Terraform Associate Certification 002 - Results

Back to result overview

Attempt 1

All domains

- 57 all
- 29 correct
- 28 incorrect
- 0 skipped
- 0 marked

Collapse all questions

Question 1 Incorrect

terraform init retrieves and caches the configuration for all remote modules.

Correct answer

True

Explanation

True. When running `terraform init`, Terraform retrieves the configuration for all remote modules specified in the main configuration file. This includes downloading any necessary plugins and modules from the Terraform Registry or other specified sources. The retrieved configuration is then cached locally to ensure consistency and speed up subsequent operations.

Your answer is incorrect

False

Explanation

False. While `terraform init` does retrieve and cache the configuration for remote modules, it does not retrieve the configuration for all remote modules. It only retrieves the configuration for the modules explicitly defined in the main configuration file. This helps in managing dependencies and ensuring that only the necessary configurations are retrieved and cached.

Overall explanation

The correct answer is True.

Explanation:

The `terraform init` command performs several important tasks to initialize a Terraform working directory. One of these tasks is retrieving and caching the configuration for all **remote modules** referenced in the configuration.

Remote modules are those that are sourced from the Terraform Registry, a Git repository, or other remote sources. When `terraform init` is run, Terraform downloads these modules and stores them locally in the `.terraform` directory so they can be used during the execution of the Terraform plan and apply workflows.

This ensures that the necessary modules are available for use without needing to fetch them repeatedly for each Terraform operation.

Question 2 Correct

Terraform configuration can only call modules from the public registry.

True

Explanation

False. Terraform configuration can call modules from various sources, including the public registry, private repositories, local file paths, and Git repositories. This flexibility allows users to leverage both public and private modules based on their specific requirements.

Your answer is correct

False

Explanation

True. Terraform configuration is not limited to calling modules only from the public registry. Users have the flexibility to use modules from different sources, such as private repositories, local file paths, and Git repositories, enabling them to customize their infrastructure provisioning based on their needs.

Overall explanation

The correct answer is False.

Explanation:

Terraform configuration is not restricted to calling modules from the public Terraform Registry. While the Terraform Registry is a convenient and popular source for modules, Terraform can also call modules from **various other locations**, such as:

- **Local directories**: Modules can reside in a local folder on your machine.
- **Version control systems**: Modules can be sourced from Git repositories like GitHub, GitLab, or Bitbucket.
- **Object storage**: Modules can be stored in an archive file (e.g., `.zip`) located in an HTTP server, AWS S3 bucket, or other storage services.
- **Private registries**: Terraform supports private module registries like those provided by Terraform Cloud or Enterprise.

This flexibility allows teams to maintain private modules or use modules stored in custom locations.

Question 3Correct

resource “kubernetes_namespace” “example” {

    name = “test”

}

A resource block is shown in the Exhibit section of this page. How would you reference the attribute name of this resource in HCL?

data.kubernetes_namespace.name

Explanation

The correct syntax for referencing the attribute name of a resource in HCL is by using the resource type followed by the resource name and then the attribute name. In this case, the resource type is "kubernetes_namespace" and the resource name is "example", so the correct reference would be "kubernetes_namespace.example.name".

resource.kubernetes_namespace.example.name

Explanation

The syntax "resource.kubernetes_namespace.example.name" is incorrect because it does not follow the correct format for referencing attributes of a resource in HCL. The correct format is to use the resource type, resource name, and attribute name in the reference.

kubernetes_namespace.test.name

Explanation

The syntax "kubernetes_namespace.test.name" is incorrect because it does not include the resource name "example" in the reference. In HCL, you need to specify both the resource type and the resource name to reference the attribute of a resource.

Your answer is correct

kubernetes_namespace.example.name

Explanation

The correct way to reference the attribute name of the resource "example" in the "kubernetes_namespace" resource block is "kubernetes_namespace.example.name". This syntax follows the correct format for referencing attributes of a resource in HCL.

Overall explanation

The correct answer is kubernetes_namespace.example.name.

Explanation:

In Terraform, to reference an attribute of a resource, you use the resource type, the resource name, and the attribute name in the format:

`<resource_type>.<resource_name>.<attribute_name>`.

For the given resource block:

```
hcl
Copy coderesource "kubernetes_namespace" "example" {
   name = "test"
}
```

- **`kubernetes_namespace`** is the resource type.
- **`example`** is the resource name.

- `name` is the attribute you want to reference.

Thus, the correct way to reference the `name` attribute of this resource is `kubernetes_namespace.example.name`.

Why the other answers are incorrect:

1. **data.kubernetes_namespace.name**:
   This format is used to reference data sources, not resources. The resource in this example is not a data source, so this is invalid.
2. **resource.kubernetes_namespace.example.name**:
   Terraform does not use the `resource` keyword for referencing attributes. This format is incorrect.
3. **kubernetes_namespace.test.name**:
   The resource name is `example`, not `test`. Using `test` would result in an error because no such resource name exists in the configuration.

Question 4 Correct

You want to use API tokens and other secrets within your team's Terraform workspaces. Where does HashiCorp recommend you store these sensitive values? (Choose three.)

Your selection is correct

In a terraform.tfvars file, securely managed and shared with your team.

Explanation

Storing sensitive values in a terraform.tfvars file that is securely managed and shared with your team is a recommended practice by HashiCorp. This allows for easy access to the secrets within the team while ensuring that they are kept secure.

Your selection is correct

In HashiCorp Vault.

Explanation

HashiCorp Vault is a secure and centralized solution for managing secrets and sensitive data. Storing API tokens and other secrets in HashiCorp Vault provides a secure and scalable way to manage and access these sensitive values within your Terraform workspaces.

In a terraform.tfvars file, checked into your version control system.

Explanation

Storing sensitive values in a terraform.tfvars file that is checked into your version control system is not recommended by HashiCorp. This practice exposes the sensitive values to potential security risks, as version control systems are not designed to securely store sensitive data.

In a plaintext document on a shared drive.

Explanation

Storing API tokens and other secrets in a plaintext document on a shared drive is a highly insecure practice. This exposes the sensitive values to unauthorized access and compromises the security of your Terraform workspaces and infrastructure.

Your selection is correct

In an HCP Terraform/Terraform Cloud variable, with the sensitive option checked.

Explanation

Storing sensitive values in an HCP Terraform/Terraform Cloud variable with the sensitive option checked is a recommended practice by HashiCorp. This allows you to securely store and manage secrets within your Terraform workspaces, ensuring that they are protected and only accessible to authorized users.

Overall explanation

The recommended places to store API tokens and other secrets for your team's Terraform workspaces are:

HashiCorp Vault: This is a secure secret management system designed specifically for storing and accessing secrets like API tokens safely.

HCP Terraform/Terraform Cloud variable, with the sensitive option checked: This allows you to securely store sensitive values directly within Terraform Cloud or HCP, ensuring they are encrypted and not exposed.

In a terraform.tfvars file, securely managed and shared with your team: While this is less secure than other methods, if the file is properly secured and not checked into version control, it can be a valid option.

Explanation

- **In HashiCorp Vault**: Provides enterprise-grade secret management, ensuring tokens and secrets are encrypted and access is tightly controlled.
- **In an HCP Terraform/Terraform Cloud variable, with the sensitive option checked**: Automatically encrypts sensitive variables and ensures they remain secure during runs.
- **In a terraform.tfvars file, securely managed and shared with your team**: Works as long as the file is not exposed or versioned in a public repository.

Incorrect Options

- **In a terraform.tfvars file, checked into your version control system**: This exposes sensitive information to everyone with access to the repository, which is highly insecure.
- **In a plaintext document on a shared drive**: Storing secrets in plaintext exposes them to accidental leaks or unauthorized access, making this a poor practice.

Question 5Correct
What happens when you execute terraform plan?

Imports all of your existing cloud provider resources to the state.
Explanation
Importing existing cloud provider resources to the state is not the primary function of executing `terraform plan`. This process is used to bring existing resources under Terraform management, but it is not directly related to the `terraform plan` command.

Installs all providers and modules referenced by configuration.
Explanation
Installing providers and modules referenced by the configuration is not the main purpose of executing `terraform plan`. This step is typically done during the initialization phase using `terraform init`, not during the planning phase.

Compares your Terraform code and local state file to the remote state file in a cloud provider and determines if any changes need to be made.
Explanation
Comparing Terraform code and local state file to the remote state file in a cloud provider is not the exact action taken when executing `terraform plan`. This comparison is part of the overall Terraform workflow, but the `terraform plan` command specifically focuses on generating an execution plan based on the current state and configuration.

Your answer is correct
Refreshes your state, then compares your state file to your Terraform configuration and creates an execution plan if any changes need to be made.
Explanation
Refreshing the state, comparing it to the Terraform configuration, and creating an execution plan if changes are needed is the correct explanation for what happens when you execute `terraform plan`. This command helps you preview the changes that Terraform will make to your infrastructure before actually applying those changes.

Overall explanation
The correct answer is:

Refreshes your state, then compares your state file to your Terraform configuration and creates an execution plan if any changes need to be made.

Explanation:

When you run `terraform plan`, Terraform first refreshes the state file to ensure it accurately reflects the current state of your resources in the cloud provider. It then compares this refreshed state file to your Terraform configuration to identify any changes needed to match the desired state defined in your code. Based on this comparison, Terraform generates an execution plan, which outlines the changes it will make if you proceed with `terraform apply`.

Why the Other Options Are Incorrect:

- **Imports all of your existing cloud provider resources to the state**: This is incorrect because `terraform plan` does not perform resource imports. Importing existing resources requires the `terraform import` command.
- **Installs all providers and modules referenced by configuration**: This is incorrect because provider and module installations happen during `terraform init`, not during `terraform plan`.
- **Compares your Terraform code and local state file to the remote state file in a cloud provider and determines if any changes need to be made**: This is partially correct but incomplete. While `terraform plan` compares the state file to the configuration, it also refreshes the state file to ensure it is up to date before generating the plan.

Question 6Correct

When you include a module block in your configuration that references a module from the Terraform Registry, the "version" attribute is required.

True

Explanation

True. When including a module block in your configuration that references a module from the Terraform Registry, the "version" attribute is indeed required to specify the version of the module to use. This ensures that the configuration uses a specific version of the module to maintain consistency and avoid unexpected changes.

Your answer is correct

False

Explanation

False. While it is recommended to include a version attribute when referencing a module from the Terraform Registry to ensure consistency and avoid unexpected changes, it is not technically required. If the version attribute is omitted, Terraform will default to using the latest version of the module, which may lead to potential issues with compatibility and stability.

Overall explanation

The correct answer is: False

Explanation:

The `version` attribute is not required when referencing a module from the Terraform Registry. If you do not specify a version, Terraform will use the latest available version of the module at the time of execution. However, it is generally a best practice to specify a version to ensure consistency and avoid unexpected changes if a new version of the module is released.

Why the Other Option Is Incorrect:

- **True**: This is incorrect because the `version` attribute is optional. It is recommended but not mandatory.

Question 7Correct

Your root module contains a variable named num_servers. Which is the correct way to pass its value to a child module with an input named servers?

${var.num.servers}

Explanation

${var.num.servers} is not the correct syntax to pass a variable value to a child module. The correct syntax for referencing a variable in Terraform is var.variable_name.

servers = num.servers

Explanation

servers = num.servers is not the correct way to pass the value of a variable from the root module to a child module. The correct syntax for passing a variable value is var.variable_name.

servers = var(num.servers)

Explanation

servers = var(num.servers) is not the correct way to pass the value of a variable from the root module to a child module. The correct syntax for referencing a variable in Terraform is var.variable_name.

Your answer is correct

servers = var.num_servers

Explanation

servers = var.num_servers is the correct way to pass the value of a variable named num.servers from the root module to a child module with an input named servers. The syntax var.variable_name is used to reference variables in Terraform.

Overall explanation

The correct answer is: servers = var.num_servers

Explanation:

When passing a variable from the root module to a child module in Terraform, the correct syntax is to reference the variable using `var.<variable_name>`. In this case, the variable `num_servers` is accessed as `var.num_servers`. This ensures that Terraform recognizes it as an input variable defined in your configuration.

Why the Other Options Are Incorrect:

- **${var.num.servers}**: This syntax is outdated. Terraform uses native HCL syntax for variables (e.g., `var.num_servers`) since version 0.12, so using `${}` is no longer necessary or supported.
- **servers = num.servers**: This is incorrect because `num.servers` is not valid Terraform syntax. You must use the `var` keyword to reference variables.
- **servers = var(num.servers)**: This syntax is invalid in Terraform. Variables are referenced with dot notation (`var.num_servers`), not with parentheses.

Question 8Correct

What does the default local Terraform backend store?

Provider plugins

Explanation

Provider plugins are not stored in the default local Terraform backend. Provider plugins are downloaded and managed by Terraform during runtime based on the configurations specified in the Terraform code.

Terraform binary

Explanation

The Terraform binary itself is not stored in the default local Terraform backend. The Terraform binary is the executable file that is used to run Terraform commands and interact with infrastructure providers.

tfplan files

Explanation

tfplan files are not stored in the default local Terraform backend. tfplan files are generated during the planning phase of a Terraform operation and contain the execution plan for the changes to be applied to the infrastructure.

Your answer is correct

State file

Explanation

The default local Terraform backend stores the state file, which contains the current state of the infrastructure managed by Terraform. This state file is used to track the resources created and managed by Terraform, allowing it to make informed decisions about how to apply changes to the infrastructure.

Overall explanation

The correct answer is: State file

Explanation:

The default local backend in Terraform stores the **state file**. This file contains information about the infrastructure that Terraform manages, including its current state and metadata. By default, the state file is stored locally on your machine in a file named `terraform.tfstate`.

Why the Other Options Are Incorrect:

- **Provider plugins**: Provider plugins are downloaded and stored in the `.terraform` directory, but they are not part of the state file. The local backend does not store provider plugins directly.

- **Terraform binary**: The Terraform binary itself is not stored in the state file or backend. You typically download and install the binary separately.
- **tfplan files**: `tfplan` files are created when you run `terraform plan`, but they are not stored by the local backend. They are just files that contain the plan of changes, and you can save them manually if you want.

Question 9 Correct

You are making changes to existing Terraform code to add some new infrastructure. When is the best time to run terraform validate?

Before you run terraform apply, so you can validate your provider credentials.

Explanation

Running terraform validate before terraform apply is not necessary for validating provider credentials. Terraform validate is used to check the syntax and configuration of your Terraform code, not the provider credentials.

Your answer is correct

Before you run terraform plan, so you can validate your code syntax.

Explanation

The best time to run terraform validate is before you run terraform plan. This allows you to validate the syntax and configuration of your Terraform code before generating an execution plan. It helps catch any errors or issues in your code early in the process.

After you run terraform apply, so you can validate your infrastructure.

Explanation

Running terraform validate after terraform apply is not recommended as it does not serve the purpose of validating your infrastructure. Terraform validate is used to check the syntax and configuration of your Terraform code, not the actual infrastructure created.

After you run terraform plan, so you can validate that your state file is consistent with your infrastructure.

Explanation

Running terraform validate after terraform plan is not necessary for validating the consistency of your state file with your infrastructure. Terraform validate is used to check the syntax and configuration of your Terraform code, not the state file.

Overall explanation

The correct answer is: Before you run terraform plan, so you can validate your code syntax.

Explanation:

You should run `terraform validate` **before** running `terraform plan` to ensure that your Terraform configuration files are syntactically correct. This command checks the syntax of your Terraform configuration files, ensuring that there are no errors or issues that would prevent them from being applied. It doesn't interact with the infrastructure or state files, but ensures that your code is valid in terms of structure and syntax.

Why the Other Options Are Incorrect:

- **Before you run terraform apply, so you can validate your provider credentials**: While `terraform validate` checks your configuration files, it does not validate credentials or interact with providers directly. Credentials are typically validated during the `terraform plan` or `terraform apply` process, depending on what operations are being performed.
- **After you run terraform apply, so you can validate your infrastructure**: `terraform apply` is responsible for making changes to your infrastructure. It is not the best time to run `terraform validate` because the goal is to check the configuration **before** applying any changes.
- **After you run terraform plan, so you can validate that your state file is consistent with your infrastructure**: `terraform validate` does not check your state file or compare it to your infrastructure. It only ensures that the code is syntactically correct, but it doesn't check runtime issues or state consistency.

Question 10 Correct

Which method for sharing Terraform modules fulfills the following criteria:

1. Keeps the module configurations confidential within your organization.

2. Supports Terraform’s semantic version constraints.

3. Provides a browsable directory of your modules.

Public Terraform module registry.
Explanation
The Public Terraform module registry is not suitable for keeping module configurations confidential within your organization as anyone can access and view the modules. It also may not support Terraform's semantic version constraints and does not provide a browsable directory of your modules.
A subfolder within your workspace.
Explanation
A subfolder within your workspace is not a method for sharing Terraform modules externally. It is limited to the scope of your workspace and does not provide a way to share modules with others outside of your organization. It also does not support Terraform's semantic version constraints or provide a browsable directory of your modules.
Your answer is correct
HCP Terraform/Terraform Cloud private registry.
Explanation
The HCP Terraform/Terraform Cloud private registry is the correct choice as it allows you to keep module configurations confidential within your organization. It supports Terraform's semantic version constraints and provides a browsable directory of your modules, making it a secure and organized way to share modules.
A Git repository containing your modules.
Explanation
A Git repository containing your modules may not keep module configurations confidential within your organization, as Git repositories can be accessed by anyone with the appropriate permissions. It may support Terraform's semantic version constraints, but it does not provide a browsable directory of your modules like the HCP Terraform/Terraform Cloud private registry does.
Overall explanation
The correct answer is: HCP Terraform/Terraform Cloud private registry.

Explanation:

The **HCP Terraform/Terraform Cloud private registry** allows you to store and share Terraform modules securely within your organization. It meets all the specified criteria:

1. **Keeps the module configurations confidential within your organization**: This registry supports private modules that are only accessible within your organization's Terraform Cloud or Terraform Enterprise account, ensuring confidentiality.
2. **Supports Terraform's semantic version constraints**: The private registry integrates with versioning and allows you to specify semantic version constraints for your modules, helping manage dependencies across different versions of the modules.
3. **Provides a browsable directory of your modules**: Terraform Cloud offers a web-based interface where you can browse and manage your private modules, making it easy for users within your organization to discover and use the modules.

Why the Other Options Are Incorrect:

- **Public Terraform module registry**: While it allows you to browse and version modules, it is a public repository, so it does not keep module configurations confidential within your organization. Anyone can access the modules published there.
- **A subfolder within your workspace**: This method doesn't provide a registry or browsing capabilities. It's simply a local storage approach that lacks versioning and a structured way of discovering modules.
- **A Git repository containing your modules**: While you can store modules in a Git repository, it doesn't offer built-in versioning support specific to Terraform or a browsable interface. You'd need to rely on external tools to manage versions and module discovery.

Question 11Incorrect
When developing a Terraform module, how would you specify the version when publishing it to the official Terraform Registry?
Terraform Registry does not support versioning modules.
Explanation
This choice is incorrect. The Terraform Registry does support versioning modules to allow users to track changes and updates to the modules.
Mention it on the module's configuration page on the Terraform Registry
Explanation

This choice is incorrect. While the module's configuration page on the Terraform Registry provides information about the module, specifying the version is not done on this page.

Correct answer

Tag a release in your module's source control repository.

Explanation

This choice is correct. Tagging a release in your module's source control repository, such as Git, is the standard way to specify the version of a Terraform module when publishing it to the official Terraform Registry.

Your answer is incorrect

Configure it in the module's Terraform code.

Explanation

This choice is incorrect. While configuring the version in the module's Terraform code is important for internal use, it is not the method used to specify the version when publishing the module to the official Terraform Registry.

Overall explanation

The correct answer is: Tag a release in your module's source control repository.

Explanation:

When you publish a Terraform module to the official Terraform Registry, you specify the version of the module by **tagging a release in your module's source control repository** (for example, GitHub). This release tag represents the version of the module, and Terraform Registry uses these tags to identify and offer different versions of the module.

Why the Other Options Are Incorrect:

- **Terraform Registry does not support versioning modules**: This is incorrect because the Terraform Registry does support versioning. Modules are versioned using Git tags, which allow users to specify which version of a module they wish to use.
- **Mention it on the module's configuration page on the Terraform Registry**: The version is not specified on the configuration page. It is determined by the Git tag in the source control repository. The Terraform Registry pulls version information based on the repository's tagged releases.
- **Configure it in the module's Terraform code**: The version is not specified directly in the module's Terraform code. Instead, the version is indicated via a release tag in the source control repository, not the code itself.

Question 12 Correct

What functionality do providers offer in Terraform? (Choose three.)

Your selection is correct

Interact with cloud provider APIs.

Explanation

Providers in Terraform allow users to interact with cloud provider APIs, enabling the provisioning and management of resources in various cloud environments. This functionality is essential for automating infrastructure deployments across different cloud platforms.

Your selection is correct

Provision resources for on-premises infrastructure services.

Explanation

Providers in Terraform also enable the provisioning of resources for on-premises infrastructure services. This functionality allows users to manage and automate infrastructure deployments in their own data centers or private cloud environments using Terraform.

Your selection is correct

Provision resources for public cloud infrastructure services.

Explanation

Another key functionality of providers in Terraform is the ability to provision resources for public cloud infrastructure services. This feature allows users to automate the deployment of resources in popular public cloud platforms such as AWS, Azure, and Google Cloud Platform.

Group a collection of Terraform configuration files that map to a single state file.

Explanation

Grouping a collection of Terraform configuration files that map to a single state file is not a functionality provided by providers in Terraform. This task is related to organizing and managing Terraform configurations within a project, rather than the capabilities offered by providers.

Enforce security and compliance policies.

Explanation

Enforcing security and compliance policies is not a direct functionality of providers in Terraform. While Terraform does offer features for managing infrastructure as code and implementing security best practices, this specific task is typically handled through other tools or processes outside of the provider functionality.

Overall explanation

The correct answers are:

Interact with cloud provider APIs.

Provision resources for on-premises infrastructure services.

Provision resources for public cloud infrastructure services.

Explanation:

- **Interact with cloud provider APIs**: Providers act as an interface between Terraform and the cloud or infrastructure platforms. They interact with the respective cloud or service APIs (like AWS, Google Cloud, Azure, etc.) to create, update, or delete resources based on the Terraform configuration.
- **Provision resources for on-premises infrastructure services**: Terraform providers can also interact with on-premises systems. Providers exist for managing infrastructure on your local network, such as VMware, OpenStack, and other on-prem solutions.
- **Provision resources for public cloud infrastructure services**: A primary function of providers is to provision resources on public cloud platforms such as AWS, Azure, or Google Cloud. They enable Terraform to manage and interact with the infrastructure in these environments.

Why the Other Options Are Incorrect:

- **Group a collection of Terraform configuration files that map to a single state file**: This refers to Terraform workspaces, not providers. Workspaces manage state and not the provisioning of infrastructure resources.
- **Enforce security and compliance policies**: While Terraform can be used in conjunction with other tools (like Sentinel, for example) to enforce security policies, the core functionality of Terraform providers is not to enforce security policies but to provision and manage infrastructure resources.

Question 13Correct

Which command(s) adds existing resources in a public cloud into Terraform state?

terraform init

Explanation

The terraform init command is used to initialize a working directory containing Terraform configuration files. It is not directly related to adding existing resources into Terraform state.

terraform plan

Explanation

The terraform plan command is used to create an execution plan for Terraform. It does not add existing resources into Terraform state but rather shows the changes that Terraform will make when applied.

terraform refresh

Explanation

The terraform refresh command is used to update the state file with the real-world infrastructure. It does not add existing resources into Terraform state but rather updates the state file with the current state of the infrastructure.

Your answer is correct

terraform import

Explanation

The terraform import command is used to import existing infrastructure into Terraform state. This command allows you to bring existing resources under Terraform management and control.

All of these

Explanation
While terraform init, terraform plan, and terraform refresh are important commands in the Terraform workflow, only the terraform import command specifically adds existing resources in a public cloud into Terraform state.
Overall explanation
The correct answer is terraform import.

Explanation:

- **terraform import**: This command is used to bring existing resources that were created outside of Terraform into Terraform's state management. It allows Terraform to track resources that were manually created or provisioned through other means, so that Terraform can manage them going forward.

Why the Other Options Are Incorrect:

- **terraform init**: This command is used to initialize a working directory containing Terraform configuration files. It sets up the backend and downloads necessary provider plugins, but it doesn't interact with the state or import resources.
- **terraform plan**: This command is used to create an execution plan that shows the changes Terraform will make to your infrastructure. It doesn't import resources; rather, it compares the current state to the desired state based on your configuration.
- **terraform refresh**: This command updates the state file with the current state of resources as defined in the cloud provider. It checks for any changes in the infrastructure, but it doesn't import resources into the state.
- **All of these**: This option is incorrect because only `terraform import` is the command specifically designed to import resources into the state.

Question 14Incorrect
Please fill the blank field(s) in the statement with the right words.

It is __ to change the Terraform backend from the default "local" backend to a different one after performing your first terraform apply.
Your answer is incorrect
mandatory
Correct answer
optional
Explanation
The correct answer is optional.

Explanation:

It is **optional** to change the Terraform backend after performing your first `terraform apply`. While it is possible to switch from the default local backend to a different backend (such as an S3 backend, Azure Storage, or Terraform Cloud), it is not mandatory. However, after making changes to the backend, you may need to migrate your state to the new backend manually. The migration process ensures that Terraform is using the correct state and that any resources it manages are accurately tracked.
Question 15Correct
What is modified when executing Terraform in refresh-only mode?
Your Terraform configuration.
Explanation
When executing Terraform in refresh-only mode, the Terraform configuration itself is not modified. Refresh-only mode is used to update the state file with the current state of the infrastructure without making any changes to the configuration.
Your Terraform plan.
Explanation
When running Terraform in refresh-only mode, the Terraform plan is not modified. Refresh-only mode is specifically used to refresh the state of the infrastructure without creating or updating any execution plans.
Your answer is correct
Your state file.
Explanation

The correct choice is C. When Terraform is executed in refresh-only mode, the state file is the only component that is modified. This mode is used to update the state file with the current state of the infrastructure without making any changes to the actual cloud infrastructure.

Your cloud infrastructure.

Explanation

When Terraform is run in refresh-only mode, the cloud infrastructure itself is not modified. This mode is specifically designed to only update the state file with the current state of the infrastructure without making any changes to the actual resources in the cloud.

Overall explanation

The correct answer is Your state file.

Explanation:

In **refresh-only mode**, Terraform updates the local state file by comparing the state file with the actual state of your infrastructure. It does not modify the configuration files, cloud infrastructure, or the plan; it simply synchronizes the local state to reflect any changes that may have occurred outside of Terraform's control (such as changes made manually in the cloud provider's console).

This command helps ensure that the state file accurately reflects the current state of your resources, which is crucial for subsequent operations like `terraform plan` and `terraform apply`.

Example of the command:

```
        bash
        Copy codeterraform plan -refresh-only
```

Why the Other Options Are Incorrect:

- **Your Terraform configuration**: The configuration is not modified during a refresh-only operation; it's just read and compared to the actual infrastructure state.
- **Your Terraform plan**: The plan is not generated or modified in refresh-only mode; it only refreshes the state.
- **Your cloud infrastructure**: No changes are made to the cloud infrastructure. Refresh-only mode doesn't create, modify, or delete any resources in your cloud provider.

Question 16 Correct

Which of the following locations can Terraform use as a private source for modules? (Choose two.)

Public repository on GitHub.

Explanation

Public repository on GitHub is not a private source for modules. Terraform can access public repositories on GitHub for modules, but it cannot be considered a private source.

Public Terraform Registry.

Explanation

Public Terraform Registry is not a private source for modules. Terraform Registry provides a public repository of modules that can be accessed by anyone, so it does not meet the criteria of a private source.

Your selection is correct

Internally hosted VCS (Version Control System) platform.

Explanation

Internally hosted VCS (Version Control System) platform can be used as a private source for modules in Terraform. By hosting the modules on an internally controlled VCS platform, organizations can ensure the privacy and security of their infrastructure code.

Your selection is correct

Private repository on GitHub.

Explanation

Private repository on GitHub can be used as a private source for modules in Terraform. By setting up a private repository on GitHub, organizations can securely store and access their Terraform modules without exposing them to the public.

Overall explanation

The correct answers are:

Internally hosted VCS (Version Control System) platform.

Private repository on GitHub.

Explanation:

Terraform can use **private** sources for modules from different repositories that are hosted internally or privately, such as:

- **Internally hosted VCS platforms**: You can host your own Git repositories or use a private VCS system like GitLab, Bitbucket, or a self-hosted Git server. Terraform allows you to reference modules hosted in these private repositories by specifying the appropriate URL in the module source.
- **Private repository on GitHub**: Similarly, you can host modules in private GitHub repositories. You can reference these modules using a Git URL and authentication (if needed) in your Terraform configuration.

Example:

```
hcl
Copy codemodule "example" {
 source = "git::https://github.com/your-org/your-private-module.git"
}
```

Why the Other Options Are Incorrect:

- **Public repository on GitHub**: While you can use public repositories on GitHub to source modules, it doesn't meet the requirement of being "private". Public repositories are openly accessible.
- **Public Terraform Registry**: This is also a public source for Terraform modules. The Terraform Registry contains publicly shared modules, which contradicts the question's focus on private sources.

Question 17Incorrect

A child module can always access variables declared in its parent module.

Your answer is incorrect

True

Explanation

This statement is incorrect. A child module cannot always access variables declared in its parent module. The scope of variables in Terraform is limited to the module where they are defined. While variables can be passed from a parent module to a child module using input variables, the child module cannot directly access variables defined in the parent module without explicitly passing them.

Correct answer

False

Explanation

This statement is correct. In Terraform, a child module cannot always access variables declared in its parent module. The scope of variables is limited to the module where they are defined. To make variables available to a child module, they need to be explicitly passed as input variables during module instantiation.

Overall explanation

Correct Answer: False

Explanation:

Child modules **cannot directly access variables declared in the parent module**. Variables must be **explicitly passed** from the parent module to the child module using input arguments in the parent configuration. This ensures clear boundaries between modules, enhancing reusability and reducing unintended dependencies.

For example:

Parent Module Configuration:

```
hcl
Copy codemodule "example_child" {
 source = "./child_module"
 input_variable = var.parent_variable
}
```

Child Module Variables Declaration:

```
hcl
Copy codevariable "input_variable" {
 description = "Variable passed from the parent module"
}
```

In this case, `parent_variable` from the parent module is explicitly passed as `input_variable` to the child module. Without this explicit passing, the child module cannot use the parent module's variables.

Question 18Incorrect

What does terraform destroy do?

Destroys the Terraform state file, leaves the infrastructure intact.

Explanation

Terraform destroy does not destroy the Terraform state file itself; it only removes the resources defined in the state file from the infrastructure.

Destroys all Terraform code files in the current directory, leaves the state file intact.

Explanation

Terraform destroy does not delete the Terraform code files; it only removes the resources defined in those files from the infrastructure.

Correct answer

Destroy all infrastructure in the Terraform state file.

Explanation

This is the correct explanation. Terraform destroy removes all the infrastructure resources that were created and managed by Terraform based on the state file.

Your answer is incorrect

Destroys all infrastructure in the configured Terraform provider.

Explanation

Terraform destroy does not delete all infrastructure in the configured provider; it only removes the resources that were created and managed by Terraform.

Overall explanation

Correct Answer: Destroy all infrastructure in the Terraform state file.

Explanation:

When you run `terraform destroy`, Terraform reads the **state file** to determine what infrastructure it has previously managed. It then destroys all resources listed in the state file. This ensures only the infrastructure tracked by Terraform is removed, leaving other unmanaged infrastructure intact.

Why the other options are incorrect:

- **Destroys the Terraform state file, leaves the infrastructure intact:** Terraform does not destroy the state file when running `terraform destroy`; it uses the state file to locate and delete managed infrastructure. The state file remains unless deleted manually.
- **Destroys all Terraform code files in the current directory, leaves the state file intact:** Terraform does not delete code files; it only destroys the resources described in the state file.
- **Destroys all infrastructure in the configured Terraform provider:** Terraform only destroys resources listed in the state file, not all resources in the provider account. For example, if you have resources not managed by Terraform, they remain untouched.

Question 19Incorrect

Which syntax check errors when you run terraform validate?

The code contains tabs for indentation instead of spaces.

Explanation

Terraform requires consistent indentation using spaces, not tabs. Using tabs for indentation can result in syntax check errors when running terraform validate.

Your answer is incorrect

There is a missing value for a variable.

Explanation
If there is a missing value for a variable in the Terraform configuration, it will result in a validation error when running terraform validate. All variables must have values assigned to them to ensure the configuration is valid.
The state file does not match the current infrastructure.
Explanation
The state file not matching the current infrastructure would not result in a syntax check error when running terraform validate. This issue would be related to the state of the infrastructure and not the syntax of the Terraform configuration.
Correct answer
None of the above.
Explanation
This choice is correct because none of the above reasons would cause syntax check errors when running terraform validate. Terraform validate specifically checks for syntax errors in the configuration files and does not validate the state file or missing variable values.
Overall explanation
Correct Answer: None of the above.

Explanation:

When you run `terraform validate`, it verifies the **syntax and internal consistency** of your Terraform configuration files. It ensures that:

- The structure of the code follows Terraform's syntax.
- Required arguments are correctly defined.
- References to other resources and modules are valid.

However, the following are **not checked** by `terraform validate`:

1. **Tabs for Indentation:** Terraform allows tabs or spaces for indentation; it doesn't enforce either.
2. **Missing Variable Values:** `terraform validate` does not check whether variable values are provided, as this is determined during `terraform plan` or `terraform apply`.
3. **State File Mismatch:** `terraform validate` does not interact with the state file or compare it to the actual infrastructure. State file validation occurs during operations like `terraform plan` or `terraform refresh`.

Thus, none of these conditions cause `terraform validate` to throw an error. It strictly ensures configuration syntax correctness and logical consistency.
Question 20Correct
In a HCP Terraform/Terraform Cloud workspace linked to a version control repository, speculative plan runs start automatically when you merge or commit change to version control.
Your answer is correct
True
Explanation
True. In a HCP Terraform/Terraform Cloud workspace linked to a version control repository, speculative plan runs are triggered automatically when changes are merged or committed to the version control system. This allows users to quickly see the potential impact of their changes before applying them to the infrastructure.
False
Explanation
False. In a HCP Terraform/Terraform Cloud workspace linked to a version control repository, speculative plan runs are indeed triggered automatically when changes are merged or committed to the version control system. This feature helps users validate their changes and understand the potential outcomes before applying them to the infrastructure.
Overall explanation
Correct Answer: True

Explanation:

In an HCP Terraform or Terraform Cloud workspace that is linked to a version control repository, speculative plan runs are automatically triggered whenever changes are pushed to the repository (such as commits or merges). This feature allows Terraform to evaluate the potential impact of changes before they are applied. It ensures greater visibility and control, helping teams identify issues or unintended consequences during the planning phase.

Question 21Incorrect

You can configure Terraform to log to a file using the TF_LOG environment variable.

Your answer is incorrect

True

Explanation

This statement is incorrect. Terraform allows you to configure logging to a file using the TF_LOG_PATH environment variable, not the TF_LOG environment variable. TF_LOG is used to set the log level for Terraform, not to specify a log file.

Correct answer

False

Explanation

This statement is correct. Terraform does not support configuring logging to a file using the TF_LOG environment variable. Instead, you can use the TF_LOG_PATH environment variable to specify the file where Terraform logs will be written.

Overall explanation

Correct Answer: False

Explanation:

The `TF_LOG` environment variable in Terraform controls the verbosity of logs but does not configure Terraform to log directly to a file. It outputs logs to `stdout` by default. If you want to log to a file, you must use the `TF_LOG_PATH` environment variable, which specifies the path to the file where logs should be written. Combining `TF_LOG` for verbosity level and `TF_LOG_PATH` for file output is the correct approach for logging to a file in Terraform.

Question 22Correct

Which of the following is not true of Terraform providers?

An individual person can write a Terraform Provider

Explanation

An individual person can write a Terraform Provider. This is true as Terraform allows individuals to create custom providers to interact with various APIs and services that are not natively supported.

A community of users can maintain a provider

Explanation

A community of users can maintain a provider. This is true as Terraform has a strong community of users who contribute to maintaining and improving various providers to support a wide range of services.

HashiCorp maintains some providers

Explanation

HashiCorp maintains some providers. This is true as HashiCorp, the company behind Terraform, maintains and supports official providers for popular services and platforms.

Cloud providers and infrastructure vendors can write, maintain, or collaborate on Terraform providers

Explanation

Cloud providers and infrastructure vendors can write, maintain, or collaborate on Terraform providers. This is true as many cloud providers and infrastructure vendors actively contribute to creating and maintaining Terraform providers to enable seamless integration with their services.

Your answer is correct

None of the above

Explanation

None of the above. This statement is incorrect as all of the statements mentioned in choices A, B, C, and D are true regarding Terraform providers.

Overall explanation

Correct Answer: None of the above

Explanation:

All the listed statements about Terraform providers are true:

- Individual developers can create Terraform providers to manage custom resources or APIs.
- Communities of users often maintain providers for less common platforms or services.
- HashiCorp actively maintains and supports many official providers, especially for major platforms like AWS, Azure, and Google Cloud.
- Cloud providers and infrastructure vendors can collaborate with HashiCorp or independently write and maintain providers to integrate their services with Terraform.

Since all options are valid, "None of the above" is the correct choice.

Question 23Correct
How is the Terraform cloud integration differ from other state backends such as S3, Consul, etc.?

Your answer is correct
It can execute Terraform runs on dedicated infrastructure in Terraform Cloud
Explanation
Terraform Cloud integration allows users to execute Terraform runs on dedicated infrastructure provided by Terraform Cloud. This ensures consistent and reliable execution of Terraform operations without the need to manage infrastructure resources locally.

It doesn't show the output of a terraform apply locally
Explanation
This choice is incorrect because the ability to show the output of a terraform apply locally is not a specific difference between Terraform Cloud integration and other state backends such as S3 or Consul. This feature is related to the terraform apply command itself, not the choice of state backend.

It is only available to paying customers
Explanation
This choice is incorrect because Terraform Cloud integration is not exclusively available to paying customers. While there are paid tiers with additional features, Terraform Cloud also offers a free tier for individual users and small teams to utilize the basic functionality.

All of the above
Explanation
This choice is incorrect because not all of the above statements are true. Only choice A accurately describes a key difference between Terraform Cloud integration and other state backends. Choices B and C do not accurately represent the unique aspects of Terraform Cloud integration.

Overall explanation
Correct Answer: It can execute Terraform runs on dedicated infrastructure in Terraform Cloud

Explanation:

Terraform Cloud is unique compared to other backends like S3 or Consul because it not only provides a remote backend for state storage but also offers additional features, such as executing Terraform runs on its own managed infrastructure. This enables teams to offload the execution of Terraform plans and applies to Terraform Cloud, ensuring consistency and reducing local setup dependencies.

- It **does** show the output of a `terraform apply` in the Terraform Cloud UI, which can also be reviewed by the team.
- Terraform Cloud is available for both free and paid tiers, with the free tier including basic functionality.
- Other backends like S3 or Consul are strictly for storing and locking state files, without the additional execution or collaboration features of Terraform Cloud.

Thus, the key differentiator is Terraform Cloud's ability to run Terraform operations on its infrastructure.

Question 24Incorrect
How would you output returned values from a child module in the Terraform CLI output?

Declare the output in the root configuration

Explanation

Declaring the output in the root configuration alone will not allow you to access the returned values from a child module in the Terraform CLI output. The output must be defined in the child module as well to be accessible.

Your answer is incorrect

Declare the output in the child module

Explanation

Declaring the output in the child module alone will not suffice to output the returned values from a child module in the Terraform CLI output. The output must also be defined in the root configuration to be accessible.

Correct answer

Declare the output in both the root and child module

Explanation

To output returned values from a child module in the Terraform CLI output, you need to declare the output in both the root and child module. This ensures that the values are properly exposed and can be accessed as needed.

None of the above

Explanation

If the output is not declared in either the root or child module, you will not be able to access the returned values from a child module in the Terraform CLI output. Declaring the output in both modules is necessary for proper functionality.

Overall explanation

Correct Answer: Declare the output in both the root and child module

Explanation:

To output returned values from a child module in the Terraform CLI output, you need to:

1. **Declare the output in the child module:** This defines the value that the child module will make available for external access. For example:
   ```
   hcl
   Copy codeoutput "example_value" {
    value = var.some_value
   }
   ```
2. **Reference the child module's output in the root module's output block:** This ensures the output from the child module is accessible and displayed when running `terraform output` in the root module. For example:
   ```
   hcl
   Copy codeoutput "example_value_from_child" {
    value = module.child_module.example_value
   }
   ```

By doing this, the root module "exposes" the child module's output, making it available in the Terraform CLI output.

Declaring the output in only one of the modules (child or root) is insufficient for CLI visibility. Both declarations are necessary to bridge the child module's values to the root and subsequently to the user.

Question 25 Correct

Setting the TF_LOG environment variable to DEBUG causes debug messages to be logged into stdout.

Your answer is correct

True

Explanation

Setting the TF_LOG environment variable to DEBUG is indeed true. When TF_LOG is set to DEBUG, Terraform will log detailed debug messages to stdout, providing additional information about the execution of Terraform commands and operations. This can be helpful for troubleshooting and understanding the inner workings of Terraform during development and deployment processes.

False

Explanation

False. This statement is incorrect. Setting the TF_LOG environment variable to DEBUG does cause debug messages to be logged into stdout. When TF_LOG is set to DEBUG, Terraform will log detailed debug messages to stdout, providing additional information about the execution of Terraform commands and operations.

Overall explanation
Correct Answer: True

Explanation:

Setting the `TF_LOG` environment variable to `DEBUG` enables Terraform's detailed debug-level logging. These debug messages are output to `stdout`, providing detailed information about Terraform's internal operations, including plugin communication, plan generation, and resource management.

This is particularly useful for troubleshooting complex issues or understanding Terraform's behavior. To enable this, you would run a command like:

```
bash
Copy codeexport TF_LOG=DEBUG
```

If you also want to log the output to a file for further analysis, you can combine it with `TF_LOG_PATH`:

```
bash
Copy codeexport TF_LOG_PATH=terraform.log
export TF_LOG=DEBUG
```

This combination will ensure debug logs are written to the specified file instead of just being shown in `stdout`.

Question 26 Incorrect
If you update the version constraint in your Terraform configuration, Terraform will update your lock file the next time you run terraform init.

Correct answer
True
Explanation
True. When you update the version constraint in your Terraform configuration file, Terraform will automatically update the lock file the next time you run `terraform init`. This ensures that the versions of the providers and modules specified in the configuration are correctly locked in the lock file for consistency and reproducibility.

Your answer is incorrect
False
Explanation
False. Updating the version constraint in your Terraform configuration does not automatically update the lock file. The lock file is only updated when you explicitly run `terraform init` after making changes to the configuration. This step is necessary to synchronize the versions of providers and modules with the updated constraints.

Overall explanation
Correct Answer: True

Explanation:

When you update the version constraint for a provider in your Terraform configuration, Terraform will detect the change and update the `terraform.lock.hcl` file the next time you run `terraform init`. The lock file is used to record the specific versions of providers that Terraform should use, ensuring consistent behavior across different runs and environments.

For example, if you change a provider version constraint in your configuration:

```
hcl
Copy codeterraform {
 required_providers {
   aws = {
     source  = "hashicorp/aws"
     version = ">= 4.0.0"
   }
 }
}
```

After running `terraform init`, the lock file will be updated to reflect the latest compatible version of the AWS provider that matches the constraint. This ensures Terraform uses the correct provider versions while also keeping the lock file in sync with the configuration changes.

Question 27 Correct

Which of the following is not considered a safe way to inject sensitive values into a Terraform Cloud workspace?

Your answer is correct

Edit the state file directly just before running terraform apply

Explanation

Editing the state file directly just before running terraform apply is not a safe way to inject sensitive values into a Terraform Cloud workspace. State files should not be manually edited, as they contain critical information about the infrastructure and should be managed securely to prevent unauthorized access to sensitive data.

Set the variable value on the command line with the -var flag

Explanation

Setting the variable value on the command line with the -var flag is a common and safe way to inject sensitive values into a Terraform Cloud workspace. This method allows you to pass sensitive information as a command-line argument without exposing it in plain text within the configuration files or state.

Write the value to a file and specify the file with the -var-file flag

Explanation

Writing the value to a file and specifying the file with the -var-file flag is a secure way to inject sensitive values into a Terraform Cloud workspace. Storing sensitive information in a separate file helps to keep the values secure and separate from the main configuration files, reducing the risk of accidental exposure.

Overall explanation

Correct Answer: Edit the state file directly just before running terraform apply

Explanation:

Editing the state file directly to inject sensitive values is not considered a safe or recommended practice. The state file is a critical component of Terraform's operations, storing information about the infrastructure, and modifying it manually can lead to corruption, inconsistencies, or security risks. Additionally, sensitive data stored in the state file is typically not encrypted, which can expose secrets to unauthorized access.

Here's why the other methods are safer:

1. **Set the variable value on the command line with the `-var` flag**: While this is not the most secure method (as it might expose the value in the command history or logs), it is a supported way to pass sensitive values during execution.
2. **Write the value to a file and specify the file with the `-var-file` flag**: This method is safer as it centralizes sensitive data in a file, which can be secured (e.g., using proper file permissions or encryption).

To securely manage sensitive values, consider using Terraform Cloud's variable management, environment variables, or tools like HashiCorp Vault.

Question 28 Incorrect

Which of these is not a benefit of remote state?

Keeping unencrypted sensitive information off disk

Explanation

Keeping unencrypted sensitive information off disk is a benefit of remote state. By storing the state file remotely, you can prevent sensitive information from being stored in plaintext on disk, enhancing security.

Correct answer

Easily share reusable code modules

Explanation

Easily share reusable code modules is not a benefit of remote state. Remote state primarily helps in storing and managing the state file in a centralized location, making it easier for collaboration and consistency in infrastructure management.

Working in a team

Explanation
Working in a team is a benefit of remote state. Remote state allows team members to collaborate more effectively by providing a centralized location for storing and managing the state file, enabling better coordination and visibility into infrastructure changes.

Your answer is incorrect
Delegate output to other teams
Explanation
Delegate output to other teams is a benefit of remote state. By using remote state, teams can share outputs and resources more efficiently, enabling better collaboration and delegation of responsibilities across different teams working on the same infrastructure.

Overall explanation
Correct Answer: Easily share reusable code modules

Explanation:

Remote state is designed to manage the Terraform state file in a centralized location, enabling collaboration and ensuring consistency when working with infrastructure. However, sharing reusable code modules is not a benefit directly related to remote state. Sharing reusable code modules is handled through the use of Terraform modules and version control systems like Git, rather than through state management.

Here's how remote state provides the other benefits:

1. **Keeping unencrypted sensitive information off disk**: Remote state backends like Terraform Cloud or AWS S3 can securely store the state file, often with encryption, avoiding the need to keep sensitive data locally on disk.
2. **Working in a team**: Remote state enables collaboration by allowing multiple team members to access and update the state file simultaneously, avoiding conflicts or overwrites.
3. **Delegate output to other teams**: Remote state can be used to share outputs (e.g., resource IDs) with other teams or projects by referencing the remote state through the `terraform_remote_state` data source.

Sharing reusable code modules is unrelated to state management and is achieved through module design and repositories.

Question 29Incorrect
Which of the following should you put into the required_providers block?

version >= 3.1
Explanation
The required_providers block in Terraform configuration files specifies the providers required for the configuration. The version constraint in this block should be in the format of "operator version_number" where the operator can be "=", ">", "<", ">=", "<=", or "~>". In this choice, the version constraint is missing the "=" sign, which is required for specifying the exact version or a range of versions.

Correct answer
version = “>= 3.1”
Explanation
The correct format for specifying the version constraint in the required_providers block in Terraform configuration files is "version = ">= 3.1"". This format allows you to define the minimum version required for the provider to work with the configuration. The "=" sign is necessary to indicate the exact version or a range of versions, as specified in this choice.

Your answer is incorrect
version ~> 3.1
Explanation
The version constraint "~> 3.1" in the required_providers block specifies a pessimistic constraint that allows any version of the provider greater than or equal to 3.1 but less than the next major release. While this format is valid for version constraints in Terraform, it is not the correct choice for the required_providers block in this scenario, as it does not match the required format of "version = ">= 3.1"".

Overall explanation
Correct Answer: version = ">= 3.1"

Explanation:

In the `required_providers` block of a Terraform configuration, you should specify the version constraint for the provider using the correct syntax. The correct way to define the version constraint is `version = ">= 3.1"`, which means the provider version should be 3.1 or higher.

Why this is correct:

- The `required_providers` block is used to declare which providers and versions your configuration depends on. The syntax `version = ">= 3.1"` explicitly defines the minimum version required, ensuring compatibility with your configuration.

Why the others are incorrect:

- **`version >= 3.1`**: This is not valid syntax for the `required_providers` block. The `version` argument must use an assignment operator (`=`) and the version constraint must be in quotes.
- **`version ~> 3.1`**: While this is valid syntax for version constraints, it means the provider version must be at least `3.1` but less than `4.0`. This is a different version constraint and does not match the requirement of allowing versions `3.1` or higher without an upper limit.

The correct format ensures clarity and follows Terraform's version constraint guidelines.

Question 30Correct

Which of the following methods, used to provision resources into a public cloud, demonstrates the concept of infrastructure as code?

curl commands manually run from a terminal

Explanation

Manually running curl commands from a terminal does not align with the concept of infrastructure as code, as it lacks the automation and repeatability aspects that are essential in IaC practices. Infrastructure as code involves defining and managing infrastructure through code, not manual commands.

A sequence of REST requests you pass to a public cloud API endpoint

Explanation

Passing a sequence of REST requests to a public cloud API endpoint is a valid method of interacting with cloud resources programmatically, but it does not fully embody the principles of infrastructure as code. Infrastructure as code typically involves using declarative configuration files or scripts to define and manage infrastructure.

Your answer is correct

A script that contains a series of public cloud CLI commands

Explanation

Using a script that contains a series of public cloud CLI commands aligns with the concept of infrastructure as code. By encapsulating the provisioning steps in a script, you can automate the deployment and management of cloud resources, ensuring consistency and reproducibility in your infrastructure setup.

A series of commands you enter into a public cloud console

Explanation

Entering a series of commands into a public cloud console is a manual and error-prone approach to provisioning resources, which does not adhere to the principles of infrastructure as code. Infrastructure as code emphasizes the use of code-based configurations that can be version-controlled, tested, and automated for efficient infrastructure management.

Overall explanation

Correct Answer: A script that contains a series of public cloud CLI commands

Explanation:

Infrastructure as Code (IaC) is the practice of managing and provisioning computing infrastructure through code rather than manual processes. IaC ensures that infrastructure is:

- Declarative or script-based.
- Version-controlled.

- Reproducible and automated.

A script containing public cloud CLI commands demonstrates IaC because it automates the provisioning and configuration of resources using code, which can be stored in version control, reused, and executed consistently across environments.

Why the other options are incorrect:

- **`curl` commands manually run from a terminal**: While `curl` can interact with APIs, running commands manually is not considered IaC because it lacks automation, reproducibility, and version control.
- **A sequence of REST requests you pass to a public cloud API endpoint**: Sending REST requests manually or through an ad-hoc process is not IaC. IaC requires defining infrastructure in a repeatable, automated way.
- **A series of commands you enter into a public cloud console**: This is a manual process and lacks the automation, repeatability, and version control necessary for IaC.

Using a script aligns with the key principles of IaC by enabling consistency, automation, and traceability in infrastructure provisioning.

Question 31 Incorrect

Which two steps are required to provision new infrastructure in the Terraform workflow? (Choose two.)

Correct selection

Plan

Explanation

The "Plan" step in the Terraform workflow is essential for generating an execution plan that shows what actions Terraform will take to change the infrastructure to match the configuration. This step helps in identifying any potential issues or conflicts before actually applying the changes.

Your selection is correct

Apply

Explanation

The "Apply" step is crucial in the Terraform workflow as it is responsible for executing the changes defined in the Terraform configuration. This step actually provisions the new infrastructure based on the execution plan generated during the planning phase.

Import

Explanation

The "Import" step in Terraform is used to import existing infrastructure into Terraform's state. While this step is important for managing existing resources, it is not directly related to provisioning new infrastructure in the Terraform workflow.

Your selection is incorrect

Init

Explanation

The "Init" step in Terraform initializes a working directory containing Terraform configuration files. While this step is necessary to set up the environment and download any required plugins, it is not directly related to provisioning new infrastructure.

Validate

Explanation

The "Validate" step in Terraform is used to validate the configuration files for syntax errors and other issues. While this step is important for ensuring the correctness of the configuration, it is not directly related to provisioning new infrastructure in the Terraform workflow.

Overall explanation

The correct steps required to provision new infrastructure in the Terraform workflow are Plan and Apply.

Here's the explanation for each option:

- **Plan**: This is a required step in the Terraform workflow. It allows Terraform to preview the changes that will be made to the infrastructure before actually applying them. It shows what will be added, modified, or destroyed. This helps avoid accidental changes and ensures that the infrastructure is modified in the expected way.
- **Apply**: This is the next step after the plan. It actually provisions or modifies the infrastructure based on the plan. Running `terraform apply` creates, updates, or destroys the resources as defined in the Terraform configuration files, and makes changes to the infrastructure.

- **Import**: This step is not required to provision new infrastructure. It is used to bring existing infrastructure into the Terraform state. It is typically used when you want to start managing resources that were created outside of Terraform. It doesn't actually provision new infrastructure.
- **Init**: While `terraform init` is an important step to initialize a Terraform working directory and set up necessary plugins and backends, it is not directly responsible for provisioning infrastructure. It is an essential step in the Terraform workflow but comes before the actual provisioning steps like `plan` and `apply`.
- **Validate**: This command is used to check the syntax and configuration of your Terraform files. It ensures there are no errors in the code but doesn't impact the actual provisioning or infrastructure changes.

In conclusion, the required steps to provision new infrastructure are **Plan** and **Apply**.

Question 32Correct

Any user can publish modules to the public Terraform Module Registry.

Your answer is correct

True

Explanation

True. Any user can publish modules to the public Terraform Module Registry. This open and collaborative platform allows users to share their modules with the Terraform community, making it easier for others to discover and use them in their infrastructure as code projects.

False

Explanation

False. Only verified publishers with a Terraform Cloud account can publish modules to the public Terraform Module Registry. This verification process ensures that the modules meet certain quality standards and helps maintain the integrity of the registry.

Overall explanation

The correct answer is True.

Any user with a GitHub account can publish modules to the public Terraform Module Registry. This registry allows users to share their modules with the community, making it easier for others to reuse and manage infrastructure as code.

Question 33Incorrect

How would you output returned values from a child module?

Declare the output in the root configuration

Explanation

Declaring the output in the root configuration alone will not allow you to access the returned values from the child module. Outputs declared in the root configuration are only accessible within the root module itself and cannot be used outside of it.

Your answer is incorrect

Declare the output in the child module

Explanation

Declaring the output in the child module alone will limit the accessibility of the returned values to only within the child module itself. This means that the values will not be available for use in the root module or any other modules.

Correct answer

Declare the output in both the root and child module

Explanation

Declaring the output in both the root and child module allows you to access the returned values from the child module in the root module. This ensures that the values are available for use in the overall configuration and can be utilized as needed in different parts of the Terraform setup.

None of the above

Explanation

Choosing "None of the above" is incorrect because declaring the output in both the root and child module is the correct approach to ensure that the returned values from the child module can be accessed and utilized in the root configuration.

Overall explanation

The correct answer is Declare the output in both the root and child module.

To output values from a child module, you first need to declare the output in the child module using the `output` block. Then, in the root module, you reference the output of the child module to access its value.

Here's an example:

1. In the child module, define an output:

```
hcl
Copy codeoutput "example_output" {
 value = aws_instance.example.id
}
```

1. In the root module, reference the child module's output:

```
hcl
Copy codemodule "child_module" {
 source = "./child_module"
}

output "child_instance_id" {
 value = module.child_module.example_output
}
```

This allows the root module to access values declared in the child module by passing them through as outputs.

Question 34Incorrect

What does running a terraform plan do?

Imports all of your existing cloud provider resources to the state file

Explanation

Running a terraform plan does not import existing cloud provider resources to the state file. It is a command used to generate an execution plan, showing what actions Terraform will take to change the infrastructure to match the configuration.

Correct answer

Compares the state file to your Terraform code and determines if any changes need to be made

Explanation

This choice is correct because running a terraform plan compares the state file, which represents the current state of the infrastructure, to the Terraform code. It then determines if any changes need to be made to align the actual state with the desired state defined in the code.

Imports all of your existing cloud provider resources to your Terraform configuration file

Explanation

Running a terraform plan does not import existing cloud provider resources to your Terraform configuration file. The configuration file defines the desired state of the infrastructure, while the state file tracks the actual state.

Your answer is incorrect

Compares your Terraform code and local state file to the remote state file in a cloud provider and determines if any changes need to be made

Explanation

This choice is incorrect because running a terraform plan does not directly compare the local state file to the remote state file in a cloud provider. Instead, it compares the state file to the Terraform code to determine the necessary changes for the infrastructure.

Overall explanation

The correct answer is Compares the state file to your Terraform code and determines if any changes need to be made.

Running `terraform plan` shows you what changes Terraform intends to make to your infrastructure based on your current configuration and the state file. It compares the configuration you wrote (your Terraform code) with the existing infrastructure (tracked in the state file) and outputs a plan of what will be added, changed, or destroyed. However, it does not make any changes—it only previews them.

Here's why the other options are incorrect:

- **Imports all of your existing cloud provider resources to the state file**: This is incorrect because importing existing resources into the state file is done with `terraform import`, not `terraform plan`.
- **Imports all of your existing cloud provider resources to your Terraform configuration file**: This is incorrect because `terraform plan` does not modify the Terraform configuration. `terraform import` is the command that adds resources to the state, but it does not modify your configuration files automatically.

- **Compares your Terraform code and local state file to the remote state file in a cloud provider and determines if any changes need to be made**: This is partially incorrect because `terraform plan` compares the state file (either local or remote) with the configuration, but it doesn't directly compare the local state with a "remote" state unless using a backend to manage the state. The comparison is typically done with the existing infrastructure tracked in the state file rather than a direct remote comparison.

Question 35Correct

How does Terraform manage most dependencies between resources?

By defining dependencies as modules and including them in a particular order

Explanation

Defining dependencies as modules and including them in a particular order is a valid approach in Terraform, but it is not the primary method for managing dependencies between resources. While module organization can help with managing complex configurations, it is not the main way Terraform handles resource dependencies.

The order that resources appear in Terraform configuration indicates dependencies

Explanation

The order that resources appear in Terraform configuration does not necessarily indicate dependencies between them. Terraform uses a dependency graph to determine the order in which resources should be created or updated, regardless of their appearance in the configuration file. Therefore, relying solely on the order of appearance is not a reliable way to manage dependencies.

Using the depends_on parameter

Explanation

The depends_on parameter in Terraform allows you to explicitly define dependencies between resources. While this can be useful in certain scenarios where you need to enforce a specific order of resource creation, it is not the primary method for managing most dependencies between resources. Terraform's dependency resolution mechanism handles most dependencies automatically.

Your answer is correct

Terraform will automatically manage most resource dependencies

Explanation

Terraform will automatically manage most resource dependencies by analyzing the configuration and building a dependency graph. This graph helps Terraform determine the correct order in which resources should be created or updated based on their dependencies. By default, Terraform handles most dependencies automatically without the need for explicit declarations or manual intervention.

Overall explanation

The correct answer is Terraform will automatically manage most resource dependencies.

Terraform automatically determines the dependencies between resources by analyzing how the resources reference one another. If one resource depends on the output or result of another (e.g., a security group ID, subnet ID, etc.), Terraform automatically understands that dependency and ensures that resources are created or updated in the correct order.

Here's why the other options are incorrect:

- **By defining dependencies as modules and including them in a particular order**: This is not correct. Modules themselves don't inherently define dependencies. Dependencies are generally managed by the resource references or the `depends_on` parameter when needed.
- **The order that resources appear in Terraform configuration indicates dependencies**: This is incorrect. The order of resources in the configuration does not determine the dependency order. Dependencies are defined by the relationships between resources, not their order in the code.
- **Using the depends_on parameter**: While the `depends_on` parameter is used to manually enforce dependencies when Terraform cannot automatically detect them, it is not required in most cases. Terraform can often figure out the dependencies by itself based on the resource references.

In summary, Terraform uses implicit dependencies based on resource references (e.g., referring to the output of another resource) and explicitly manages dependencies only when necessary through the `depends_on` parameter.

Question 36Incorrect

Variables declared within a module are accessible outside of the module.

Your answer is incorrect

True

Explanation

True. Variables declared within a module are accessible outside of the module, allowing for flexibility and reusability of variables across different parts of the Terraform configuration. This can be useful for passing values between modules or sharing common variables throughout the configuration.

Correct answer

False

Explanation

False. Variables declared within a module are scoped to that specific module and are not accessible outside of it. This ensures encapsulation and prevents unintended variable conflicts or dependencies between different parts of the Terraform configuration.

Overall explanation

The correct answer is False.

Variables declared within a module are **not** directly accessible outside of the module. These variables are scoped to the module itself. If you need to make the values of the variables accessible outside the module, you should declare **outputs** in the module and reference those outputs from the root configuration.

Here's why:

- Variables in a module are used to configure the module itself but do not expose any data outside the module unless you explicitly define outputs.
- Outputs are used to expose values from within a module, making them accessible to the root configuration or other modules.

For example, if you define a variable inside a module but don't declare an output, you cannot access that variable from outside the module. You would need to define an output in the module like so:

```
hcl
Copy codeoutput "example_output" {
 value = var.example_variable
}
```

This output can then be accessed from the root module or another module using the syntax `module.<module_name>.<output_name>`.

Question 37Incorrect

The ________ determines how Terraform creates, updates, or deletes resources.

Your answer is incorrect

Terraform configuration

Explanation

Terraform configuration defines the infrastructure resources, their relationships, and configurations in a declarative language. It specifies what resources should be created, updated, or deleted, but it does not directly determine how Terraform performs these actions.

Correct answer

Terraform core

Explanation

Terraform core is responsible for the main functionality of Terraform, including resource management, state management, and execution planning. It determines how Terraform interacts with the infrastructure providers to create, update, or delete resources based on the configuration provided.

Terraform provider

Explanation

Terraform provider is responsible for translating the Terraform configuration into API calls to interact with specific infrastructure providers (such as AWS, Azure, or Google Cloud). While the provider plays a crucial role in resource management, it does not directly determine how Terraform performs resource actions.

Terraform provisioner

Explanation

Terraform provisioner is used to execute scripts on local or remote machines as part of resource creation or updates. While provisioners can be used to configure resources, they do not determine how Terraform creates, updates, or deletes resources in the infrastructure.

Overall explanation

The correct answer is Terraform core.

Terraform core is responsible for determining how to create, update, or delete resources based on the configurations provided. The core takes the Terraform configuration, interprets it, and applies the necessary changes to manage infrastructure.

Here's why the other options are incorrect:

- **Terraform configuration**: The configuration defines what resources are needed, but it's the Terraform core that processes these configurations and determines the actions (create, update, delete) on the resources.
- **Terraform provider**: A provider is responsible for interacting with the APIs of specific services (e.g., AWS, Azure). While the provider manages the actual implementation details of creating, updating, or deleting resources, it is the Terraform core that coordinates these actions based on the configuration.
- **Terraform provisioner**: Provisioners are used to execute scripts or commands on the resources after they are created or updated, but they don't directly determine the resource lifecycle (create, update, delete).

The **Terraform core** manages the resource lifecycle and determines the necessary actions based on the configuration and the current state of the infrastructure.

Question 38 Correct

In a Terraform Cloud workspace linked to a version control repository, speculative plan runs start automatically when you merge or commit changes to version control.

Your answer is correct

True

Explanation

True. In a Terraform Cloud workspace linked to a version control repository, speculative plan runs are triggered automatically when changes are merged or committed to the version control system. This allows users to preview the potential changes before applying them to the infrastructure, ensuring that any unintended consequences can be identified and addressed.

False

Explanation

False. In a Terraform Cloud workspace linked to a version control repository, speculative plan runs do not start automatically when changes are merged or committed to version control. Speculative plan runs need to be manually triggered by the user to preview the potential changes before applying them to the infrastructure.

Overall explanation

The correct answer is True.

In Terraform Cloud, when a workspace is linked to a version control repository, speculative plan runs are triggered automatically when you commit or merge changes to the repository. This feature allows Terraform Cloud to perform a dry-run of your changes (without applying them) and provide insights into potential changes before they are actually applied, helping to identify any issues or unexpected changes ahead of time.

This process ensures that any changes made to the code in version control can be previewed for correctness before being applied to the actual infrastructure, promoting safer and more controlled infrastructure management.

Question 39 Correct

You can access state stored with the local backend by using the terraform_remote_state data source.

True

Explanation

True. The terraform_remote_state data source is used to access state data from a remote backend, not from a local backend. When using a local backend, state data is stored locally on the machine where Terraform is being run, and there is no need to use terraform_remote_state to access it.

Your answer is correct

False

Explanation

False. The terraform_remote_state data source is specifically designed to access state data from a remote backend, not from a local backend. When using a local backend, state data is stored locally on the machine where Terraform is being run, and there is no need to use terraform_remote_state to access it.

Overall explanation

The correct answer is False.

The **terraform_remote_state** data source is used to access state stored in remote backends, such as AWS S3, Terraform Cloud, or others. However, it cannot be used to access state stored with the **local backend**.

For local state, Terraform doesn't provide a specific data source to retrieve state, as it is directly stored on the local machine in the `terraform.tfstate` file. If you need to share state between different configurations, you would need to configure a remote backend instead.

Question 40 Incorrect

When using a module from the public Terraform Module Registry, the following parameters are required attributes in the module block. (Choose two.)

Correct selection

Each of the module’s required inputs

Explanation

Each of the module’s required inputs are necessary attributes in the module block when using a module from the public Terraform Module Registry. These inputs define the configuration values that must be provided to the module for it to function correctly.

Correct selection

The module’s source address

Explanation

The module’s source address is a required attribute in the module block when using a module from the public Terraform Module Registry. This address specifies the location of the module in the registry, allowing Terraform to download and use the module in the configuration.

Terraform Module Registry account token

Explanation

The Terraform Module Registry account token is not a required attribute in the module block when using a module from the public Terraform Module Registry. This token is typically used for authentication and access control purposes, but it is not directly related to specifying the module's configuration.

Your selection is incorrect

Each of the module’s dependencies (example: submodules)

Explanation

Each of the module’s dependencies, such as submodules, are not required attributes in the module block when using a module from the public Terraform Module Registry. Dependencies are managed internally by Terraform and do not need to be explicitly specified in the module block.

Your selection is incorrect

The version of the module

Explanation

The version of the module is not a required attribute in the module block when using a module from the public Terraform Module Registry. While specifying a version can help ensure consistency and predictability in the configuration, it is not mandatory for using a module from the registry.

Overall explanation

The correct answer is Each of the module’s required inputs and The module’s source address.

Here’s why:

- **Each of the module's required inputs**: When using a module, you need to provide the required input variables that the module expects. These inputs are defined by the module author and are essential for the module to function properly. If you don't provide these inputs, Terraform will return an error.
- **The module's source address**: The source address is required to define where Terraform should fetch the module from. In the case of using the public Terraform Module Registry, this would be a reference to the module's location on the registry (e.g., `terraform-aws-modules/vpc/aws`).

Why the other options are incorrect:

- **Terraform Module Registry account token**: An account token is not required to use modules from the public registry. Public modules are accessible without authentication.
- **Each of the module's dependencies (example: submodules)**: While modules can depend on other modules, it is not mandatory to specify dependencies in the module block itself. Dependencies are implicitly handled by Terraform based on the module's internal configuration.
- **The version of the module**: While it's possible and often recommended to specify a version of a module, it is not strictly required. If you don't specify a version, Terraform will use the latest version available.

Question 41 Incorrect

Please fill the blank field(s) in the statement with the right words.

You need to migrate a workspace to use a remote backend. After updating your configuration, what command do you run to perform the migration?

Type your answer in the field provided. The text field is not case-sensitive and all variations of the correct answer are accepted.

__

Your answer is incorrect

terraform init -migrate

Correct answer

terraform init -migrate-state

Question 42 Incorrect

Why should secrets not be hard coded into Terraform code? (Choose two.)

Correct selection

It makes the code less reusable.

Explanation

Hard coding secrets into Terraform code makes the code less reusable because each time the code needs to be reused in a different environment or by a different team, the secrets would need to be manually updated. This increases the risk of exposing sensitive information and makes the code less flexible and adaptable.

Correct selection

Terraform code is typically stored in version control, as well as copied to the systems from which it's run. Any of those may not have robust security mechanisms.

Explanation

Terraform code is typically stored in version control systems and may be copied to various systems for execution. Hard coding secrets into the code increases the risk of exposing sensitive information, as version control systems may not have robust security mechanisms in place to protect the secrets. This practice violates security best practices and can lead to potential security breaches.

Your selection is incorrect

The Terraform code is copied to the target resources to be applied locally and could expose secrets if a target resource is compromised.

Explanation

While it is true that Terraform code is copied to the target resources for local application, this choice does not directly address the reasons why secrets should not be hard coded into Terraform code. The focus should be on the security implications of hard coding secrets, rather than the process of applying Terraform code to resources.

Your selection is incorrect

All passwords should be rotated on a quarterly basis.

Explanation
The statement about rotating passwords on a quarterly basis is not directly related to the question of why secrets should not be hard coded into Terraform code. While password rotation is a good security practice, it does not address the specific risks associated with hard coding secrets into Terraform code.

Overall explanation
The correct answers are:

- **It makes the code less reusable.**
  Hard coding secrets directly in Terraform code reduces the reusability of the code because it binds the configuration to a specific set of credentials. The goal of Infrastructure as Code (IaC) is to keep configurations flexible and reusable across different environments or projects, but hardcoded secrets make it difficult to reuse the code without modifying it.
- **Terraform code is typically stored in version control, as well as copied to the systems from which it's run. Any of those may not have robust security mechanisms.**
  Terraform code, including secrets, may end up in version control repositories (e.g., GitHub) or shared across systems, which might not have adequate security measures in place. If secrets are hardcoded in the code, they can be exposed to unauthorized access when the code is viewed or shared.

These two reasons highlight why hardcoding secrets is considered a bad practice.

Question 43Correct
Where can Terraform not load a provider from?

Your answer is correct
Source code
Explanation
Terraform cannot load a provider directly from the source code. Providers need to be compiled into plugins that Terraform can use. Loading directly from the source code is not a supported method for loading providers.

Plugins directory
Explanation
Terraform can load providers from the plugins directory. This directory contains the compiled provider plugins that Terraform can use to interact with different cloud platforms and services.

Official HashiCorp distribution on releases.hashicorp.com
Explanation
Terraform can load providers from the official HashiCorp distribution on releases.hashicorp.com. This is a trusted source for obtaining provider plugins that are compatible with Terraform and maintained by HashiCorp.

Provider plugin cache
Explanation
Terraform can load providers from the provider plugin cache. The provider plugin cache stores downloaded provider plugins locally to improve performance and reduce the need to download them repeatedly.

Overall explanation
The correct answer is Source code.

Here's why the other options are incorrect:

- **Plugins directory**: This is a valid place from where Terraform can load providers. When you install a provider, Terraform typically places it in the **plugins directory**. It will automatically use the provider plugin from this directory to interact with the cloud or service you're provisioning.
- **Official HashiCorp distribution on releases.hashicorp.com**: Terraform providers can be loaded from the official HashiCorp provider distribution, located on **releases.hashicorp.com**. When you run commands like `terraform init`, Terraform will download the necessary providers from this source if they are not found locally.
- **Provider plugin cache**: Terraform uses a **provider plugin cache** to store the downloaded provider plugins. If a provider has been previously downloaded, Terraform will use it from this cache rather than re-downloading it, making the process more efficient.

Thus, **Source code** is the only option that does not work for loading providers. Providers must be compiled and distributed as binaries, not directly loaded from source code by Terraform.

Question 44Incorrect
terraform init retrieves the source code for all referenced modules.

Correct answer
True
Explanation
True. When running `terraform init`, Terraform retrieves the source code for all referenced modules specified in the configuration files. This ensures that the necessary modules are available locally for use during the Terraform execution process.

Your answer is incorrect
False
Explanation
False. Terraform init does retrieve the necessary modules specified in the configuration files. This step is essential to ensure that Terraform can access and use the modules during the infrastructure provisioning process.

Overall explanation
The correct answer is True.

Here's why:

When you run `terraform init`, it initializes your Terraform configuration by downloading the necessary provider plugins and retrieving the source code for all referenced modules. This includes modules that are stored in the Terraform Module Registry, Git repositories, or local file paths. The `terraform init` command ensures that all dependencies (such as provider plugins and modules) are available for Terraform to use during the execution of `terraform plan` or `terraform apply`.

Why False is incorrect: If you run `terraform init`, it does indeed retrieve the source code for all modules that are referenced in your configuration, ensuring that the Terraform environment is set up correctly with all dependencies before running further Terraform commands.

Question 45Incorrect
Running terraform fmt without any flags in a directory with Terraform configuration files will check the formatting of those files without changing their contents.

Your answer is incorrect
True
Explanation
This statement is incorrect. When running `terraform fmt` without any flags, it will not only check the formatting of the Terraform configuration files but also automatically update them to comply with the standard formatting rules. Therefore, the files' contents will be changed during this process.

Correct answer
False
Explanation
This statement is correct. Running `terraform fmt` without any flags in a directory with Terraform configuration files will not only check the formatting of those files but also automatically update them to comply with the standard formatting rules. This means that the contents of the files will be changed to adhere to the formatting standards.

Overall explanation
The correct answer is False.

Here's why:

Running `terraform fmt` without any flags will **format** the Terraform configuration files according to Terraform's standard style guidelines. This command will modify the contents of the files to ensure they follow proper indentation, alignment, and other formatting conventions. It does not just check the formatting, but actually updates the files if needed.

Why True is incorrect: While it's true that `terraform fmt` checks the formatting of the files, it will also make changes to the files to ensure they conform to Terraform's style conventions. If you don't want `terraform fmt` to modify the files, you would need to use the `-check` flag, like this: `terraform fmt -check`, which will only check formatting without making any changes.

Question 46 Correct
Which provider authentication method prevents credentials from being stored in the state file?

Your answer is correct
Using environment variables
Explanation
Using environment variables is the recommended method for provider authentication as it prevents credentials from being stored in the state file. Environment variables allow you to set sensitive information outside of your Terraform configuration, reducing the risk of exposing credentials in the state file.

Specifying the login credentials in the provider block
Explanation
Specifying the login credentials directly in the provider block is not a secure method for authentication as it can lead to credentials being stored in the state file. This approach increases the risk of exposing sensitive information and is not recommended for preventing credentials from being stored in the state file.

Setting credentials as Terraform variables
Explanation
Setting credentials as Terraform variables may seem like a convenient option, but it does not prevent credentials from being stored in the state file. Terraform variables are part of the configuration and can be included in the state file, potentially exposing sensitive information.

None of the above
Explanation
None of the above choices provide a secure method for preventing credentials from being stored in the state file. It is crucial to use environment variables for provider authentication to ensure that sensitive information is not exposed in the state file.

Overall explanation
The correct answer is Using environment variables.

Explanation: Using environment variables for provider authentication prevents credentials from being stored in the state file. Terraform will read the credentials from the environment when it interacts with the provider, ensuring that sensitive information such as access keys or tokens are not stored in your configuration files or state files.

Why the other options are incorrect:

- **Specifying the login credentials in the provider block**: If you specify credentials directly in the provider block, they may be saved in the configuration file, potentially leading to sensitive information being exposed. In the case of `terraform apply`, sensitive data might also be written to the state file.
- **Setting credentials as Terraform variables**: Similar to specifying credentials in the provider block, storing credentials as Terraform variables can lead to sensitive data being included in the state file, which is not ideal for security.
- **None of the above**: This option is incorrect because using environment variables is the recommended method to prevent credentials from being stored in the state file.

Question 47 Correct
When does Sentinel enforce policy logic during a Terraform Enterprise run?

Before the plan phase
Explanation
Sentinel does not enforce policy logic before the plan phase. It is involved in evaluating and enforcing policies during the execution of Terraform Enterprise runs, but not before the plan phase.

During the plan phase
Explanation
Sentinel does not enforce policy logic during the plan phase. The plan phase is where Terraform generates an execution plan, but Sentinel's policy enforcement occurs at a different stage of the Terraform Enterprise run.

Your answer is correct
Before the apply phase
Explanation

Sentinel enforces policy logic before the apply phase in a Terraform Enterprise run. This means that Sentinel evaluates and enforces policies before any changes are actually applied to the infrastructure, ensuring that the defined policies are met before any modifications take place.

After the apply phase

Explanation

Sentinel does not enforce policy logic after the apply phase. Once the apply phase is completed, Sentinel's role in evaluating and enforcing policies is typically concluded, as its primary function is to ensure compliance and governance before changes are made to the infrastructure.

Overall explanation

The correct answer is Before the apply phase.

Explanation: Sentinel enforces policy logic just before the apply phase during a Terraform Enterprise run. This is done after the plan phase has completed, allowing Sentinel to evaluate whether the proposed changes meet the defined policies before they are actually applied to the infrastructure.

Why the other options are incorrect:

- **Before the plan phase**: Sentinel doesn't enforce policies before the plan phase. The plan phase is when Terraform creates an execution plan, and Sentinel policies need the plan to evaluate against.
- **During the plan phase**: While policies can evaluate the execution plan during the run, they specifically apply before the changes are actually made to the infrastructure, which happens in the apply phase.
- **After the apply phase**: Sentinel policies are not enforced after the apply phase. Once the resources have been applied, it is too late to evaluate or enforce policies for the run.

This ensures policies are enforced while still in a controllable, evaluative stage before any infrastructure changes take place.

Question 48Incorrect

You can reference a resource created with for_each using a Splat (*) expression.

Your answer is incorrect

True

Explanation

True. You can indeed reference a resource created with for_each using a Splat (*) expression in Terraform. This allows you to access attributes of each instance created by the for_each loop, making it a powerful feature for managing multiple instances of a resource.

Correct answer

False

Explanation

False. While Splat expressions can be used to reference attributes of resources created without using for_each, they cannot be used to reference resources created with for_each. In this case, you would need to use the index syntax to access specific instances created by the for_each loop.

Overall explanation

The correct answer is False.

Explanation: You cannot use a Splat (`*`) expression directly to reference resources created with `for_each`. In Terraform, when using `for_each`, each resource instance is referenced by its key, and you cannot use the Splat expression in the same way you would with a resource created using `count`.

For a resource created with `for_each`, you must reference each specific instance using its key rather than a Splat expression.

Why the other option is incorrect:

- **True**: This is incorrect because Splat expressions do not work in the same way for resources created using `for_each`. To access resources created with `for_each`, you need to reference the individual keys or iterate over them explicitly.

For example, when using `for_each`, you would reference each resource as `resource_name[key]`, not using `*`.

Question 49 Incorrect

Which of the following is not a way to trigger terraform destroy?

Your answer is incorrect

Using the destroy command with auto-approve

Explanation

Using the destroy command with auto-approve is a valid way to trigger the terraform destroy operation. This command will automatically approve the destruction of all resources without requiring manual confirmation.

Running terraform destroy from the correct directory and then typing "yes" when prompted in the CLI

Explanation

Running terraform destroy from the correct directory and then typing "yes" when prompted in the CLI is a valid way to trigger the terraform destroy operation. This manual confirmation step ensures that the user is aware of the resources being destroyed.

Passing --destroy at the end of a plan request

Explanation

Passing --destroy at the end of a plan request is not a valid way to trigger the terraform destroy operation. The --destroy flag is not a recognized option in Terraform commands and will not initiate the destruction of resources.

Correct answer

Delete the state file and run terraform apply

Explanation

Deleting the state file and running terraform apply is not a valid way to trigger the terraform destroy operation. This action will not explicitly trigger the destruction of resources and may lead to unexpected behavior in the Terraform environment.

Overall explanation

The correct answer is Delete the state file and run terraform apply.

Explanation:

- **Using the destroy command with auto-approve**: This is a valid way to trigger a `terraform destroy`. The `auto-approve` flag automatically accepts the destruction plan and applies it without manual intervention.
- **Running terraform destroy from the correct directory and then typing "yes" when prompted in the CLI**: This is also a valid way to trigger a `terraform destroy`. When you run `terraform destroy`, it will prompt for confirmation before proceeding. Typing "yes" will confirm and apply the destruction.
- **Passing --destroy at the end of a plan request**: This option is valid as well. You can pass the `--destroy` flag when running `terraform plan` to generate a plan that will destroy resources, and then apply that plan with `terraform apply`.
- **Delete the state file and run terraform apply**: This is **not** a valid way to trigger `terraform destroy`. Deleting the state file causes Terraform to lose the state of the infrastructure, meaning it would try to create all the resources again when running `terraform apply`. This wouldn't trigger destruction, but rather re-creation of resources because Terraform assumes they don't exist anymore.

Question 50 Incorrect

All Terraform Cloud tiers support team management and governance.

Your answer is incorrect

True

Explanation

Terraform Cloud tiers have different features and capabilities, and not all tiers support team management and governance. It is important to choose the appropriate tier based on the organization's needs and requirements.

Correct answer

False

Explanation

This choice is correct because not all Terraform Cloud tiers support team management and governance. It is essential to select the right tier that aligns with the organization's governance and management needs.

Overall explanation

The correct answer is False.

Explanation:

- **Terraform Cloud** has different tiers, and not all tiers offer the same level of features. Team management and governance features, such as role-based access control (RBAC), are typically available in the **Team & Governance** and **Business** tiers. The free tier, **Terraform Cloud Free**, provides limited functionality and may not include all team management and governance capabilities.

So, while team management and governance are available in higher-tier plans, they are not available in all tiers, particularly the free tier.

Question 51 Incorrect

You are writing a child Terraform module which provisions an AWS instance. You want to make use of the IP address returned in the root configuration. You name the instance resource "main".

Which of these is the correct way to define the output value using HCL2?

Correct answer

output "instance_ip_addr" {

value = "${aws_instance.main.private_ip}"

}

Explanation

This choice is correct because it uses the correct syntax for defining an output in HCL2. The "output" keyword is followed by the name of the output value, in this case, "instance_ip_addr". The "value" attribute is used to specify the value that will be returned, which is the private IP address of the AWS instance named "main" in this scenario. The interpolation syntax "${}" is used to reference the attribute of the AWS instance resource.

Your answer is incorrect

output "instance_ip_addr" {

return aws_instance.main.private_ip

}

Explanation

This choice is incorrect because it does not use the correct syntax for defining an output in HCL2. The "output" keyword should be followed by the name of the output value, as shown in the correct choice. Additionally, the "return" keyword is not used in HCL2 for defining outputs. The correct way to specify the value to be returned is by using the "value" attribute.

Overall explanation

The correct answer is output "instance_ip_addr" {

value = "${aws_instance.main.private_ip}"

}

Explanation: In Terraform, when defining an output value, you need to use the correct syntax to reference resource attributes. In **HCL2** (HashiCorp Configuration Language 2), the interpolation syntax (`"${}"`) is optional, but you can still use it for clarity.

The correct way to define the output value for the IP address of an instance is:

```
hcl
Copy codeoutput "instance_ip_addr" {
 value = aws_instance.main.private_ip
}
```

Why output "instance_ip_addr" {

value = "${aws_instance.main.private_ip}"

} **is correct:**

- The correct syntax is using `value = aws_instance.main.private_ip`, which directly references the `private_ip` attribute of the `aws_instance.main` resource.

Why output "instance_ip_addr" {

  return aws_instance.main.private_ip

} **is incorrect:**

- The `return` keyword is not used in Terraform's output definitions. The correct syntax requires using `value =`, not `return`.

Question 52Correct
You have to initialize a Terraform backend before it can be configured.

True
Explanation
Initializing a Terraform backend is a necessary step before configuring it. This process allows Terraform to set up the backend configuration, such as storing state files remotely, enabling locking, and configuring access controls. Without initializing the backend, Terraform will not be able to manage the infrastructure state effectively.

Your answer is correct
False
Explanation
This choice is correct because initializing a Terraform backend is not a mandatory step before configuring it. While initializing the backend is recommended for best practices, it is not strictly required. Terraform can still be used to manage infrastructure without initializing a backend, but it may lead to challenges in state management and collaboration among team members.
Overall explanation
The correct answer is False.

Explanation: You do not need to initialize the backend before it can be configured in Terraform. The backend configuration is usually included in the Terraform configuration file (`main.tf` or other `.tf` files), and when you run `terraform init`, Terraform initializes both the working directory and the backend, which includes configuring and setting up the backend if it's specified in the configuration.

So, the initialization happens during the `terraform init` command, not beforehand.

Question 53Incorrect
You need to constrain the GitHub provider to version 2.1 or greater.

Which of the following should you put into the Terraform 0.12 configuration's provider block?

version >= 2.1
Explanation
The syntax "version >= 2.1" is not valid in the provider block of Terraform configuration. The correct syntax for specifying a version constraint is different.

Your answer is incorrect
version ~> 2.1
Explanation
The syntax "version ~> 2.1" is not valid in the provider block of Terraform configuration. The correct syntax for specifying a version constraint is different.

version = "≤ 2.1"
Explanation
The syntax "version = "≤ 2.1"" is not valid in the provider block of Terraform configuration. The correct syntax for specifying a version constraint is different.

Correct answer
version = ">= 2.1"
Explanation

The correct syntax for constraining the GitHub provider to version 2.1 or greater in the Terraform configuration's provider block is "version = ">= 2.1". This syntax ensures that the provider version used is 2.1 or any version greater than 2.1.

Overall explanation

The correct answer is version = ">= 2.1"

Explanation:

To constrain the version of a provider in Terraform, the `version` argument is used inside the provider block. Here's what the different options do:

- **version >= 2.1**: This is incorrect because it lacks the quotation marks that are required in the provider configuration.
- **version ~> 2.1**: The `~>` operator would allow versions 2.1.x but would not allow any major or minor version changes beyond that (e.g., it would allow 2.1.3 but not 2.2.0). While this could be useful in some cases, it's not the same as specifying "2.1 or greater" for any higher version.
- **version = "≤ 2.1"**: This is invalid syntax and won't work because the `≤` operator is not valid in Terraform.
- **version = ">= 2.1"**: This is correct. It specifies that the version of the GitHub provider must be 2.1 or greater, which is the desired behavior.

Thus, **version = ">= 2.1"** is the correct way to define the provider version constraint for version 2.1 or greater.

Question 54Incorrect

Which option cannot be used to keep secrets out of Terraform configuration files?

Environment Variables

Explanation

Environment Variables are a common method used to keep secrets, such as API keys or passwords, out of Terraform configuration files. By storing sensitive information in environment variables, you can ensure that they are not exposed in plain text within your Terraform code.

Correct answer

Mark the variable as sensitive

Explanation

Marking a variable as sensitive in Terraform is a valid method to keep secrets out of configuration files. When a variable is marked as sensitive, its value will not be displayed in Terraform output or state files, helping to protect sensitive information from being exposed.

A Terraform provider

Explanation

A Terraform provider is not typically used to keep secrets out of configuration files. Providers are used to interact with APIs and services, but they do not inherently provide a mechanism for storing or managing sensitive information securely.

Your answer is incorrect

A -var flag

Explanation

The -var flag in Terraform is used to pass variables from the command line when running Terraform commands. While it can be used to provide values for variables, including sensitive ones, it does not inherently keep secrets out of configuration files. It is more of a runtime configuration option rather than a method for securely managing secrets.

Overall explanation

The correct answer is Mark the variable as sensitive.

Explanation:

- **Environment Variables**: Environment variables are a common and secure method to pass secrets into Terraform without hardcoding them into the configuration files. This method is widely used to keep secrets out of version control.
- **Mark the variable as sensitive**: Marking a variable as `sensitive` in Terraform only prevents Terraform from displaying the value in the output. However, the sensitive value can still be stored in the state file. This method does not keep secrets entirely out of the Terraform configuration or state, as they are still present in the state file and can be accessed by anyone with access to it.

- **A Terraform provider**: While a Terraform provider is used to interact with infrastructure, it doesn't specifically handle secrets or prevent them from being exposed. Providers can use environment variables or other secure methods for authentication but don't directly store secrets in the configuration files.
- **A -var flag**: The `-var` flag allows you to pass variables into Terraform at runtime. It can be used securely with sensitive information, as the value is passed at the command line level and is not hardcoded into configuration files.

So, **Mark the variable as sensitive** is the correct answer because it only prevents secrets from being displayed in Terraform's output, not from being stored in the state file, making it less secure in comparison to the other methods.

Question 55 Correct

Please fill the blank field(s) in the statement with the right words.

FILL BLANK -

In the below configuration, how would you reference the module output vpc_id?

```
module "vpc" {

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

  cidr = "10.0.0.0/16"

  name = "test-vpc"

}
```

__

Your answer is correct

module.vpc.vpc_id

Question 56 Correct

How would you reference the Volume IDs associated with the ebs_block_device blocks in this configuration?

```
resource "aws_instance" "example" {

ami = "ami-abc123"

instance_type - "t2.micro"

ebs_block_device {

device_name = "sda2"

volume_size = 16

}

ebs_block_device {

device_name = "sda3"

volume_size = 20

}

}
```

aws_instance.example.ebs_block_device.[*].volume_id

Explanation
The syntax used in this choice is incorrect. The correct way to reference multiple elements within a block in Terraform is by using the asterisk (*) symbol, not the dot (.) symbol. Therefore, this syntax is invalid for referencing the Volume IDs associated with the ebs_block_device blocks in the configuration.

aws_instance.example.ebs_block_device.volume_id
Explanation
The syntax used in this choice is incorrect. When referencing attributes within a block in Terraform, you need to use the asterisk (*) symbol to specify all elements within that block. Therefore, directly referencing volume_id after ebs_block_device without specifying all elements is not the correct way to access the Volume IDs associated with the ebs_block_device blocks.

aws_instance.example.ebs_block_device[sda2,sda3].volume_id
Explanation
The syntax used in this choice is incorrect. When referencing specific elements within a block in Terraform, you should use the element's index or key, not a comma-separated list of values. Therefore, the syntax ebs_block_device[sda2,sda3].volume_id is not valid for referencing the Volume IDs associated with the ebs_block_device blocks in the configuration.

Your answer is correct
aws_instance.example.ebs_block_device.*.volume_id
Explanation
This choice is correct. The syntax aws_instance.example.ebs_block_device.*.volume_id is the correct way to reference the Volume IDs associated with all ebs_block_device blocks in the configuration. The asterisk (*) symbol is used to specify all elements within the ebs_block_device block, allowing you to access the Volume IDs associated with each block.

Overall explanation
The correct answer is aws_instance.example.ebs_block_device.*.volume_id.

Explanation:

- **aws_instance.example.ebs_block_device.*.volume_id**:
  - This option is correct. It uses the `.*` syntax, which is used to iterate over all `ebs_block_device` elements in the list and retrieve the `volume_id` from each of them. This is the correct way to access the `volume_id` from all `ebs_block_device` blocks associated with the `aws_instance`.

Why the other options are wrong:

- **aws_instance.example.ebs_block_device.[*].volume_id**:
  - This option is incorrect because the square brackets `[*]` syntax is not valid in Terraform for referencing all elements in a list. The correct syntax uses `.*` instead of `[*]` to get all `ebs_block_device` entries.
- **aws_instance.example.ebs_block_device.volume_id**:
  - This is incorrect because it references the `volume_id` of the first `ebs_block_device` block only, not all blocks. Since there are multiple `ebs_block_device` blocks, this will return the `volume_id` of only the first block, and not all of them.
- **aws_instance.example.ebs_block_device[sda2,sda3].volume_id**:
  - This option is incorrect because Terraform does not allow referencing list items by specific attributes (like `device_name`) in this way. You need to use the index or the `.*` operator to get the volume IDs of all `ebs_block_device` blocks.

So, the correct approach is to use **aws_instance.example.ebs_block_device.*.volume_id** to get all the `volume_id` values from each `ebs_block_device` block.

Question 57 Incorrect
You are creating a Terraform configuration which needs to make use of multiple providers, one for AWS and one for Datadog.

Which of the following provider blocks would allow you to do this?

provider {

"aws" {

```
profile = var.aws_profile

region = var.aws_region

}

"datadog" {

api_key = var.datadog_api_key

app_key = var.datadog_app_key

}

}
```

Explanation

This choice is incorrect because the provider block is not correctly structured. In Terraform, each provider should be defined separately with its own block, not nested within a single provider block. Therefore, this configuration would not allow you to use multiple providers for AWS and Datadog.

Correct answer

```
provider "aws" {

profile = var.aws_profile

region = var.aws_region

}

provider "datadog" {

api_key = var.datadog_api_key

app_key = var.datadog_app_key

}
```

Explanation

This choice is correct because it correctly defines two separate provider blocks for AWS and Datadog. Each provider block specifies the necessary configuration parameters for the respective provider, allowing you to use multiple providers in your Terraform configuration.

Your answer is incorrect

```
terraform {

provider "aws" {

profile = var.aws_profile

region = var.aws_region

}

provider "datadog" {

api_key = var.datadog_api_key

app_key = var.datadog_app_key

}
```

}

Explanation

This choice is incorrect because the provider blocks are nested within a terraform block, which is not the correct syntax for defining providers in Terraform. Provider blocks should be defined at the root level of the configuration file, as shown in Choice B. Therefore, this configuration would not allow you to use multiple providers for AWS and Datadog.

Overall explanation

The correct answer is provider "aws" {

profile = var.aws_profile

region = var.aws_region

}

provider "datadog" {

api_key = var.datadog_api_key

app_key = var.datadog_app_key

}

Explanation:

- provider "aws" {
  profile = var.aws_profile
  region = var.aws_region
  }

  provider "datadog" {
  api_key = var.datadog_api_key
  app_key = var.datadog_app_key
  }
    - This option correctly defines two separate provider blocks, one for AWS and one for Datadog. Each provider block is named with the appropriate provider type (`aws` and `datadog`) and configured with the respective variables (like `aws_profile`, `aws_region`, `datadog_api_key`, and `datadog_app_key`). This is the correct way to define and use multiple providers in Terraform.

Why the other options are wrong:

- provider {
  "aws" {
  profile = var.aws_profile
  region = var.aws_region
  }

  "datadog" {
  api_key = var.datadog_api_key
  app_key = var.datadog_app_key
  }
  }
    - This option is incorrect because the provider blocks are wrapped in an additional pair of curly braces without the correct provider block names (like `provider "aws"` and `provider "datadog"`). Terraform expects the provider blocks to be defined with the provider name, but here it incorrectly tries to define them as keys within a single object.

- terraform {
  provider "aws" {
  profile = var.aws_profile
  region = var.aws_region
  }
  provider "datadog" {
  api_key = var.datadog_api_key
  app_key = var.datadog_app_key
  }
  }
    - This option is incorrect because it incorrectly uses the `terraform` block, which is meant for configuration settings such as backend configuration, and not for defining providers. Providers should be defined directly outside the `terraform` block, as in option B.