

**Question 1 Incorrect**

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`?

`aws_instance.bryan-demo["2"]`

**Explanation**

The resource address `aws_instance.bryan-demo["2"]` is not the correct format when using the `for_each` meta-argument in Terraform. The key-value pairs provided in the `for_each` block are used as keys to uniquely identify each instance, so the correct format includes the specific key, in this case, "vault".

**Your answer is incorrect**

`aws_instance.bryan-demo.vault`

**Explanation**

The resource address format `aws_instance.bryan-demo.vault` is not valid when using the `for_each` meta-argument in Terraform. The correct format includes the key enclosed in square brackets, such as `aws_instance.bryan-demo["vault"]`, to properly reference the instance.

[Continue](#) [Retake test](#)



`for_each` meta-argument in Terraform. The correct format includes the key enclosed in square brackets, such as `aws_instance.bryan-demo["vault"]`, to properly reference the instance related to "vault".

**Correct answer**

`aws_instance.bryan-demo["vault"]`

**Explanation**

The correct resource address format for the instance related to "vault" when using the `for_each` meta-argument in Terraform is `aws_instance.bryan-demo["vault"]`. This format allows Terraform to uniquely identify and manage each instance based on the key-value pair provided in the `for_each` block.

`aws_instance.bryan-demo[1]`

**Explanation**

The resource address `aws_instance.bryan-demo[1]` is not the correct format when using the `for_each` meta-argument in Terraform. The key-value pairs provided in the `for_each` block are used as keys to uniquely identify each instance, so the correct format includes the specific key, in this case, "vault".

**Overall 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:

```
1 | resource "aws_instance" "bryan-demo" {
```

[Continue](#) [Retake test](#)



#### Overall 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:

```
1 | resource "aws_instance" "bryan-demo" {
2 |   # ...
3 |   for_each = {
4 |     "terraform": "infrastructure",
5 |     "vault": "security",
6 |     "consul": "connectivity",
7 |     "nomad": "scheduler",
8 |   }
9 | }
```

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.

<https://developer.hashicorp.com/terraform/cli/v1.1.x/state/resource-addressing>

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

Continue Retake test

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

#### Question 2 Incorrect

What Terraform command will launch an interactive console to evaluate and experiment with expressions?

`terraform cmdline`

#### Explanation

The Terraform command "terraform cmdline" is not a valid command in Terraform. It does not exist in the Terraform documentation or official commands list.

`terraform cli`

#### Explanation

The Terraform command "terraform cli" is not a valid command in Terraform. It does not exist in the Terraform documentation or official commands list.

#### Correct answer

`terraform console`

#### Explanation

The correct Terraform command to launch the Interactive console is "terraform console". This command allows users to evaluate and experiment with expressions in an interactive manner.

Continue Retake test

Your answer is incorrect

**terraform get**

#### Explanation

The Terraform command "terraform get" is used to download and update modules defined in the configuration. It is not related to launching an interactive console for evaluating expressions.

#### Overall 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:

1. 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.
2. 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.
3. 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,

[Continue](#) [Retake test](#)



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.

<https://developer.hashicorp.com/terraform/cli/commands/console>

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

#### ! Question 3 Incorrect

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

4

#### Explanation

Using 4 spaces between each nesting level is not the recommended practice by HashiCorp for writing Terraform code.

**Correct answer**

2

#### Explanation

HashiCorp recommends using 2 spaces between each nesting level in Terraform code for better readability and maintainability.

[Continue](#) [Retake test](#)



○ 5

**Explanation**

Using 5 spaces between each nesting level is not the recommended practice by HashiCorp for writing Terraform code, as it deviates from the standard indentation guidelines provided by HashiCorp.

**Your answer is incorrect**

✗ 1

**Explanation**

Using only 1 space between each nesting level is not the recommended practice by HashiCorp for writing Terraform code, as it may lead to confusion and difficulty in understanding the code structure.

**Overall 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:

```
1 | resource "aws_instance" "example" {
2 |   ami           = "ami-0c55b159cbfafe1f0"
3 |   instance_type = "t2.micro"
4 |
5 |   tags = {
6 |     Name = "example-instance"
7 |   }
8 | }
```

Continue Retake test ⚙️ ↻

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.

[Check this link](#) for more information

**Domain**

Objective 3 - Understand Terraform Basics

✓ **Question 4 Correct**

Which are some of the benefits of using Infrastructure as Code in an organization? (select three)

**Your selection is correct**

- IaC uses a human-readable configuration language to help you write infrastructure code quickly

**Explanation**

IaC utilizes a human-readable configuration language, making it easier for developers and operators to understand, write, and maintain infrastructure code efficiently.

**Your selection is correct**

- IaC code can be used to manage infrastructure on multiple cloud platforms

**Explanation**

IaC code is platform-agnostic and can be used to manage infrastructure across various cloud platforms, such as AWS, Azure, Google Cloud, and others.

Continue Retake test ⚙️ ↻

IaC is written as an imperative approach, where specific commands need to be executed in the correct order

#### Explanation

This statement is incorrect. IaC follows a declarative approach, where the desired state of the infrastructure is defined, rather than specific commands to be executed in a particular order.

Your selection is correct

IaC allows you to commit your configurations to version control to safely collaborate on infrastructure

#### Explanation

Using Infrastructure as Code (IaC) allows for configurations to be stored in version control, enabling collaboration, tracking changes, and ensuring consistency in infrastructure deployment.

#### Overall 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

Continue Retake test



#### Incorrect Answer:

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 what happens. This often speeds things up because resources can be created/managed in parallel when there aren't any implicit or explicit dependencies.

<https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code>

<https://www.terraform.io/use-cases/infrastructure-as-code>

#### Domain

Objective 1 - Understand Infrastructure as Code Concepts

#### Question 5 Incorrect

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

Your answer is incorrect

TF\_DEBUG

#### Explanation

The environment variable 'TF\_DEBUG' is not used to enable detailed logging for Terraform. It is typically used for debugging purposes and not for detailed logging.

TF\_TRACE

#### Explanation

The environment variable 'TF\_TRACE' is not used to enable detailed logging for Terraform. It is typically used for tracing and providing additional information during execution.

Continue Retake test



`TF_TRACE`

#### Explanation

The environment variable `TF\_TRACE` is not used to enable detailed logging for Terraform. It is typically used for tracing and providing additional information during execution.

`Correct answer`

`TF_LOG`

#### Explanation

The correct environment variable to enable detailed logging for Terraform is `TF\_LOG`. Setting this variable will provide detailed logs for troubleshooting and debugging purposes.

`TF_INFO`

#### Explanation

The environment variable `TF\_INFO` is not used to enable detailed logging for Terraform. It is typically used to display informational messages and not for detailed logging purposes.

#### Overall 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.

<https://developer.hashicorp.com/terraform/internals/debugging>

[Continue](#) [Retake test](#)



#### Domain

Objective 8 - Read, Generate, and Modify Configuration

`Question 6 Correct`

True or False? 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.

`False`

#### Explanation

The statement is false because the `terraform plan -refresh-only` command is used to create a plan that only refreshes the state without updating it to match changes made to remote objects outside of Terraform. It does not update the Terraform state.

`Your answer is correct`

`True`

#### Explanation

The statement is true because the `terraform plan -refresh-only` command is specifically designed to only refresh the Terraform state to match any changes made to remote objects outside of Terraform. It does not apply those changes to the state.

#### Overall 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

[Continue](#) [Retake test](#)



**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`

<https://developer.hashicorp.com/terraform/cli/commands/plan#planning-modes>

<https://developer.hashicorp.com/terraform/cli/commands/refresh>

#### Domain

Objective 7 - Implement and Maintain State

#### Question 7 Incorrect

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

`terraform unlock`

#### Explanation

The Terraform command 'terraform unlock' does not exist in the Terraform documentation. Therefore, this command is not valid and cannot be used to remove the lock on the state for the current configuration.

Your answer is incorrect

`terraform state-unlock`

#### Explanation

The Terraform command 'terraform state-unlock' does not exist in the Terraform documentation. This command is not a valid command for removing the lock on the state file.

Correct answer

`terraform force-unlock`

#### Explanation

The correct Terraform command to remove the lock on the state for the current configuration is 'terraform force-unlock'. This command is specifically designed to force unlock the state file and allow modifications to be made.

Removing the lock on a state file is not possible

#### Explanation

It is possible to remove the lock on a state file in Terraform using the 'terraform force-unlock' command. This command allows the user to forcibly unlock the state file and make necessary changes.

#### Overall 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.

<https://developer.hashicorp.com/terraform/cli/commands/force-unlock>

Continue Retake test



Continue Retake test



**Domain**

Objective 7 - Implement and Maintain State

**Question 8 Correct**

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

```
terraform {  
    required_version = ">= 1.7.5"  
}
```

- versions before Terraform 1.7.5 were not approved by HashiCorp to be used in production

**Explanation**

This choice is incorrect because the snippet is focused on specifying the minimum required version of Terraform, not on the approval status of older versions by HashiCorp for production use.

- The user wants to ensure that the application being deployed is a minimum version of 1.7.5

**Explanation**

This choice is incorrect because the snippet is actually specifying the minimum version of Terraform required, not the version of the application being deployed.

- this ensures that all Terraform providers are above a certain version to match the application being deployed

[Continue](#) [Retake test](#)

**Explanation**

This choice is incorrect as the snippet specifically targets the Terraform version required to run the configuration, not the versions of Terraform providers.

Your answer is correct

- The user wants to specify the minimum version of Terraform that is required to run the configuration

**Explanation**

The snippet specifies the minimum version of Terraform required to run the configuration, ensuring compatibility and preventing potential issues that may arise from using older versions.

**Overall 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 1.7.5, you could include the following `terraform` block in your configuration to ensure that Terraform 1.7.5 or later is used:

```
1 | terraform {  
2 |   required_version = ">= 1.7.5"  
3 | }
```

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.

<https://developer.hashicorp.com/terraform/language/settings#specifying-a-required-terraform-version>

[Continue](#) [Retake test](#)

**Domain**

Objective 3 - Understand Terraform Basics

**Question 9 Correct**

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?

- The `.terraform.d` directory in the current working directory

**Explanation**

The `:terraform.d` directory is not the correct location for storing provider plugins. The correct directory for storing provider plugins is `.terraform/providers` within the current working directory.

[Continue](#) [Retake test](#)



Your answer is correct

The `.terraform/providers` directory in the current working directory

**Explanation**

The provider plugins are downloaded and stored in the `'.terraform/providers'` directory within the current working directory. This directory is specifically used by Terraform to manage provider plugins.

The `.terraform.plugins` directory in the current working directory

**Explanation**

The `'.terraform.plugins'` directory does not exist in Terraform's standard directory structure. The correct directory for storing provider plugins is `'.terraform/providers'` within the current working directory.

`/etc/terraform/plugins`

**Explanation**

The `'/etc/terraform/plugins'` directory is not the default location for storing provider plugins in Terraform. Provider plugins are typically stored in the `'.terraform/providers'` directory within the current working directory.

Continue Retake test  

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)



```

    aws_vpc_environment
      .terraform
        providers/registry.terraform.io/hashicorp/aws
          terraform-provider-aws_v3.74.1_x5 ←
          environment
      terraform.tfstate.d
  
```

<https://developer.hashicorp.com/terraform/plugin#installing-plugins>

**Domain**  
Objective 3 - Understand Terraform Basics

**Question 10 Incorrect**

Why might users want to utilize Sentinel or OPA with Terraform Cloud in their infrastructure workflow? (select four)

Your selection is correct

To provide real-time feedback on potential security risks in Terraform configurations during the development process

**Explanation**

Sentinel and OPA can provide real-time feedback on potential security risks present in Terraform configurations during the development process. This allows developers to identify and mitigate security issues early in the development cycle.

Continue Retake test  

#### Explanation

Sentinel and OPA can provide real-time feedback on potential security risks present in Terraform configurations during the development process. This allows developers to identify and address security issues early on, reducing the risk of vulnerabilities in the infrastructure.

Your selection is incorrect

- To allow users to bypass version control and directly apply changes to production

#### Explanation

Using Sentinel and OPA with Terraform Cloud does not allow users to bypass version control. Instead, it enforces policies and checks to ensure that changes comply with set standards before being applied to production. This helps in maintaining a secure and compliant infrastructure.

Your selection is correct

- Organizations can enforce resource naming conventions or approved machine images for improved consistency and clarity

#### Explanation

Organizations can use Sentinel and OPA to enforce resource naming conventions and approved machine images, which helps in maintaining consistency and clarity across the infrastructure. This can prevent naming conflicts and ensure that only approved resources are used.

Your selection is correct

- Sentinel and OPA enable automated policy checks to enforce compliance standards before applying changes to production environments

#### Explanation

Sentinel and OPA enable automated policy checks that can be used to enforce compliance

Continue Retake test



Correct selection

- Sentinel and OPA can enhance security by preventing unauthorized changes to your managed infrastructure

#### Explanation

Sentinel and OPA can enhance security by allowing users to define and enforce policies that prevent unauthorized changes to infrastructure. This helps in maintaining the integrity and security of the managed infrastructure.

#### Overall explanation

Using Sentinel and OPA with Terraform Cloud provides several benefits that enhance the overall management and security of your infrastructure. Using Sentinel with Terraform Cloud provides a powerful mechanism to enforce policies, increase security, and maintain compliance in your infrastructure deployments. It gives you greater control and confidence in managing your cloud resources while promoting best practices and reducing the risk of misconfiguration. Here are some specific reasons and examples on why you would use Sentinel with Terraform Cloud:

- Policy Enforcement:** Sentinel and OPA enable you to define and enforce policies that govern the configurations and changes made to your infrastructure. You can create custom policies tailored to your organization's needs, ensuring compliance with regulatory requirements, security best practices, and internal standards.
- Automated Governance:** With Sentinel and OPA, you can implement automated governance and compliance checks in your Terraform workflows. This means that every time changes are proposed or applied, Sentinel or OPA evaluates those changes against the defined policies, automatically preventing non-compliant configurations from being deployed.
- Enhanced Security:** By incorporating Sentinel and OPA into your Terraform Cloud environment, you can bolster your infrastructure security. Sentinel can flag and block any potentially risky configurations, helping to minimize security vulnerabilities and ensuring that only approved, secure changes are allowed.
- Version-controlled Policies:** Sentinel and OPA policies are defined as code, which means they can be stored in version control alongside your Terraform

Continue Retake test



4. **Version-controlled Policies:** Sentinel and OPA policies are defined as code, which means they can be stored in version control alongside your Terraform configurations. This allows your policies to be managed, reviewed, and updated through the same version control system, improving collaboration and maintaining a history of policy changes.
5. **Custom Approval Workflows:** You can create customized approval workflows based on policy conditions using Sentinel or OPA. This means that changes to the infrastructure can be automatically approved or flagged for manual review, depending on the defined policies, ensuring tighter control over infrastructure modifications.
6. **Preventing Costly Mistakes:** Sentinel and OPA policies can help catch potential mistakes or misconfigurations before they impact your infrastructure. By running policy checks in real-time, you can identify issues early on and avoid costly downtime or unexpected behavior caused by incorrect configurations.
7. **Consistency and Best Practices:** Utilizing Sentinel/OPA allows you to enforce consistent naming conventions, tagging standards, and other best practices across your infrastructure. This consistency leads to improved manageability and makes it easier for teams to collaborate effectively.
8. **Auditing and Compliance Reporting:** With Sentinel's logging and reporting capabilities, you can track policy decisions and changes made to your infrastructure over time. This audit trail is valuable for compliance purposes and can be used to demonstrate adherence to regulatory requirements.

**Wrong Answer:**

- Using Sentinel or OPA would NOT allow you to bypass version control. In fact, with TFC, your workspace would likely be configured to ONLY manage changes to your environment using code that is committed to a linked code repository.

<https://developer.hashicorp.com/terraform/cloud-docs/policy-enforcement>

**Domain**

Objective 9 - Understand Terraform Cloud Capabilities

 **Question 11 Correct**

True or False? Each Terraform workspace uses its own state file to manage the infrastructure associated with that particular workspace.

[Continue](#) [Retake test](#)  

**Domain**

Objective 9 - Understand Terraform Cloud Capabilities

 **Question 11 Correct**

True or False? Each Terraform workspace uses its own state file to manage the infrastructure associated with that particular workspace.

**Your answer is correct**

**True**

**Explanation**

Each Terraform workspace indeed uses its own state file to manage the infrastructure specific to that workspace. This separation of state files allows for isolation and independent management of resources within each workspace.

**False**

**Explanation**

This statement is incorrect. In Terraform, each workspace is designed to have its own state file to keep track of the infrastructure associated with that workspace. Sharing state files between workspaces can lead to conflicts and inconsistencies in managing resources.

**Overall 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.

[Continue](#) [Retake test](#)  

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....

<https://developer.hashicorp.com/terraform/language/state/workspaces#workspace-internals>

#### Domain

Objective 7 - Implement and Maintain State

#### ! Question 12 Incorrect

After many years of using Terraform Community (Free), 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?

Correct answer

- the same Terraform version that was used to perform the migration

#### Explanation

The new workspace in Terraform Cloud will be configured to use the same Terraform version that was used to perform the migration. This ensures compatibility and consistency with the existing state and configuration.

- the most recent version of Terraform available

#### Explanation

The new workspace will not automatically be configured to use the most recent version of Terraform available. It will retain the version used during the migration process to prevent any compatibility issues.

- the latest major release of Terraform

#### Explanation

The new workspace will not be configured to use the latest major release of Terraform automatically. It will stick to the version used during the migration to maintain stability and avoid any unexpected changes.

Your answer is incorrect

- whatever version is defined in the `terraform` block

#### Explanation

The new workspace will not be configured to use whatever version is defined in the 'terraform' block of the configuration. It will continue to use the version that was used during the migration process to ensure a smooth transition and consistent behavior.

#### Overall 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.

<https://developer.hashicorp.com/terraform/tutorials/cloud/cloud-versions>

Continue

Retake test



Continue

Retake test



**Domain**

Objective 9 - Understand Terraform Cloud Capabilities

**Question 13 Incorrect**

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

**Your answer is incorrect** **TypeScript****Explanation**

TypeScript is not a valid syntax for expressing Terraform configurations. Terraform primarily uses HCL and JSON for configuration files.

 **YAML****Explanation**

YAML is not a valid syntax for expressing Terraform configurations. Terraform primarily uses HCL and JSON for configuration files.

**Correct answer** **JSON****Explanation**

Terraform can be expressed in JSON syntax in addition to HCL. JSON is a popular choice for configuration files due to its simplicity and compatibility with various systems.

**Continue** **Retake test** **XML****Explanation**

XML is not a valid syntax for expressing Terraform configurations. Terraform primarily uses HCL and JSON for configuration files.

**Overall 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:

```
1 | # HCL syntax Example
2 | # terraform { }
3 |
4 |
5 | # JSON syntax Example
6 | {
7 |   "terraform": {}
8 | }
```

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.

<https://github.com/hashicorp/hcl/blob/main/hclsyntax/spec.md>

<https://developer.hashicorp.com/terraform/language/syntax/json>

**Continue** **Retake test**

**Domain**

Objective 3 - Understand Terraform Basics

**Question 14 Correct**

True or False? By default, the `terraform destroy` command will prompt the user for confirmation before proceeding.

 False**Explanation**

The statement is incorrect. In reality, the terraform destroy command does prompt the user for confirmation by default. This is to prevent unintended deletion of resources and to give the user a chance to review the actions that will be taken.

**Your answer is correct** True**Explanation**

By default, the terraform destroy command does prompt the user for confirmation before proceeding to ensure that the user is aware of the resources that will be deleted. This is a safety measure to prevent accidental deletion of important infrastructure.

**Overall 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.

**Continue** **Retake test****Domain**

Objective 3 - Understand Terraform Basics

**Question 14 Correct**

True or False? By default, the `terraform destroy` command will prompt the user for confirmation before proceeding.

 False**Explanation**

The statement is incorrect. In reality, the terraform destroy command does prompt the user for confirmation by default. This is to prevent unintended deletion of resources and to give the user a chance to review the actions that will be taken.

**Your answer is correct** True**Explanation**

By default, the terraform destroy command does prompt the user for confirmation before proceeding to ensure that the user is aware of the resources that will be deleted. This is a safety measure to prevent accidental deletion of important infrastructure.

**Overall 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.

**Continue** **Retake test**

Your selection is correct

GitHub.com

**Explanation**

Terraform Cloud supports integration with GitHub.com, allowing users to easily connect their Terraform Cloud workspace to their GitHub repositories for version control and collaboration.

Your selection is incorrect

CVS Version Control

**Explanation**

CVS Version Control is not a supported VCS provider for Terraform Cloud. Terraform Cloud does not offer integration with CVS repositories for version control and collaboration.

Your selection is correct

Bitbucket Cloud

**Explanation**

Bitbucket Cloud is another supported VCS provider for Terraform Cloud, enabling users to integrate their Bitbucket repositories with Terraform Cloud for seamless version control and collaboration.

Correct selection

Azure DevOps Server

**Explanation**

- - - - -

Correct selection

Azure DevOps Server

**Explanation**

Terraform Cloud supports integration with Azure DevOps Server, allowing users to connect their Azure DevOps repositories with Terraform Cloud for version control and collaboration in a Microsoft environment.

**Overall explanation**

Terraform Cloud supports the following VCS providers as of March 2024:

- [GitHub.com](#)
- [GitHub App for TFE](#)
- [GitHub.com \(OAuth\)](#)
- [GitHub Enterprise](#)
- [GitLab.com](#)
- [GitLab EE and CE](#)
- [Bitbucket Cloud](#)
- [Bitbucket Data Center](#)
- [Azure DevOps Server](#)
- [Azure DevOps Services](#)

<https://developer.hashicorp.com/terraform/cloud-docs/vcs#supported-vcs-providers>

**Domain**

Objective 9 - Understand Terraform Cloud Capabilities

 **Question 16 Correct**

What tasks can the `terraform state` command be used for in Terraform?

Continue Retake test  

Continue Retake test  

 Question 16 Correct

What tasks can the `terraform state` command be used for in Terraform?

- there is no such command

**Explanation**

The 'terraform state' command does exist in Terraform and can be used for various state management tasks. It is not accurate to say that there is no such command.

Your answer is correct

- modify the current state, such as removing items

**Explanation**

The 'terraform state' command can indeed be used to modify the current state by removing items. This is useful for managing the state of resources in Terraform.

- create a new state file

**Explanation**

The 'terraform state' command is not used to create a new state file. State files are typically created and managed automatically by Terraform during the execution of the infrastructure code.

- refresh the existing state

**Explanation**

The 'terraform state' command is not used to refresh the existing state. The 'terraform

**Explanation**

The 'terraform state' command is not used to refresh the existing state. The 'terraform refresh' command is specifically designed for that purpose.

[Continue](#) [Retake test](#)



**Overall 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:

1. 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.
2. 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.
3. 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.
4. 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.

<https://developer.hashicorp.com/terraform/cli/commands/state/list>

<https://developer.hashicorp.com/terraform/cli/state>

**Domain**

Objective 4 - Use Terraform Outside of Core Workflow

[Continue](#) [Retake test](#)



✓ **Question 17 Correct**

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

**number**

**Explanation**

The variable type 'number' is a valid type in Terraform. Numbers are used to represent numerical values in Terraform configurations.

**bool**

**Explanation**

The variable type 'bool' is a valid type in Terraform. Booleans are used to represent true or false values in Terraform configurations.

**map**

**Explanation**

The variable type 'map' is a valid type in Terraform. Maps are used to define key-value pairs in Terraform configurations.

**Your answer is correct**

**Continue** **Retake test**  

**Your answer is correct**

**float**

**Explanation**

In Terraform, the variable type 'float' is not a valid type. Terraform supports variable types such as 'string', 'map', 'bool', and 'number', but not 'float'.

**string**

**Explanation**

The variable type 'string' is a valid type in Terraform. Strings are commonly used for representing text values in Terraform configurations.

**Overall 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)

```
1 | variable "practice-exam" {  
2 |   description = "bryan's terraform associate practice exams"  
3 |   type        = string  
4 |   default     = "highly-rated"  
5 | }
```

<https://developer.hashicorp.com/terraform/language/expressions/types>

**Continue** **Retake test**  

 Question 18 Correct

Harry has deployed resources on Azure 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 is company policy to manage production workloads using IaC, how can Harry bring these resources under Terraform management without negatively impacting the availability of the deployed resources?

- run a `terraform get` to retrieve other resources that are not under Terraform management

**Explanation**

Running '`terraform get`' is used to retrieve modules from remote repositories, not to bring existing resources under Terraform management. This command is not suitable for the scenario where Harry needs to manage resources created outside of Terraform.

- 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.

**Explanation**

Rewriting the Terraform configuration file to deploy new resources and manually deleting the resources created by Ron and Ginny is not a recommended approach as it involves unnecessary manual intervention and potential data loss. This method does not align with the best practice of managing resources using IaC.

Your answer is correct

- use `terraform import` or the `import` block to import the existing resources under Terraform management

**Explanation**

Using '`terraform import`' or the '`import`' block allows Harry to bring the existing resources

[Continue](#) [Retake test](#)  

**Explanation**

Using '`terraform import`' or the '`import`' block allows Harry to bring the existing resources under Terraform management without disrupting the availability of the deployed resources. This method ensures that the resources are managed by Terraform while preserving their current state.

- resources created outside of Terraform cannot be managed by Terraform

**Explanation**

The statement that resources created outside of Terraform cannot be managed is incorrect. Terraform provides the capability to import existing resources and manage them through the use of '`terraform import`' or the '`import`' block. This choice does not align with the functionality provided by Terraform for managing existing resources.

**Overall 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:

1. 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. You can also create an `import` block to pull in resources under Terraform management.
2. 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.
3. 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

[Continue](#) [Retake test](#)  

3. 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.

4. 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.*

<https://developer.hashicorp.com/terraform/language/import>

<https://developer.hashicorp.com/terraform/cli/commands/import>

#### Domain

Objective 4 - Use Terraform Outside of Core Workflow

#### ✓ Question 19 Correct

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?

`consul`

#### Explanation

The "consul" backend type is a supported backend type in Terraform. Consul is a popular choice

`consul`

#### Explanation

The "consul" backend type is a supported backend type in Terraform. Consul is a popular choice for storing Terraform state.

`local`

#### Explanation

The "local" backend type is a supported backend type in Terraform. It stores the state file on the local disk where Terraform is being run.

Your answer is correct

`github`

#### Explanation

The "github" backend type is not a supported backend type in Terraform. Terraform does not have built-in support for storing state in a GitHub repository.

`s3`

#### Explanation

The "s3" backend type is a supported backend type in Terraform. It allows storing the state file in an Amazon S3 bucket.

#### Overall explanation

Continue Retake test

Continue Retake test

#### Explanation

The "s3" backend type is a supported backend type in Terraform. It allows storing the state file in an Amazon S3 bucket.

#### Overall 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

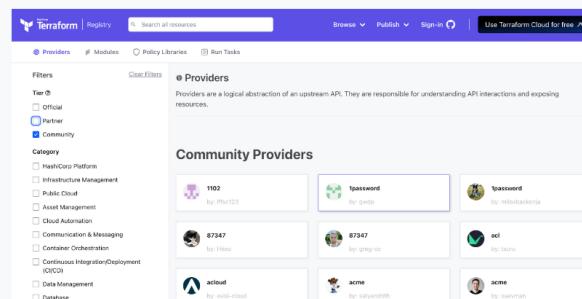
Continue Retake test  

#### Domain

Objective 7 - Implement and Maintain State

#### Question 20 Correct

True or False? Using the latest versions of Terraform, `terraform init` cannot automatically download community providers.



The screenshot shows the Terraform Registry interface. In the top navigation bar, there are links for 'Terraform', 'Registry', 'Search providers...', 'Browse', 'Publish', 'Sign-in', and 'Use Terraform Cloud for free'. Below the navigation, there are tabs for 'Providers', 'Modules', 'Policy Libraries', and 'Run Tasks'. Under the 'Providers' tab, there are sections for 'Filters' (with 'Tier' dropdown set to 'Official') and 'Providers' (with a sub-section for 'Community'). The 'Community Providers' section displays a grid of provider icons and names, including '1102' (by mhetzler), 'spassword' (by spwitt), 'password' (by mhetzler), '87347' (by hclio), '87347' (by greg-rc), 'acl' (by hclio), 'awscloud' (by awscloud), 'acme' (by svenfriis), and 'acme' (by ryanman).

Your answer is correct

False

#### Explanation

The statement "False" is correct because using the latest versions of Terraform, the command `terraform init` can automatically download community providers. This functionality simplifies the process of integrating community providers into Terraform configurations, enhancing the overall user experience.

Continue Retake test  

**Explanation**

The statement "False" is correct because using the latest versions of Terraform, the command 'terraform init' can automatically download community providers. This functionality simplifies the process of integrating community providers into Terraform configurations, enhancing the overall user experience.

True

**Explanation**

The statement "True" is incorrect because using the latest versions of Terraform, the command 'terraform init' can automatically download community providers. This feature allows users to easily access and utilize community-created providers without manual intervention.

**Overall 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.

<https://www.hashicorp.com/blog/automatic-installation-of-third-party-providers-with-terraform-0-13>

[Continue](#) [Retake test](#)

**Domain**

Objective 3 - Understand Terraform Basics

**!** Question 21 Incorrect

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

- 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**

The information about maintaining resource dependencies in a file called 'resource.dependencies' and the location in the 'terraform.d' folder is inaccurate. Terraform does not require providers to maintain a separate file for dependencies.

**Correct answer**

- 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.

**Explanation**

Terraform automatically analyzes expressions within a resource block to identify dependencies on other resources. This allows Terraform to determine the correct order of operations when creating, updating, or destroying resources.

The Terraform binary contains a built-in reference map of all defined Terraform

[Continue](#) [Retake test](#)



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.

#### Explanation

The statement regarding a built-in reference map of resource dependencies and the need to update Terraform versions is incorrect. Terraform does not rely on a static map of dependencies within the binary.

Your answer is incorrect

Resource dependencies are handled automatically by the `depends_on` meta-argument, which is set to true by default.

#### Explanation

Resource dependencies are not handled by setting the 'depends\_on' meta-argument to true by default. Terraform uses implicit ordering requirements based on expressions within resource blocks.

#### Overall 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.

<https://developer.hashicorp.com/terraform/language/resources>

Continue Retake test  

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

#### Question 22 Correct

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 = "5.7.0"

  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
}
```

these are where the `variable declarations` are created so Terraform is aware of these variables within the calling module

#### Explanation

The declarations in the code are not where variable declarations are created. They are actually instances where the variables are being assigned values passed from the calling module. Variable declarations are typically defined in separate variable files or within the same module.

these are the `outputs` that the child module will return

Continue Retake test  

- these are the **outputs** that the child module will return

#### Explanation

The declarations in the Terraform code are not outputs that the child module will return. Outputs are used to expose certain values to other parts of the Terraform configuration, whereas these declarations are used as input variables.

Your answer is correct

- these are **variables** that are passed into the child module likely used for resource creation

#### Explanation

The declarations like `name`, `cidr`, and `azs` are variables that are being passed into the child module for resource creation. These variables allow for customization and flexibility in configuring the VPC module according to specific requirements.

- the **value** of these variables will be obtained from values created within the child module

#### Explanation

The values of the variables like name, cidr, and azs are not obtained from values created within the child module. Instead, these variables are typically defined and assigned values in the calling module or through input variables provided during the Terraform execution.

#### Overall 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.

Continue Retake test  

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

For more information on Terraform modules and input variables, I recommend checking out the Terraform documentation: <https://www.terraform.io/docs/modules/index.html>

<https://learn.hashicorp.com/tutorials/terraform/module-use#set-values-for-module-input-variables>

#### Domain

Objective 5 - Interact with Terraform Modules

#### ! Question 23 Incorrect

You are using a Terraform Cloud workspace linked to a GitHub repo to manage production workloads in your environment. After approving a merge request, what `default` action can you expect to be triggered on the workspace?

Correct answer

- A speculative plan will be run to show the potential changes to the managed environment and validate the changes against any applicable Sentinel policies

#### Explanation

After approving a merge request, Terraform Cloud will run a speculative plan to show the potential changes that will be applied to the managed environment. This allows users to review

Continue Retake test  

potential changes that will be applied to the managed environment. This allows users to review and validate the changes against any applicable Sentinel policies before applying them.

- The workspace will trigger a set of tests, such as `terratest` and `terraform validate`, to ensure the code is valid and can be successfully executed by the specific version of Terraform configured for the workspace.

#### Explanation

Triggering tests like terratest and terraform validate to ensure code validity and successful execution is not the default action after approving a merge request in a Terraform Cloud workspace linked to a GitHub repo. The default action is running a speculative plan.

- The workspace will run a speculative plan and automatically apply the changes without any required interaction from the user

#### Explanation

The workspace will not automatically apply the changes without user interaction. Instead, a speculative plan will be run to show the potential changes, and the user will need to manually apply the changes after reviewing them.

Your answer is incorrect

- Terraform Cloud will automatically execute a `terraform destroy` operation on your production workloads, and apply the new committed code stored in the GitHub repo

#### Explanation

Automatically executing a terraform destroy operation on production workloads and applying new code stored in the GitHub repo is not the default action triggered on the workspace after approving a merge request. The default action is running a speculative plan to show potential changes.

Continue Retake test  

#### Overall explanation

After approving a merge request that modifies Terraform configurations in a GitHub repository linked to a Terraform Cloud workspace, the default action that can be expected to run automatically is a "**speculative plan**" operation.

When you merge a pull request or push changes to the main branch (or any branch you have configured as the trigger for the workspace), Terraform Cloud typically triggers a plan operation. During this plan phase, Terraform examines the proposed changes to your infrastructure and displays a list of actions it would take if applied. It's a way to preview the changes before actually making them.

The plan output shows what resources Terraform would create, modify, or delete, which allows you to review and validate the expected changes. After reviewing the plan, you can then manually apply the changes to your infrastructure through the Terraform Cloud workspace.

Note: You can absolutely configure a Terraform workspace to automatically apply the changes to the code, although that is generally not recommended, **nor is it the default action**.

#### Wrong Answers:

- TFC does not automatically run a speculative plan and apply the changes unless you specifically configure the workspace to do so. This is not the default action that would be triggered when you commit new code to the repo
- TFC does not run external tests, such as `terratest` and `terraform validate`, on your code when you commit it to a repo
- TFC, or Terraform in general, does not destroy managed infrastructure when executing a plan and apply. It will only modify the resources needed to ensure the managed restructure now matches the desired state.

<https://developer.hashicorp.com/terraform/cloud-docs/run/remote-operations>

#### Domain

Objective 9 - Understand Terraform Cloud Capabilities

#### Question 24 Correct

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

Continue Retake test  

wants to create these new resources using modules so she easily reuse the code. 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?

Your answer is correct

- add an `output` block to the `subnet` module and retrieve the value using `module.subnet.subnet_id` for the `load balancer` module

**Explanation**

Adding an output block to the subnet module allows the `subnet_id` to be exposed as an output variable. This output variable can then be retrieved using `module.subnet.subnet_id` in the load balancer module, enabling Sara to pass the `subnet_id` between modules.

- references to resources that are created within a module cannot be used within other modules

**Explanation**

Contrary to this statement, references to resources created within a module can be used in other modules by defining output variables. Therefore, this choice is incorrect as it provides inaccurate information regarding the usability of resources created within modules.

- publish the module to a Terraform registry first

**Explanation**

Publishing the module to a Terraform registry is not a solution to the problem of passing the `subnet_id` between modules. While it may be a good practice for sharing modules with others, it does not address the immediate issue Sara is experiencing.

- move the `subnet` and `load balancer` resource into the main configuration file so they can easily be referenced

**Explanation**

Moving the subnet and load balancer resources into the main configuration file may make them easier to reference, but it does not address the specific issue of passing the `subnet_id` from the subnet module to the load balancer module. This solution does not fix the problem Sara is facing.

**Overall 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.

<https://learn.hashicorp.com/tutorials/terraform/module-use#define-root-output-values>

**Domain**

Objective 5 - Interact with Terraform Modules

Continue Retake test



Continue Retake test



**!** Question 25 Incorrect

Which of the following statements represents the most accurate statement about the Terraform language?

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

**Explanation**

This statement is incorrect. Terraform is not a mutable language and it is not imperative. It is an immutable, declarative Infrastructure as Code provisioning language based on HashiCorp Configuration Language (HCL) or JSON. YAML is not the primary language used for Terraform configuration files.

**Correct answer**

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

**Explanation**

Terraform is indeed an immutable and declarative Infrastructure as Code provisioning language. It allows users to define the desired state of their infrastructure and Terraform will make the necessary changes to reach that state. The language is based on HashiCorp Configuration Language (HCL) or JSON for configuration files.

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

**Explanation**

This statement is inaccurate. Terraform is not an imperative language; it is declarative. Users define the desired state of their infrastructure without specifying the exact steps to achieve that state. This allows Terraform to determine the most efficient way to make changes.

[Continue](#) [Retake test](#)



**Your answer is incorrect**

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

**Explanation**

This statement is incorrect. Terraform is not a mutable language; it is immutable. It focuses on defining the desired state of infrastructure and making changes to reach that state, rather than directly modifying existing infrastructure.

**Overall 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.

**Incorrect Answers:**

Terraform is not a configuration management tool - <https://developer.hashicorp.com/terraform/intro/vs/chef-puppet>

**Domain**

Objective 1 - Understand Infrastructure as Code Concepts

**!** Question 26 Incorrect

In the following code snippet, the type of Terraform `block` is identified by which string?

[Continue](#) [Retake test](#)



1 Question 26 Incorrect

In the following code snippet, the type of Terraform **block** is identified by which string?

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

**instance\_type**

**Explanation**

The string "instance\_type" is not the type of Terraform block in this code snippet. It is the attribute key used to specify the instance type for the AWS instance resource.

**Correct answer**

**resource**

**Explanation**

The type of Terraform block is identified by the keyword "resource" in this code snippet. This keyword indicates that a new AWS instance resource is being defined.

**t2.micro**

**Explanation**

The string "t2.micro" is not the type of Terraform block in this code snippet. It is the value assigned to the instance\_type attribute of the AWS instance resource.

[Continue](#) [Retake test](#)



**Your answer is incorrect**

**db**

**Explanation**

The string "db" is not the type of Terraform block in this code snippet. It is the name given to this specific AWS instance resource block.

**Overall explanation**

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

The format of a resource block configuration is as follows:

```
1 | resource "TYPE" "NAME" {
2 |   [CONFIGURATION_KEY = CONFIGURATION_VALUE]
3 |   ...
4 | }
```

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

[Continue](#) [Retake test](#)



#### CONFIGURATION\_KEY

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

```
1 | resource "aws_instance" "example" {
2 |   ami           = "ami-0323c3dd2da7fb37d"
3 |   instance_type = "t2.micro"
4 | }
```

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.

<https://developer.hashicorp.com/terraform/language/resources>

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

##### Question 27 Incorrect

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

place a space in between each line

1 type = "A"  
2  
3 ttl = "300"

Continue Retake test

##### Question 27 Incorrect

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

place a space in between each line

1 type = "A"  
2  
3 ttl = "300"  
4  
5 zone\_id = aws\_route53\_zone.primary.zone\_id

#### Explanation

HashiCorp does not recommend placing a space in between each line for consecutive arguments with single-line values at the same nesting level. This practice may introduce unnecessary whitespace and reduce code compactness.

put arguments in alphabetical order

1 name = "www.example.com"  
2 records = [aws\_eip.lb.public\_ip]  
3 type = "A"  
4 ttl = "300"  
5 zone\_id = aws\_route53\_zone.primary.zone\_id

#### Explanation

Putting arguments in alphabetical order is not a recommendation from HashiCorp. It may not provide any significant benefit in terms of readability or maintainability of the Terraform configuration.

Correct answer

Continue Retake test

**Correct answer**

**align the equals signs**



```
1 | ami           = "abc123"
2 | instance_type = "t2.micro"
```

**Explanation**

HashiCorp recommends aligning the equals signs for better readability and consistency in the code. This helps in quickly identifying and understanding the key-value pairs in the configuration.

**Your answer is incorrect**

**place all arguments using a variable at the top**



```
1 | ami = var.aws_ami
2 | instance_type = var.instance_size
3 | subnet_id = "subnet-0bb1c79de3EXAMPLE"
4 | tags = {
5 |   Name = "HelloWorld"
6 | }
```

**Explanation**

Placing all arguments using a variable at the top is not a recommendation from HashiCorp. While using variables can improve code maintainability, there is no specific guideline to place all arguments using variables at the top.

**Overall explanation**

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

[Continue](#)

[Retake test](#)



**Correct answer**

**align the equals signs**



```
1 | ami           = "abc123"
2 | instance_type = "t2.micro"
```

**Explanation**

HashiCorp recommends aligning the equals signs for better readability and consistency in the code. This helps in quickly identifying and understanding the key-value pairs in the configuration.

**Your answer is incorrect**

**place all arguments using a variable at the top**



```
1 | ami = var.aws_ami
2 | instance_type = var.instance_size
3 | subnet_id = "subnet-0bb1c79de3EXAMPLE"
4 | tags = {
5 |   Name = "HelloWorld"
6 | }
```

**Explanation**

Placing all arguments using a variable at the top is not a recommendation from HashiCorp. While using variables can improve code maintainability, there is no specific guideline to place all arguments using variables at the top.

**Overall explanation**

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

[Continue](#)

[Retake test](#)



**Explanation**

This code snippet defines a data block for retrieving information about an AWS AMI (Amazon Machine Image) based on specific criteria like owners and tags. This data can then be used within the Terraform configuration to make decisions or set attributes based on the retrieved information.

```
1 module "deploy-servers" {
2   source = "./app-cluster"
3
4   servers = 5
5 }
```

**Explanation**

This code snippet defines a module block for deploying servers from a specified source. While modules are useful for organizing and reusing Terraform configurations, this snippet does not focus on retrieving information about existing resources for use in the configuration.

```
1 locals {
2   service_name = "forum"
3   owner        = "Community Team"
4 }
```

**Explanation**

This code snippet defines local values for service\_name and owner, which can be used for storing and reusing values within the Terraform configuration. However, it does not directly involve retrieving information about existing resources for use in the configuration.

```
1 provider "google" {
2   project = "acme-app"
3   region  = "us-central1"
4 }
```

**Explanation**

This code snippet defines a provider block for Google Cloud Platform, specifying the project and region. While providers are essential for interacting with cloud platforms, this snippet does not directly address retrieving information about existing resources for use in Terraform configuration.

```
1 resource "aws_instance" "web" {
2   ami           = "ami-1a1b2c3d4"
3   instance_type = "t2.micro"
4 }
```

**Explanation**

This code snippet defines a resource block for creating an AWS EC2 instance with specific attributes like AMI and instance type. While creating new resources is a common task in Terraform, this snippet does not address retrieving information about existing resources for use in the configuration.

**Overall 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:

```
1 data "aws ami" "example" {
2   most_recent = true
3
4   filter {
5     name  = "name"
6     values = ["amzn2-ami-hvm-2.0.*-x86_64-gp2"]
7   }
8 }
```

[Continue](#) [Retake test](#)



[Continue](#) [Retake test](#)



```

4   filter {
5     name  = "name"
6     values = ["amzn2-ami-hvm-2.0.*-x86_64-gp2"]
7   }
8
9   filter {
10    name  = "virtualization-type"
11    values = ["hvm"]
12  }
13 }
14
15 resource "aws_instance" "example" {
16   ami           = data.aws_ami.example.id
17   instance_type = "t2.micro"
18 }

```

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.

<https://developer.hashicorp.com/terraform/language/data-sources>

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

#### Question 29 Incorrect

Which of the following is a `valid` variable name in Terraform?

Your answer is incorrect

`version`

[Continue](#) [Retake test](#)



#### Question 29 Incorrect

Which of the following is a `valid` variable name in Terraform?

Your answer is incorrect

`version`

#### Explanation

"version" is not a valid variable name in Terraform as it is a reserved keyword

`lifecycle`

#### Explanation

"lifecycle" is not a valid variable name in Terraform as it is a reserved keyword used for defining resource lifecycle configuration and cannot be used as a variable name.

`invalid`

#### Explanation

This is a valid variable name in Terraform as it follows the naming conventions for variables, which allow alphanumeric characters and underscores, and must start with a letter or underscore.

`count`

#### Explanation

[Continue](#) [Retake test](#)



**count**

#### Explanation

"count" is not a valid variable name in Terraform as it is a reserved keyword used for resource iteration and cannot be used as a variable name.

#### Overall 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 `source`, `version`, `providers`, `count`, `for_each`, `lifecycle`, `depends_on`, `locals`.

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.

<https://developer.hashicorp.com/terraform/tutorials/configuration-language/count>

<https://developer.hashicorp.com/terraform/language/values/variables#declaring-an-input-variable>

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

#### ❗ Question 30 Incorrect

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

[Continue](#) [Retake test](#)  

```
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]  
}
```

The **AMI** used for the EC2 instance

#### Explanation

The AMI used for the EC2 instance is not an implicit dependency in this code snippet, as the dependency is explicitly defined using the `depends_on` attribute for the `aws_instance` resource.

The EIP with an id of **ami-2757f631**

#### Explanation

The EIP with an id of "ami-2757f631" is not an implicit dependency in this code snippet, as the EIP resource is dependent on the EC2 instance, not on the specific EIP id mentioned in the code.

**Your answer is incorrect**

The S3 bucket labeled **company\_data**

#### Explanation

The implicit dependency in the code is not the S3 bucket labeled "company\_data", as there is no direct relationship between the `aws_instance` and the `aws_s3_bucket` in the code provided.

[Continue](#) [Retake test](#)  

Your answer is incorrect

The S3 bucket labeled `company_data`

#### Explanation

The implicit dependency in the code is not the S3 bucket labeled "company\_data", as there is no direct relationship between the `aws_instance` and the `aws_s3_bucket` in the code provided.

Correct answer

The EC2 instance labeled `web_server`

#### Explanation

The implicit dependency in the code is the EC2 instance labeled "web\_server" because the `aws_eip` resource depends on the `aws_instance.web_server.id` for its `instance` attribute.

#### Overall 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_lambda_function`.

Continue Retake test



#### Domain

Objective 8 - Read, Generate, and Modify Configuration

#### Question 31 Correct

True or False? 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.

True

#### Explanation

This choice is incorrect. Terraform relies on state files to track the current state of infrastructure resources and compare it to the desired state declared in configuration files. Without a state file, Terraform would not be able to accurately determine the changes needed to align real-world resources with the desired state.

Your answer is correct

False

#### Explanation

This choice is correct. Terraform requires a state file to store information about the current state of infrastructure resources. By inspecting this state file, Terraform can determine the necessary changes to bring the real-world resources in line with the desired state specified in the configuration files. Without a state file, Terraform would not be able to perform this validation.

#### Overall explanation

**State is a necessary requirement for Terraform to function.** And in the scenarios where Terraform may be able to run even without state, doing so would require shifting resources

Continue Retake test



#### Overall 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.

<https://developer.hashicorp.com/terraform/language/state/purpose>

#### Domain

Objective 2 - Understand Terraform's Purpose (vs other IaC)

#### ! Question 32 Incorrect

True or False? When using the Terraform provider for Vault, the tight integration between these HashiCorp tools provides the ability to mask secrets in the state file.

Your answer is incorrect

True

#### Explanation

False. The Terraform provider for Vault does not inherently provide the ability to mask secrets in the state file. It is important to handle sensitive information securely in Terraform configurations.

Correct answer

False

[Continue](#) [Retake test](#)



#### Explanation

Correct. The statement is false because the tight integration between Terraform and Vault does not automatically mask secrets in the state file. Developers need to implement secure practices to handle secrets effectively.

#### Overall 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 to store sensitive information or use Terraform's data sources to retrieve the information from the environment 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 configuration files.

<https://learn.hashicorp.com/tutorials/terraform/secrets-vault>

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

#### ! Question 33 Incorrect

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

Correct answer

```
1 | variable "public_subnet_cidr_blocks" {  
2 |   type = list(string)  
3 |   default = [  
4 |     "10.0.1.0/24",  
5 |     "10.0.1.0/24",  
6 |     "10.0.1.0/24",  
7 |     "10.0.1.0/24",  
8 |   ]
```

[Continue](#) [Retake test](#)



**Correct answer**

```
1 | variable "public_subnet_cidr_blocks" {  
2 |   type = list(string)  
3 |   default = [  
4 |     "10.0.1.0/24",  
5 |     "10.0.1.8/24",  
6 |     "10.0.1.16/24",  
7 |     "10.0.1.0/24",  
8 |   ]  
9 | }
```

**Explanation**

The syntax provided correctly defines a list of strings for the variable "public\_subnet\_cidr\_blocks" in Terraform. The values are enclosed in double quotes and separated by commas within square brackets.

```
1 | variable "public_subnets" {  
2 |   description = "The number of public subnets for VPC"  
3 |   type        = list(string)  
4 |   default     = 2  
5 | }
```

**Explanation**

The syntax provided defines a variable for the number of public subnets, not a list of strings. It is not the correct syntax for defining a list of strings in Terraform.

```
1 | variable "resource_tags" {  
2 |   description = "Tags to set for all resources"  
3 |   type        = list(string)  
4 |   default     = {  
5 |     project    = "exam-prep",  
6 |     environment = "prod"  
7 |     instructor = "krausen"  
8 |   }  
9 | }
```

**Explanation**

The syntax provided defines a map of key-value pairs, not a list of strings. It is not the correct syntax for defining a list of strings in Terraform.

**Your answer is incorrect**

```
1 | variable "aws_region" {  
2 |   description = "AWS region"  
3 |   type        = list(string)  
4 |   default     = "us-west-1"  
5 | }
```

**Explanation**

The syntax provided defines a variable for the AWS region as a single string value, not a list of strings. It is not the correct syntax for defining a list of strings in Terraform.

**Overall 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:

1. 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:

```
1 | variable "example_list" {  
2 |   type = list(string)  
3 | }
```

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

```
1 | variable "example_list" {  
2 |   type = list(string)  
3 |   default = ["string1", "string2", "string3"]  
4 | }
```

[Continue](#) [Retake test](#)



[Continue](#) [Retake test](#)



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

```
1 | variable "example_list" {
2 |   type = list(string)
3 |   default = ["string1", "string2", "string3"]
4 | }
```

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.

<https://developer.hashicorp.com/terraform/tutorials/configuration-language/variables#list-public-and-private-subnets>

<https://developer.hashicorp.com/terraform/language/expressions/types#list>

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

#### Question 34 Incorrect

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`?

- Henry should run a `terraform plan -refresh=true` to update the state for the new DNS resource

#### Explanation

Running '`terraform plan -refresh=true`' is not the appropriate action for resolving the error related to a new provider. The correct step is to run '`terraform init`' to download the Infoblox plugin before planning and applying the changes.

[Continue](#) [Retake test](#)



#### Correct answer

- since a new provider has been introduced, `terraform init` needs to be run to download the Infoblox plugin

#### Explanation

Running '`terraform init`' is necessary when a new provider is introduced to download the required plugin. This ensures that Terraform has access to the Infoblox provider and can properly manage the DNS record resource.

#### Your answer is incorrect

- 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

#### Explanation

While it is good practice to organize resources into modules, the error Henry is facing is not due to mixing resources from different providers. Running '`terraform init`' to download the Infoblox plugin is the immediate step to resolve the issue.

- the Azure plugin doesn't support Infoblox directly, so Henry needs to put the DNS resource in another configuration file

#### Explanation

The issue is not related to the Azure plugin not supporting Infoblox. The correct action is to run '`terraform init`' to download the Infoblox plugin, rather than moving the DNS resource to another configuration file.

#### Overall explanation

[Continue](#) [Retake test](#)



#### Overall 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:

1. **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.
2. 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.
3. 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.
4. 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.

<https://developer.hashicorp.com/terraform/cli/commands/init>

#### Domain

Objective 6 - Use the core Terraform workflow

Continue Retake test



#### Question 35 Correct

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?

`terraform state rm aws_s3_bucket.data-bucket`

#### Explanation

The `'terraform state rm'` command is not the correct option for updating the local name without replacing the existing resource. This command is used to remove a resource from the Terraform state, which is not the desired action in this situation where the goal is to update the local name without replacing the resource.

`terraform apply -replace aws_s3_bucket.data-bucket`

#### Explanation

The `'terraform apply -replace'` command is not the correct option for updating the local name

Continue Retake test



`terraform apply -replace aws_s3_bucket.data-bucket`

#### Explanation

The 'terraform apply -replace' command is not the correct option for updating the local name without replacing the existing resource. This command is used to force Terraform to replace a specific resource during the apply process, which is not the desired outcome in this scenario.

`terraform apply -refresh-only`

#### Explanation

The 'terraform apply -refresh-only' command is not the correct option for updating the local name without replacing the existing resource. This command is used to only refresh the state of the resources without making any changes to the infrastructure.

Your answer is correct

`terraform state mv aws_s3_bucket.data-bucket aws_s3_bucket.prod-encrypted-data-s3-bucket`

#### Explanation

The correct command to update the local name without replacing the existing resource is to use the 'terraform state mv' command. This command will move the existing state object to the new local name specified, ensuring that Terraform does not replace the resource.

[Continue](#) [Retake test](#)



#### Overall 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 aws_s3_bucket.data-bucket`

#### Explanation

The 'terraform apply -replace' command is not the correct option for updating the local name without replacing the existing resource. This command is used to force Terraform to replace a specific resource during the apply process, which is not the desired outcome in this scenario.

`terraform apply -refresh-only`

#### Explanation

The 'terraform apply -refresh-only' command is not the correct option for updating the local name without replacing the existing resource. This command is used to only refresh the state of the resources without making any changes to the infrastructure.

Your answer is correct

`terraform state mv aws_s3_bucket.data-bucket aws_s3_bucket.prod-encrypted-data-s3-bucket`

#### Explanation

The correct command to update the local name without replacing the existing resource is to use the 'terraform state mv' command. This command will move the existing state object to the new local name specified, ensuring that Terraform does not replace the resource.

[Continue](#) [Retake test](#)



#### Overall 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

[Continue](#) [Retake test](#)



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

Your answer is correct.

```
1 | variable "example" {
2 |   description = "This is a variable description"
3 |   type        = list(string)
4 |   default     = {}
5 | }
```

**Explanation**

The variable declaration in Choice A is going to result in an error because the default value is assigned as an empty map []. The type specified for the variable is list(string), so assigning an empty map as the default value is not valid and will cause an error.

1 | variable "example" {
2 | description = "This is a test"
3 | type = map
4 | default = {"one" = 1, "two" = 2, "Three" = "3"}
5 | }

**Explanation**

This variable declaration is valid as it includes a description, type, and default value for the variable "example". The type specified is a map, and the default value is a valid map with key-value pairs.

1 | variable "docker\_ports" {
2 | type = list(object({
3 | internal = number
4 | external = number
5 | protocol = string
6 | }))
7 | }

**Explanation**

This is a valid variable declaration for the variable "docker\_ports". It specifies the type as a list of

**Explanation**

This is a valid variable declaration for the variable "docker\_ports". It specifies the type as a list of objects with specific attributes (internal, external, protocol). The declaration is complete and does not contain any errors.

1 | variable "example" {}

**Explanation**

This is a valid variable declaration with an empty block for the variable "example". This is a common way to declare a variable without specifying any additional attributes or values.

**Overall 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]`.

**Domain**

Objective 8 - Read, Generate, and Modify Configuration

**!** Question 37 Incorrect

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?

Your answer is incorrect.

Continue Retake test



Continue Retake test



**Your answer is incorrect**

`terraform workspace -create stage`

**Explanation**  
The command "terraform workspace -create stage" is not the correct syntax for creating a new workspace in Terraform. The correct syntax uses the "new" keyword without the hyphen.

`terraform workspace -new stage`

**Explanation**  
The command "terraform workspace -new stage" is not the correct syntax for creating a new workspace in Terraform. The correct syntax uses the "new" keyword without the hyphen.

`terraform workspace create stage`

**Explanation**  
The command "terraform workspace create " is not the correct syntax for creating a new workspace in Terraform. The correct syntax uses the "new" keyword instead of "create".

**Correct answer**

`terraform workspace new stage`

**Explanation**  
The correct command to create a new workspace in Terraform is "terraform workspace new ". This command will create a new workspace named "stage" as requested by the user.

Continue Retake test ⚙ ↻

**Your answer is incorrect**

`terraform workspace -create stage`

**Explanation**  
The command "terraform workspace -create stage" is not the correct syntax for creating a new workspace in Terraform. The correct syntax uses the "new" keyword without the hyphen.

`terraform workspace -new stage`

**Explanation**  
The command "terraform workspace -new stage" is not the correct syntax for creating a new workspace in Terraform. The correct syntax uses the "new" keyword without the hyphen.

`terraform workspace create stage`

**Explanation**  
The command "terraform workspace create " is not the correct syntax for creating a new workspace in Terraform. The correct syntax uses the "new" keyword instead of "create".

**Correct answer**

`terraform workspace new stage`

**Explanation**  
The correct command to create a new workspace in Terraform is "terraform workspace new ". This command will create a new workspace named "stage" as requested by the user.

Continue Retake test ⚙ ↻

**Your answer is incorrect**

`terraform workspace -create stage`

**Explanation**  
The command "terraform workspace -create stage" is not the correct syntax for creating a new workspace in Terraform. The correct syntax uses the "new" keyword without the hyphen.

`terraform workspace -new stage`

**Explanation**  
The command "terraform workspace -new stage" is not the correct syntax for creating a new workspace in Terraform. The correct syntax uses the "new" keyword without the hyphen.

`terraform workspace create stage`

**Explanation**  
The command "terraform workspace create " is not the correct syntax for creating a new workspace in Terraform. The correct syntax uses the "new" keyword instead of "create".

**Correct answer**

`terraform workspace new stage`

**Explanation**  
The correct command to create a new workspace in Terraform is "terraform workspace new ". This command will create a new workspace named "stage" as requested by the user.

Continue Retake test  

```
1 | $ terraform workspace select dev
2 | Switched to workspace "dev".
https://developer.hashicorp.com/terraform/cli/commands/workspace/select
```

**Domain**  
Objective 4 - Use Terraform Outside of Core Workflow

**Question 39 Incorrect**

Anyone can publish and share modules on the [Terraform Public 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 Registry? (select three)

The module must be PCI/HIPPA compliant.

**Explanation**  
The statement about the module needing to be PCI/HIPPA compliant is incorrect, as there is no such requirement mentioned for publishing modules on the Terraform Public Registry. Compliance with these standards would depend on the specific use case of the module.

**Correct selection**

Module repositories must use this three-part name format, `terraform-<PROVIDER>-<NAME>`.

**Explanation**  
The requirement for module repositories to follow the terraform-- naming format is accurate, as this naming convention helps organize and categorize modules on the Terraform Public Registry.

Continue Retake test  

```
1 | $ terraform workspace select dev
2 | Switched to workspace "dev".
https://developer.hashicorp.com/terraform/cli/commands/workspace/select
```

#### Domain

Objective 4 - Use Terraform Outside of Core Workflow

#### !

 Question 39 Incorrect

Anyone can publish and share modules on the [Terraform Public 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 Registry? (select three)

- The module must be PCI/HIPPA compliant.

#### Explanation

The statement about the module needing to be PCI/HIPPA compliant is incorrect, as there is no such requirement mentioned for publishing modules on the Terraform Public Registry. Compliance with these standards would depend on the specific use case of the module.

#### Correct selection

- Module repositories must use this three-part name format, `terraform-<PROVIDER>-<NAME>`.

#### Explanation

The requirement for module repositories to follow the `terraform--` naming format is accurate, as this naming convention helps organize and categorize modules on the Terraform Public Registry.

[Continue](#) [Retake test](#)



- **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.

<https://developer.hashicorp.com/terraform/registry/modules/publish#requirements>

#### Domain

Objective 5 - Interact with Terraform Modules

#### !

 Question 40 Incorrect

Provider dependencies are created in several different ways. Select the valid provider dependencies from the following list: (select three)

#### Your selection is correct

- Existence of any resource instance belonging to a particular provider in the current state.

[Continue](#) [Retake test](#)



**Explanation**

The existence of any resource instance belonging to a particular provider in the current state signifies a dependency on that provider, as Terraform needs access to the provider to manage the state of those resources.

**Correct selection**

- Explicit use of a provider block in configuration, optionally including a version constraint.

**Explanation**

The explicit use of a provider block in the configuration, along with an optional version constraint, establishes a direct dependency on that specific provider for the resources being managed.

- Existence of any provider plugins found locally in the working directory.

**Explanation**

The existence of provider plugins locally in the working directory does not establish a direct dependency on the provider for the resources being managed. Terraform relies on the configuration and state to determine provider dependencies.

**Correct selection**

- Use of any resource block or data block in the configuration, belonging to a particular provider

**Explanation**

When any resource block or data block is used in the configuration, it indicates a dependency on the provider associated with those blocks, as they are responsible for managing those specific resources.

[Continue](#) [Retake test](#)

**Overall 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.

<https://developer.hashicorp.com/terraform/language/providers/requirements>

<https://developer.hashicorp.com/terraform/cli/commands/providers>

**Domain**

Objective 3 - Understand Terraform Basics

**!** Question 41 Incorrect

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         = "a4c9357eadde"  
  route_table_id = "rtb-a221958bc5892eade331"  
}
```

**Correct answer**

- the Terraform public registry

**Explanation**

The code specifies a source from "terraform-vault-aws-tgw/hcp", which is a typical format for modules stored in the Terraform public registry. This choice is correct based on the information

[Continue](#) [Retake test](#)



**Explanation**

The code specifies a source from "terraform-vault-aws-tgw/hcp", which is a typical format for modules stored in the Terraform public registry. This choice is correct based on the information provided in the code snippet.

- in a Terraform Cloud private registry

**Explanation**

The code does not indicate that the module is stored in a Terraform Cloud private registry. It specifies a source from "terraform-vault-aws-tgw/hcp", which is more indicative of the Terraform public registry. This choice is incorrect based on the information provided in the code snippet.

- a local code repository on your network

**Explanation**

The code specifies a source from "terraform-vault-aws-tgw/hcp", which indicates that the module is not stored in a local code repository on your network. It is referencing an external source.

Your answer is incorrect

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

**Explanation**

The code specifies a source from "terraform-vault-aws-tgw/hcp", but it does not indicate that the module is stored in a local file under a directory named "terraform/vault-aws-tgw/hcp". This choice is incorrect based on the information provided in the code snippet.

[Continue](#) [Retake test](#)  

**Overall explanation**

You can use the Terraform Public 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 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:

```
1 | module "vpc" {
2 |   source = "terraform-aws-modules/vpc"
3 |   version = "2.34.0"
4 |
5 |   # Add any required variables and configuration here
6 | }
```

The `source` attribute specifies the module source, which is the repository on the Terraform Public 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:

```
1 | module "vpc" {
2 |   source = "terraform-aws-modules/vpc"
3 |   version = "2.34.0"
4 |
5 |   name = "my-vpc"
6 |   cidr = "10.0.0.0/16"
7 |   azs = ["us-west-2a", "us-west-2b", "us-west-2c"]
8 | }
```

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:

```
1 | output "vpc_id" {
2 |   value = module.vpc.vpc_id
3 | }
```

You can find more information on using modules from the Terraform Public Registry in the Terraform documentation: <https://www.terraform.io/docs/configuration/modules.html>

[Continue](#) [Retake test](#)  

**Domain**

Objective 5 - Interact with Terraform Modules

**Question 42 Correct**

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 answers 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  
}
```

**Your answer is correct**

```
1 | resource "aws_security_group" "vault_elb" {  
2 |     name      = "${var.name_prefix}-vault-elb"  
3 |     description = "Vault ELB"  
4 |     vpc_id     = var.vpc_id  
5 | }
```

**Explanation**

The syntax error in the original code is that the description value is missing quotation marks. By adding the quotes around "Vault ELB", the code will be corrected and the provisioning process will not throw an error.

```
1 | resource "aws_security_group" "vault_elb" {  
2 |     name      = "${var.name_prefix}-vault-elb"  
3 |     description = ${Vault ELB}  
4 |     vpc_id     = var.vpc_id  
5 | }
```

**Continue** **Retake test****Explanation**

In this choice, the description value is enclosed in "\$()" which is used for variable interpolation. Since "Vault ELB" is a string and not a variable, it should be enclosed in double quotes instead of "\$()".

```
1 | resource "aws_security_group" "vault_elb" {  
2 |     name      = "${var.name_prefix}-vault-elb"  
3 |     description = [Vault ELB]  
4 |     vpc_id     = var.vpc_id  
5 | }
```

**Explanation**

The description value in this choice is enclosed in square brackets, which is incorrect syntax for a string in Terraform. Strings should be enclosed in double quotes, so the correct syntax would be to use double quotes around "Vault ELB".

```
1 | resource "aws_security_group" "vault_elb" {  
2 |     name      = "${var.name_prefix}-vault-elb"  
3 |     description = var.Vault ELB  
4 |     vpc_id     = var.vpc_id  
5 | }
```

**Explanation**

The description value in this choice is missing quotation marks and includes an incorrect variable reference. To fix the syntax error, the description should be enclosed in double quotes like "Vault ELB".

**Overall explanation**

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

**Continue** **Retake test**

1. 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.

2. 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.**

<https://developer.hashicorp.com/terraform/language/syntax/configuration#arguments-and-blocks>

## Domain

Objective 3 - Understand Terraform Basics

### ! Question 43 Incorrect

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

custom VCS integration

#### Explanation

Custom VCS integration refers to the ability to connect Terraform Cloud to a version control system to manage infrastructure as code, but it does not specifically address the publishing and maintenance of custom modules within an organization.

**Correct answer**

private registry

#### Explanation

The private registry feature in Terraform Cloud allows users to publish and maintain custom modules within their organization, providing a secure and controlled environment for sharing infrastructure configurations.

**Your answer is incorrect**

Terraform registry

#### Explanation

The Terraform registry is a public repository of modules that can be used by any Terraform user, but it is not specific to an organization and does not provide the same level of control as a private registry.

remote runs

#### Explanation

Remote runs in Terraform Cloud allow users to execute Terraform configurations in a remote environment, but it does not directly relate to the publishing and maintenance of custom modules within an organization.

#### Overall 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.

<https://www.datocms-assets.com/2885/1602500234-terraform-full-feature-pricing-tablev2-1.pdf>

## Domain

[Continue](#) [Retake test](#)  

[Continue](#) [Retake test](#)  

**Question 44 Incorrect**

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 = {
      source = "hashicorp/aws"
      version ~> "5.36.0"
    }
  }
}
```

Your selection is correct

 AWS provider version: 5.36.9**Explanation**

The version "5.36.9" satisfies the constraint expression "~> 5.36.0" as it falls within the same minor version range (5.36.x).

Your selection is incorrect

 AWS provider version: 5.37.0**Explanation**

The version "5.37.0" does not satisfy the constraint expression "~> 5.36.0" as it is beyond the specified minor version range.

Continue Retake test



Correct selection

 AWS provider version: 5.36.3**Explanation**

The version "5.36.3" satisfies the constraint expression "~> 5.36.0" as it falls within the same minor version range (5.36.x).

 AWS provider version: 5.3.1**Explanation**

The version "5.3.1" does not satisfy the constraint expression "~> 5.36.0" as it is not within the specified minor version range.

**Overall explanation**

In Terraform, `required_providers` act as traffic controllers for your infrastructure tools. They ensure all modules use the right versions of providers like AWS or Azure, avoiding compatibility issues and guaranteeing everyone plays by the same rules. Think of them as a clear roadmap for your infrastructure setup, leading to consistent, predictable, and secure deployments.

A version constraint is a [string literal](#) containing one or more conditions, which are separated by commas.

Each condition consists of an operator and a version number.

Version numbers should be a series of numbers separated by periods (like `1.2.0`), optionally with a suffix to indicate a beta release:

- `~>`: Allows only the *rightmost* version component to increment. This format is referred to as the *optimistic constraint* operator. For example, to allow new patch releases within a specific minor release, use the full version number:
  - `~> 1.0.4`: Allows Terraform to install `1.0.5` and `1.0.10` but not `1.1.0`.
  - `~> 1.1`: Allows Terraform to install `1.2` and `1.10` but not `2.0`.

Continue Retake test



**Domain**

Objective 3 - Understand Terraform Basics

**Question 45 Incorrect**

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]
}
```

non-dependency resource

**Explanation**

The 'depends\_on' argument in Terraform does not indicate a non-dependency resource. It specifically defines a dependency relationship between resources.

Your answer is incorrect

implicit dependency

**Explanation**

The 'depends\_on' argument in Terraform creates an explicit dependency between resources, not an implicit one. This means that Terraform will ensure that the resource specified in 'depends\_on' is created or updated before the dependent resource.

[Continue](#) [Retake test](#)  

**Explanation**

The 'depends\_on' argument in Terraform creates an explicit dependency between resources, not an implicit one. This means that Terraform will ensure that the resource specified in 'depends\_on' is created or updated before the dependent resource.

internal dependency

**Explanation**

The 'depends\_on' argument in Terraform does not establish an internal dependency. It explicitly defines a dependency on another resource in the Terraform configuration.

Correct answer

explicit dependency

**Explanation**

The 'depends\_on' argument in Terraform creates an explicit dependency between resources. This means that Terraform will wait for the specified resource to be created or updated before proceeding with the dependent resource.

**Overall 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:

[Continue](#) [Retake test](#)  

```

1 | resource "azurerm_virtual_network" "vnet" {
2 |   name           = "example-vnet"
3 |   address_space  = ["10.0.0.0/16"]
4 | }
5 |
6 | resource "azurerm_subnet" "subnet" {
7 |   name           = "example-subnet"
8 |   virtual_network_name = azurerm_virtual_network.vnet.name
9 |   address_prefix  = "10.0.1.0/24"
10| }
11|
12| resource "azurerm_network_interface" "nic" {
13|   name           = "example-nic"
14|   location       = azurerm_virtual_network.vnet.location
15|   subnet_id      = azurerm_subnet.subnet.id
16|   depends_on     = [
17|     azurerm_subnet.subnet,
18|     azurerm_virtual_network.vnet
19|   ]
20| }

```

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.

<https://learn.hashicorp.com/tutorials/terraform/dependencies>

## Domain

Objective 8 - Read, Generate, and Modify Configuration

Continue Retake test

### Question 46 Correct

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?

Your answer is correct

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

#### Explanation

Removing the Cloud SQL instance from Terraform management using the '`terraform state rm`' command ensures that the instance is not included in the resources to be destroyed when running '`terraform destroy`'. This allows Freddy and Jason to delete all other resources without impacting the database.

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

#### Explanation

Taking a snapshot of the database and recreating it after running '`terraform destroy`' is not an efficient solution. This process involves manual steps and does not utilize Terraform's infrastructure as code capabilities to manage resources.

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

#### Explanation

Running '`terraform destroy`' without removing the Cloud SQL resource from the configuration

Continue Retake test

file will result in the database being destroyed along with other resources. Modifying the configuration file to include only the Cloud SQL resource and then applying the changes will not prevent the database from being deleted.

- delete the entire state file using the `terraform state rm` command and manually delete the other resources in GCP

#### Explanation

Deleting the entire state file using the `terraform state rm` command and manually deleting other resources in GCP is not recommended. This approach bypasses Terraform's management of resources and can lead to inconsistencies in the infrastructure state.

#### Overall 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:

1. Identify the resource you want to preserve. In this example, let's assume you want to preserve a resource named `prod_db`.
2. Run `terraform state list` to see a list of all Terraform-managed resources.
3. Run `terraform state rm` for each resource you want to keep, like the `prod_db`. For example:

```
1 | terraform state rm google_sql_database_instance.prod_db
2 | terraform state rm aws_instance.another_instance
```
4. 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.

<https://developer.hashicorp.com/terraform/cli/commands/state/rm>

[Continue](#) [Retake test](#)



#### Domain

Objective 7 - Implement and Maintain State

#### Question 47 Correct

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&m"]
}

output "give_me_candy" {
  value = element(var.candy_list, 2)
}
```

- m&m's

#### Explanation

[Continue](#) [Retake test](#)



**Explanation**

The output value of the code snippet is not "m&ms" because the element function is used to access the element at index 2 in the candy\_list variable, which is "reeces" and not "m&ms".

 **kitkat****Explanation**

The output value of the code snippet is not "kitkat" because the element function is used to access the element at index 2 in the candy\_list variable, which is "reeces" and not "kitkat".

**Your answer is correct** **reeces****Explanation**

The output value of the code snippet is "reeces" because the element function is used to access the element at index 2 in the candy\_list variable, which is "reeces".

 **snickers****Explanation**

The output value of the code snippet is not "snickers" because the element function is used to access the element at index 2 in the candy\_list variable, which is "reeces" and not "snickers".

**Overall 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`.

[Continue](#) [Retake test](#)  

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

[https://developer.hashicorp.com/terraform/language/functions/index\\_function](https://developer.hashicorp.com/terraform/language/functions/index_function)

<https://developer.hashicorp.com/terraform/language/functions/element>

**Domain**

Objective 8 - Read, Generate, and Modify Configuration

**1 Question 48 Incorrect**

Where does Terraform Community (Free) store the `local` state for workspaces?

 **directory called `terraform.workspaces.tfstate`****Explanation**

The directory structure '`terraform.workspaces.tfstate`' is not the standard location where Terraform Community (Free) stores the local state for workspaces. The correct directory structure is '`terraform.tfstate.d/`'.

**Your answer is incorrect** **a file called `terraform.tfstate`****Explanation**

Terraform Community (Free) does not store the local state for workspaces in a single file called

[Continue](#) [Retake test](#)  

'terraform.tfstate'. Instead, it uses a directory structure to store state files for each workspace individually.

- a file called `terraform.tfstate.backup`

**Explanation**

The file 'terraform.tfstate.backup' is not where Terraform Community (Free) stores the local state for workspaces. The correct location is a directory called 'terraform.tfstate.d'.

**Correct answer**

- directory called `terraform.tfstate.d/<workspace_name>`

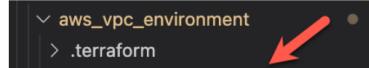
**Explanation**

Terraform Community (Free) stores the local state for workspaces in a directory called 'terraform.tfstate.d/'. This directory structure allows for separate state files for each workspace, making it easier to manage and maintain the state data.

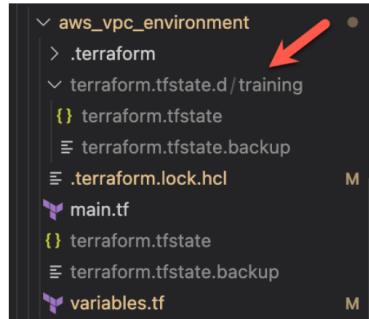
**Overall explanation**

Terraform Community (Free) 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 named `terraform.tfstate`.



Continue Retake test ⚙️ ↻



<https://developer.hashicorp.com/terraform/cli/workspaces#workspace-internals>

**Domain**

Objective 7 - Implement and Maintain State

**Question 49 Incorrect**

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`?

Your answer is incorrect

5

**Explanation**

Continue Retake test ⚙️ ↻

Your answer is incorrect

5

#### Explanation

This choice is incorrect as Terraform provisions 10 resources concurrently by default, not 5.

20

#### Explanation

This choice is incorrect as Terraform provisions 10 resources concurrently by default, not 20.

Correct answer

10

#### Explanation

Terraform by default provisions 10 resources concurrently during a ‘terraform apply’ command to speed up the provisioning process and reduce the overall time taken.

50

#### Explanation

This choice is incorrect as Terraform provisions 10 resources concurrently by default, not 50.

#### Overall 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:

```
1 | terraform {  
2 |   parallelism = 20  
3 | }
```

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

<https://developer.hashicorp.com/terraform/internals/graph#walking-the-graph>

#### Domain

Objective 6 - Use the core Terraform workflow

#### Question 50 Correct

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"  
}
```

[Continue](#) [Retake test](#)



[Continue](#) [Retake test](#)



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 parameter is required to use multiple provider blocks of the same type, but with distinct configurations, such as cloud regions, namespaces, or other desired settings?

Your answer is correct

**alias**

#### Explanation

The correct additional parameter required to use multiple provider blocks of the same type with distinct configurations is the "alias" parameter. This allows you to differentiate between the different configurations of the same provider type.

**multi**

#### Explanation

The "multi" parameter is not a valid parameter in Terraform for defining multiple configurations of the same provider type. The correct parameter to achieve this is the "alias" parameter.

**version**

**version**

#### Explanation

The "version" parameter is not used to define multiple configurations of the same provider type in Terraform. It is typically used to specify the version of the provider being used.

**label**

#### Explanation

The "label" parameter is not a valid parameter in Terraform for defining multiple configurations of the same provider type. To differentiate between configurations, the "alias" parameter should be used.

#### Overall 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:

```
1 | provider "aws" {  
2 |   region = "us-east-1"  
3 | }  
4 |  
5 | provider "aws" {  
6 |   region = "us-west-1"  
7 |   alias = "west"  
8 | }
```

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.

**Continue** **Retake test**



**Continue** **Retake test**



**Domain**

Objective 3 - Understand Terraform Basics

✓ **Question 51 Correct**

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

`terraform show`

**Explanation**

The choice `terraform show` is incorrect as this command is used to provide human-readable output from a state or plan file. It does not perform validation checks for errors within modules, attribute names, and value types.

`terraform fmt`

**Explanation**

The choice `terraform fmt` is incorrect as this command is used to rewrite Terraform configuration files to a canonical format and style. It does not check for errors within modules, attribute names, and value types.

`terraform format`

**Explanation**

The choice `terraform format` is incorrect as this command is not a valid Terraform command. The correct command for formatting Terraform configuration files is `terraform fmt`.

**Your answer is correct**

`terraform validate`

**Explanation**

The correct choice is `terraform validate` because this command is specifically designed to check and report errors within modules, attribute names, and value types to ensure they are syntactically valid and internally consistent.

**Overall 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.

<https://developer.hashicorp.com/terraform/cli/commands/validate>

**Domain**

Objective 6 - Use the core Terraform workflow

[Continue](#) [Retake test](#)



[Continue](#) [Retake test](#)



**Domain**

Objective 6 - Use the core Terraform workflow

**Question 52 Incorrect**

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.

 **TF\_VAR\_VALUE****Explanation**

The prefix `TF_VAR_VALUE` is not the correct format for setting input variables using environment variables in Terraform. The correct prefix is `TF_VAR`.

 **TF\_ENV****Explanation**

The prefix `TF_ENV` is not the correct format for setting input variables using environment variables in Terraform. The correct prefix is `TF_VAR`.

**Correct answer** **TF\_VAR****Explanation**

The correct prefix string for setting input variables using environment variables in Terraform is `TF_VAR`. This prefix is recognized by Terraform to assign values to variables.

**Continue** **Retake test****Your answer is incorrect** **TF\_ENV\_VAR****Explanation**

The prefix `TF_ENV_VAR` is not the correct format for setting input variables using environment variables in Terraform. The correct prefix is `TF_VAR`.

**Overall 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:

```
1 | variable "instructor_name" {  
2 |   type = string  
3 | }
```

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:

```
1 | $ export TF_VAR_instructor_name="bryan"  
2 | $ terraform apply
```

<https://developer.hashicorp.com/terraform/cli/config/environment-variables>

**Domain**

Objective 3 - Understand Terraform Basics

**Continue** **Retake test**

 Question 53 Incorrect

Terraform is distributed as a single binary and available for many different platforms. Select all Operating Systems that Terraform is available for. (select five)

Your selection is incorrect

AIX

Explanation

Terraform is not available for AIX, making this choice incorrect.

Your selection is correct

Solaris

Explanation

Terraform is not available for Solaris, making this choice incorrect.

Your selection is correct

Windows

Explanation

Terraform is available for Windows, making this choice correct.

Your selection is correct

Linux

Explanation

Terraform is available for Linux, making this choice correct.

Correct selection

FreeBSD

Explanation

Terraform is available for FreeBSD, making this choice correct.

Your selection is correct

macOS

Explanation

Terraform is available for macOS, making this choice correct.

Overall explanation

Terraform is a cross-platform tool and can be installed on several operating systems, including:

1. Windows: Terraform can be installed on **Windows** operating systems using the Windows installer.
2. **macOS**: Terraform can be installed on macOS using the macOS installer or using Homebrew.
3. Linux: Terraform can be installed on **Linux** operating systems using the binary distribution or through package management systems, such as apt or yum.
4. **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,

[Continue](#) [Retake test](#)



[Continue](#) [Retake test](#)



**Question 54 Incorrect**True or False? 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
}

```

**Correct answer** True**Explanation**

The code snippet provided shows an implicit dependency in Terraform. The resource "aws\_volume\_attachment" "attach\_data\_volume" depends on both "aws\_ebs\_volume.data" and "aws\_instance.web" resources without explicitly specifying the dependency using the "depends\_on" attribute. Terraform automatically detects this relationship and ensures that the dependencies are resolved in the correct order during the execution.

**Continue** **Retake test**  **Your answer is incorrect** False**Explanation**

This statement is incorrect. The code snippet clearly demonstrates an implicit dependency between the resources "aws\_volume\_attachment" "attach\_data\_volume", "aws\_ebs\_volume.data", and "aws\_instance.web". In Terraform, implicit dependencies are recognized based on the resource references within the configuration, even if they are not explicitly defined using the "depends\_on" attribute.

**Overall 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:

```

1 | resource "aws_instance" "example" {
2 |     ami           = "ami-0c55b159cbfafe1f0"
3 |     instance_type = "t2.micro"
4 |
5 |     depends_on = [
6 |         aws_ebs_volume.example
7 |     ]
8 |
9 |

```

**Continue** **Retake test**  

```
9
10 | resource "aws_ebs_volume" "example" {
11 |   availability_zone = "us-west-2a"
12 |   size                = 1
13 | }
```

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.

<https://developer.hashicorp.com/terraform/tutorials/certification-associate-tutorials-003/dependencies>

#### Domain

Objective 8 - Read, Generate, and Modify Configuration

#### Question 55 Incorrect

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?

- run `terraform apply -lock=false` when executing the changes made to the configuration

#### Explanation

Running `terraform apply -lock=false` will instruct Terraform to not hold a state lock during the operation. This is dangerous if others might concurrently run commands against the same workspace. Using this method does not allow Oscar to validate the changes without affecting production workloads.

- run `terraform refresh` to compare his existing configuration file against the current

- run `terraform refresh` to compare his existing configuration file against the current one

#### Explanation

`terraform refresh` is used to update the state file with the real-world infrastructure. It does not provide a preview of the changes that will be made, so it is not suitable for Oscar to validate the changes without impacting existing workloads.

Your answer is incorrect

- run a `terraform validate` to ensure the changes won't impact the production workloads

#### Explanation

While `terraform validate` checks the syntax and configuration of the Terraform files, it does not provide a preview of the changes that will be made. Therefore, it is not the best option for Oscar to validate the changes without impacting existing workloads.

Correct answer

- run a `terraform plan` and validate the changes that will be made

#### Explanation

Running a `terraform plan` allows Oscar to preview the changes that will be made to the infrastructure without actually applying them. This way, he can validate the changes and ensure they won't negatively impact existing workloads before making any modifications.

#### Overall 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

Continue Retake test



Continue Retake test



#### Overall 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.

<https://developer.hashicorp.com/terraform/cli/commands/plan>

#### Domain

Objective 6 - Use the core Terraform workflow

#### Question 56 Incorrect

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)

`terraform plan`

#### Explanation

The `terraform plan` command is used to create an execution plan for applying changes to the infrastructure. It does not download or update modules.

**Your selection is incorrect**

`terraform refresh`

#### Explanation

The `terraform refresh` command is used to update the state file with the current state of the infrastructure. It does not download or update modules.

#### Explanation

The `terraform refresh` command is used to update the state file with the current state of the infrastructure. It does not download or update modules.

**Correct selection**

`terraform init`

#### Explanation

The `terraform init` command is used to initialize a working directory containing Terraform configuration files. It downloads any modules specified in the configuration.

**Your selection is correct**

`terraform get`

#### Explanation

The `terraform get` command is used to download modules from the module registry or a version control system, making them available for use in the configuration.

#### Overall explanation

The two Terraform commands used to download and update modules are:

1. `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.
2. `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

[Continue](#) [Retake test](#)



[Continue](#) [Retake test](#)



2. `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.

<https://learn.hashicorp.com/tutorials/terraform/module-create?in=terraform/modules#install-the-local-module>

#### Domain

Objective 5 - Interact with Terraform Modules

#### ✓ Question 57 Correct

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?

- 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 command '`terraform state rm aws_instance.web`' removes the specified resource from the state file, prompting Terraform to recreate the instance during the next apply. This method is not recommended in this scenario as it removes the resource entirely from the state.

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

Continue Retake test  

2. `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.

<https://learn.hashicorp.com/tutorials/terraform/module-create?in=terraform/modules#install-the-local-module>

#### Domain

Objective 5 - Interact with Terraform Modules

#### ✓ Question 57 Correct

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?

- 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 command '`terraform state rm aws_instance.web`' removes the specified resource from the state file, prompting Terraform to recreate the instance during the next apply. This method is not recommended in this scenario as it removes the resource entirely from the state.

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

Continue Retake test  

2. `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.

<https://learn.hashicorp.com/tutorials/terraform/module-create?in=terraform/modules#install-the-local-module>

#### Domain

Objective 5 - Interact with Terraform Modules

#### Question 57 Correct

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?

- 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 command `terraform state rm aws\_instance.web` removes the specified resource from the state file, prompting Terraform to recreate the instance during the next apply. This method is not recommended in this scenario as it removes the resource entirely from the state.

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

[Continue](#) [Retake test](#)

