Using an enhanced storage backend allows you to execute your Terraform on infrastructure either locally or in Terraform Cloud. Note that this ***enhanced storage backend*** term has now been deprecated by Terraform but it's likely to show up in the test for a while.

Note: *In Terraform versions prior to 1.1.0, backends were also classified as being 'standard' or 'enhanced', where the latter term referred to the ability of the remote backend to store state*

and perform Terraform operations. This classification has been removed, clarifying the primary purpose of backends. Refer to Using Terraform Cloud for details about how to store state, execute remote operations, and use Terraform Cloud directly from Terraform.

173. Given the definition below, what Terraform feature is being described?

"helps you share Terraform providers and Terraform modules across your organization. It includes support for versioning, a searchable list of available providers and modules, and a configuration designer to help you build new workspaces faster."

a. Terraform Workspaces
b. HashiCorp Sentinel
c. CDK for Terraform
d. Private Module Registry

Explanation

This definition is describing the Private Module Registry...
Terraform Cloud's private registry works similarly to the public Terraform Registry and helps you share Terraform providers and Terraform modules across your organization. It includes support for versioning, a searchable list of available providers and modules, and a configuration designer to help you build new workspaces faster.

174. Given the following code snippet, what is the managed resource name for this particular resource?

```
resource "aws_vpc" "prod-vpc" {
  cidr_block = var.vpc_cidr
   tags = {
    Name       = var.vpc_name
    Environment = "demo_environment"
    Terraform   = "true"
  }
   enable_dns_hostnames = true
}
```

a. demo environment
b. prod-vpc
c. aws_vpc
d. resource.aws_vpc

Explanation

prod-vpc is the managed resource name for this resource. More specifically, you'd use aws_vpc.prod-vpc address to refer to this resource in your code, like this:

```
output "vpc_id" {
  value = aws_vpc.prod-vpc.id
}
```

175. As you are developing new Terraform code, you are finding that you are constantly repeating the same expression over and over throughout your code, and you are worried about the effort that will be needed if this expression needs to be changed. What feature of Terraform can you use to avoid repetition and make your code easier to maintain?

a. data block
b. locals
c. remote backend
d. terraform graph

Explanation

A local value assigns a name to an expression, so you can use it multiple times within a configuration without repeating it. The expressions in local values are not limited to literal constants; they can also reference other values in the configuration in order to transform or combine them, including variables, resource attributes, or other local values.

You can use local values to simplify your Terraform configuration and avoid repetition. Local values (locals) can also help you write a more readable configuration by using meaningful names rather than hard-coding values. If overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.

Use local values only in moderation, in situations where a single value or result is used in many places and that value is likely to be changed in the future. The ability to easily change the value in a central place is the key advantage of local values.

176. You have a configuration file that you've deployed to one AWS region already but you want to deploy the same configuration file to a second AWS region without making changes to the configuration file. What feature of Terraform can you use to accomplish this?

a. `terraform get`
b. `terraform import`
c. `terraform plan`
d. `terraform workspace`

Explanation

Workspaces should be used in this scenario to create separate state files for each regional deployment. When you use a workspace in Terraform OSS, you get a brand new state file to work with that is completely separate from the original. Therefore, you can modify environment variables or other values and use the same Terraform without negatively impacting resources that were deployed in any other workspace.

To create a new workspace, you can use the command `terraform workspace new <name>`

`terraform plan` - this command compares the current infrastructure against the desired state (configuration file) and proposes changes to your infrastructure. This is also commonly referred to as a dry run.

`terraform get` - this command is used to download modules

`terraform import` - this command can be used to import existing resources and pull them under Terraform management

177. Rather than having to scan and inspect every resource on every run, Terraform relies on what feature to help manage resources?

a. local variables
b. providers
c. environment variables
d. state

Explanation
Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to map real-world resources to your configuration, keep track of metadata, and improve performance for large infrastructures.
This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.
environment variables - these can be used to provide values for variables or other Terraform configurations, but this is now how Terraform manages resources
providers - these are Terraform plugins that enable communication to a platform's API to allow Terraform to provision resources, but it's not the feature that it uses to manage the resources that it is managing.
local variables - these provide values to use for the deployment and management of deployed resources in Terraform. It doesn't use these to manage resources

178. What actions does a `terraform init` perform for you?

a. compares the current configuration to the prior state and notes any differences
b. ensures any configuration file that ends with a `.tf` file extension is syntactically valid and internally consistent
c. ensures that all Terraform files match the canonical formatting and style
d. downloads plugins and retrieves the source code for referenced modules

Explanation
One of the functions that a `terraform init` does for you is download any referenced modules/plugins so they can be used locally. This is required before you run most other terraform CLI commands.
- ensures any configuration file that ends with a .tf file extension is syntactically valid and internally consistent - this is the job of `terraform validate`, not `terraform init`
- ensures that all terraform files match the canonical formatting and style - this is done by the CLI command `terraform fmt`
- compares the current configuration to the prior state and notes any differences - nope, that's what a `terraform plan` will do for you, not `terraform init`

179. You are managing multiple resources using Terraform running in AWS. You want to destroy all the resources except for a single web server. How can you accomplish this?

a. delete the web server resource block from the configuration file and run a `terraform apply`
b. run a `terraform state rm` to remove it from state and then destroy the remaining resources by running `terraform destroy`
c. run a `terraform import` command against the web server and then execute a `terraform destroy`
d. change to a different workspace and run a `terraform destroy`

Explanation
To accomplish this, you can delete the resource from state so Terraform no longer knows anything about it. Then you can run a `terraform destroy` to destroy the remaining resources. During the destroy, Terraform won't touch the web server since it is no longer managing it. This

is similar to how Terraform won't impact existing resources that it does not know about when creating, modifying, and destroying resources in your local or public cloud infrastructure.
To delete a resource from state, you can use the `terraform state rm <address>` command, which will effectively make Terraform "forget" the object while it continues to exist in the remote system.
change to new workspace and run terraform destroy - changing to a different workspace would have no impact on any resources in the current workspace. It could, however, impact any resources that were provisioned with the second workspace that you change to
terraform import & terraform destroy - using the import command here is basically the opposite of what we want to do. We want Terraform to "forget" about a particular resource, and the import command pulls existing resources under Terraform management
delete the block and run terraform apply - this would actually destroy ONLY the web server that we want to keep, so essentially doing the exact opposite of what we're trying to accomplish

180. Given the code snippet below, how would you identify the `arn` to be used in the output block that was retrieved by the data block?

```
data "aws_s3_bucket" "data-bucket" {
  bucket = "my-data-lookup-bucket-btk"
}
...
output "s3_bucket_arn" {
  value = ????
}
```

a. **`data.aws_s3_bucket.data-bucket.arn`**
b. `data.aws_s3_bucket.arn`
c. `aws_s3_bucket.data-bucket`
d. `data.aws_s3_bucket.data-bucket`

Explanation
To refer to a resource, you'd use `<block type>.<resource type>.<name>` . In this case, the `<block type>` is **data**, the `<resource type>` is **aws_s3_bucket**, and then you'd add the **arn** attribute at the end.

181. A coworker provided you with Terraform configuration file that includes the code snippet below. Where will Terraform download the referenced module from?

```
terraform {
  required_providers {
    kubernetes = {
      source = "hashicorp/kubernetes"
      version = "2.6.1"
    }
  }
}

provider "kubernetes" {
  # Configuration below
```

...

a. from the configured VCS provider in the `hashicorp/kubernetes` repo
b. from the `hashicorp/kubernetes` directory where Terraform is executed
c. the official Terraform public module registry
d. from the official Kubernetes public GitHub repo

Explanation

When a module is located at `hashicorp/<name>`, Terraform download it from the official Terraform pubic module registry. This is specified by the `source` argument within the `module` block. The module installer supports the following source types:

- Local paths
- Terraform Registry
- GitHub
- Bitbucket
- Generic Git, Mercurial repositories
- HTTP URLs
- S3 buckets
- GCS buckets
- Modules in Package Sub-directories

182. You are using Terraform to manage some of your AWS infrastructure. You notice that a new version of the provider now includes additional functionality you want to take advantage of. What command do you need to run to upgrade the provider?

a. `terraform providers`
b. `terraform init -upgrade`
c. `terraform plan`
d. `terraform get hashicorp/aws`

Explanation

To upgrade an existing provider that you have already downloaded, you need to run `terraform init -upgrade` . This command will upgrade all previously-selected plugins to the newest version that complies with the configuration's version constraints. This will cause Terraform to ignore any selections recorded in the dependency lock file, and to take the newest available version matching the configured version constraints.

183. What flag would you use to perform a dry-run of your changes and save the proposed changes to a file for future use?

a. `terraform plan -file=bryan`
b. `terraform plan -save=bryan`
c. `terraform plan -output=bryan`
d. `terraform plan -out=bryan`

Explanation

Make sure to know that you need to use the flag `-out` to save a `terraform plan` output so you can execute it later

184. What command can you use to display details about the resource as shown below?

```
resource "aws_internet_gateway" "demo" {
  vpc_id = aws_vpc.vpc.id
  tags = {
    Name = "demo_igw"
  }
}
```

a. terraform display aws_internet_gateway.demo
b. terraform state show aws_internet_gateway.demo
c. terraform state list aws_internet_gateway.demo
d. terraform state rm aws_internet_gateway.demo

Explanation

terraform state show ADDRESS will show the attributes of a single resource, therefore the answer is aws_internet_gateway.demo

185. What command can be used to get an interactive console to evaluate expressions in your Terraform code?

a. terraform console
b. terraform graph
c. terraform env
d. terraform help

Explanation

terraform console [options]
This command provides an interactive command-line console for evaluating and experimenting with expressions.

186. By default, in what file does Terraform store its state file?

a. terraform.tfstate
b. terraform.state
c. terraformstate.hcl
d. state.json

Explanation

Terraform stores its state in a file called terraform.tfstate

187. You have the following code snippet as part of your Terraform configuration. How would you reference the id of the *s3_bucket*?

```
data "aws_s3_bucket" "data_bucket" {
  bucket = "my-data-lookup-bucket-bk"
}
```

a. data_bucket.id
b. data.aws_s3_bucket.data_bucket.id
c. data.data_bucket.id
d. aws_s3_bucket.data_bucket.id

Explanation

You would use data.<resource type>.<resource name>.id

188. How can you reference all of the subnets that are created by this resource block?

```
#Deploy the private subnets
resource "aws_subnet" "private_subnets" {
 for_each        = var.private_subnets
 vpc_id          = aws_vpc.vpc.id
 cidr_block      = cidrsubnet(var.vpc_cidr, 8, each.value)
 availability_zone = tolist(data.aws_availability_zones.available.names)[each.value]
  tags = {
   Name     = each.key
   Terraform = "true"
 }
}
```

a. aws_subnet.private_subnets[*]
b. aws_subnet.private_subnets.id
c. aws_subnet.private_subnets(each.value)
d. aws_subnet.private_subnets[0,3]

Explanation

You can reference all of the subnets created by this for_each by using a [*] at the end of the resource address like this aws_subnet.private_subnets[*]

189. What command can be used to ensure your code is syntactically valid and internally consistent?

a. terraform validate
b. terraform plan
c. terraform env
d. terraform fmt

Explanation

terraform validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state. It is thus primarily useful for general verification of reusable modules, including the correctness of attribute names and value types.

190. You have recently added new resource blocks to your configuration from a different provider. What command do you need to run before you can run a terraform plan/apply?

a. terraform init
b. terraform validate
c. terraform get
d. terraform fmt

Explanation

You need to run a terraform init in order to download the provider for the new resource blocks you added

191. You want Terraform to redeploy a specific resource that it is managing. What command should you use to mark the resource for replacement?

a. terraform fmt
b. terraform apply -replace
c. terraform plan -destroy
d. terraform destroy

Explanation

You would mark the resource for replacement using terraform apply -replace.

NOTE: This used to be terraform taint and has been replaced with terraform apply -replace

192. How is the Terraform remote backend different than other state backends such as S3, Consul, etc.?

a. It can execute Terraform runs on dedicated infrastructure on premises or in Terraform Cloud
b. It doesn't show the output of a terraform apply locally
c. It is only available to paying customers
d. All of the above

Explanation: The Terraform remote backend offers various functionalities, including the ability to execute Terraform runs in dedicated infrastructure either on-premises or in Terraform Cloud, providing better control and scalability.

193. What is the workflow for deploying new infrastructure with Terraform?

a. terraform plan to import the current infrastructure to the state file, make code changes, and terraform apply to update the infrastructure.
b. Write a Terraform configuration, run terraform show to view proposed changes, and terraform apply to create new infrastructure.
c. terraform import to import the current infrastructure to the state file, make code changes, and terraform apply to update the infrastructure.
d. Write a Terraform configuration, run terraform init, run terraform plan to view planned infrastructure changes, and terraform apply to create new infrastructure.

Explanation: The correct workflow involves initializing Terraform in a directory, creating a configuration, planning changes using **terraform plan**, and applying those changes using **terraform apply**.

194. You run a local-exec provisioner in a null resource called null_resource.run_script and realize that you need to rerun the script.
Which of the following commands would you use first?

a. terraform taint null_resource.run_script
b. terraform apply -target=null_resource.run_script
c. terraform validate null_resource.run_script
d. terraform plan -target=null_resource.run_script

Explanation: terraform taint marks a resource for recreation. Running this command on the null resource will trigger the provisioner to execute again during the next **terraform apply**.

195. Which provisioner invokes a process on the resource created by Terraform?

a. remote-exec
b. null-exec
c. local-exec
d. file

Explanation: The **local-exec** provisioner runs a command on the machine where Terraform is executed after a resource is created.

196. Which of the following is not true of Terraform providers?

a. Providers can be written by individuals
b. Providers can be maintained by a community of users
c. Some providers are maintained by HashiCorp
d. Major cloud vendors and non-cloud vendors can write, maintain, or collaborate on Terraform providers
e. None of the above

Explanation: All statements are true. Providers can be written by individuals, maintained by communities or HashiCorp, and are contributed by both major cloud vendors and non-cloud vendors.

197. What command does Terraform require the first time you run it within a configuration directory?

a. terraform import
b. terraform init
c. terraform plan
d. terraform workspace

Explanation: **terraform init** initializes a directory containing Terraform configuration files. It installs necessary plugins and initializes a working directory.

198. You have deployed a new webapp with a public IP address on a cloud provider. However, you did not create any outputs for your code. What is the best method to quickly find the IP address of the resource you deployed?

a. Run terraform output ip_address to view the result
b. In a new folder, use the terraform_remote_state data source to load in the state file, then write an output for each resource that you find the state file
c. Run terraform state list to find the name of the resource, then terraform state show to find the attributes including public IP address
d. Run terraform destroy then terraform apply and look for the IP address in stdout

Explanation: **terraform state** commands help inspect the state file, allowing you to list resources and view their attributes, including the public IP address, without destroying and recreating resources or modifying the configuration.

199. Which of the following is not a key principle of infrastructure as code?

a. Versioned infrastructure
b. Golden images
c. Idempotence
d. Self-describing infrastructure

Explanation: Golden images refer to pre-configured, fully patched machine images ready for deployment, but it's not a key principle of infrastructure as code, which focuses on defining infrastructure through code, versioning, idempotence, and self-describing infrastructure.*1*

200. What is the provider for this fictitious resource?

```
resource "aws_vpc" "main" {
    name = "test"
}
```

a. vpc
b. main
c. aws
d. teston #14*To*

Explanation: The provider for managing resources related to Amazon Web Services (AWS) is typically defined as "aws" in Terraform configurations.*pic 1*

201. If you manually destroy infrastructure, what is the best practice reflecting this change in Terraform?

a. **Run terraform refresh**
b. It will happen automatically
c. Manually update the state fire
d. Run terraform import

Explanation: If infrastructure is manually destroyed, you should update Terraform state to reflect that change using **terraform import**. This helps Terraform recognize the current state of the resource.5*Topic 1*

202. What is not processed when running a terraform refresh?

a. State file
b. Configuration file
c. Credentials
d. Cloud provider

Explanation: **terraform refresh** updates the state file with the current state of resources in the infrastructure, but it does not process the configuration file.6*Topic 1*

203. What information does the public Terraform Module Registry automatically expose about published modules?

a. Required input variables
b. Optional inputs variables and default values
c. Outputs
d. All of the above
e. None of the above

Explanation: The public Terraform Module Registry automatically exposes required input variables, optional input variables with default values, and outputs about published modules.

204. Which of the following is not a valid string function in Terraform?

a. split
b. join
c. slice
d. chompestion

Explanation: **chomp** is not a valid string function in Terraform.#20*Topic 1*

205. You have recently started a new job at a retailer as an engineer. As part of this new role, you have been tasked with evaluating multiple outages that occurred during

peak shopping time during the holiday season. Your investigation found that the team is manually deploying new compute instances and configuring each compute instance manually. This has led to inconsistent configuration between each compute instance. How would you solve this using infrastructure as code?

a. Implement a ticketing workflow that makes engineers submit a ticket before manually provisioning and configuring a resource
b. Implement a checklist that engineers can follow when configuring compute instances
c. Replace the compute instance type with a larger version to reduce the number of required deployments
d. Implement a provisioning pipeline that deploys infrastructure configurations committed to your version control system following code reviews

Explanation: Implementing a provisioning pipeline using infrastructure as code allows for consistent deployment and configuration of compute instances. It ensures that configurations are version-controlled, reviewed, and deployed systematically, reducing inconsistencies.

206. Why would you use the terraform taint command?

a. When you want to force Terraform to destroy a resource on the next apply
b. When you want to force Terraform to destroy and recreate a resource on the next apply
c. When you want Terraform to ignore a resource on the next apply
d. When you want Terraform to destroy all the infrastructure in your workspace

Explanation: terraform taint marks a resource for recreation on the next apply, forcing Terraform to destroy and recreate it.

207. When should you use the force-unlock command?

a. You see a status message that you cannot acquire the lock
b. You have a high priority change
c. Automatic unlocking failed
d. You apply failed due to a state lock

Explanation: terraform force-unlock releases the state lock manually in case the apply failed due to a state lock.

208. Terraform can import modules from a number of sources `" which of the following is not a valid source?

a. FTP server
b. GitHub repository
c. Local path
d. Terraform Module Registry

Explanation: Terraform can import modules from GitHub repositories, local paths, and the Terraform Module Registry, but not directly from an FTP server.

209. Which of the following is available only in Terraform Enterprise or Cloud workspaces and not in Terraform CLI?

a. Secure variable storage
b. Support for multiple cloud providers
c. Dry runs with terraform plan
d. Using the workspace as a data source

Explanation: Secure variable storage is typically a feature available in Terraform Enterprise or Cloud workspaces, providing secure handling of sensitive data.

210. Which of the following is the correct way to pass the value in the variable num_servers into a module with the input servers?
    a. servers = num_servers
    b. servers = variable.num_servers
    c. servers = var(num_servers)
    d. servers = var.num_servers

Explanation: When passing variables to a module, you reference the variable using **variable** keyword followed by the variable name. So, to pass **num_servers** into a module's **servers** input variable, you'd use **servers = variable.num_servers**.

211. What does the default "local" Terraform backend store?
    a. tfplan files
    b. Terraform binary
    c. Provider plugins
    d. State file

Explanation: The "local" backend stores the state file, keeping track of the infrastructure Terraform manages.

212. You have multiple team members collaborating on infrastructure as code (IaC) using Terraform, and want to apply formatting standards for readability. How can you format Terraform HCL (HashiCorp Configuration Language) code according to standard Terraform style convention?
    a. Run the terraform fmt command during the code linting phase of your CI/CD process
    b. Designate one person in each team to review and format everyone's code
    c. Manually apply two spaces indentation and align equal sign "=" characters in every Terraform file (*.tf)
    d. Write a shell script to transform Terraform files using tools such as AWK, Python, and sed

Explanation: **terraform fmt** is used to automatically format Terraform code according to the standard style conventions, and it can be integrated into the CI/CD pipeline for consistent formatting.

213. What value does the Terraform Cloud/Terraform Enterprise private module registry provide over the public Terraform Module Registry?
    a. The ability to share modules with public Terraform users and members of Terraform Enterprise Organizations
    b. The ability to tag modules by version or release
    c. The ability to restrict modules to members of Terraform Cloud or Enterprise organizations
    d. The ability to share modules publicly with any user of Terraform

Explanation: The private module registry in Terraform Cloud/Enterprise allows restricting module access to specific organization members, providing greater control over module distribution and access.

214. Which task does terraform init not perform?

a. Sources all providers present in the configuration and ensures they are downloaded and available locally
b. Connects to the backend
c. Sources any modules and copies the configuration locally
d. Validates all required variables are present

Explanation: **terraform init** initializes a working directory but does not validate all required variables. Variable validation occurs during the **terraform plan** or **terraform apply** phase.

215. Which argument(s) is (are) required when declaring a Terraform variable?
a. type
b. default
c. description
d. All of the above
e. None of the above

Explanation: Among the choices provided, specifying the type of a variable is required while declaring a Terraform variable. Default and description are optional.

216. When using a module block to reference a module stored on the public Terraform Module Registry such as:

```
module "consul" {
  source = "hashicorp/consul/aws"
}
```

How do you specify version 1.0.0?
a. Modules stored on the public Terraform Module Registry do not support versioning
b. Append ?ref=v1.0.0 argument to the source path
c. Add version = "1.0.0" attribute to module block
d. Nothing ג€" modules stored on the public Terraform Module Registry always default to version 1.0.0

Explanation: To specify a particular version, you add the **version** attribute to the module block, like **version = "1.0.0"**. If omitted, it defaults to version 1.0.0.

217. Where does the Terraform local backend store its state?
a. In the /tmp directory
b. In the terraform file
c. In the terraform.tfstate file
d. In the user's terraform.state file

Explanation: The state for the Terraform local backend is stored in the **terraform.tfstate** file by default.

218. Which option can not be used to keep secrets out of Terraform configuration files?
a. A Terraform provider

b. Environment variables
c. A -var flag
d. secure string

Explanation: There isn't a direct reference to a "secure string" to keep secrets out of Terraform configuration files. Instead, best practices involve using options like Terraform providers, environment variables, and flags like **-var**.

219. What is one disadvantage of using dynamic blocks in Terraform?
a. They cannot be used to loop through a list of values
b. Dynamic blocks can construct repeatable nested blocks
c. They make configuration harder to read and understand
d. Terraform will run more slowly

Explanation: While dynamic blocks offer flexibility, they can sometimes make the configuration less readable and understandable due to their varying nature.

220. Examine the following Terraform configuration, which uses the data source for an AWS AMI.What value should you enter for the ami argument in the AWS instance resource?

```
data "aws_ami" "ubuntu" {
  ...
}

resource "aws_instance" "web" {
  ami = _____________________
  instance_type = "t2.micro"

  tags = {
    Name = "HelloWorld"
  }
}
```

a. aws_ami.ubuntu
b. data.aws_ami.ubuntu
c. data.aws_ami.ubuntu.id
d. aws_ami.ubuntu.id

Explanation: The correct value for referencing the AMI ID from the data source in an AWS instance resource is **data.aws_ami.ubuntu.id**.

221. You have never used Terraform before and would like to test it out using a shared team account for a cloud provider. The shared team account already contains 15 virtual machines (VM). You develop a Terraform configuration containing one VM, perform terraform apply, and see that your VM was created successfully. What should you do to delete the newly-created VM with Terraform?
a. The Terraform state file contains all 16 VMs in the team account. Execute terraform destroy and select the newly-created VM.

b. The Terraform state file only contains the one new VM. Execute terraform destroy.
c. Delete the Terraform state file and execute Terraform apply.
d. Delete the VM using the cloud provider console and terraform apply to apply the changes to the Terraform state file.

Explanation: To remove the newly created VM, deleting the Terraform state file and executing Terraform apply will sync the state with the cloud provider, removing the additional VM.

222. What is the name assigned by Terraform to reference this resource?

```
resource "azurerm_resource_group" "dev" {
  name = "test"
  location = "westus"
}
```

a. dev
b. azurerm_resource_group
c. azurerm
d. test

Explanation: The name assigned by Terraform to reference a resource typically relates to the resource type or a specific resource, resembling a description or category in the code.

- dev seems generic and is less likely to be the assigned name for a specific resource.
- azurerm_resource_group is more specific and resembles a resource type, possibly referring to an Azure Resource Group in an Azure provider configuration.
- azurerm could be a bit ambiguous; it might refer to the provider or a top-level resource, but it's not clear.
- test is a generic term and is unlikely to be a reference to a specific resource.

223. Where in your Terraform configuration do you specify a state backend?
a. The terraform block
b. The resource block
c. The provider block
d. The datasource block

Explanation: The state backend configuration, which determines where Terraform stores its state, is specified within the **terraform** block in the configuration file. It defines where the state file is stored, allowing collaboration and remote state management.

224. What command should you run to display all workspaces for the current configuration?
a. terraform workspace
b. terraform workspace show
c. terraform workspace list
d. terraform show workspace

Explanation: This command lists all the available workspaces for the current configuration. It helps manage and switch between different environments or sets of infrastructure configurations.

225. Which of these is the best practice to protect sensitive values in state files?
   a. Blockchain
   b. Secure Sockets Layer (SSL)
   c. Enhanced remote backends
   d. Signed Terraform providers

Explanation: Enhanced remote backends, such as using encrypted remote state storage options like Amazon S3 with encryption enabled or Azure Blob Storage with appropriate access controls, provide better security for sensitive values stored in Terraform state files.

226. When does terraform apply reflect changes in the cloud environment?
   a. Immediately
   b. However long it takes the resource provider to fulfill the request
   c. After updating the state file
   d. Based on the value provided to the -refresh command line argument
   e. None of the above

Explanation: Terraform apply initiates changes in the cloud environment by instructing the resource provider to make modifications. The time taken for these changes to reflect in the cloud environment depends on the provider's speed in fulfilling the requests.

227. How would you reference the "name" value of the second instance of this fictitious resource?

```
resource "aws_instance" "web" {
  count = 2
  name = "terraform-${count.index}"
}
```

   a. element(aws_instance.web, 2)
   b. aws_instance.web[1].name
   c. aws_instance.web[1]
   d. aws_instance.web[2].name
   e. aws_instance.web.*.name

Explanation: This references the "name" attribute of the second instance of the AWS resource named "web". The **[2]** index signifies the second element in the list of instances.

228. A Terraform provider is not responsible for:
   a. Understanding API interactions with some service
   b. Provisioning infrastructure in multiple clouds
   c. Exposing resources and data sources based on an API
   d. Managing actions to take based on resource differences

Explanation: While Terraform providers handle understanding API interactions, exposing resources, and provisioning infrastructure, they are not directly responsible for deciding actions based on differences between resources.

229. What is terraform refresh intended to detect?
   a. Terraform configuration code changes
   b. Empty state files
   c. State file drift

d. Corrupt state files

Explanation: Terraform refresh reconciles the state Terraform knows about (as recorded in the state file) with the real-world infrastructure, detecting any discrepancies or drift between them.

230. Which of the following is not a valid Terraform collection type?
   a. list
   b. map
   c. tree
   d. set

Explanation: In Terraform, the collection types are **list**, **map**, and **set**. However, there isn't a native "tree" type used explicitly as a collection type within Terraform.

231. How can terraform plan aid in the development process?
   a. Validates your expectations against the execution plan without permanently modifying state
   b. Initializes your working directory containing your Terraform configuration files
   c. Formats your Terraform configuration files
   d. Reconciles Terraform's state against deployed resources and permanently modifies state using the current status of deployed resources

Explanation: **terraform plan** provides an execution plan showing what actions Terraform will take when **apply** is called. It helps validate changes without actually applying them, preventing accidental changes to infrastructure.

232. You would like to reuse the same Terraform configuration for your development and production environments with a different state file for each. Which command would you use?
   a. terraform import
   b. terraform workspace
   c. terraform state
   d. terraform init

Explanation: Using Terraform workspaces allows you to maintain separate state files for different environments while using the same configuration. It helps manage multiple environments with ease.

233. What is the name assigned by Terraform to reference this resource?

```
mainresource “google_compute_instance” “main” {
     name = “test”
}
```

   a. compute_instance
   b. main
   c. google
   d. teat

Explanation: This is an example of how Terraform might assign a name to reference a specific resource. The naming convention usually follows the type of resource being created.

234. You're building a CI/CD (continuous integration/ continuous delivery) pipeline and need to inject sensitive variables into your Terraform run. How can you do this safely?

a. **Pass variables to Terraform with a -var flag**
b. Copy the sensitive variables into your Terraform code
c. Store the sensitive variables in a secure_vars.tf file
d. Store the sensitive variables as plain text in a source code repository

Explanation: Using the **-var** flag allows you to pass sensitive variables securely without storing them directly in the Terraform configuration or exposing them in version control systems.

235. Your security team scanned some Terraform workspaces and found secrets stored in a plaintext in state files. How can you protect sensitive data stored in Terraform state files?
a. Delete the state file every time you run Terraform
b. Store the state in an encrypted backend
c. Edit your state file to scrub out the sensitive data
d. Always store your secrets in a secrets.tfvars file.

Explanation: Utilizing an encrypted backend for state storage (like Amazon S3 with server-side encryption, Azure Blob Storage with encryption at rest, etc.) helps safeguard sensitive data stored in Terraform state files.

236. You want to know from which paths Terraform is loading providers referenced in your Terraform configuration (*.tf files). You need to enable debug messages to find this out. Which of the following would achieve this?
a. Set the environment variable TF_LOG=TRACE
b. Set verbose logging for each provider in your Terraform configuration
c. Set the environment variable TF_VAR_log=TRACE
d. Set the environment variable TF_LOG_PATH

Explanation: Setting **TF_LOG=TRACE** as an environment variable instructs Terraform to generate debug-level logs, allowing you to trace and identify from which paths Terraform is loading the providers referenced in your configuration.

237. How is terraform import run?
a. As a part of terraform init
b. As a part of terraform plan
c. As a part of terraform refresh
d. By an explicit call
e. All of the above

Explanation: Running **terraform import** requires an explicit call by the user to import existing infrastructure into Terraform's state management. It's not implicitly executed during **terraform init**, **plan**, or **refresh**.

238. You have a simple Terraform configuration containing one virtual machine (VM) in a cloud provider. You run terraform apply and the VM is created successfully. What will happen if you delete the VM using the cloud provider console, and run terraform apply again without changing any Terraform code?
a. Terraform will remove the VM from state file
b. Terraform will report an error
c. Terraform will not make any changes
d. Terraform will recreate the VM

Explanation: Terraform operates based on the state it manages. Since the state reflects that the VM is not present, running **terraform apply** again without changes in code won't trigger any action as it aligns with the current state.

239. Which of these options is the most secure place to store secrets foe connecting to a Terraform remote backend?
    a. Defined in Environment variables
    b. Inside the backend block within the Terraform configuration
    c. Defined in a connection configuration outside of Terraform
    d. None of above

Explanation: Storing secrets in a secure external configuration (like a secrets management system, not directly within Terraform configurations or environment variables) is a more secure approach to manage sensitive credentials.

240. Your DevOps team is currently using the local backend for your Terraform configuration. You would like to move to a remote backend to begin storing the state file in a central location. Which of the following backends would not work?
    a. Amazon S3
    b. Artifactory
    c. Git
    d. Terraform Cloud

Explanation: Git itself doesn't function as a backend for Terraform state. The other options—Amazon S3, Artifactory, and Terraform Cloud—are valid options for remote backends.

241. Which backend does the Terraform CLI use by default?
    a. Terraform Cloud
    b. Consul
    c. Remote
    d. Local

Explanation: By default, Terraform uses the local backend if a different backend is not explicitly specified in the configuration.

242. When you initialize Terraform, where does it cache modules from the public Terraform Module Registry?
    a. On disk in the /tmp directory
    b. In memory
    c. On disk in the .terraform sub-directory
    d. They are not cached

Explanation: Terraform caches modules from the public Terraform Module Registry on disk in the **.terraform** directory within the working directory.

243. You write a new Terraform configuration and immediately run terraform apply in the CLI using the local backend. Why will the apply fail?
    a. Terraform needs you to format your code according to best practices first
    b. Terraform needs to install the necessary plugins first
    c. The Terraform CLI needs you to log into Terraform cloud first
    d. Terraform requires you to manually run terraform plan first

Explanation: Running **terraform apply** immediately after writing a new configuration might fail because Terraform requires downloading and installing necessary plugins to execute the plan and apply actions.

244. What features stops multiple admins from changing the Terraform state at the same time?
    a. Version control
    b. Backend types
    c. Provider constraints
    d. State locking

Explanation: State locking is a feature in Terraform that prevents concurrent operations from multiple users by locking the state file. It ensures only one user can make changes at a time to avoid conflicts and data corruption.

245. A fellow developer on your team is asking for some help in refactoring their Terraform code. As part of their application's architecture, they are going to tear down an existing deployment managed by Terraform and deploy new. However, there is a server resource named aws_instance.ubuntu[1] they would like to keep to perform some additional analysis. What command should be used to tell Terraform to no longer manage the resource?
    a. terraform apply rm aws_instance.ubuntu[1]
    b. terraform state rm aws_instance.ubuntu[1]
    c. terraform plan rm aws_instance.ubuntu[1]
    d. terraform delete aws_instance.ubuntu[1]

Explanation: This command removes the resource instance from Terraform's state file, effectively untracking it, and no longer managing it during subsequent Terraform operations.

246. You need to constrain the GitHub provider to version 2.1 or greater. Which of the following should you put into the Terraform 0.12 configuration's provider block?
    a. version >= 2.1
    b. version ~> 2.1
    c. version = ƛ2.1 =>€ƛ€
    d. version = ƛ2.1 =<€ƛ€

Explanation: In the provider block, using the **version** argument with a constraint (**>=**) allows specifying a minimum version requirement for the provider. This ensures that Terraform uses the specified version or newer for the provider.

247. You just scaled your VM infrastructure and realized you set the count variable to the wrong value. You correct the value and save your change. What do you do next to make your infrastructure match your configuration?
    a. Run an apply and confirm the planned changes
    b. Inspect your Terraform state because you want to change it
    c. Reinitialize because your configuration has changed
    d. Inspect all Terraform outputs to make sure they are correct

Explanation: Running **terraform apply** will evaluate the changes in the configuration against the current state and execute the necessary actions to align the infrastructure with the corrected configuration.

248. Terraform provisioners that require authentication can use the _____ block.

a. **connection**
b. credentials
c. secrets
d. ssh

Explanation: The **connection** block in Terraform is used to configure authentication details (like SSH credentials, usernames, keys, etc.) required by provisioners to connect to resources during provisioning.

249. Terraform validate reports syntax check errors from which of the following scenarios?
a. Code contains tabs indentation instead of spaces
b. There is missing value for a variable
c. The state files does not match the current infrastructure
d. None of the above

Explanation: terraform validate checks the syntax and validity of Terraform configuration files but doesn't specifically catch scenarios like indentation errors, missing values for variables, or mismatches between state files and infrastructure.

250. Which of the following is allowed as a Terraform variable name?
a. count
b. name
c. source
d. version

Explanation: All the listed options (**count**, **name**, **source**, **version**) can be used as variable names in Terraform.

251. What type of block is used to construct a collection of nested configuration blocks?
a. for_each
b. repeated
c. nesting
d. dynamic

Explanation: The **for_each** block in Terraform is used to create multiple instances of nested configuration blocks based on a map or set of values, allowing the construction of collections of configurations.n #91*Topic 1*

252. If writing Terraform code that adheres to the Terraform style conventions, how would you properly indent each nesting level compared to the one above it?
a. With four spaces
b. With a tab
c. With three spaces
d. With two spaces

Explanation: Terraform conventionally recommends using four spaces to properly indent each level in nested blocks for better readability and consistency.

253. Which of the following is not an action performed by terraform init?
a. Create a sample main.tf file
b. Initialize a configured backend
c. Retrieve the source code for all referenced modules

d. Load required provider plugins

Explanation: Terraform init initializes the working directory, sets up the backend, retrieves provider plugins, and fetches module sources but doesn't create a sample main.tf file.

254. How can you trigger a run in a Terraform Cloud workspace that is connected to a Version Control System (VCS) repository?

a. Only Terraform Cloud organization owners can set workspace variables on VCS connected workspaces
b. Commit a change to the VCS working directory and branch that the Terraform Cloud workspace is connected to
c. Only members of a VCS organization can open a pull request against repositories that are connected to Terraform Cloud workspaces
d. Only Terraform Cloud organization owners can approve plans in VCS connected workspaces

Explanation: By committing a change to the connected VCS repository, Terraform Cloud detects the changes and triggers a run in the associated workspace.

255. You need to deploy resources into two different cloud regions in the same Terraform configuration. To do that, you declare multiple provider configurations as follows:

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

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

What meta-argument do you need to configure in a resource block to deploy the resource to the `us-west-2` AWS region?

a. alias = west
b. provider = west
c. provider = aws.west
d. alias = aws.west

Explanation: Using the alias meta-argument in a resource block, you can specify which provider configuration the resource should use. In this case, **alias** allows targeting the specific provider configuration defined for the **us-west-2** region.

256. You have declared an input variable called environment in your parent module. What must you do to pass the value to a child module in the configuration?

a. Add node_count = var.node_count
b. Declare the variable in a terraform.tfvars file
c. Declare a node_count input variable for child module
d. Nothing, child modules inherit variables of parent module

Explanation: To pass values from a parent module to a child module, you need to declare the corresponding input variables in the child module and assign the values in the parent module's invocation of the child module.#101*Topic 1*

257. Which option cannot be used to keep secrets out of Terraform configuration files?
    a. Environment Variables
    b. Mark the variable as sensitive
    c. A Terraform provider
    d. A -var flag

Explanation: While environment variables, marking variables as sensitive, and using **-var** flags can help keep secrets out of Terraform configuration files, Terraform providers themselves don't inherently facilitate keeping secrets out of configuration files.

258. Which of the following arguments are required when declaring a Terraform output?
    a. sensitive
    b. description
    c. default
    d. value

Explanation: While **description** provides context for the output, **value** is the actual output value returned by the Terraform configuration. **sensitive** and **default** are optional arguments for Terraform outputs.

259. Your risk management organization requires that new AWS S3 buckets must be private and encrypted at rest. How can Terraform Enterprise automatically and proactively enforce this security control?
    a. With a Sentinel policy, which runs before every apply
    b. By adding variables to each TFE workspace to ensure these settings are always enabled
    c. With an S3 module with proper settings for buckets
    d. Auditing cloud storage buckets with a vulnerability scanning tool

Explanation: Sentinel policies in Terraform Enterprise can enforce certain constraints, ensuring that security controls, like bucket privacy and encryption at rest, are always met before any infrastructure changes are applied.

260. What does terraform import allow you to do?
    a. Import a new Terraform module
    b. Use a state file to import infrastructure to the cloud
    c. Import provisioned infrastructure to your state file
    d. Import an existing state file to a new Terraform workspace

Explanation: terraform import helps bring existing, already provisioned infrastructure into Terraform's management by importing it into the Terraform state.

261. How would you reference the Volume IDs associated with the ebs_block_device blocks in this configuration?

```
resource "aws_instance" "example" {
  ami = "ami-abc123"
  instance_type - "t2.micro"

ebs_block_device {
  device_name = "sda2"
  volume_size = 16
}

ebs_block_device {
  device_name = "sda3"
  volume_size = 20
 }
}
```

a. aws_instance.example.ebs_block_device.[*].volume_id
b. aws_instance.example.ebs_block_device.volume_id
c. aws_instance.example.ebs_block_device[sda2,sda3].volume_id
d. aws_instance.example.ebs_block_device.*.volume_id

Explanation: This syntax references all the **volume_id** attributes associated with **ebs_block_device** blocks within the **aws_instance.example** resource.

262. What does state locking accomplish?
a. Copies the state file from memory to disk
b. Encrypts any credentials stored within the state file
c. Blocks Terraform commands from modifying the state file
d. Prevents accidental deletion of the state file

Explanation: State locking helps prevent concurrent modifications to the Terraform state, ensuring that only one process can modify the state at any given time.111*Topic 1*

263. You just upgraded the version of a provider in an existing Terraform project. What do you need to do to install the new provider?
a. Run terraform apply -upgrade
b. Run terraform init -upgrade
c. Run terraform refresh
d. Upgrade your version of Terraform.

Explanation: When you upgrade a provider version, you can use **terraform init -upgrade** to download and install the updated provider version.

264. When you use a remote backend that needs authentication, HashiCorp recommends that you:

a. Use partial configuration to load the authentication credentials outside of the Terraform code

b. Push your Terraform configuration to an encrypted git repository

c. Write the authentication credentials in the Terraform configuration files

d. Keep the Terraform configuration files in a secret store

Explanation: It's advisable to manage sensitive information (like credentials) externally and use partial configuration within Terraform to reference these values securely.

265. You have a simple Terraform configuration containing one virtual machine (VM) in a cloud provider. You run terraform apply and the VM is created successfully. What will happen if you terraform apply again immediately afterwards without changing any Terraform code?

a. Terraform will terminate and recreate the VM

b. Terraform will create another duplicate VM

c. Terraform will apply the VM to the state file

d. Nothing

Explanation: If no changes have been made to the Terraform code but you attempt to apply it again immediately, Terraform detects there are no changes and does not perform any actions.

266. A junior admin accidentally deleted some of your cloud instances. What does Terraform do when you run terraform apply?

a. Build a completely brand new set of infrastructure

b. Tear down the entire workspace infrastructure and rebuild it

c. Rebuild only the instances that were deleted

d. Stop and generate an error message about the missing instances

Explanation: Terraform compares the current state (defined in the configuration) with the actual state and only makes necessary changes.

267. You have created a main.tr Terraform configuration consisting of an application server, a database, and a load balancer. You ran terraform apply and all resources were created successfully. Now you realize that you do not actually need the load balancer so you run terraform destroy without any flags What will happen?

a. Terraform will destroy the application server because it is listed first in the code

b. Terraform will prompt you to confirm that you want to destroy all the infrastructure

c. Terraform will destroy the main.tf file

d. Terraform will prompt you to pick which resource you want to destroy

e. Terraform will immediately destroy all the infrastructure

Explanation: Running **terraform destroy** without flags will destroy all the resources declared in the configuration.

268. Which type of block fetches or computes information for use elsewhere in a Terraform configuration?

a. provider

b. resource

c. local

d. data

Explanation: The **data** block fetches or computes information to be used within the Terraform configuration, allowing access to information without creating or modifying resources.

269. You have just developed a new Terraform configuration for two virtual machines with a cloud provider. You would like to create the infrastructure for the first time. Which Terraform command should you run first?
    a. terraform apply
    b. terraform plan
    c. terraform show
    d. terraform init

Explanation: To initialize the configuration and the backend, **terraform init** should be executed first before any other Terraform command.

270. What does Terraform use .terraform.lock.hcl file for?
    a. Tracking provider dependencies
    b. There is no such file
    c. Preventing Terraform runs from occurring
    d. Storing references to workspaces which are locked

Explanation: This file helps track and lock provider versions and dependencies within a Terraform configuration.

271. You've used Terraform to deploy a virtual machine and a database. You want to replace this virtual machine instance with an identical one without affecting the database. What is the best way to achieve this using Terraform?
    a. Use the terraform state rm command to remove the VM from state file
    b. Use the terraform taint command targeting the VMs then run terraform plan and terraform apply
    c. Use the terraform apply command targeting the VM resources only
    d. Delete the Terraform VM resources from your Terraform code then run terraform plan and terraform apply

Explanation: Tainting the VM resource will cause Terraform to destroy and recreate it without affecting other resources.

272. How do you specify a module's version when publishing it to the public Terraform Module Registry?
    a. The module's configuration page on the Terraform Module Registry
    b. Terraform Module Registry does not support versioning modules
    c. The release tags in the associated repo
    d. The module's Terraform code

Explanation: By tagging releases in the associated repository, you can specify the version of a module when publishing it to the Terraform Module Registry.

273. To check if all code in a Terraform configuration with multiple modules is properly formatted without making changes, what command should be run?
    a. terraform fmt -check
    b. terraform fmt -write-false
    c. terraform fmt ز€"list -recursive
    d. terraform fmt -check -recursive

Explanation: This command checks if the Terraform configuration adheres to the formatting rules without applying any changes.

274. As a member of the operations team, you need to run a script on a virtual machine created by Terraform. Which provision is best to use in your Terraform code?

a. null-exlμ
b. local-exec
c. remote-exec
d. file

Explanation: **remote-exec** provisioner allows executing scripts on a remote machine, which fits the scenario of running a script on a VM.

275. You are using a networking module in your Terraform configuration with the name label my_network. In your main configuration you have the following code:

```
output: "net_id" {
  value = module.my_network.vnet_id
}
```

When you run terraform validate, you get the following error:

```
Error: Reference to undeclared output value

  on main.tf line 12, in output "net_id":
  12:    value = module.my_network.vnet_id
```

What must you do to successfully retrieve this value from your networking module?

a. Define the attribute vnet_id as a variable in the networking module
b. Change the referenced value to module.my_network.outputs.vnet_id
c. Define the attribute vnet_id as an output in the networking module
d. Change the referenced value to my_network.outputs.vnet_id

Explanation: Defining **vnet_id** as an output in the networking module allows it to be accessed from outside the module.

276. You are writing a child Terraform module which provisions an AWS instance. You want to make use of the IP address returned in the root configuration. You name the instance resource "main". Which of these is the correct way to define the output value using HCL2?

a.

```
output "instance_ip_addr" {
  value = "${aws_instance.main.private_ip}"
}
```

b.

```
output "instance_ip_addr" {
  return aws_instance.main.private_ip
}
```

Explanation: Has to be Answer A. no such definition as "return"uestion #131*Topic 1*

277. Which of the following statements about Terraform modules is not true?
   a. Modules must be publicly accessible
   b. Modules can be called multiple times
   c. Module is a container for one or more resources
   d. Modules can call other modules

Explanation: Modules don't have to be publicly accessible; they can reside locally or in private repositories.

278. Which Terraform collection type should you use to store key/value pairs?
   a. tuple
   b. set
   c. map
   d. list

Explanation: Maps are used to store key/value pairs, making them suitable for such data.

279. When do you need to explicitly execute terraform refresh?
   a. Before every terraform plan
   b. Before every terraform apply
   c. Before every terraform import
   d. None of the above

Explanation: **terraform refresh** is not required before **terraform plan**, **apply**, or **import** as it retrieves the current state of resources managed by Terraform.

280. What advantage does an operations team that uses infrastructure as code have?
   a. The ability to delete infrastructure
   b. The ability to update existing infrastructure
   c. The ability to reuse best practice configurations and settings
   d. The ability to autoscale a group of servers

Explanation: Infrastructure as code allows teams to reuse proven configurations, enhancing consistency and reliability.

281. You have modified your Terraform configuration to fix a typo in the Terraform ID of a resource from aws_security_group.http to aws_security_group.http

Original configuration:

```
resource "aws_security_group" "htp" {
  name = "http"
  ingress {
    from_port   = "80"
    to_port     = "80"
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Updated configuration:

```
resource "aws_security_group" "http" {
  name = "http"
  ingress {
    from_port   = "80"
    to_port     = "80"
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Which of the following commands would you run to update the ID in state without destroying the resource?

a. terraform mv aws_security_group.htp aws_security_group.http
b. terraform apply
c. terraform refresh

Explanation: terraform refresh reconciles the Terraform state with real-world resources without modifying the resources themselves.

282. You are creating a Terraform configuration which needs to make use of multiple providers, one for AWS and one for Datadog. Which of the following provider blocks would allow you to do this?

a.

```
provider {
 "aws" {
   profile = var.aws_profile
   region  = var.aws_region
 }

 "datadog" {
   api_key = var.datadog_api_key
   app_key = var.datadog_app_key
 }
}
```

b.

```
provider "aws" {
  profile = var.aws_profile
  region  = var.aws_region
}

provider "datadog" {
  api_key = var.datadog_api_key
  app_key = var.datadog_app_key
}
```

c.

```
terraform {
 provider "aws" {
   profile = var.aws_profile
   region  = var.aws_region
 }

provider "datadog" {
 api_key = var.datadog_api_key
   app_key = var.datadog_app_key
 }
}
```

Explanation: B is correct as you are configuring the provider not declaring it terraform { required_providers { aws = { source = "hashicorp/aws" version = "~> 4.0" } } }

283. Which of the following is not a way to trigger terraform destroy?
   a. Using the destroy command with auto-approve
   b. Running terraform destroy from the correct directory and then typing "yes" when prompted in the CLI
   c. Passing --destroy at the end of a plan request
   d. **Delete the state file and run terraform apply**

Explanation: While deleting the state file can lead to re-creating resources, it does not explicitly trigger **terraform destroy**.43*Topic 1*

284. Which of the following is not an advantage of using infrastructure as code operations?
   a. Self-service infrastructure deployment
   b. Troubleshoot via a Linux diff command
   c. Public cloud console configuration workflows
   d. Modify a count parameter to scale resources API driven workflows

Explanation: The advantages of infrastructure as code do not directly involve using public cloud console configuration workflows.

285. You're writing a Terraform configuration that needs to read input from a local file called id_rsa.pub.

Which built-in Terraform function can you use to import the file's contents as a string?

a. fileset("id_rsa.pub")
b. filebase64("id_rsa.pub")
c. templatefile("id_rsa.pub")
d. file("id_rsa.pub")

Explanation: The **file** function reads the content of a file and returns it as a string in Terraform. #

286. How does Terraform determine dependencies between resources?

a. Terraform automatically builds a resource graph based on resources, provisioners, special meta-parameters, and the state file, if present.
b. Terraform requires all dependencies between resources to be specified using the depends_on parameter
c. Terraform requires resources in a configuration to be listed in the order they will be created to determine dependencies
d. Terraform requires resource dependencies to be defined as modules and sourced in order

Explanation: Terraform constructs a resource graph based on dependencies declared and inferred from the configuration, using this to establish the execution order.

287. Once a new Terraform backend is configured with a Terraform code block, which command(s) is (are) used to migrate the state file?

a. terraform apply
b. terraform push
c. terraform destroy, then terraform apply
d. terraform init

Explanation: Once you've configured a new backend in Terraform, the **terraform init** command is used to initialize the backend and can be used for migrating the state file to the new backend.

288. What does this code do?

```
terraform {
  required_providers {
    aws = "~> 3.0"
  }
}
```

a. Requires any version of the AWS provider >= 3.0 and < 4.0
b. Requires any version of the AWS provider >= 3.0
c. Requires any version of the AWS provider after the 3.0 major release, like 4.1
d. Requires any version of the AWS provider > 3.0

Explanation: This version constraint specifies that any version of the AWS provider equal to or greater than 3.0 but less than 4.0 is acceptable for use in the configuration. It does not include version 4.0 but does include versions like 3.1, 3.2, etc., up to version 3.9.

MCQ with More than One Correct Option

1. You need to enable logging for Terraform and persist the logs to a specific file. What two environment variables can be set to enable logs and write them to a file? (select two)

a. `TF_ENABLE_LOG=true`

b. `TF_LOG=TRACE`

c. `TF_LOG_PATH="<file_path>"`

d. `TF_LOG_OUTPUT="<file_path>"`

Explanation

Terraform has detailed logs which can be enabled by setting the `TF_LOG` environment variable to any value. This will cause detailed logs to appear on stderr. 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.

To persist logged output you can set `TF_LOG_PATH` in order to force the log to always be appended to a specific file when logging is enabled. Note that even when `TF_LOG_PATH` is set, `TF_LOG` must be set in order for any logging to be enabled.

The wrong answers provided in this question are not valid environment variables that you can use with Terraform.

2. Which of the following are true statements regarding Terraform? (select three)

a. Terraform can orchestrate large-scale, multi-cloud infrastructure deployments

b. Terraform is cloud-agnostic

c. A single configuration file can use multiple providers

d. Terraform can manage dependencies within a single cloud, but not cross-cloud

Explanation

Terraform can indeed manage dependencies across multiple cloud providers. That is a huge benefit of using Terraform since it's cloud-agnostic, it doesn't care where the resources are deployed. It can still manage implicit or explicit dependencies between resources regardless of where they are deployed.

3. Which of the following are tasks that `terraform apply` can perform? (select three)

a. destroy infrastructure previously deployed with Terraform

b. update existing infrastructure with new configurations

c. provision new infrastructure

d. import existing infrastructure

Explanation

The `terraform apply` command executes the actions proposed in a Terraform plan. This works the same regardless of whether you have existing infrastructure deployed and changes are needed, or if you are just deploying your infrastructure for the first time. The `terraform apply` command can also destroy infrastructure by passing the `-destroy` flag as well.

Running a terraform apply cannot import infrastructure to pull under Terraform management. That's the job of the terraform import command

4. Which of the following Terraform versions does ***NOT*** support Sentinel? (select two)

a. Terraform Enterprise
b. Terraform OSS/CLI
c. Terraform Cloud (free)
d. Terraform Cloud for Business

Explanation

Terraform OSS does not support Sentinel, nor does the Free version of Terraform Cloud

5. What of the following are benefits of using Infrastructure as Code? (select three)

a. the ability to programmatically deploy infrastructure
b. the reduction of misconfigurations that could lead to security vulnerabilities and unplanned downtime
c. reducing vulnerabilities in your publicly-facing applications
d. your infrastructure configurations can be version controlled and stored in a code repository alongside the application code

Explanation

Reducing vulnerabilities in your publicly-facing applications is NOT a benefit of using IaC since IaC is geared towards deploying infrastructure and applications, but not determining whether your application is secure.

Terraform does not reduce vulnerabilities in your applications. You CAN pair Terraform with other tools that do this through a CI?CD pipeline or something like that, but Terraform will not do this natively.

6. Which of the features below is available in the free version of Terraform Cloud? (select three)

a. State Management
b. Private Module Registry
c. Remote Operations
d. Single Sign-On

Explanation

Single Sign-On is a feature of Terraform Enterprise and Terraform Cloud for Business. It is NOT available in Terraform Cloud (free tier)

7. Terraform relies on state in order to create and manage resources. Which of the following is true regarding the state file? (select four)

a. state should be modified by editing the file directly
b. remote state is required when more than one person wants to manage the infrastructure managed by Terraform
c. **it may contain sensitive data that you might not want others to view**
d. the state file is formatted using JSON
e. by default, state is stored in a file named terraform.tfstate in the current working directory

Explanation

In order to properly and correctly manage your infrastructure resources, Terraform stores the state of your managed infrastructure. Terraform uses this state on each execution to plan and

make changes to your infrastructure. This state must be stored and maintained on each execution so future operations can perform correctly.
During execution, Terraform will examine the state of the currently running infrastructure, determine what differences exist between the current state and the revised desired state, and indicate the necessary changes that must be applied. When approved to proceed, only the necessary changes will be applied, leaving existing, valid infrastructure untouched.
By default, Terraform will store the state in a JSON-formatted file named terraform.tfstate in the current working directory. The file should never be modified directly if state needs to be manually changed. You should use the `terraform state` command to modify state where possible.
The local state file is NOT encrypted (unless your local disk is encrypted or using something like Windows Bitlocker). This is why many organizations use a remote backend, which will store the state file in an encrypted store. This also allows multiple team members to work with the state file rather than storing it on an individual's laptop/desktop.

8. Which of the following are *collection* or *structural types* that can be used when declaring a variable in order to group values together? (select four)

a. object
b. bool
c. number
d. map
e. list
f. tuple
g. string

Explanation
As you continue to work with Terraform, you're going to need a way to organize and structure data. This data could be input variables that you are giving to Terraform, or it could be the result of resource creation, like having Terraform create a fleet of web servers or other resources. Either way, you'll find that data needs to be organized yet accessible so it is referenceable throughout your configuration. The Terraform language uses the following types for values:
* **string**: a sequence of Unicode characters representing some text, like "hello".
* **number**: a numeric value. The number type can represent both whole numbers like 15 and fractional values like 6.283185.
* **bool**: a boolean value, either true or false. bool values can be used in conditional logic.
* **list (or tuple)**: a sequence of values, like ["us-west-1a", "us-west-1c"]. Elements in a list or tuple are identified by consecutive whole numbers, starting with zero.
* **map (or object)**: a group of values identified by named labels, like {name = "Mabel", age = 52}. Maps are used to store key/value pairs.
Strings, numbers, and bools are sometimes called primitive types. Lists/tuples and maps/objects are sometimes called complex types, structural types, or collection types.

9. Which of the following options are ***not*** available in Terraform OSS/CLI and Terraform Cloud (free)? (select three)

a. Public Module Registry
b. Single Sign-On (SSO)
c. Sentinel

d. Workspaces
e. Audit Logging

Explanation

Single Sign-On requires Terraform Cloud for Business or Terraform Enterprise. It is NOT available in Terraform OSS or Terraform Cloud (free)
Sentinel is available in Terraform Cloud (Team & Governance), Terraform Enterprise, and Terraform Cloud for Business. It is NOT available in Terraform OSS or Terraform Cloud (free).
Audit Logging is available in Terraform Cloud (Team & Governance), Terraform Enterprise, and Terraform Cloud for Business. It is NOT available in Terraform OSS or Terraform Cloud (free).
Public Module Registry is available to users of any version of Terraform.
Workspaces are a feature of all versions of Terraform, both Terraform OSS/Cloud and all other paid versions.

10. When you add a new module to a configuration, Terraform must download it before it can be used. What two commands can be used to download and update modules? (select two)

a. `terraform get`
b. `terraform plan`
c. `terraform refresh`
d. `terraform init`

Explanation

The two Terraform commands used to download and update modules are:
`terraform init`: This command downloads and updates the required modules for the Terraform configuration. It also sets up the backend for state storage if specified in the configuration.
`terraform get`: This command is used to download and update modules for a Terraform configuration. It can be used to update specific modules by specifying the module name and version number, or it can be used to update all modules by simply running the command without any arguments.
It's important to note that **`terraform init`** is typically run automatically when running other Terraform commands, so you may not need to run **`terraform get`** separately. However, if you need to update specific modules, running **`terraform get`** can be useful.

11. Terraform Cloud is more powerful when you integrate it with your version control system (VCS) provider. Select all the supported VCS providers from the answers below. (select four)

a. CVS Version Control
b. GitHub Enterprise
c. Azure DevOps Server
d. Bitbucket Cloud
e. GitHub.com

Explanation

Terraform Cloud supports the following VCS providers as of February 2023:
- GitHub
- GitHub.com (OAuth)
- GitHub Enterprise

- GitLab.com
- GitLab EE and CE
- Bitbucket Cloud
- Bitbucket Server
- Azure DevOps Server
- Azure DevOps Services

12. Terraform is distributed as a single binary and available for many different platforms. Select all Operating Systems that Terraform is available for. (select five)

a. AIX
b. Linux
c. Solaris
d. **macOS**
e. FreeBSD
f. Windows

Explanation

Terraform is a cross-platform tool and can be installed on several operating systems, including:
Windows: Terraform can be installed on **Windows** operating systems using the Windows installer.
macOS: Terraform can be installed on macOS using the macOS installer or using Homebrew.
Linux: Terraform can be installed on **Linux** operating systems using the binary distribution or through package management systems, such as apt or yum.
Unix: Terraform can be installed on Unix-like operating systems using the binary distribution.
There is no Terraform binary for AIX. Terraform is available for macOS, FreeBSD, OpenBSD, Linux, Solaris, and Windows.

13. Which are some of the benefits of using Infrastructure as Code in an organization? (select three)

a. IaC uses a human-readable configuration language to help you write infrastructure code quickly
b. IaC is written as an imperative approach, where specific commands need to be executed in the correct order
c. IaC code can be used to manage infrastructure on multiple cloud platforms
d. IaC allows you to commit your configurations to version control to safely collaborate on infrastructure

Explanation

Infrastructure as Code has many benefits. For starters, IaC allows you to create a blueprint of your data center as code that can be ***versioned***, ***shared***, and ***reused***. Because IaC is code, it can (and should) be stored and managed in a **code repository**, such as GitHub, GitLab, or Bitbucket. Changes can be proposed or submitted via Pull Requests (PRs), which can help ensure a proper workflow, enable an approval process, and follow a typical development lifecycle.
One of the primary reasons that Terraform (or other IaC tools) are becoming more popular is because they are mostly ***platform agnostic***. You can use Terraform to provision and manage resources on various platforms, SaaS products, and even local infrastructure.

IaC is generally ***easy to read*** (and develop). Terraform is written in HashiCorp Configuration Language (HCL), while others may use YAML or solution-specific languages (like Microsoft ARM). But generally, IaC code is easy to read and understand

IaC is written using a ***declarative*** approach (***not imperative***), which allows users to simply focus on what the eventual target configuration should be, and the tool manages the process of how that happens. This often speeds things up because resources can be created/managed in parallel when there aren't any implicit or explicit dependencies.

14. Anyone can publish and share modules on the Terraform Public Module Registry, and meeting the requirements for publishing a module is extremely easy. What are some of the requirements that must be met in order to publish a module on the Terraform Public Module Registry? (select three)

a. The module must be PCI/HIPPA compliant.
b. The module must be on GitHub and must be a public repo.
c. Module repositories must use this three-part name format, *terraform-<PROVIDER>-<NAME>*.
d. The registry uses tags to identify module versions. Release tag names must be for the format *x.y.z*, and can optionally be prefixed with a v .

Explanation

The list below contains all the requirements for publishing a module. Meeting the requirements for publishing a module is extremely easy. The list may appear long only to ensure we're detailed, but adhering to the requirements should happen naturally.

GitHub. The module must be on GitHub and must be a public repo. This is only a requirement for the public registry. If you're using a private registry, you may ignore this requirement

Named terraform-<PROVIDER>-<NAME>. Module repositories must use this three-part name format, where <NAME> reflects the type of infrastructure the module manages and <PROVIDER> is the main provider where it creates that infrastructure.

The <NAME> segment can contain additional hyphens. Examples: terraform-google-vault or terraform-aws-ec2-instance.

Repository description. The GitHub repository description is used to populate the short description of the module. This should be a simple one-sentence description of the module.

Standard module structure. The module must adhere to the standard module structure. This allows the registry to inspect your module and generate documentation, track resource usage, parse submodules and examples, and more.

x.y.z tags for releases. The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2. To publish a module initially, at least one release tag must be present. Tags that don't look like version numbers are ignored.

15. Provider dependencies are created in several different ways. Select the valid provider dependencies from the following list: (select three)

a. Explicit use of a provider block in configuration, optionally including a version constraint.
b. Existence of any provider plugins found locally in the working directory.

c. **Existence of any resource instance belonging to a particular provider in the current *state*.**
d. **Use of any resource belonging to a particular provider in a resource or data block in the configuration.**

Explanation

The existence of a provider plugin found locally in the working directory does not itself create a provider dependency. The plugin can exist without any reference to it in the Terraform configuration.

16. When using constraint expressions to signify a version of a provider, which of the following are valid provider versions that satisfy the expression found in the following code snippet: (select two)

```
terraform {
  required_providers {
    aws = "~> 1.2.0"
  }
}
```

a. **Terraform 1.2.3**
b. Terraform 1.3.0
c. **Terraform 1.2.9**
d. Terraform 1.3.1

Explanation

In a **required_version** parameter in Terraform, the tilde (**~**) symbol followed by the greater than symbol (**>**) specifies a "compatible with" version constraint.

For example, if your Terraform configuration specifies **required_version = "~> 1.12.0"**, Terraform will accept any version of Terraform 1.12 that is greater than or equal to version 1.12.0 and less than 1.13.0. In other words, Terraform will accept any version of Terraform 1.12 that is considered compatible with version 1.12.0.

This type of version constraint is useful when your Terraform configuration uses features that are available in a specific version of Terraform, but you also want to allow for later versions of Terraform that are compatible with that version. This allows you to specify a minimum required version of Terraform, while also allowing for later versions that are compatible with your configuration.

Note that version constraints specified using the tilde and greater than symbols are specific to Terraform, and they are not a standard part of the Semantic Versioning specification.

17. What are some of the features of Terraform state? (select three)

a. **mapping configuration to real-world resources**
b. **determining the correct order to destroy resources**
c. **increased performance**
d. inspection of cloud resources

Explanation

Terraform state is a necessary requirement for Terraform to function. It is often asked if it is possible for Terraform to work without state, or for Terraform to not use state and just inspect real world resources on every run. Terraform state is required for: and the benefits it provides.

- Mapping to the Real World
- Metadata
- Performance
- Syncing

18. What are some problems with how infrastructure was traditionally managed before Infrastructure as Code? (select three)

a. Traditional deployment methods are not able to meet the demands of the modern business where resources tend to live days to weeks, rather than months to years

b. Requests for infrastructure or hardware often required a ticket, increasing the time required to deploy applications

c. Traditionally managed infrastructure can't keep up with cyclic or elastic applications

d. Pointing and clicking in a management console is a scalable approach and reduces human error as businesses are moving to a multi-cloud deployment model

Explanation

Traditionally, infrastructure was managed using manual processes and user interfaces, which could lead to several problems, including:

Configuration drift: With manual configuration, it can be difficult to ensure that all infrastructure components are consistently configured. Over time, differences in configuration can accumulate, leading to configuration drift, where systems in the same environment are no longer identical.

Lack of standardization: Manual configuration can also result in inconsistencies across environments, which can make it difficult to manage and troubleshoot infrastructure. For example, different environments may have different versions of software or different security settings, making it hard to replicate issues or ensure consistent behavior.

Slow provisioning: Provisioning infrastructure manually can be time-consuming, especially for complex configurations or when setting up multiple resources. This can lead to delays in development and deployment, as teams may need to wait for infrastructure to be set up before they can begin work.

Human error: Manual provisioning and configuration is prone to human error, which can lead to security vulnerabilities, performance issues, or downtime. For example, a misconfigured firewall rule could leave systems open to attack, or a typo in a configuration file could cause a system to crash.

Difficulty in documentation: With manual configuration, it can be challenging to keep documentation up to date and accurate. This can make it hard for teams to understand how infrastructure is configured, what changes have been made, and how to troubleshoot issues.

Overall, these problems can make it difficult to manage infrastructure at scale and can lead to increased costs, reduced agility, and increased risk of errors and downtime. Infrastructure as Code helps to address many of these issues by providing a standardized, repeatable, and automated way to manage infrastructure resources.

19. What advantages does Terraform offer over using a provider's native tooling for deploying resources in multi-cloud environments? (select three)

a. Terraform simplifies management and orchestration, helping operators build large-scale, multi-cloud infrastructure

b. **Terraform can help businesses deploy applications on multiple clouds and on-premises infrastructure**
c. **Terraform can manage cross-cloud dependencies**
d. Terraform is not cloud-agnostic and can only be used to deploy resources across a single public cloud at a time

Explanation

Terraform offers several advantages over using a provider's native tooling for deploying resources in multi-cloud environments, including:

Multi-cloud support: Terraform provides a consistent interface for managing infrastructure resources across multiple cloud providers, including AWS, Azure, Google Cloud, and more. This allows organizations to use a single tool for managing their entire multi-cloud environment rather than needing to learn and use multiple provider-specific tools.

Standardized configuration: Terraform uses a declarative configuration language to define infrastructure resources, which can be used to define resources across multiple providers in a standardized way. This provides consistency and reduces the need for provider-specific knowledge.

Idempotent execution: Terraform only makes changes to infrastructure resources if the desired state differs from the current state, which means that it can be safely run multiple times without causing unintended changes. This reduces the risk of configuration drift and ensures that infrastructure remains consistent over time.

Plan preview: Terraform can generate a plan that shows the changes it will make to infrastructure resources before it applies them. This provides visibility into changes and helps to reduce the risk of unintended consequences.

Collaboration and version control: Terraform configurations can be stored in version control systems, allowing multiple team members to collaborate on infrastructure changes. This provides a centralized location for documentation, change history, and issue tracking, making it easier to manage infrastructure changes over time.

Overall, using Terraform for deploying resources in multi-cloud environments can provide a consistent, standardized, and efficient approach to managing infrastructure across multiple providers.

20. Which of the following connection types are supported by the `remote-exec` provisioner? (select two)

a. rdp
b. ssh
c. winrm
d. smb

Explanation

The **`remote-exec`** provisioner in Terraform is used to execute commands on a resource after it has been created over an **SSH** or **WinRM** connection. The supported connection types for **`remote-exec`** depend on the type of resource being provisioned and the underlying operating system.

For Linux-based resources, the **`remote-exec`** provisioner supports the following connection types:

SSH (Secure Shell) over TCP (Transmission Control Protocol)
For Windows-based resources, the **remote-exec** provisioner supports the following connection types:
WinRM (Windows Remote Management) over HTTP (Hypertext Transfer Protocol) or HTTPS (HTTP Secure)
Note that both **SSH** and **WinRM** connections can be configured to use specific usernames and passwords or SSH keys, depending on the resource being provisioned and the security requirements of the environment.
It is worth noting that while **remote-exec** provisioner can be useful in certain situations, it should generally be used as a last resort due to the potential risks and complexities involved in executing remote commands on resources. Whenever possible, it is recommended to use Terraform's native resource types to manage your infrastructure, rather than relying on external scripts or tools.

21. HashiCorp offers multiple versions of Terraform, including Terraform open-source, Terraform Cloud, and Terraform Enterprise. Which of the following Terraform feature is available in the **Enterprise** and T**erraform Cloud for Business** editions? (select four)

a. Private Network Connectivity
b. Self-Managed Installation
c. Private Module Registry
d. Audit Logs
e. SAML/SSO

Explanation
While a ton of features are available to open source and Cloud users, there are still a few features that are part of the Enterprise offering, which is geared toward enterprise requirements. With the introduction of Terraform Cloud for Business, almost all features are now available for a hosted Terraform deployment. To see what specific features are part of Terraform Cloud and Terraform Enterprise, check out this link.
Information about Answers:
- Private Network Connectivity is available for Terraform Enterprise since it is installed locally in your data center or cloud environment. However, it is also available in Terraform Cloud for Business since you can use a self-hosted agent to communicate with local/private hosts in your environment.
- SAML/SSO is available in BOTH Terraform Enterprise and Terraform Cloud for Business.
- Private Module Registry is available in every version of Terraform except for Open-Source.
- Audit Logging is available in Terraform Enterprise AND Terraform Cloud for Business.
- Self-Managed Installation is available for Terraform Enterprise; technically, you manage your installation for Terraform open source as well. However, since it's a SaaS offering, you can't "self-manage" a Terraform Cloud deployment. This excludes the fact that you can deploy cloud agents in your environment, but that's an extension of TFC and not TFC itself.

22. From the answers below, select the advantages of using Infrastructure as Code. (select four)

a. Safely test modifications using a "dry run" before applying any actual changes
b. Easily change and update existing infrastructure
c. Provide reusable modules for easy sharing and collaboration

d. Provide a codified workflow to develop customer-facing applications
e. Easily integrate with application workflows (GitLab Actions, Azure DevOps, CI/CD tools)

Explanation

Infrastructure as Code is **not** used to develop applications, but it can be used to help deploy or provision those applications to a public cloud provider or on-premises infrastructure.
All of the others are benefits to using Infrastructure as Code over the traditional way of managing infrastructure, regardless if it's public cloud or on-premises.

23. Using multi-cloud and provider-agnostic tools provides which of the following benefits? (select two)

a. slower provisioning speed allows the operations team to catch mistakes before they are applied
b. can be used across major cloud providers and VM hypervisors
c. operations teams only need to learn and manage a single tool to manage infrastructure, regardless of where the infrastructure is deployed
d. increased risk due to all infrastructure relying on a single tool for management

Explanation

Using a tool like Terraform can be advantageous for organizations deploying workloads across multiple public and private cloud environments. Operations teams only need to learn a single tool, a single language, and can use the same tooling to enable a DevOps-like experience and workflows.

24. In regards to Terraform state file, select all the statements below which are correct: (select four)

a. the state file is always encrypted at rest
b. using the `mask` feature, you can instruct Terraform to mask sensitive data in the state file
c. when using local state, the state file is stored in plain-text
d. the Terraform state can contain sensitive data, therefore the state file should be protected from unauthorized access
e. Terraform Cloud always encrypts state at rest
f. storing state remotely can provide better security

Explanation

Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive." The state contains resource IDs and all resource attributes. For resources such as databases, this may contain initial passwords.
When using local state, state is stored in plain-text JSON files.
If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.
Storing Terraform state remotely can provide better security. As of Terraform 0.9, Terraform does not persist state to the local disk when remote state is in use, and some backends can be configured to encrypt the state data at rest.

25. What is the purpose of using the `local-exec` provisioner? (select two)

a. ensures that the resource is only executed in the local infrastructure where Terraform is deployed

b. executes a command on the resource to invoke an update to the Terraform state

c. to execute one or more commands on the machine running Terraform

d. to invoke a local executable

Explanation

In Terraform, the `local-exec` provisioner is used to execute a command on the machine running Terraform, rather than on a remote resource.

The `local-exec` provisioner is often used to perform actions that cannot be accomplished using Terraform's built-in resource types or to execute local scripts or commands to perform additional setup or configuration after creating infrastructure resources.

26. When configuring a remote backend in Terraform, it might be a good idea to purposely omit some of the required arguments to ensure secrets and other relevant data are not inadvertently shared with others. What are the ways the remaining configuration can be added to Terraform so it can initialize and communicate with the backend? (select three)

a. interactively on the command line

b. directly querying HashiCorp Vault for the secrets

c. use the -backend-config=PATH to specify a separate config file

d. command-line key/value pairs

Explanation

You do not need to specify every required argument in the backend configuration. Omitting certain arguments may be desirable to avoid storing secrets, such as access keys, within the main configuration. When some or all of the arguments are omitted, we call this a *partial configuration*.

With a partial configuration, the remaining configuration arguments must be provided as part of the initialization process. There are several ways to supply the remaining arguments:

Interactively: Terraform will interactively ask you for the required values unless interactive input is disabled. Terraform will not prompt for optional values.

File: A configuration file may be specified via the `init` command line. To specify a file, use the `-backend-config=PATH` option when running `terraform init`. If the file contains secrets it may be kept in a secure data store, such as Vault, in which case it must be downloaded to the local disk before running Terraform.

Command-line key/value pairs: Key/value pairs can be specified via the `init` command line. Note that many shells retain command-line flags in a history file, so this isn't recommended for secrets. To specify a single key/value pair, use the `-backend-config="KEY=VALUE"` option when running `terraform init`.

27. What are the benefits of using Infrastructure as Code? (select five)

a. Infrastructure as Code is easily repeatable, allowing the user to reuse code to deploy similar, yet different resources

b. Infrastructure as Code gives the user the ability to recreate an application's infrastructure for disaster recovery scenarios

c. Infrastructure as Code is relatively simple to learn and write, regardless of a user's prior experience with developing code

d. Infrastructure as Code easily replaces development languages such as Go and .Net for application development

e. **Infrastructure as Code provides configuration consistency and standardization among deployments**
f. **Infrastructure as Code allows a user to turn a manual task into a simple, automated deployment**

Explanation

Infrastructure as Code (IaC) refers to the practice of managing and provisioning infrastructure resources through code, rather than manual processes or user interfaces. Some of the benefits of using IaC include:

Consistency and repeatability: IaC allows for the creation of infrastructure in a consistent and repeatable way. This ensures that the same infrastructure can be deployed across multiple environments (e.g. development, testing, production) with minimal differences, reducing the risk of issues due to configuration drift or environment-specific issues.

Speed and agility: IaC allows for rapid provisioning and scaling of infrastructure resources, reducing the time it takes to set up and modify infrastructure. This enables teams to quickly respond to changing business needs or shifting workloads, without the delays associated with manual provisioning processes.

Version control: IaC code can be stored in version control systems, allowing teams to track changes over time and revert to previous versions if necessary. This provides a history of infrastructure changes and ensures that teams are always working with the most up-to-date version of the infrastructure code.

Collaboration and documentation: IaC code can be shared and collaborated on, allowing teams to work together to design and maintain infrastructure resources. It also provides a single source of truth for documentation, making it easier to understand how the infrastructure is configured and how it has changed over time.

Cost savings: IaC can help reduce infrastructure costs by allowing teams to more effectively manage resources, optimize usage, and avoid over-provisioning. It can also reduce the need for manual intervention, which can save time and reduce the risk of errors.

Overall, using IaC can help organizations achieve greater consistency, speed, agility, collaboration, and cost savings in their infrastructure management practices.

28. Frank has a file named `main.tf` which is shown below. Which of the following statements are true about this code? (select two)

```
module "servers" {
  source = "./app-cluster"
   servers = 5
}
```

a. **`app-cluster` is the child module**
b. **`main.tf` is the calling module**
c. `main.tf` is the child module
d. `app-cluster` is the calling module

Explanation

To *call* a module means to include the contents of that module into the configuration with specific values for its input variables. Modules are called from within other modules

using `module` blocks. A module that includes a `module` block like this is the *calling module* of the child module.
The label immediately after the `module` keyword is a local name, which the calling module can use to refer to this instance of the module.

29. Which of the following actions are performed during a `terraform init`? (select three)
    a. downloads the required modules referenced in the configuration
    b. initializes the backend configuration
    c. provisions the declared resources in your configuration
    d. downloads the providers/plugins required to execute the configuration

Explanation
The `terraform init` command is used to initialize a working directory containing Terraform configuration files. This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control. It is safe to run this command multiple times.

30. Which of the following Terraform files should be ignored by Git when committing code to a repo? (select two)
    a. terraform.tfvars
    b. terraform.tfstate
    c. outputs.tf
    d. variables.tf

Explanation
When using Terraform with Git, it is generally recommended to ignore certain files in order to avoid committing sensitive or unnecessary information to your repository. The specific files that should be ignored may vary depending on your project and configuration, but as a general rule, you should ignore the following files:
.terraform directory: This directory contains local Terraform state files, which should not be committed to the repository.
terraform.tfstate and **terraform.tfstate.backup**: These files contain the current state of your infrastructure, and should not be committed to the repository.
.tfvars files: These files may contain sensitive information, such as passwords or API keys, and should be kept out of version control. Instead, you can use environment variables or other secure methods to pass this information to Terraform.
***.tfplan** files: These files contain the plan generated by Terraform when applying changes to your infrastructure, and may include sensitive information such as resource IDs. They should not be committed to the repository.
To ignore these files in Git, you can add them to your **.gitignore** file.

31. *Published modules* via the Terraform Registry provide which of the following benefits? (select four)
    a. allow browsing version histories
    b. support from any code repo
    c. support versioning
    d. show examples and READMEs

e. **automatically generated documentation**

Explanation

Public modules are managed via ***Git and GitHub***. Publishing a module takes only a few minutes. Once a module is published, you can release a new version of a module by simply pushing a properly formed Git tag. The module must be on GitHub and must be a public repo. This is only a requirement for the public registry. If you're using a private registry, you may ignore this requirement.

The key here is that HashiCorp uses **GitHub** for published modules.

32. Kristen is using modules to provision an Azure environment for a new application. She is using the following code and specifying a version of her virtual machine module to ensure she's calling the correct module. Which of the following options provides support for the versioning of a module? (select two)

```
module "compute" {
  source  = "Azure/compute/azurerm"
  version = "5.1.0"
   resource_group_name = "production_web"
  vnet_subnet_id      = azurerm_subnet.aks-default.id
}
```

a. **public module registry**
b. **private module registry**
c. modules stored in GitLab
d. local file paths

Explanation

Version constraints are supported only for modules installed from a module registry, such as the public Terraform Registry or Terraform Cloud's private module registry. Other module sources can provide their own versioning mechanisms within the source string itself, or might not support versions at all. In particular, modules sourced from local file paths do not support `version`; since they're loaded from the same source repository, they always share the same version as their caller.

33. HashiCorp offers multiple versions of Terraform to meet the needs of individuals to large enterprises. Which of the following offerings provides access to a private module registry? (select three)

a. **Terraform Cloud - Business**
b. Terraform Open-Source
c. **Terraform Cloud - Free**
d. **Terraform Cloud - Team & Governance**

Explanation

The Private Module Registry is available in all versions of Terraform except for Open Source.

	OSS	Cloud			Self-Hosted
Infrastructure as Code		FREE	TEAM & GOVERNANCE	BUSINESS	ENTERPRISE
Infrastructure as Code (HCL)	✓	✓	✓	✓	✓
Workspaces	✓	✓	✓	✓	✓
Variables	✓	✓	✓	✓	✓
Runs (separate plan and apply)	✓	✓	✓	✓	✓
Resource Graph	✓	✓	✓	✓	✓
Providers	✓	✓	✓	✓	✓
Modules	✓	✓	✓	✓	✓
Public Module Registry	✓	✓	✓	✓	✓
Collaborative Infrastructure as Code		FREE	TEAM & GOVERNANCE	BUSINESS	ENTERPRISE
Remote State		✓	✓	✓	✓
VCS Connection		✓	✓	✓	✓
Workspace Management		✓	✓	✓	✓
Secure Variable Storage		✓	✓	✓	✓
Remote Runs		✓	✓	✓	✓
Private Module Registry		✓	✓	✓	✓
Team Management & Governance		FREE	TEAM & GOVERNANCE	BUSINESS	ENTERPRISE
Team Management			✓	✓	✓
Sentinel Policy as Code Management			✓	✓	✓
Cost Estimation			✓	✓	✓
Advanced Security, Compliance, and Governance		FREE	TEAM & GOVERNANCE	BUSINESS	ENTERPRISE
Single Sign On (SSO)				✓	✓
Audit Logging				✓	✓
Self-Hosted Agents				✓	
Self-Service Infrastructure		FREE	TEAM & GOVERNANCE	BUSINESS	ENTERPRISE
Configuration Designer				✓	✓
ServiceNow Integration				✓	✓
Performance Operations		FREE	TEAM & GOVERNANCE	BUSINESS	ENTERPRISE
Concurrent Runs		Up to 1	Up to 2	✓	✓
Operations	Local CLI	Cloud	Cloud	Cloud	Private
Support		FREE	TEAM & GOVERNANCE	BUSINESS	ENTERPRISE
Community	✓	✓			
Bronze			✓	✓	✓
Silver				✓	✓
Gold				✓	✓

34. Aaron is new to Terraform and has a single configuration file that is ready to be deployed. Which of the following can be true about this configuration file? (select three)

a. Aaron's configuration file can deploy applications in both AWS and GCP

b. the configuration file can deploy both QA and Staging infrastructure for applications

c. the state can be disabled when deploying to multiple clouds to prevent sensitive data from being shared across cloud platforms

d. the state file can be stored in Azure but provision applications in AWS

Explanation

There are a ton of benefits of deploying with Terraform and the solution is very capable of managing deployments across multiple clouds. However, state is still required and cannot be disabled.

35. Terraform Cloud provides organizations with many features not available to those running Terraform open-source to deploy infrastructure. Select the ADDITIONAL features that organizations can take advantage of by moving to Terraform Cloud. (select three)

a. public module registry
b. private module registry
c. providers
d. remote runs
e. VCS connection

Explanation

Terraform Cloud offers many features, even in the free version, that organizations can quickly take advantage of. This is the best table that compares the features available in Terraform OSS vs. Terraform Cloud and Terraform Enterprise.

36. What happens when you apply a Terraform configuration using `terraform apply`? (select two)

a. Terraform makes infrastructure changes defined in your configuration.
b. Terraform recreates all the infrastructure defined in the configuration file
c. Terraform updates the state file with configuration changes made during the execution
d. Terraform downloads any required plugins
e. Terraform formats your configuration to the standard canonical format and style

Explanation

The `terraform apply` command is used to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a `terraform plan` execution plan.

37. Both Terraform CLI and Terraform Cloud offer a feature called "workspaces". Which of the following statements are true regarding workspaces? (select three)

a. Terraform Cloud manages infrastructure collections with a workspace whereas CLI manages collections of infrastructure resources with a persistent working directory
b. Each CLI workspace coincides with a different VCS repo
c. Run history is logged in a file underneath the working directory of a CLI workspace
d. CLI workspaces are alternative state files in the same working directory
e. Terraform Cloud maintains the state version and run history for each workspace

Explanation

Workspaces are similar concepts in all versions of Terraform, although they behave differently depending on the platform they are being used on.

38. When using a Terraform provider, it's common that Terraform needs credentials to access the API for the underlying platform, such as VMware, AWS, or Google Cloud. While there are many ways to accomplish this, what are three options that you can provide these credentials? (select three)

a. using a remote backend
b. directly in the provider block by hardcoding or using a variable
c. use environment variables
d. integrated services, such as AWS IAM or Azure Managed Service Identity

Explanation

You can use methods such as static credentials, environment variables, share credentials/configuration file, or other methods. For example, the AWS provider can use many different options as seen here:

- You can directly specify credentials within the Terraform configuration using variables or hardcoded values.
- Terraform can also pick up credentials from environment variables, providing a flexible way to handle authentication without hardcoding them into configuration files.
- Leveraging platform-specific integrated identity and access management services like AWS IAM or Azure Managed Service Identity allows Terraform to access credentials securely without explicitly storing them within the configuration.

Each provider is different, and you should check the documentation to see what is supported for each one you want to use.

39. Which of the following are the benefits of using *modules* in Terraform? (select three)

a. allows modules to be stored anywhere accessible by Terraform
b. supports versioning to maintain compatibility
c. supports modules stored locally or remotely
d. enables code reuse

Explanation

All of these are examples of the benefits of using Terraform modules except where they can be stored. Modules can only be supported in certain sources:

- Modules can be versioned, enabling better management and tracking of changes to ensure compatibility with different configurations.
- Modules can be stored either locally or in remote repositories, offering flexibility in organizing and accessing modular components.
- Modules facilitate code reuse, allowing you to define and encapsulate reusable components that can be used across different Terraform configurations.

40. Infrastructure as Code (IaC) provides many benefits to help organizations deploy application infrastructure much faster than clicking around in the console. What are the additional benefits of IaC? (select three)

a. creates a blueprint of your data center
b. eliminates parallelism
c. allows infrastructure to be versioned
d. can always be used to deploy the latest features and services
e. code can easily be shared and reused

Explanation

Infrastructure is described using a high-level configuration syntax. This allows a blueprint of your datacenter to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.

Infrastructure as Code almost always uses parallelism to deploy resources faster. And depending on the solution being used, it doesn't always have access to the latest features and services available on cloud platforms or other solutions.

41. What happens if multiple users attempt to run a `terraform apply` simultaneously when using a remote backend? (select two)

a. the Terraform apply will work for both users
b. if the backend does not support locking, the state file could become corrupted
c. if the backend supports locking, the first *terraform apply* will lock the file for changes, preventing the second user from running the *apply*
d. both users will get an error

Explanation

If the state is configured for remote state, the backend selected will determine what happens. If the backend supports locking, the file will be locked for the first user, and that user's configuration will be applied. The second user's `terraform apply` will return an error that the state is locked.

If the remote backend does not support locking, the state file could become corrupted, since multiple users are trying to make changes at the same time.

42. Which of the following Terraform CLI commands are valid? (select five)

a. `$ terraform initialize`
b. `$ terraform fmt`
c. `$ terraform login`
d. `$ terraform show`
e. `$ terraform workspace select`
f. `$ terraform delete`
g. `terraform apply -refresh-only`

Explanation

`terraform delete` and `terraform initialize` are not valid Terraform CLI commands.

Correct Answers:

The `terraform apply -refresh-only` command creates a plan whose goal is only to update the Terraform state and any root module output values to match changes made to remote objects outside of Terraform.

he `terraform fmt` command is used to rewrite Terraform configuration files to a canonical format and style.

The `terraform workspace select` command is used to choose a different workspace to use for further operations.

The `terraform show` command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

The `terraform login` command can be used to automatically obtain and save an API token for Terraform Cloud, Terraform Enterprise, or any other host that offers Terraform services.

43. There are an endless number of benefits of using Terraform within your organization. Which of the following are true statements regarding Terraform. (select three)

a. A single Terraform configuration file can be used to manage multiple providers
b. Terraform can simplify both management and orchestration of deploying large-scale, multi-cloud infrastructure
c. Terraform can manage dependencies within a single cloud, but not cross-cloud
d. Terraform is cloud-agnostic but requires a specific provider for the cloud platform

Explanation

All of the answers are benefits to using Terraform, except `Terraform can manage dependencies within a single cloud, but not cross-cloud`. Terraform isn't limited to ***only*** managing dependencies for a single cloud, it can manage dependencies across multiple cloud providers as well.

44. What CLI commands will completely tear down and delete all resources that Terraform is currently managing? (select two)

a. `terraform apply -destroy`
b. `terraform plan -destroy`
c. `terraform destroy`
d. `terraform apply -delete`

Explanation

The `terraform destroy` command is a convenient way to destroy all remote objects managed by a particular Terraform configuration.
While you will typically not want to destroy long-lived objects in a production environment, Terraform is sometimes used to manage ephemeral infrastructure for development purposes, in which case you can use `terraform destroy` to conveniently clean up all of those temporary objects once you are finished with your work.
This command is just a convenience alias for the following command:
`terraform apply -destroy`
For that reason, this command accepts most of the options that `terraform apply` accepts, although it does not accept a plan file argument and forces the selection of the "destroy" planning mode.
While `terraform plan -destroy` is a valid command, it only creates a speculative destroy plan to see what the effect of destroying would be `terraform apply -delete` is not a valid Terraform command

45. You are worried about unauthorized access to the Terraform state file since it might contain sensitive information. What are some ways you can protect the state file? (select two)

a. replicate the state file to an encrypted storage device
b. store in a remote backend that encrypts state at rest
c. enable native encryption in Terraform as configured in the `terraform` block
d. use the S3 backend using the `encrypt` option to ensure state is encrypted

Explanation

If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.

Storing state remotely can provide better security. As of Terraform 0.9, Terraform does not persist state to the local disk when remote state is in use, and some backends can be configured to encrypt the state data at rest.
Terraform Cloud always encrypts state at rest and protects it with TLS in transit. Terraform Cloud also knows the identity of the user requesting state and maintains a history of state changes. This can be used to control access and track activity. Terraform Enterprise also supports detailed audit logging.
The S3 backend supports encryption at rest when the encrypt option is enabled. IAM policies and logging can be used to identify any invalid access. Requests for the state go over a TLS connection.
Replication? replicating the state file to another location won't prevent the original file from being accessed.
Encryption? As of today, Terraform doesn't support any type of native encryption capability when writing and managing state.

46. You want to use Terraform to deploy resources across your on-premises infrastructure and a public cloud provider. However, your internal security policies require that you have full control over both the operating system and deployment of Terraform binaries. What versions of Terraform can you use for this? (select two)

a. Terraform Cloud for Business
b. Terraform OSS/CLI
c. Terraform Cloud (free)
d. Terraform Enterprise

Explanation
Terraform OSS and Terraform Enterprise are versions of Terraform that can be installed locally on your own servers, therefore giving you the ability to manage both the Terraform binary and the underlying operating system where Terraform runs.
Although Terraform Cloud for Business does offer Cloud Agents that could be used to provision resources on your local infrastructure on-premises, it is a hosted solution and you would NOT have full control over the operating system that runs the Terraform platform.
Terraform Cloud (free) does not meet either of these use cases since you can't deploy to on-premises nor can you manage the underlying operating system since it's a hosted service.

47. When using Terraform, where can you install providers from? (select four)

a. plugins directory
b. Terraform registry
c. the provider's source code
d. Terraform plugin cache
e. official HashiCorp releases site

Explanation
Providers can be installed using multiple methods, including downloading from a Terraform public or private registry, the official HashiCorp releases page, a local plugins directory, or even from a plugin cache. Terraform cannot, however, install directly from the source code.

48. When using collection types for variables in Terraform, which of the following ***two*** statements are true? (select two)

a. maps are defined inside of square brackets, like this: `[ name = "John" age = 52 ]`
b. lists are defined inside of curly braces, like this: `{"value1", "value2", "value3"}`
c. maps are defined inside of curly braces, like this: `{ name = "John" age = 52 }`
d. lists are defined inside of square brackets, like this: `["value1", "value2", "value3"]`

Explanation

Lists/tuples are represented by a pair of ***square brackets*** containing a comma-separated sequence of values, like `["a", 15, true]`.

Maps/objects are represented by a pair of ***curly braces*** containing a series of `<KEY> = <VALUE>` pairs.

49. Which of the following are true regarding Terraform variables? (select two)

a. variables marked as sensitive are still stored in the state file, even though the values are obfuscated from the CLI output
b. the default value will be found in the state file if no other value was set for the variable
c. the variable name can be found in the state file to allow for easy searching
d. the description of a variable will be written to state to help describe the contents of the state file

Explanation

When it comes to working with variables, the value that is used in the Terraform configuration will be stored in the state file, regardless of whether the sensitive argument was set to `true`. However, the value will not be shown in the CLI output if the value was to be exported by an output block.

Beyond the value, you won't find the variable name or description in the state file because they are simply used on the development side of Terraform, and not the backend operational aspect of how Terraform works.

50. Which of the following statements are ***true*** about using `terraform import`? (select three)

a. using `terraform import` will bring the imported resource under Terraform management and add the new resource to the state file
b. the `terraform import` command will automatically update the referenced Terraform resource block after the resource has been imported to ensure consistency
c. the `resource address` (example: *aws_instance.web*) and `resource ID` (example: *i-abdcef12345*) must be provided when importing a resource
d. you must update your Terraform configuration for the imported resource *before* attempting to import the resource

Explanation

`terraform import` can be used to import resources into Terraform so they can be managed by Terraform moving forward. Any resources that are imported will be added to Terraform state so they can be managed like any other resource. Before you can use the terraform import command, you MUST develop the resource block for the resource that will be imported. For example, if you are planning to import an Azure virtual machine, you must add an azurerm_virtual_machine block with the proper configurations.

When you run the `terraform import` command, you will need to reference the `resource address` - like *azure_virtual_machine.web-server* - and the `resource ID` - like the ID of the virtual machine in Azure - as the two required parameters.
`terraform import azurerm_virtual_machine.web-server 090556DA-D4FA-764F-A9F1-63614EDA019A`
`terraform import` will ***NOT*** create the resource block for you. You must create the resource block in your Terraform configuration *before* using the import command

51. You are using Terraform OSS and need to spin up a copy of your GCP environment in a second region to test some new features. You create a new workspace. Which of the following is true about this new workspace? (select four)

a. changes to this workspace won't impact other workspaces
b. it uses the same Terraform code in the current directory
c. it has its own state file
d. it uses a different Terraform backend
e. you can use a different variables file for this workspace if needed

Explanation
Terraform workspaces (OSS) allow you to create a new workspace to execute the same Terraform but with a ***different state file***. This feature will enable you to run the same Terraform with different configurations without modifying Terraform code or impacting any existing workspaces. Terraform states out with the `default` workspace, and that's the workspace you are using unless you create and switch to a new workspace.
Remember that Terraform Cloud and Enterprise also have Workspaces, but they behave slightly differently. In Cloud and Ent, each workspace is still isolated from others, meaning it has its own state. Still, often these workspaces point to different code repositories and use completely different Terraform configuration files.
To create a new workspace, you'd run:
$ terraform workspace new btk
Created and switched to workspace "btk"!
You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration.
To list all of the existing workspaces, you can run (note the * indicates the workspace you are using):
$ terraform workspace list
default*
btk
bryan-dev
temp-workspace
When using workspaces, you're essentially using the same Terraform configuration files. Therefore the backend will remain the same for all of your workspaces. That makes this answer incorrect.

52. Which of the following are true about Terraform providers? (select four)

a. they allow anybody to write a provider and publish it to the registry

b. providers can be written and maintained by an outside organization, such as AWS, F5, or Microsoft

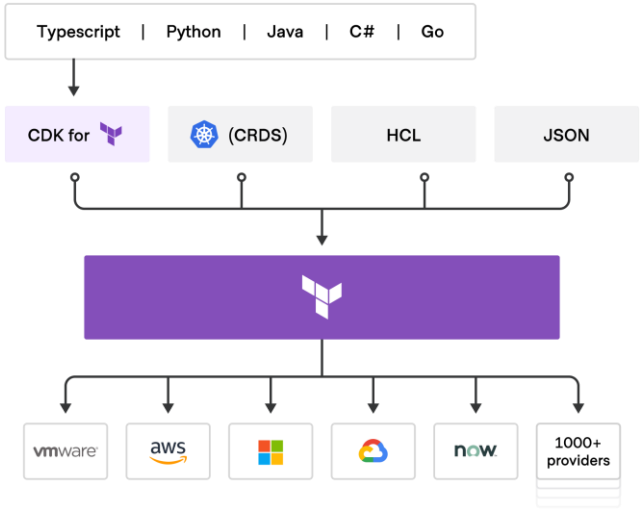
c. all providers are automatically included when downloading Terraform

d. some providers are community-supported

e. some providers are maintained by HashiCorp

Explanation

The cool part about providers is that anybody can write or contribute to them. This includes individuals would just want to contribute to open-source projects, manufacturers/platform vendors that want to ensure providers are up to date, or HashiCorp themselves. If you find that a provider doesn't provide the capabilities you need, you can develop the new capabilities and submit a PR for review. If approved, those changes would now become part of the Terraform provider that millions of people could use. Pretty cool!



Providers are treated as plugins for Terraform, and during a terraform init process, the required providers are downloaded to the local machine that is executing Terraform so they can be used. Therefore, not all providers are included with Terraform when you download the latest version from terraform.io.

53. Which of the following tasks does `terraform init` perform? (select three)

a. caches the source code locally for referenced modules

b. creates a sample Terraform configuration file in the working directory

c. updates your state file based on any new changes

d. downloads required providers used in your configuration file

e. prepares the working directory for use with Terraform

Explanation

The `terraform init` command performs several different initialization steps in order to prepare the current working directory for use with Terraform. Some of these steps include downloading any referenced providers (like AWS, Azure, GCP, etc.), caching the source code for modules in the local directory so they can be used, and other steps to prepare the working directory to be used with Terraform.

Note that there are quite a few options that you can use with `terraform init` to perform operations that you might need when using Terraform. These operations might include state migrations or upgrading providers.
`terraform init` does NOT create a sample Terraform configuration file. Actually, I don't know if there are any native Terraform commands that will create a .tf file for you.
You can run `terraform init` over and over again and it will not change/modify your state file.

54. Your team is using Terraform, and multiple team members need to be able to manage the infrastructure. You need to support state locking to reduce the chance of corrupting the state file. What backends can you use to meet these requirements? (select three)

a. `kubernetes` backend
b. `s3` backend (with DynamoDB)
c. `consul` backend
d. `local` backend

Explanation
Not all Terraform backends are created equal. Some backends act like plain "remote disks" for state files; others support *locking* the state while operations are being performed, which helps prevent conflicts and inconsistencies.
Kubernetes, Consul, and S3 backends all support state locking. S3 supports state locking with the help of DynamoDB.
While the `local` backend does support locking via system APIs, you can't use the `local` backend to share the state across your team.

55. Your co-worker has decided to migrate Terraform state to a remote backend. They configure Terraform with the backend configuration, including the type, location, and credentials. However, you want to better secure this configuration. Rather than storing them in plaintext, where should you store the credentials? (select two)

a. use a variable
b. credentials file
c. on the remote system
d. environment variables

Explanation
Some backends allow providing access credentials directly as part of the configuration for use in unusual situations, for pragmatic reasons. However, in normal use, HashiCorp ***does not*** recommend including access credentials as part of the backend configuration. Instead, leave those arguments completely unset and provide credentials via the credentials files or environment variables that are conventional for the target system, as described in the documentation for each backend.
Use a variable? Well, you could use a variable but that wouldn't really improve security here, since variable defaults or configurations are also stored in plaintext.
On the remote system? I don't think this is even a viable option. The creds would need to be read by the local system that is executing Terraform.

56. Which of the following are advantages of using infrastructure as code (IaC) for your day-to-day operations? (select three)

a. enables self-service for developers and operators alike

b. API-driven workflows

c. ensures the security of applications provisioned on managed infrastructure

d. provides the ability to version control the infrastructure and application architecture

Explanation

Using Infrastructure as Code (IaC) like Terraform, CloudFormation, etc. enables organizations to completely change the way they deploy applications and the underlying infrastructure to support them. Rather than click around in a console, ***IaC enables API-driven workflows*** for deploying resources in public clouds, private infrastructure, and other SaaS and PaaS services. When developing IaC, organizations can now use a Version Control System, such as GitHub, GitLab, Bitbucket, etc. ***to store and version its code***. When changes are needed, the existing code can be cloned, modified, and merged back into the main repository by way of a pull or merge request. These requests can follow a traditional workflow where approvals are needed before they are deployed to a production environment.

By moving your configurations to code and publishing them to a shared repository, or registry, different operators in the organization can now easily consume this code without even knowing how to write Terraform. They can simply provide the required values by way of variables and entire environments can be provisioned to support their application(s).

While Terraform can indeed help with the security of your applications, it won't guarantee it. You can combine Terraform with other tools, such as LaunchDarkly, SonarQube, DeepScan, and more using a CI/CD pipeline, but again, it doesn't guarantee the security of your application. Security is the responsibility of everybody involved in the deployment of an application, starting with the developer(s) themselves all the way to the people responsible for the day-to-day operations of the application.

57. Which of the following Terraform versions offer the ability to use a private module registry? (select three)

a. Terraform OSS/CLI

b. Terraform Cloud (free)

c. Terraform Enterprise

d. Terraform Cloud for Business

Explanation

Terraform Cloud (all versions) and Terraform Enterprise offer the use of a private module registry. Among all the features you get with Terraform OSS, all other versions get these features, at a minimum:

- State management
- Remote operations
- Private module registry
- Community support

You do not have the ability to use a private module registry with Terraform OSS.

58. What are some of the benefits that Terraform *providers* offer to users? (select three)

a. enables the deployment of resources to multiple platforms, such as public cloud, private cloud, or other SaaS, PaaS, or IaaS services

b. abstracts the target platform's API from the end-user

c. enforces security compliance across multiple cloud providers

d. enables a plugin architecture that allows Terraform to be extensible without having to update Terraform core

Explanation

Terraform enables its users to interact with a platform's API without requiring the end-user to understand individual APIs for the targeted platform. This allows a user to easily provision and manage resources across many different platforms without having to understand the API for each individual backend. This benefit makes users more efficient and reduces the administrative burden for understanding and troubleshooting each one.

Terraform can support any platform that has an API, including public, private, and other offerings on the market today. If it has an API, a provider can be written to allow Terraform to manage it. Don't believe me? Check out the Spotify or Domino's Pizza Terraform provider :)

Lastly, by using providers, HashiCorp can enable the extensibility of Terraform without having to modify Terraform core for each supported platform. Each provider, or plugin, can be downloaded as needed to extend the functionality of Terraform itself.

Now, while Terraform can help you standardize security configurations and settings across multiple clouds, it won't enforce it outside of your `terraform apply`. In other words, it doesn't act like a configuration management tool that can constantly watch for changes to change the configuration back to the desired state.

59. You have provisioned some virtual machines (VMs) on Google Cloud Platform (GCP) using the gcloud command line tool. However, you are standardizing with Terraform and want to manage these VMs using Terraform instead. What are the two things you must do to achieve this? (Choose two.)
    a. Provision new VMs using Terraform with the same VM names
    b. Use the terraform import command for the existing VMs
    c. Write Terraform configuration for the existing VMs
    d. Run the terraform import-gcp command

Explanation: You need to import the existing VMs into Terraform state using **terraform import** and write Terraform configuration for managing these VMs to transition to managing them with Terraform.

60. Which two steps are required to provision new infrastructure in the Terraform workflow? (Choose two.)
    a. Destroy
    b. Apply
    c. Import
    d. Init
    e. Validate

Explanation: To provision new infrastructure using Terraform, you need to initialize the working directory (**terraform init**) and then apply the configuration (**terraform apply**).

61. You have used Terraform to create an ephemeral development environment in the cloud and are now ready to destroy all the infrastructure described by your Terraform configuration. To be safe, you would like to first see all the infrastructure that will be deleted by Terraform. Which command should you use to show all of the resources that will be deleted? (Choose two.)
    a. Run terraform plan -destroy.

b. This is not possible. You can only show resources that will be created.
c. Run terraform state rm *.
d. Run terraform destroy and it will first output all the resources that will be deleted before prompting for approval.

Explanation: Both **terraform plan -destroy** and **terraform destroy** with the **-destroy** flag show the resources that will be deleted before actual deletion occurs.

62. You have declared a variable called var.list which is a list of objects that all have an attribute id. Which options will produce a list of the IDs? (Choose two.)
    a. { for o in var.list : o => o.id }
    b. var.list[*].id
    c. [var.list[*].id]
    d. [for o in var.list : o.id]

Explanation: These options correctly access the **id** attribute within the list of objects in **var.list**.

63. What features does the hosted service Terraform Cloud provide? (Choose two.)
    a. Automated infrastructure deployment visualization
    b. Automatic backups
    c. Remote state storage
    d. A web-based user interface (UI)

Explanation: Terraform Cloud provides remote state storage, allowing teams to store their state files securely in the cloud. It also offers a web-based UI for managing Terraform configurations, workspaces, and collaboration.

64. When using Terraform to deploy resources into Azure, which scenarios are true regarding state files? (Choose two.)
    a. When a change is made to the resources via the Azure Cloud Console, the changes are recorded in a new state file
    b. When a change is made to the resources via the Azure Cloud Console, Terraform will update the state file to reflect them during the next plan or apply
    c. When a change is made to the resources via the Azure Cloud Console, the current state file will not be updated
    d. When a change is made to the resources via the Azure Cloud Console, the changes are recorded in the current state file

Explanation: Changes made outside of Terraform (e.g., Azure Cloud Console) can affect the state file and are reflected during the next Terraform plan or apply.

65. Which of the following does terraform apply change after you approve the execution plan? (Choose two.)
    a. Cloud infrastructure
    b. The .terraform directory
    c. The execution plan
    d. State file
    e. Terraform code

Explanation: Terraform applies the changes described in the execution plan to the cloud infrastructure and updates the state file to reflect the current state of the infrastructure.

66. How can a ticket-based system slow down infrastructure provisioning and limit the ability to scale? (Choose two.)
    a. A full audit trail of the request and fulfillment process is generated
    b. A request must be submitted for infrastructure changes
    c. As additional resources are required, more tickets are submitted
    d. A catalog of approved resources can be accessed from drop down lists in a request form

Explanation: Ticket-based systems often introduce manual approval processes, slowing down provisioning as each change requires approval, and they might bottleneck as more resources require more tickets and approvals.

67. You have used Terraform to create an ephemeral development environment in the cloud and are now ready to destroy all the infrastructure described by your Terraform configuration. To be safe, you would like to first see all the infrastructure that will be deleted by Terraform. Which command should you use to show all of the resources that will be deleted? (Choose two.)
    a. Run terraform plan -destroy
    b. Run terraform show -destroy
    c. Run terraform destroy and it will first output all the resources that will be deleted before prompting for approval
    d. Run terraform show -destroy

Explanation: Both commands can provide details on the resources that Terraform plans to destroy.

68. What does Terraform use providers for? (Choose three.)
    a. Provision resources for on-premises infrastructure services
    b. Simplify API interactions
    c. Provision resources for public cloud infrastructure services
    d. Enforce security and compliance policies
    e. Group a collection of Terraform configuration files that map to a single state file

Explanation: Providers enable interactions with APIs, provision resources, and encapsulate configurations related to a specific service or infrastructure.#146*Topic*

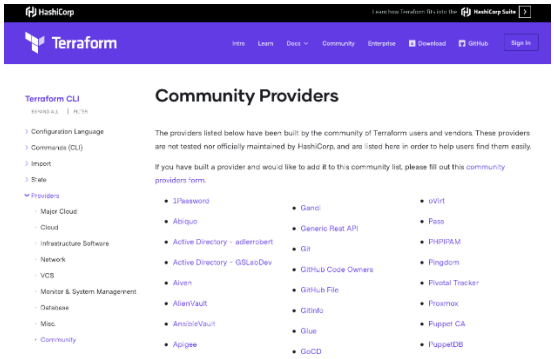
69. Which parameters does terraform import require? (Choose two.)

a. Path
b. Provider
c. Resource ID
d. Resource address

Explanation: When using **terraform import**, you need to specify the resource's ID (the identifier for the resource in the provider) and the resource's address (the address used within the Terraform configuration).

True or False

1. Using the latest versions of Terraform, terraform init cannot automatically download community providers.



   a. True
   b. False

Explanation

False. With the latest versions of Terraform, **terraform init** can automatically download community providers. More specifically, this feature was added with Terraform 0.13. Before 0.13, **terraform init** would **NOT** download community providers.

Terraform includes a built-in provider registry that allows you to easily install and manage the providers you need for your Terraform configuration. When you run **terraform init**, Terraform will check your configuration for any required providers and download them automatically if they are not already installed on your system. This includes both official Terraform providers and community-maintained providers.

To use a community-maintained provider in your Terraform configuration, you need to specify the provider in your configuration using the **provider** block and include the provider's source repository in your configuration. Terraform will download and install the provider automatically when you run **terraform init**, provided that the provider is available in the Terraform provider registry.

2. Each Terraform workspace uses its own state file to manage the infrastructure associated with that particular workspace.

   a. True
   b. False

Explanation

True. Each Terraform workspace uses its own state file to manage the infrastructure associated with that workspace. This allows Terraform to manage multiple sets of infrastructure independently and avoid conflicts. Each Terraform workspace has its own Terraform state file that keeps track of the resources and their attributes, so changes made in one workspace will not affect the infrastructure managed by other workspaces.

In fact, having different state files provides the benefits of workspaces, where you can separate the management of infrastructure resources so you can make changes to specific resources without impacting resources in others....

3. When using the Terraform provider for Vault, the tight integration between these HashiCorp tools provides the ability to mask secrets in the state file.

a. False

b. True

Explanation

False. By default, Terraform does not provide the ability to mask secrets in the Terraform plan and state files regardless of what provider you are using. While Terraform and Vault are both developed by HashiCorp and have a tight integration, masking secrets in Terraform plans and state files requires additional steps to securely manage sensitive information.

One common approach is to use environment variables or Terraform input variables to store sensitive information, and then use Terraform's data sources to retrieve the information from the environment or input variables, rather than hardcoding the information into the Terraform configuration. This helps to ensure that sensitive information is not stored in plain text in the Terraform plan or state files.

4. The `terraform plan -refresh-only` command is used to create a plan whose goal is only to update the Terraform state to match any changes made to remote objects outside of Terraform.

a. False

b. True

Explanation

The **`terraform plan -refresh-only`** command is used in Terraform to update the state of your infrastructure in memory without making any actual changes to the infrastructure. The **`-refresh-only`** flag tells Terraform to only update its understanding of the current state of the infrastructure and not to make any changes.

When you run **`terraform plan -refresh-only`**, Terraform will query the current state of your infrastructure and update its internal state to reflect what it finds. This can be useful if you want to ensure that Terraform has the most up-to-date information about your infrastructure before generating a plan, without actually making any changes.

It is important to note that while the **`terraform plan -refresh-only`** command updates Terraform's internal state, it does not modify the Terraform state file on disk. The Terraform state file is only updated when Terraform actually makes changes to the infrastructure.

Note that this command replaced the deprecated command **`terraform refresh`**

5. Rather than use a state file, Terraform can inspect cloud resources on every run to validate that the real-world resources match the desired state.

a. False

b. True

Explanation

State is a necessary requirement for Terraform to function. And in the scenarios where Terraform may be able to get away without state, doing so would require shifting massive amounts of complexity from one place (state) to another place (the replacement concept).

To support mapping configuration to resources in the real world, Terraform uses its own state structure. Terraform can guarantee one-to-one mapping when it creates objects and records

their identities in the state. Terraform state also serves as a performance improvement - rather than having to scan every single resource to determine the current state of each resource.

6. Terraform Enterprise offers the ability to use Terraform to deploy infrastructure in your local on-premises datacenter as well as a public cloud platform, such as AWS, Azure, or GCP.

a. True

b. False

Explanation

Terraform Enterprise is often installed on-premises or in a public cloud and provides the ability to deploy resources using any provider/plugin that is supported by Terraform. TFE gives you many features that are not available with Terraform OSS. Many of these features, however, are available in Terraform Cloud, though.

7. By default, the `terraform destroy` command will prompt the user for confirmation before proceeding.

a. True

b. False

Explanation

True. By default, Terraform will prompt for confirmation before proceeding with the **`terraform destroy`** command. This prompt allows you to verify that you really want to destroy the infrastructure that Terraform is managing before it actually does so.

Terraform destroy will always prompt for confirmation before executing unless passed the **`-auto-approve`** flag.

```
$ terraform destroy
Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.
   Enter a value: yes
```

8. The following code is an example of an ***implicit dependency*** in Terraform

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
resource "aws_ebs_volume" "data" {
  availability_zone = "us-west-2a"
  size              = 1
  tags = {
    Name = "data-volume"
  }
}
resource "aws_volume_attachment" "attach_data_volume" {
  device_name = "/dev/xvdf"
  volume_id   = aws_ebs_volume.data.id
  instance_id = aws_instance.web.id
```

}

a. True

b. False

Explanation

Terraform ***implicit dependencies*** refer to the dependencies between resources in a Terraform configuration but are not **explicitly** defined in the configuration. Terraform uses a graph to track these implicit dependencies and ensures that resources are created, updated, and deleted in the correct order.

For example, suppose you have a Terraform configuration that creates a virtual machine and a disk. In that case, Terraform will implicitly depend on the disk being created before the virtual machine because the virtual machine needs the disk to function. Terraform will automatically create the disk first and then create the virtual machine.

Sometimes, Terraform may miss an implicit dependency, resulting in an error when you run **terraform apply**. In these cases, you can use the **depends_on** argument to explicitly declare the dependency between resources. For example:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
   depends_on = [
    aws_ebs_volume.example
  ]
}
resource "aws_ebs_volume" "example" {
  availability_zone = "us-west-2a"
  size              = 1
}
```

In this example, the **aws_instance** resource depends on the **aws_ebs_volume** resource, and Terraform will create the **aws_ebs_volume** resource first and then the **aws_instance** resource. In general, Terraform implicit dependencies are handled automatically, but sometimes it may be necessary to use the **depends_on** argument to ensure that resources are created in the correct order.

9. Starting in Terraform v0.12, the Terraform language now has built-in syntax for creating lists using the [and] delimiters, replacing and deprecating the list () function.

a. True

b. False

Explanation

The list function is deprecated. From Terraform v0.12, the Terraform language has built-in syntax for creating lists using the [and] delimiters. Use the built-in syntax instead.

The list function will be removed in a future version of Terraform.

10. Multiple providers can be declared within a single Terraform configuration file.

a. True

b. False

Explanation

True. Multiple providers can be declared within a single Terraform configuration file. In fact, it is common to declare multiple providers within a single configuration file, particularly when managing resources across multiple cloud providers.
When declaring multiple providers within a single configuration file, each provider should have a unique configuration block that specifies its name, source, and any required settings or credentials. Here's an example of what a configuration block for two different providers might look like:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
    google = {
      source = "hashicorp/google"
    }
  }
}
provider "aws" {
  region = "us-west-2"
  access_key = "ACCESS_KEY"
  secret_key = "SECRET_KEY"
}
provider "google" {
  project = "my-project"
  credentials = file("path/to/credentials.json")
}
```

In this example, we have declared two providers (**aws** and **google**) within a single configuration file. The **terraform** block declares the required providers, while the **provider** blocks specify the provider-specific settings and credentials.
When executing Terraform commands, you can use the **-target** option to specify which provider you want to apply changes to. For example, you could apply changes to the AWS provider by running **terraform apply -target=aws**.

11. Provisioners should only be used as a last resort.

a. True

b. False

Explanation

Provisioners in Terraform are used to execute scripts or commands on a resource after it has been created. While provisioners can be useful in certain situations, such as configuring a server or setting up a database, ***it is generally true that provisioners should only be used as a last resort.***
The reason for this is that using provisioners can make your Terraform code less predictable and harder to manage. When you use provisioners, you introduce external dependencies and

additional complexity into your infrastructure configuration. This can make it harder to reproduce your infrastructure, as well as harder to troubleshoot issues that arise.
Instead of using provisioners, it is generally recommended to use native Terraform resources and data sources to manage your infrastructure. This approach makes your code more predictable, easier to manage, and less error-prone.
However, there may be situations where provisioners are necessary or unavoidable, such as when working with legacy systems or when a resource does not support the necessary configuration options. In these cases, provisioners can be a useful tool, but they should be used judiciously and with care, and you should be aware of the potential risks and downsides of using provisioners in your infrastructure configuration.

12. You can migrate the Terraform backend but only if there are no resources currently being managed.

a. False

b. True

Explanation

If you are already using Terraform to manage infrastructure, you probably want to transfer to another backend, such as Terraform Cloud, so you can continue managing it. By migrating your Terraform state, you can hand off infrastructure without de-provisioning anything.

13. State is a requirement for Terraform to function.

a. False

b. True

Explanation

True. State is a fundamental concept in Terraform that keeps track of the resources Terraform manages, their configuration, and their current state. Terraform uses this information to determine the differences between the desired state and the current state and to generate a plan for creating, updating, or deleting resources to match the desired state.
The state file is a critical component of Terraform and is required for its proper functioning. It is typically stored remotely in a shared location, such as a storage service or version control system, to allow multiple members of a team to collaborate on infrastructure changes.

14. Workspaces provide similar functionality in the open-source and Terraform Cloud versions of Terraform.

a. False

b. True

Explanation

Workspaces, managed with the `terraform workspace` command, isn't the same thing as Terraform Cloud's workspaces. Terraform Cloud workspaces act more like completely separate working directories. CLI workspaces (OSS) are just alternate state files.

15. A `terraform plan` is a required step before running a `terraform apply`?

a. True

b. False

Explanation

If no explicit plan file is given on the command line, `terraform apply` will create a new plan automatically and prompt for approval to apply it

16. Before a `terraform validate` can be run, the directory must be initialized.

a. False

b. True

Explanation

Validation requires an initialized working directory with any referenced plugins and modules installed. If the directory is NOT initialized, it will result in an error.

```
$ terraform validate
Error: Could not load plugin
Plugin reinitialization required. Please run "terraform init".
Plugins are external binaries that Terraform uses to access and manipulate
resources. The configuration provided requires plugins which can't be located,
don't satisfy the version constraints, or are otherwise incompatible.
Terraform automatically discovers provider requirements from your
configuration, including providers used in child modules. To see the
requirements and constraints, run "terraform providers".
Failed to instantiate provider "registry.terraform.io/hashicorp/aws" to obtain
schema: unknown provider "registry.terraform.io/hashicorp/aws"
```

17. A `main.tf` file is always required when using Terraform?

a. True

b. False

Explanation

Although main.tf is the standard name, it's not necessarily required. Terraform will look for any file with a `.tf` or `.tf.json` extension when running terraform commands.

18. A remote backend configuration is required for using Terraform.

a. False

b. True

Explanation

This is false. If you don't provide a backend configuration, Terraform will use the local default backend. **Remote Backends are completely optional**. You can successfully use Terraform without ever having to learn or use a remote backend. However, they do solve pain points that afflict teams at a certain scale. If you're an individual, you can likely get away with never using backends.

19. Any sensitive values referenced in the Terraform code, even as variables, will end up in plain text in the state file.

a. False

b. True

Explanation

Any values that are retrieved in a data block or referenced as variables will show up in the state file.

20. Terraform is designed to work only with public cloud platforms, and organizations that wish to use it for on-premises infrastructure (private cloud) should look for an alternative solution.

a. False

b. True

Explanation

Terraform is designed to work with almost any infrastructure that provides an API. Terraform is very frequently used to provision infrastructure atop VMware infrastructure, along with traditional, physical security or infrastructure service solutions.

21. When developing Terraform code, you *must* include a provider block for each unique provider so Terraform knows which ones you want to download and use.

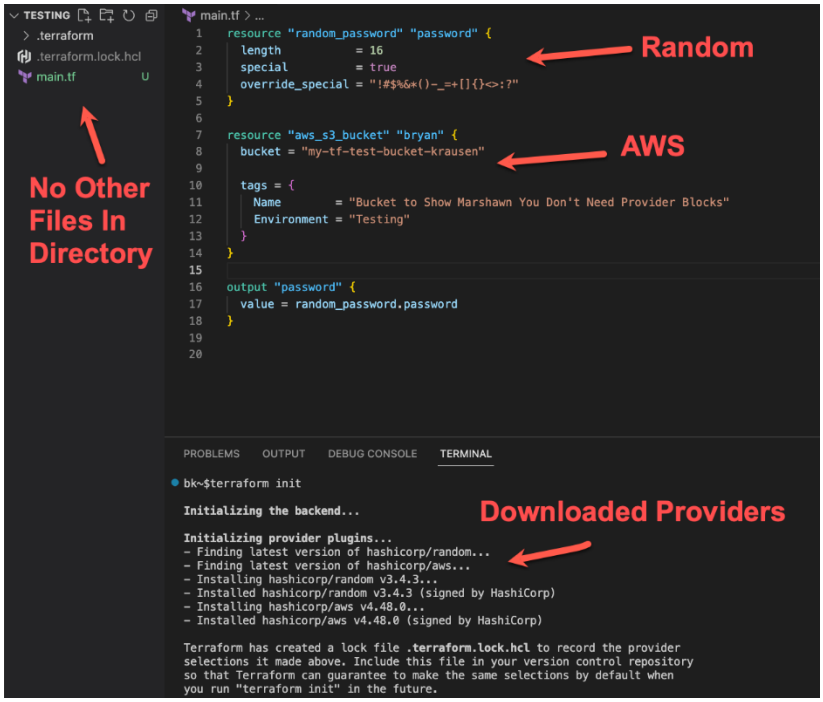
a. True

b. False

Explanation

Unlike many other objects in the Terraform language, a `provider` block may be omitted if its contents would otherwise be empty. Terraform assumes an empty default configuration for any provider that is not explicitly configured. In other words, if you don't have any specific configurations for your provider, you may indeed leave it out of your configuration.

To prove this out, I created a `.tf` file that includes a resource from the **RANDOM** provider as well as the **AWS** provider and omitted any provider blocks in my configuration. After running a `terraform init`, you can clearly see that Terraform understands what providers the resources are from and downloads the correct provider plugins. Thus proving that you do NOT need a `Provider` block to use Terraform.



22. Terraform can only manage dependencies between resources if the `depends_on` argument is explicitly set for the dependent resources.

a. True

b. False

Explanation

The most common source of dependencies is an implicit dependency between two resources or modules. That means that Terraform builds a dependency map (aka resource graph) to help determine what resources it can create in parallel, and what resources are dependent on others based on interpolation used within the configuration.

23. `min`, `max`, `format`, `join`, `trim`, and `length` are examples of different expressions in Terraform.

a. True

b. False

Explanation

These are actually examples of Terraform functions, not expressions. Expressions would be something more in the line of `string`, `number`, `bool`, `null`, etc.

For the exam, you should go through these and understand what they do at a high level as you could get questions on using a few of them.

24. Official Terraform providers and modules are owned and maintained by HashiCorp.

a. True

b. False

Explanation

This is true. If a module or provider is marked as official, it is owned and maintained by HashiCorp themselves. *In fact, I copied the sentence in the question straight off the official Terraform registry page :)*

There are other modules/providers available in the registry that are maintained by third-party partners, or even individuals. This also means that not all of the modules published to the Terraform registry are validated or verified by HashiCorp. Many folks will use the public module registry as a starting place to create their own custom modules needed to meet requirements.

25. The terraform graph command can be used to generate a visual representation of a configuration or execution plan.

a. True

b. False

Explanation

The `terraform graph` command is used to generate a visual representation of either a configuration or execution plan. The output is in the DOT format, which can be used by GraphViz to generate charts.

26. You can continue using your local Terraform CLI to execute `terraform plan` and `terraform apply` operations while using Terraform Cloud as the backend.

a. True

b. False

Explanation

If you have migrated or configured your state to use Terraform Cloud using the backend configuration, you can continue using your local Terraform CLI to execute operations while using Terraform Cloud. You can even specify the workspace you want to execute the operation in.

To configure the backend to use Terraform Cloud, you can add something like this:

```
terraform {
  cloud {
```

```
  organization = "bryan"
  workspaces {
   tags = ["app"]
  }
 }
}
```

27. If you have properly locked down access to your state file, it is safe to provide sensitive values inside of your Terraform configuration.
    a. True
    b. False

Explanation

Ok, so this was sort of a trick question because locking down your state file really has nothing to do with storing sensitive values inside of your Terraform configuration. Remember that most, if not all, of your configuration, will likely be committed to a code repository. Anybody, or any machine, with access to that code repo would now be able to read the sensitive values that were hardcoded in your Terraform configuration.

Best practice here is to provide your sensitive values OUTSIDE of Terraform, like storing and retrieving them from a secrets management platform like Vault, or using environment variables.

28. When using Terraform Cloud, committing code to your version control system (VCS) can automatically trigger a speculative plan.
    a. True
    b. False

Explanation

When workspaces are linked to a VCS repository, Terraform Cloud can automatically initiate Terraform runs when changes are committed to the specified branch.

29. You can move Terraform state between supported backends at any time, even after running your first `terraform apply`.
    a. False
    b. True

Explanation

You can change your backend configuration at any time. You can change both the configuration itself as well as the type of backend (for example from "consul" to "s3").

Terraform will automatically detect any changes in your configuration and request a reinitialization. As part of the reinitialization process, Terraform will ask if you'd like to migrate your existing state to the new configuration. This allows you to easily switch from one backend to another.

30. `terraform validate` will validate the syntax of your HCL files.
    a. False
    b. True

Explanation

The `terraform validate` command validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider APIs, etc.

Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state. It is thus primarily useful for general verification of reusable modules, including the correctness of attribute names and value types.

31. In both Terraform OSS and Terraform Cloud/Enterprise, workspaces provide similar functionality of using a separate state file for each workspace.

a. False

b. True

Explanation

This is true. When you create a new workspace using Terraform OSS/CLI using the `terraform workspace new` command, you will be working with a separate state file when working with that workspace. You can easily change between workspaces and their respective state file using the `terraform workspace select` command.

The same is true in Terraform Cloud/Enterprise. When you create a new workspace, you'll be working with a dedicated state file for that particular workspace. It doesn't share a state file with any other workspace.

32. Under special circumstances, Terraform can be used without state.

a. **False**

b. True

Explanation

State is a hard requirement for Terraform - there's no getting around it. You can have the state stored locally or you can configure a remote backend to store it somewhere else. But overall, state is always required for Terraform.

33. Infrastructure as code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface.

a. False

b. True

Explanation

This is true, although there are tools out there that have UIs to deploy IaC. However, the goal is to reduce or eliminate the need to use a UI to deploy infrastructure and applications.

34. In order to use the `terraform console` command, the CLI must be able to lock state to prevent changes.

a. False

b. True

Explanation

The `terraform console` command will read the Terraform configuration in the current working directory and the Terraform state file from the configured backend so that interpolations can be tested against both the values in the configuration and the state file.

When you execute a `terraform console` command, you'll get this output:

```
$ terraform console
Acquiring state lock. This may take a few moments...
>
```

35. When you are referencing a module, you must specify the version of the module in the calling module block.

a. False

b. True

Explanation

While it's not required, we recommend explicitly constraining the acceptable version numbers to avoid unexpected or unwanted changes. Use the `version` argument in the `module` block to specify versions.

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"
   servers = 3
}
```

36. In Terraform OSS, workspaces generally use the same code repository while workspaces in Terraform Enterprise/Cloud are often mapped to different code repositories.

a. False

b. True

Explanation

Workspaces in OSS are often used within the same working directory while workspaces in Enterprise/Cloud are often (but not required) mapped to unique repos.

37. A `provider` block is required in every configuration file so Terraform can download the proper plugin.

a. True

b. False

Explanation

You don't have to specify a provider block since Terraform is smart enough to download the right provider based on the specified resources. That said, Terraform official documentation states that Terraform configurations must declare which providers they require so that Terraform can install and use them. Although Terraform CAN be used without declaring a plugin, you should follow best practices and declare it along with the `required_version` argument to explicitly set the version constraint.

38. Running a `terraform apply` will fail if you do not run a `terraform plan` first.

a. True

b. False

Explanation

You do NOT need to run a `terraform plan` before running `terraform apply`. When you execute a `terraform apply`, it will actually run its own "plan" to make sure it knows what resources to update.

The most straightforward way to use `terraform apply` is to run it without any arguments at all, in which case it will automatically create a new execution plan (as if you had run `terraform plan`) and then prompt you to approve that plan, before taking the indicated actions.

39. Input variables that are marked as sensitive are NOT written to Terraform state.

a. True

b. False

Explanation

While the value is not shown in the Terraform CLI output, the value will still be written to state. This is why it's important to secure your state file wherever possible.

40. The terraform.tfstate file always matches your currently built infrastructure.

a. True

b. False

Explanation: The **terraform.tfstate** file represents the state of your infrastructure as recorded by Terraform during its last execution. Changes made directly to the infrastructure, without using Terraform, won't be reflected in this file.

41. One remote backend configuration always maps to a single remote workspace.

a. True

b. False

Explanation: Each remote backend configuration in Terraform typically corresponds to a single remote workspace. Workspaces help isolate Terraform state and configurations for different environments or purposes.

42. A provider configuration block is required in every Terraform configuration.

Example:

```
provider "provider_name" {
   ...
}
```

a. True

b. False

Explanation: Provider configurations are necessary only when you're working with resources specific to a particular provider. Not every Terraform configuration requires provider blocks.

43. Terraform variables and outputs that set the "description" argument will store that description in the state file.

a. True

b. **False**

Explanation: Descriptions set in variables or outputs don't affect the state file. The state file stores the state of resources managed by Terraform, not their descriptive metadata.**#13*Topic***

44. If a module uses a local values, you can expose that value with a terraform output.

a. True

b. False

Explanation: You can expose local values from a module as outputs using **terraform output**.n

45. You should store secret data in the same version control repository as your Terraform configuration.

a. True

b. **False** #19*Top*

Explanation: Storing secret data in version control isn't recommended due to security reasons. Instead, use secure storage solutions or environment variables.

46. terraform init initializes a sample main.tf file in the current directory.

a. True
b. False

Explanation: **terraform init** initializes Terraform in a directory, downloading necessary plugins and configuring the working directory, but it does not create a sample **main.tf** file.

47. Terraform requires the Go runtime as a prerequisite for installation.
    a. True
    b. False

Explanation: Terraform is distributed as a pre-built binary executable for different operating systems and does not require the Go runtime as a prerequisite for installation.

48. terraform validate validates the syntax of Terraform files.
    a. True
    b. False

Explanation: **terraform validate** checks the syntax of Terraform configuration files for errors.

49. A Terraform provisioner must be nested inside a resource configuration block.
    a. True
    b. False

Explanation: Provisioners in Terraform are associated with resource creation and are defined within resource blocks to perform actions on the created resource.

50. Terraform can run on Windows or Linux, but it requires a Server version of the Windows operating system.
    a. True
    b. False

Explanation: Terraform can run on Windows and Linux, and it doesn't specifically require a Server version of Windows; it works on various Windows versions and Linux distributions.

51. Only the user that generated a plan may apply it.
    a. True
    b. False

Explanation: In most cases, multiple users with access to the state file can apply the plan generated by another user.

52. Setting the TF_LOG environment variable to DEBUG causes debug messages to be logged into syslog.
    a. True
    b. False

Explanation: Changing the **TF_LOG** environment variable to **DEBUG** increases the level of detail in Terraform logs, aiding in debugging by providing more information in the terminal or the configured output destination.

53. In Terraform 0.13 and above, outside of the required_providers block, Terraform configurations always refer to providers by their local names.
    a. True
    b. False

Explanation: In Terraform 0.13 and later versions, configurations outside of the **required_providers** block typically reference providers using their local names, allowing for a more consistent and manageable provider configuration within the main configuration.

54. Terraform providers are always installed from the Internet.

a. True
b. False

Explanation: While Terraform providers can be obtained from various sources, including online repositories like the Terraform Registry, they can also be installed manually or sourced from private repositories and local filesystems, not necessarily relying on the internet.

55. Terraform provisioners can be added to any resource block.
    a. True
    b. False

Explanation: Terraform provisioners can indeed be added to any resource block to perform tasks like initializing resources, executing commands after resource creation, or implementing custom actions.

56. A Terraform local value can reference other Terraform local values.
    a. True
    b. False

Explanation: Terraform local values can indeed reference other Terraform local values. This allows for the creation of derived values or variables based on previously defined local values within the same module.

57. When running the command terraform taint against a managed resource you want to force recreation upon, Terraform will immediately destroy and recreate the resource.
    a. True
    b. False

Explanation: Running **terraform taint** marks a resource as tainted, but it does not immediately destroy and recreate the resource. Instead, it signifies to Terraform that the resource needs to be recreated on the next **terraform apply** run.

58. All standard backend types support state storage, locking, and remote operations like plan, apply and destroy.
    a. True
    b. False

Explanation: Standard backend types like Amazon S3, Azure Blob Storage, Google Cloud Storage, and others support state storage, locking mechanisms, and remote operations for Terraform commands such as **plan**, **apply**, and **destroy**.

59. In contrast to Terraform Open Source, when working with Terraform Enterprise and Cloud Workspaces, conceptually you could think about them as completely separate working directories.
    a. True
    b. False

Explanation: Terraform Enterprise and Cloud Workspaces share a similar conceptual structure as Terraform Open Source, where they maintain workspaces for different environments but within the same configuration and repository context.

60. Terraform can only manage resource dependencies if you set them explicitly with the depends_on argument.
    a. True
    b. False

Explanation: While the **depends_on** argument is one way to set explicit dependencies between resources, Terraform also automatically infers dependencies based on resource references within the configuration, allowing it to manage resource ordering implicitly.

61. Module variable assignments are inherited from the parent module and do not need to be explicitly set.
    a. True
    b. False

Explanation: Module variable assignments need to be explicitly set when a module is invoked. While default values can be defined in the module, these values are not automatically inherited; they must be explicitly provided when using the module.

62. HashiCorp Configuration Language (HCL) supports user-defined functions.
    a. True
    b. False

Explanation: As of Terraform 0.15 (the last release as of my last update), HCL does not support defining custom or user-defined functions. However, it provides various built-in functions for manipulating values and performing tasks within the configuration.

63. If a module declares a variable with a default, that variable must also be defined within the module.
    a. True
    b. False

Explanation: When a variable in a module has a default value, it doesn't need to be explicitly provided when using the module. The module can operate using the default value unless a different value is explicitly passed when calling the module.

64. terraform validate validates that your infrastructure matches the Terraform state file.
    a. True
    b. False

Explanation: **terraform validate** checks the syntax of the Terraform files to ensure they are valid and correctly formatted but doesn't compare or validate against the actual infrastructure state.

65. A module can always refer to all variables declared in its parent module.
    a. True
    b. False

Explanation: By default, a module cannot directly access variables from its parent module unless these variables are explicitly passed to the module as inputs.

66. All modules published on the official Terraform Module Registry have been verified by HashiCorp.
    a. True
    b. False

Explanation: While the official Terraform Module Registry is a trusted source, not all modules listed have been directly verified by HashiCorp.

67. You have to initialize a Terraform backend before it can be configured.
    a. True
    b. False

Explanation: Initializing a Terraform backend is a necessary step before configuring it for state storage and operations.

68. Terraform plan updates your state file.
    a. True
    b. False

Explanation: **terraform plan** only creates an execution plan and does not update the state file.

69. All Terraform Cloud tiers support team management and governance.
    a. True
    b. False

Explanation: All Terraform Cloud tiers offer team management and governance capabilities to various extents.

70. Terraform variable names are saved in the state file.
    a. True
    b. False

Explanation: Terraform variable names are not saved in the state file; they are stored in the configuration files and can be overridden with values provided by users.

71. Terraform Cloud is available only as a paid offering from HashiCorp.
    a. True
    b. False

Explanation: Terraform Cloud has both free and paid tiers, offering varying levels of features and support.

72. You can reference a resource created with for_each using a Splat (*) expression.
    a. True
    b. False

Explanation: The splat (*) expression can be used to reference attributes of resources created using for_each.

Fill in the Blanks

1. *Fill in the correct answers below:*

Infrastructure as Code (IaC) makes infrastructure changes _____, _____, _____, and _____. (select four)

- **a. consistent**
- **b. predictable**
- c. highly-available
- **d. idempotent**
- **e. repeatable**

Explanation

IaC makes changes idempotent, consistent, repeatable, and predictable. Without IaC, scaling up infrastructure to meet increased demand may require an operator to remotely connect to each machine and then manually provision and configure many servers by executing a series of commands/scripts. They might open multiple sessions and move between screens, which often results in skipped steps or slight variations between how work is completed, necessitating rollbacks. Perhaps a command was run incorrectly on one instance and reverted before being re-run correctly.

2. *Fill in the blank from the correct answer below:*

A Terraform module (usually the root module of a configuration) can *call* other modules to include their resources into the configuration. A module that has been called by another module is often referred to as _____.

- a. sourced module
- **b. child module**
- c. called module
- **d.** parent module

Explanation

A Terraform module (usually the root module of a configuration) can *call* other modules to include their resources into the configuration. A module that has been called by another module is often referred to as a *child module.*

Child modules can be called multiple times within the same configuration, and multiple configurations can use the same child module.

3. Environment variables can be used to set the value of input variables. The environment variables must be in the format "____"_<*variablename*>.

Select the correct prefix string from the following list.

- a. TF_ENV
- b. TF_VAR_VALUE
- c. TF_ENV_VAR
- **d. TF_VAR**

Explanation

Terraform allows you to use environment variables to set values in your Terraform configuration. This can be useful for specifying values specific to the environment in which Terraform is

running or providing values that can be easily changed without modifying the Terraform configuration.
To use a variable in Terraform, you need to define the variable using the following syntax in your Terraform configuration:

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

You can then set the value of the environment variable when you run Terraform by exporting the variable in your shell before running any Terraform commands:

```
$ export TF_VAR_instructor_name="bryan"
$ terraform apply
```

4. *Select two answers to complete the following sentence:*

Before a new provider can be used, it must be _____ and _____. (select two)

a. uploaded to source control
b. approved by HashiCorp
c. declared or used in a configuration file
d. initialized

Explanation

Each time a new provider is added to configuration -- either explicitly via a provider block or by adding a resource from that provider -- Terraform must initialize the provider before it can be used. Initialization downloads and installs the provider's plugin so that it can later be executed.

5. *Select the answer below that completes the following statement:*

Terraform Cloud can be managed from the CLI but requires _____?

a. authentication using MFA
b. an API token
c. a username and password
d. a TOTP token

Explanation

API and CLI access are managed with API tokens, which can be generated in the Terraform Cloud UI. Each user can generate any number of personal API tokens, which allow access with their own identity and permissions. Organizations and teams can also generate tokens for automating tasks that aren't tied to an individual user.

6. *Select the feature below that best completes the sentence:*

The following list represents the different types of _____ available in Terraform.

max
min
join
replace
length
range

a. named values
b. data sources
c. functions
d. backends

Explanation
The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values. The Terraform language does not support user-defined functions, and only the functions built into the language are available for use.

7. You need to specify a dependency manually. What resource meta-parameter can you use to make sure Terraform respects the dependency?

depends_on
Explanation: depends_on is used as a resource meta-parameter to specify dependencies and ensure Terraform handles resource creation or deletion in the correct order.

8. Which flag would you add to terraform plan to save the execution plan to a file?

-out
Explanation: -out. To save the execution plan to a file, you'd use the command **terraform plan -out=<filename>**

9. What is the name of the default file where Terraform stores the state? Type your answer in the field provided.

terraform.tfstate.
Explanation: terraform.tfstate. This is the default filename where Terraform stores the state unless overridden in the configuration.

10. In the below configuration, how would you reference the module output vpc_id?

```
module "vpc" {
  source = "terraform-and-modules/vpc/aws"
  cidr = "10.0.0.0/16"
  name = "test-vpc"
}
```

module.<MODULE_NAME>.vpc_id
Explanation:
vpc_id would be **module.<MODULE_NAME>.vpc_id**, replacing **<MODULE_NAME>** with the actual name of the module where the output is defined.

11. A terraform apply can not _____ infrastructure.
    a. change
    b. destroy
    c. provision
    d. import

Explanation: terraform apply can change, destroy, or provision infrastructure but it does not import existing infrastructure into Terraform's management. The import command is used separately to bring existing resources under Terraform's control.

12. Most Terraform providers interact with _____.
    a. API
    b. VCS Systems
    c. Shell scripts
    d. None of the above

Explanation: Terraform providers are interfaces to different infrastructure APIs (such as AWS, Azure, Google Cloud) allowing Terraform to manage resources by making API calls to these services.

13. A Terraform backend determines how Terraform loads state and stores updates when you execute _____.
    a. apply
    b. taint
    c. destroy
    d. All of the above
    e. None of the above

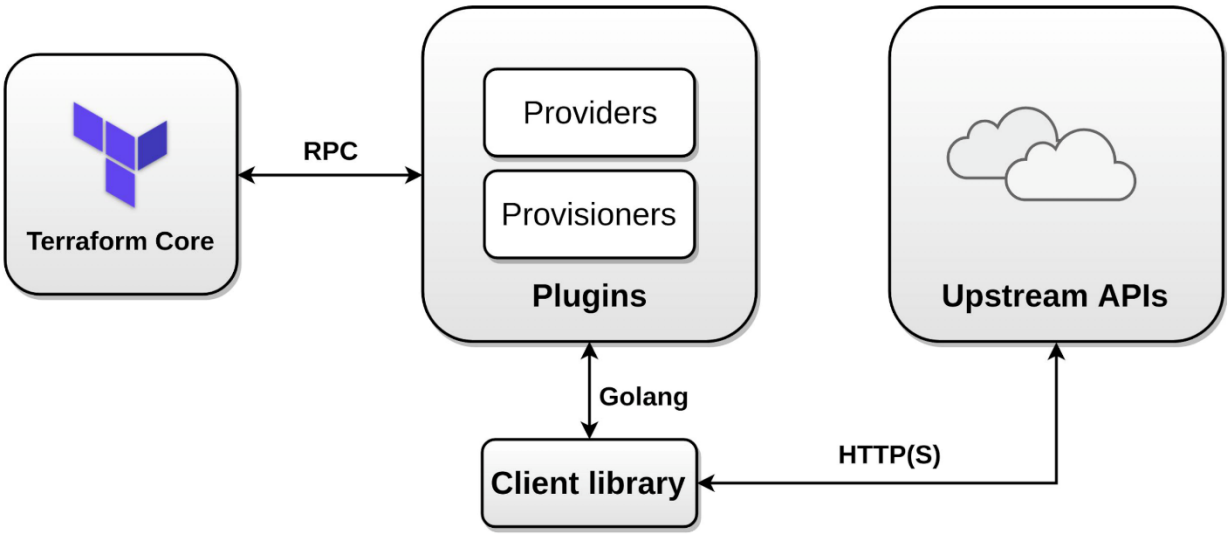
Explanation: A Terraform backend is responsible for managing the state, regardless of whether you're applying, tainting, or destroying resources.

Notes

Introduction

- An open source provisioning declarative tool that based on Infrastructure as a Code paradigm
- designed on immutable infrastructure principles
- Written in Golang and uses own syntax – HCL (Hashicorp Configuration Language), but also supports JSON
- Helps to evolve the infrastructure, safely and predictably
- Applies Graph Theory to IaaC and provides Automation, Versioning and Reusability
- Terraform is a multipurpose composition tool:
  - Composes multiple tiers (SaaS/PaaS/IaaS)
  - A plugin-based architecture model
- Terraform is not a cloud agnostic tool. It embraces all major Cloud Providers and provides common language to orchestrate the infrastructure resources
- Terraform is not a configuration management tool and other tools like chef, ansible exists in the market.

Terraform Architecture



Terraform Providers (Plugins)

- provide abstraction above the upstream API and is responsible for understanding API interactions and exposing resources.
- Invoke only upstream APIs for the basic CRUD operations
- Providers are unaware of anything related to configuration loading, graph theory, etc.
- supports multiple provider instances using alias *for e.g. multiple aws provides with different region*
- can be integrated with any API using providers framework

- Most providers configure a specific infrastructure platform (either cloud or self-hosted).
- can also offer local utilities for tasks like generating random numbers for unique resource names.

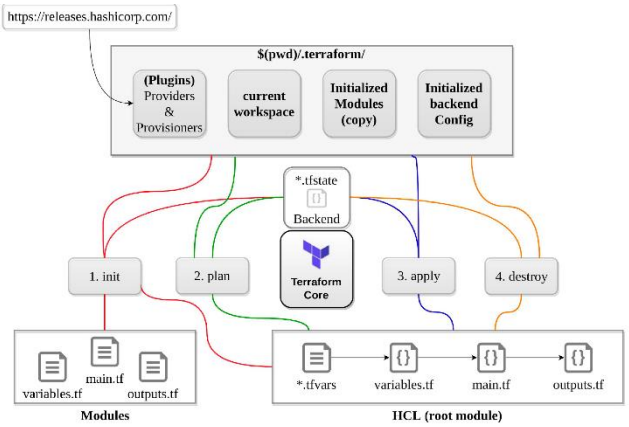
Terraform Provisioners

- run code locally or remotely on resource creation
    - ❖ local exec executes code on the machine running terraform
    - ❖ remote exec
        - ➢ runs on the provisioned resource
        - ➢ supports ssh and winrm
    - ❖ requires inline list of commands
- should be used as a last resort
- are defined within the resource block.
- support types – Create and Destroy
    - ❖ if creation time fails, resource is tainted if provisioning failed, by default. (next apply it will be re-created)
    - ❖ behavior can be overridden by setting the on_failure to continue, which means ignore and continue
    - ❖ for destroy, if it fails – resources are not removed

Terraform Workspaces

- helps manage multiple distinct sets of infrastructure resources or environments with the same code.
- just need to create needed workspace and use them, instead of creating a directory for each environment to manage
- state files for each workspace are stored in the directory terraform.tfstate.d
- terraform workspace new dev creates a new workspace and switches to it as well
- terraform workspace select dev helps select workspace
- terraform workspace list lists the workspaces and shows the current active one with *
- does not provide strong separation as it uses the same backend

Terraform Workflow



init

- initializes a working directory containing Terraform configuration files.
- performs
  - ❖ backend initialization , storage for terraform state file.
  - ❖ modules installation, downloaded from terraform registry to local path
  - ❖ provider(s) plugins installation, the plugins are downloaded in the sub-directory of the present working directory at the path of .terraform/plugins
- supports -upgrade to update all previously installed plugins to the newest version that complies with the configuration's version constraints
- is safe to run multiple times, to bring the working directory up to date with changes in the configuration
- does not delete the existing configuration or state

validate

- validates syntactically for format and correctness.
- is used to validate/check the syntax of the Terraform files.
- verifies whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state.
- A syntax check is done on all the terraform files in the directory, and will display an error if any of the files doesn't validate.

plan

- create a execution plan
- traverses each vertex and requests each provider using parallelism
- calculates the difference between the last-known state and
  the current state and presents this difference as the output of the terraform plan operation to user in their terminal
- does not modify the infrastructure or state.
- allows a user to see which actions Terraform will perform prior to making any changes to reach the desired state
- will scan all *.tf files in the directory and create the plan
- will perform refresh for each resource and might hit rate limiting issues as it calls provider APIs
- all resources refresh can be disabled or avoided using
  - ❖ -refresh=false or
  - ❖ target=xxxx or
  - ❖ break resources into different directories.
- supports -out to save the plan

apply

- apply changes to reach the desired state.
- scans the current directory for the configuration and applies the changes appropriately.
- can be provided with a explicit plan, saved as out from terraform plan
- If no explicit plan file is given on the command line, terraform apply will create a new plan automatically and prompt for approval to apply it
- will modify the infrastructure and the state.

- if a resource successfully creates but fails during provisioning,
    - ❖ Terraform will error and mark the resource as “tainted”.
    - ❖ A resource that is tainted has been physically created, but can’t be considered safe to use since provisioning failed.
    - ❖ Terraform also does not automatically roll back and destroy the resource during the apply when the failure happens, because that would go against the execution plan: the execution plan would’ve said a resource will be created, but does not say it will ever be deleted.
- does not import any resource.
- supports -auto-approve to apply the changes without asking for a confirmation
- supports -target to apply a specific module

refresh

- used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure
- does not modify infrastructure, but does modify the state file

destroy

- destroy the infrastructure and all resources
- modifies both state and infrastructure
- terraform destroy -target can be used to destroy targeted resources
- terraform plan -destroy allows creation of destroy plan

import

- helps import already-existing external resources, not managed by Terraform, into Terraform state and allow it to manage those resources
- Terraform is not able to auto-generate configurations for those imported modules, for now, and requires you to first write the resource definition in Terraform and then import this resource

taint

- marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.
- will not modify infrastructure, but does modify the state file in order to mark a resource as tainted. Infrastructure and state are changed in next apply.
- can be used to taint a resource within a module

fmt

- format to lint the code into a standard format

console

- command provides an interactive console for evaluating expressions.

Terraform Modules

- enables code reuse
- supports versioning to maintain compatibility
- stores code remotely
- enables easier testing

- enables encapsulation with all the separate resources under one configuration block
- modules can be nested inside other modules, allowing you to quickly spin up whole separate environments.
- can be referred using source attribute
- supports Local and Remote modules
    - ❖ Local modules are stored alongside the Terraform configuration (in a separate directory, outside of each environment but in the same repository) with source path ./ or ../
    - ❖ Remote modules are stored externally in a separate repository, and supports versioning
- supports following backends
    - ❖ Local paths
    - ❖ Terraform Registry
    - ❖ GitHub
    - ❖ Bitbucket
    - ❖ Generic Git, Mercurial repositories
    - ❖ HTTP URLs
    - ❖ S3 buckets
    - ❖ GCS buckets
- Module requirements
    - ❖ must be on GitHub and must be a public repo, if using public registry.
    - ❖ must be named terraform-<PROVIDER>-<NAME>, where <NAME> reflects the type of infrastructure the module manages and <PROVIDER> is the main provider where it creates that infrastructure. *for e.g. terraform-google-vault or terraform-aws-ec2-instance*.
    - ❖ must maintain x.y.z tags for releases to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a v *for example, v1.0.4 and 0.9.2*. Tags that don't look like version numbers are ignored.
    - ❖ must maintain a Standard module structure, which allows the registry to inspect the module and generate documentation, track resource usage, parse submodules and examples, and more.

Terraform Read and write configuration

sample.tf

```
// terraform settings
terraform {
  required_version = ">= 0.12.26" // ensure Terraform 0.12 or greater is used
}

# provider configuration
provider "aws" {
  region = "us-east-2"
}

// Local block - resuable values
locals {
  standard_tags = {
    environment = "prod"
  }
}

# defines the variable
variable "instance_type" {
  type    = string // data type, other supported are bool, number, list, map, object, tuple
  default = "t2.micro"
}

// data block - find the latest available centos AMI
data "aws_ami" "centos" {
  most_recent = true
  filter {
    name   = "name"
    values = ["CentOS Linux 7 x86_64 HVM EBS *"]
  }
}

// resource creating a private s3 bucket
resource "aws_s3_bucket" "example" {
  acl = "private"
}

resource "aws_instance" "local_name" {
  ami           = data.aws_ami.centos.id // ami from data
  instance_type = var.instance_type      // type from variable
  depends_on = [aws_s3_bucket.example] // explicit dependency
  // tags help define metata for the resource
  dynamic "tag" {
    for_each = local.standard_tags
    content {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}

// resource creating a elastic ip
resource "aws_eip" "ip" {
  vpc      = true
  instance = aws_instance.example_a.id // uses implicit dependencies by referencing the resosurce
}

// Output Values
output "aws_instance_eip" {
  value = aws_eip.eip // output value
}

output "aws_instance_private_ip" {
  value = aws_instance.local_name.private_ip
}
```

Resources

- resource are the most important element in the Terraform language that describes one or more infrastructure objects, such as compute instances etc
- resource type and local name together serve as an identifier for a given resource and must be unique within a module *for e.g. aws_instance.local_name*

Data Sources

- data allow data to be fetched or computed for use elsewhere in Terraform configuration
- allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another separate Terraform configuration

Variables

- variable serve as parameters for a Terraform module and act like function arguments
- allows aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations
- can be defined through multiple ways
    - ❖ command line for e.g.-var="image_id=ami-abc123"
    - ❖ variable definition files .tfvars or .tfvars.json. By default, terraform automatically loads
        - ➢ Files named exactly terraform.tfvars or terraform.tfvars.json.
        - ➢ Any files with names ending in .auto.tfvars or .auto.tfvars.json
        - ➢ file can also be passed with -var-file
    - ❖ environment variables can be used to set variables using the format TF_VAR_name

Environment variables

- terraform.tfvars file, if present.
- terraform.tfvars.json file, if present.
- Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.
- Any -var and -var-file options on the command line, in the order they are provided.Terraform loads variables in the following order, with later sources taking precedence over earlier ones:

Local Values

- locals assigns a name to an expression, allowing it to be used multiple times within a module without repeating it.
- are like a function's temporary local variables.
- helps to avoid repeating the same values or expressions multiple times in a configuration.

Output

- are like function return values.
- output can be marked as containing sensitive material using the optional sensitive argument, which prevents Terraform from showing its value in the list of outputs. However, they are still stored in the state as plain text.
- In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>.

Named Values

- is an expression that references the associated value for
  e.g. aws_instance.local_name, data.aws_ami.centos, var.instance_type etc.
- support Local named values for e.g count.index

Dependencies

- identifies implicit dependencies as Terraform automatically infers when one resource depends on another by studying the resource attributes used in interpolation expressions for e.g aws_eip on resource aws_instance
- explicit dependencies can be defined using depends_on where dependencies between resources that are *not* visible to Terraform

Data Types

- supports primitive data types of
  - ❖ string, number and bool
  - ❖ Terraform language will automatically convert number and bool values to string values when needed
- supports complex data types of
  - ❖ list – a sequence of values identified by consecutive whole numbers starting with zero.
  - ❖ map – a collection of values where each is identified by a string label.
  - ❖ set – a collection of unique values that do not have any secondary identifiers or ordering.
- supports structural data types of
  - ❖ object – a collection of named attributes that each have their own type
  - ❖ tuple – a sequence of elements identified by consecutive whole numbers starting with zero, where each element has its own type.

Built-in Functions

- includes a number of built-in functions that can be called from within expressions to transform and combine values for e.g. min, max, file, concat, element, index, lookup etc.
- does not support user-defined functions

Dynamic Blocks

- acts much like a for expression, but produces nested blocks instead of a complex typed value. It iterates over a given complex value, and generates a nested block for each element of that complex value.

Terraform Comments

- supports three different syntaxes for comments:
  - ❖ #
  - ❖ //
  - ❖ /* and */

Terraform Backends

- determines how state is loaded and how an operation such as apply is executed
- are responsible for storing state and providing an API for optional state locking
- needs to be initialized

- if switching the backed for the first time setup, Terraform provides a migration option
- helps
    - ❖ collaboration and working as a team, with the state maintained remotely and state locking
    - ❖ can provide enhanced security for sensitive data
    - ❖ support remote operations
- supports local vs remote backends
    - ❖ local (default) backend stores state in a local JSON file on disk
    - ❖ remote backend stores state remotely like S3, OSS, GCS, Consul and support features like remote operation, state locking, encryption, versioning etc.
- supports partial configuration with remaining configuration arguments provided as part of the initialization process
- Backend configuration doesn't support interpolations.
- GitHub is not the supported backend type in Terraform.

Terraform State Management

- state helps keep track of the infrastructure Terraform manages
- stored locally in the terraform.tfstate
- recommended not to edit the state manually
- Use terraform state command
    - mv – to move/rename modules
    - rm – to safely remove resource from the state. (destroy/retain like)
    - pull – to observe current remote state
    - list & show – to write/debug modules

State Locking

- happens for all operations that could write state, if supported by backend
- prevents others from acquiring the lock & potentially corrupting the state
- backends which support state locking are
    - azurerm
    - Hashicorp consul
    - Tencent Cloud Object Storage (COS)
    - etcdv3
    - Google Cloud Storage GCS
    - HTTP endpoints
    - Kubernetes Secret with locking done using a Lease resource
    - AliCloud Object Storage OSS with locking via TableStore
    - PostgreSQL
    - AWS S3 with locking via DynamoDB
    - Terraform Enterprise
- Backends which do not support state locking are
    - artifactory
    - etcd
- can be disabled for most commands with the -lock flag
- use force-unlock command to manually unlock the state if unlocking failed

State Security

- can contain sensitive data, depending on the resources in use for e.g passwords and keys
- using local state, data is stored in plain-text JSON files
- using remote state, state is held in memory when used by Terraform. It may be encrypted at rest, if supported by backend for e.g. S3, OSS

Terraform Logging

- debugging can be controlled using TF_LOG , which can be configured for different levels TRACE, DEBUG, INFO, WARN or ERROR, with TRACE being the more verbose.
- logs path can be controlled TF_LOG_PATH. TF_LOG needs to be specified.

Terraform Cloud and Terraform Enterprise

- Terraform Cloud provides Cloud Infrastructure Automation as a Service. It is offered as a multi-tenant SaaS platform and is designed to suit the needs of smaller teams and organizations. Its smaller plans default to one run at a time, which prevents users from executing multiple runs concurrently.
- Terraform Enterprise is a private install for organizations who prefer to self-manage. It is designed to suit the needs of organizations with specific requirements for security, compliance and custom operations.
- Terraform Cloud provides features
    - ❖ **Remote Terraform Execution** – supports Remote Operations for Remote Terraform execution which helps provide consistency and visibility for critical provisioning operations.
    - ❖ **Workspaces** – organizes infrastructure with workspaces instead of directories. Each workspace contains everything necessary to manage a given collection of infrastructure, and Terraform uses that content whenever it executes in the context of that workspace.
    - ❖ **Remote State Management** – acts as a remote backend for the Terraform state. State storage is tied to workspaces, which helps keep state associated with the configuration that created it.
    - ❖ **Version Control Integration –** is designed to work directly with the version control system (VCS) provider.
    - ❖ **Private Module Registry** – provides a private and central library of versioned & validated modules to be used within the organization
    - ❖ **Team based Permission System** – can define groups of users that match the organization's real-world teams and assign them only the permissions they need
    - ❖ **Sentinel Policies** – embeds the Sentinel policy-as-code framework, which lets you define and enforce granular policies for how the organization provisions infrastructure. Helps eliminate provisioned resources that don't follow security, compliance, or operational policies.
    - ❖ **Cost Estimation** – can display an estimate of its total cost, as well as any change in cost caused by the proposed updates
    - ❖ Security – encrypts state at rest and protects it with TLS in transit.
- Terraform Enterprise features
    - ❖ includes all the Terraform Cloud features with

    - **Audit** – supports detailed audit logging and tracks the identity of the user requesting state and maintains a history of state changes.
    - **SSO/SAML** – SAML for SSO provides the ability to govern user access to your applications.
- Terraform Enterprise currently supports running under the following operating systems for a Clustered deployment:
    - Ubuntu 16.04.3 – 16.04.5 / 18.04
    - Red Hat Enterprise Linux 7.4 through 7.7
    - CentOS 7.4 – 7.7
    - Amazon Linux
    - Oracle Linux
    - Clusters currently don't support other Linux variants.
- Terraform Cloud currently supports following VCS Provider
    - GitHub.com
    - GitHub.com (OAuth)
    - GitHub Enterprise
    - GitLab.com
    - GitLab EE and CE
    - Bitbucket Cloud
    - Bitbucket Server
    - Azure DevOps Server
    - Azure DevOps Services
- A Terraform Enterprise install that is provisioned on a network that does not have Internet access is generally known as an **air-gapped install**. These types of installs require you to pull updates, providers, etc. from external sources vs. being able to download them directly.