#2 Terraform Associate Certification 003 - Results

Back to result overview

Attempt 1

All domains

57 all
38 correct
19 incorrect
0 skipped
0 marked

Collapse all questions

Question 1 **Correct**

Terraform can only manage resource dependencies if you set them explicitly with the depends_on argument.

Your answer is correct

False

Explanation

This choice is correct because Terraform can manage resource dependencies without explicitly setting them using the depends_on argument. Terraform uses the resource configuration to determine the dependencies between resources and ensures that they are created or destroyed in the correct order. While setting dependencies explicitly can provide more control, it is not a requirement for Terraform to manage resource dependencies effectively.

True

Explanation

Setting resource dependencies explicitly with the depends_on argument is a common practice in Terraform to ensure that resources are created or destroyed in the correct order. However, Terraform can still manage resource dependencies even without explicitly setting them using the depends_on argument. Terraform analyzes the resource dependencies based on the resource configuration and automatically determines the correct order of resource creation or destruction.

Overall explanation

Correct Answer: False

Explanation:

Terraform automatically manages most dependencies between resources based on the references in your configuration. For example, if one resource references an attribute of another resource, Terraform understands that there is a dependency and will create, update, or delete the resources in the correct order. The `depends_on` argument is only needed when there is an implicit dependency that Terraform cannot detect on its own.

For example, if you have two resources that don't directly reference each other but should be created in a specific order, you could use `depends_on` to enforce that order. In most cases, however, Terraform can handle dependencies without `depends_on` by analyzing resource relationships.

Resources

Create resource dependencies

Question 2 **Correct**

As a member of the operations team, you need to run a script on a virtual machine created by Terraform. Which provision is best to use in your Terraform code?

null-exec

Explanation

The provisioner "null_resource" is used to execute a local provisioner block on a resource that doesn't support remote execution. It does not directly execute scripts on a virtual machine created by Terraform, so it is not the best choice for running a script on a VM in this scenario.

local-exec

Explanation

The provisioner "local-exec" is used to execute a command locally on the machine running Terraform. It does not directly execute scripts on the virtual machine created by Terraform, so it is not the best choice for running a script on a VM in this scenario.

Your answer is correct

remote-exec

Explanation

The provisioner "remote-exec" is specifically designed to execute scripts on a remote machine after it has been created by Terraform. It allows you to run scripts on the virtual machine created by Terraform, making it the best choice for running a script on a VM in this scenario.

file

Explanation

The provisioner "file" is used to copy files from the machine running Terraform to the machine being provisioned. It does not execute scripts on the virtual machine, so it is not the best choice for running a script on a VM in this scenario.

Overall explanation

Correct Answer: `remote-exec`

Explanation:

The `remote-exec` provisioner is best suited for running a script or command directly on a remote virtual machine created by Terraform. This provisioner connects to the VM over SSH (for Linux) or WinRM (for Windows) and then executes the specified commands on the remote system. This is ideal for your scenario, where you need to run a script on the virtual machine itself.

- `null-exec` does not exist in Terraform, so it's not a valid option.
- `local-exec` runs a command on the machine where Terraform is executed (the local machine), not on the remote VM, so it wouldn't work for running scripts on a remote instance.
- `file` is used to transfer files to the remote instance but does not execute scripts or commands. It's typically paired with `remote-exec` if you need to upload a script file and then execute it.

Resources

remote-exec Provisioner

Question 3Correct

How can you trigger a run in a Terraform Cloud workspace that is connected to a Version Control System (VCS) repository?

Only Terraform Cloud organization owners can set workspace variables on VCS connected workspaces

Explanation

This choice is incorrect because setting workspace variables on VCS connected workspaces is not the method to trigger a run in Terraform Cloud. Workspace variables are used for storing sensitive information or configuration settings, but they do not directly trigger a run.

Your answer is correct

Commit a change to the VCS working directory and branch that the Terraform Cloud workspace is connected to

Explanation

This choice is correct because committing a change to the VCS working directory and branch that the Terraform Cloud workspace is connected to will automatically trigger a run in the workspace. Terraform Cloud monitors the connected VCS repository for changes and initiates a run when new commits are detected.

Only members of a VCS organization can open a pull request against repositories that are connected to Terraform Cloud workspaces

Explanation

This choice is incorrect because opening a pull request against repositories connected to Terraform Cloud workspaces is not the method to trigger a run. Pull requests are typically used for code review and collaboration, but they do not directly trigger Terraform runs in the workspace.

Only Terraform Cloud organization owners can approve plans in VCS connected workspaces

Explanation

This choice is incorrect because approving plans in VCS connected workspaces is not the method to trigger a run. Approving plans is a separate step in the Terraform Cloud workflow that occurs after a run has been initiated, executed, and its plan reviewed.

Overall explanation

Correct Answer: Commit a change to the VCS working directory and branch that the Terraform Cloud workspace is connected to

Explanation:

When a Terraform Cloud workspace is connected to a Version Control System (VCS) repository, any change made to the repository, such as a commit to the working directory or branch linked to the workspace, will automatically trigger a new run. This allows Terraform Cloud to detect the changes and plan accordingly.

The other options are incorrect:

- Terraform Cloud workspace variables can be set by users with appropriate permissions, not just organization owners.
- Any collaborator with the correct permissions can open a pull request, not just members of the VCS organization.
- The ability to approve plans in VCS-connected workspaces is not restricted to organization owners; users with the appropriate permissions can approve plans.

Resources
UI- and VCS-driven Run Workflow

Question 4 **Incorrect**
What does Terraform use **.terraform.lock.hcl** file for?

Correct answer
Tracking provider dependencies
Explanation
The .terraform.lock.hcl file is used by Terraform to track provider dependencies. It ensures that the correct versions of providers are used during Terraform runs, helping to maintain consistency and avoid unexpected behavior due to provider version mismatches.

Your answer is incorrect
Storing references to workspaces which are locked
Explanation
The .terraform.lock.hcl file does not store references to workspaces that are locked. Its primary purpose is to track and manage provider dependencies to ensure consistent and reliable Terraform runs.

There is no such file
Explanation
This choice is incorrect as the .terraform.lock.hcl file does exist in Terraform projects and is used for tracking provider dependencies. It is not a file that does not exist.

Preventing Terraform runs from occurring
Explanation
The .terraform.lock.hcl file is not used for preventing Terraform runs from occurring. It is specifically used for managing provider dependencies and versions in a Terraform project.

Overall explanation
Correct Answer: Tracking provider dependencies

Explanation:

Terraform uses the `.terraform.lock.hcl` file to track and lock specific versions of provider dependencies required by a configuration. This file ensures that the exact same versions of providers are used in subsequent Terraform operations, regardless of changes or updates in the provider's version outside the configuration. By locking these versions, it helps maintain consistency across environments and team members, preventing unexpected issues caused by provider version differences.

- "There is no such file" is incorrect because `.terraform.lock.hcl` is automatically generated by Terraform to handle provider version locking.
- "Preventing Terraform runs from occurring" is incorrect, as this file does not prevent Terraform from running; it simply locks provider versions.
- "Storing references to workspaces which are locked" is incorrect because `.terraform.lock.hcl` does not manage workspace locks; it is solely for provider version tracking.

Resources
Dependency Lock File

Question 5 **Correct**
What does `terraform import` allow you to do?

Your answer is correct

Import provisioned infrastructure to your state file

Explanation

Terraform import allows you to import provisioned infrastructure into your Terraform state file. This is useful for bringing existing resources under Terraform management without recreating them.

Use a state file to import infrastructure to the cloud

Explanation

Terraform import does not involve using a state file to import infrastructure to the cloud. It is specifically used to import existing infrastructure into your Terraform state file.

Import a new Terraform module

Explanation

Terraform import does not allow you to import a new Terraform module. It is used for importing existing resources into your Terraform configuration.

Import an existing state file to a new Terraform workspace

Explanation

Terraform import does not involve importing an existing state file to a new Terraform workspace. It is focused on importing existing resources into your Terraform configuration.

Overall explanation

Correct Answer: Import provisioned infrastructure to your state file

Explanation: `terraform import` allows you to bring existing, provisioned infrastructure into Terraform's management by importing it into the state file. This process does not modify or create the resource itself in the cloud but rather links the existing resource to your Terraform state. After the import, Terraform can manage and track that resource as part of your configuration.

- The other options are incorrect because `terraform import` is specifically for importing resources into the state file, not for creating new modules or managing state files directly across workspaces.

Resources

Terraform Import

Question 6Correct

`terraform validate` validates that your infrastructure matches the Terraform state file.

Your answer is correct

False

Explanation

This statement is correct. Terraform validate does not validate that your infrastructure matches the Terraform state file. It is used to check the syntax and configuration of your Terraform files before applying them to your infrastructure.

True

Explanation

This statement is incorrect. Terraform validate does not validate that your infrastructure matches the Terraform state file. Instead, it checks the syntax and configuration of your Terraform files to ensure they are valid and can be successfully applied.

Overall explanation

Correct Answer: False

Explanation: `terraform validate` does not check if your infrastructure matches the Terraform state file. Instead, it validates the syntax and structure of your Terraform configuration files (e.g., `.tf` files). It checks for correct syntax, valid variable definitions, and proper resource configurations but does not interact with the state or the actual infrastructure.

- To check the actual state of your infrastructure against the configuration, you would use `terraform plan`, which compares the Terraform state with the current infrastructure and plans any necessary changes.

Resources

Terraform validate command

Question 7Incorrect

How can `terraform plan` help in the development process?

Initializes your working directory containing your Terraform configuration files

Explanation

The choice of initializing your working directory containing Terraform configuration files is related to the terraform init command, not the terraform plan command. Terraform init is used to initialize a working directory containing Terraform configuration files and download any necessary plugins.

Formats your Terraform configuration files

Explanation

Formatting Terraform configuration files is done using the terraform fmt command, not the terraform plan command. Terraform fmt is used to rewrite Terraform configuration files to a canonical format and style.

Your answer is incorrect

Reconciles Terraform's state against deployed resources and permanently modifies state using the current status of deployed resources

Explanation

Reconciling Terraform's state against deployed resources and permanently modifying state using the current status of deployed resources is done through the terraform apply command, not the terraform plan command. Terraform apply is used to apply the changes required to reach the desired state of the configuration.

Correct answer

Validates your expectations against the execution plan without permanently modifying state

Explanation

Terraform plan is a command that generates an execution plan showing what actions Terraform will take to change the infrastructure to match the current configuration. It allows you to validate your expectations against the execution plan without actually making any changes to the infrastructure state. This helps in ensuring that the changes you are about to apply are correct and align with your intentions before actually applying them.

Overall explanation

Validates your expectations against the execution plan without permanently modifying state

The `terraform plan` command is used to preview the changes Terraform will make before applying them. It shows a detailed execution plan, allowing you to verify that your configuration will produce the expected changes in the infrastructure, without making any permanent modifications to the state or resources. This is essential for validating your changes and preventing unintended outcomes.

Resources

Command: plan

Question 8Correct

How is terraform import run?

All of the above

Explanation

None of the above choices are correct. Terraform import is not automatically run as part of terraform init, plan, or refresh. It requires an explicit call to the terraform import command to import existing infrastructure resources.

Your answer is correct

By an explicit call

Explanation

Terraform import is run by an explicit call. When you want to import existing infrastructure into Terraform, you need to explicitly call the terraform import command to bring that resource under Terraform management.

As a part of terraform plan

Explanation

Terraform import is not run as a part of terraform plan. Terraform plan is used to create an execution plan for the infrastructure changes before actually applying them.

As a part of terraform init

Explanation

Terraform import is not run as a part of terraform init. Terraform init is used to initialize a working directory containing Terraform configuration files.

As a part of terraform refresh

Explanation

Terraform import is not run as a part of terraform refresh. Terraform refresh is used to reconcile the state Terraform knows about (via the state file) with the real-world infrastructure.

Overall explanation
Correct Answer: By an explicit call

Explanation:

`terraform import` is used to import existing resources into Terraform's state management, allowing you to bring infrastructure managed outside of Terraform into its control. It is not automatically triggered by `terraform init`, `terraform plan`, or `terraform refresh`. Instead, it requires an explicit command to be run by the user. For example, you would run `terraform import <resource_type>.<resource_name> <resource_id>` to import a specific resource into the Terraform state.

- The other options are incorrect because:
  - `terraform init` is used for initializing the working directory, not for importing resources.
  - `terraform plan` generates a plan for changes, but it does not import existing resources.
  - `terraform refresh` updates the Terraform state to reflect the current state of the infrastructure, but it does not import new resources into the state.

Resources
Import Usage

Question 9Correct
When using Terraform to deploy resources into Azure, which scenarios are true regarding state files? (Choose two.)

Your selection is correct
When a change is made to the resources via the Azure Cloud Console, the current state file will not be updated
Explanation
If changes are made to resources in Azure using the Azure Cloud Console, the current state file maintained by Terraform will not be automatically updated. It is important to run a plan or apply operation in Terraform to synchronize the state file with the actual infrastructure.

Your selection is correct
When a change is made to the resources via the Azure Cloud Console, Terraform will update the state file to reflect them during the next plan or apply
Explanation
Terraform is designed to track the state of the infrastructure it manages. When changes are made to resources in Azure using the Azure Cloud Console, Terraform will update the state file to reflect these changes during the next plan or apply operation.

When a change is made to the resources via the Azure Cloud Console, the changes are recorded in a new state file
Explanation
When a change is made to the resources via the Azure Cloud Console, Terraform does not automatically create a new state file. Instead, it updates the existing state file to reflect the changes during the next plan or apply operation.

When a change is made to the resources via the Azure Cloud Console, the changes are recorded in the current state file
Explanation
Contrary to this statement, when changes are made to resources via the Azure Cloud Console, Terraform does not automatically record these changes in the current state file. Instead, Terraform requires a plan or apply operation to update the state file with the changes made outside of Terraform.

Overall explanation
Correct Answer:

- **When a change is made to the resources via the Azure Cloud Console, Terraform will update the state file to reflect them during the next plan or apply**
- **When a change is made to the resources via the Azure Cloud Console, the current state file will not be updated**

Explanation:

- When changes are made directly to resources outside of Terraform (such as via the Azure Cloud Console), Terraform will not immediately know about those changes. The state file only tracks what Terraform has created or managed. During the

next `terraform plan` or `terraform apply`, Terraform will compare the existing state file with the current infrastructure state and try to reconcile differences, updating the state file if necessary.

- In this case, the state file will not be updated automatically as the change was made outside of Terraform. Terraform will detect the difference when you run a plan and decide how to adjust the infrastructure based on the configuration and the current state.
- Terraform does not create new state files when changes are made via the Azure Cloud Console. The state file remains the same, and any changes detected are reconciled during the next Terraform operation.

Question 10 **Incorrect**

If writing Terraform code that adheres to the Terraform style conventions, how would you properly indent each nesting level compared to the one above it?

Your answer is incorrect

With a tab

Explanation

Using a tab for indenting each nesting level is not the preferred method for Terraform code. Consistent indentation is essential for readability and maintainability, and using tabs can lead to inconsistent formatting across different editors and platforms.

With four spaces

Explanation

Indenting each nesting level with four spaces is not the correct way to adhere to Terraform style conventions. The recommended indentation for Terraform code is two spaces for each nesting level.

Correct answer

With two spaces

Explanation

Properly indenting each nesting level with two spaces is the correct way to adhere to Terraform style conventions. Consistent indentation helps improve code readability, maintainability, and collaboration among team members working on the same Terraform project.

With three spaces

Explanation

Indenting each nesting level with three spaces is not the standard practice for Terraform code. The Terraform style conventions recommend using two spaces for indentation to maintain consistency and readability in the codebase.

Overall explanation

Correct Answer: With two spaces

Explanation:

In Terraform, the standard style convention is to use two spaces for indentation at each nesting level. This helps maintain consistency and readability across Terraform configuration files. Adhering to this convention makes your code cleaner and easier to collaborate on, particularly in larger teams. Using tabs or other forms of indentation (like four spaces or three spaces) is not recommended as it deviates from the community-driven conventions.

Resources

Terraform config language style guide

Question 11 **Incorrect**

Terraform and Terraform providers must use the same major version number in a single configuration.

Correct answer

False

Explanation

This statement is correct. Terraform and Terraform providers do not need to use the same major version number in a single configuration. It is possible to use different versions of Terraform providers with a specific version of Terraform, as long as they are compatible and do not create any issues.

Your answer is incorrect

True

Explanation

This statement is incorrect. Terraform and Terraform providers do not necessarily have to use the same major version number in a single configuration. Different versions of Terraform providers can be used with a specific version of Terraform as long as they are compatible and do not cause any conflicts.

Overall explanation

Correct Answer: False

Explanation:

Terraform and its providers do not need to have the same major version number in a single configuration. You can use different versions of Terraform and its providers as long as they are compatible. Terraform follows a versioning scheme that allows flexibility in provider versioning, and the Terraform CLI supports using providers of different versions through version constraints.

However, it is important to ensure that the provider version is compatible with the version of Terraform you are using, which is why specifying provider version constraints in your configuration can help manage this compatibility.

Resources

Terraform version constraints

Question 12 **Correct**

Which of these options is the most secure place to store secrets for connecting to a Terraform remote backend?

None of above

Explanation

Choosing none of the above options means not securely storing secrets for connecting to a Terraform remote backend. It is crucial to follow security best practices and choose a secure method, such as using environment variables, to protect sensitive information.

Your answer is correct

Defined in Environment variables

Explanation

Storing secrets in environment variables is considered a best practice for security as they are not stored directly in the Terraform configuration files. Environment variables are not committed to version control and provide an extra layer of security by keeping sensitive information separate from the codebase.

Inside the backend block within the Terraform configuration

Explanation

Storing secrets inside the backend block within the Terraform configuration is not recommended for security reasons. This approach exposes the secrets within the configuration files, which can be a security risk if the files are shared or exposed unintentionally.

Defined in a connection configuration outside of Terraform

Explanation

Storing secrets in a connection configuration outside of Terraform can be a viable option, but it may not be as secure as using environment variables. Depending on how the connection configuration is managed, there may be risks of exposure or unauthorized access to the secrets.

Overall explanation

Correct Answer: **Defined in Environment variables**

Explanation: Storing secrets in environment variables is generally the most secure method for managing sensitive data, such as credentials for connecting to a Terraform remote backend. Environment variables are not stored in your Terraform configuration or state files, reducing the risk of accidental exposure, especially if the code is checked into a version control system. Many CI/CD systems and Terraform Cloud allow you to securely store these variables without hardcoding them in files.

- Storing secrets **inside the backend block within the Terraform configuration** is not secure because the Terraform configuration files might be stored in version control or shared, potentially exposing sensitive data.
- Storing secrets **defined in a connection configuration outside of Terraform** is a less secure option because it can still be exposed depending on the system's access and storage mechanism.
- **None of the above** is incorrect because environment variables are recognized as the best practice for securely storing sensitive data like API keys or secrets.

Resources
Credentials and sensitive data

Question 13 Correct
You would like to reuse the same Terraform configuration for your development and production environments with a different state file for each.
Which command would you use?

terraform init
Explanation
The `terraform init` command is used to initialize a working directory containing Terraform configuration files. It is not specifically designed for managing different state files for different environments.

Your answer is correct
terraform workspace
Explanation
The `terraform workspace` command allows you to manage multiple states for the same configuration. By creating separate workspaces for development and production environments, you can maintain different state files while using the same configuration.

terraform state
Explanation
The `terraform state` command is used to interact with Terraform state. While it is essential for managing state, it does not directly address the need for different state files for different environments.

terraform import
Explanation
The `terraform import` command is used to import existing infrastructure into Terraform. It is not directly related to managing different state files for different environments.

Overall explanation
Correct Answer: terraform workspace

Explanation:

Using `terraform workspace` allows you to create and switch between multiple workspaces in Terraform. Each workspace has its own separate state file, which enables you to use the same Terraform configuration code for different environments (like development and production) while maintaining distinct state files for each environment. This avoids conflicts and helps keep resources organized by environment.

- **terraform import** is used to import existing infrastructure into the Terraform state but does not manage multiple states for different environments.
- **terraform state** is for manipulating Terraform's state, such as moving resources or removing them, but it does not create or manage separate workspaces.
- **terraform init** initializes a working directory containing Terraform configuration files but does not support creating multiple environments with different states.

Using workspaces is the best method in this case for managing multiple environments with separate state files.

Resources
Workspaces

Question 14 Incorrect
Module variable assignments are inherited from the parent module and do not need to be explicitly set.

Correct answer
False
Explanation
This statement is correct. Module variable assignments are not inherited from the parent module and must be explicitly set within each module. This allows for better control and isolation of variables within each module, preventing unintended variable conflicts or dependencies.

Your answer is incorrect

True

Explanation

This statement is incorrect. Module variable assignments are not inherited from the parent module by default. Each module must explicitly set its own variable assignments, and they are not automatically passed down from the parent module.

Overall explanation

Correct Answer: False

Explanation:

In Terraform, module variables are not automatically inherited from the parent module. You must explicitly pass values for module variables when you call a module. If you do not pass a value for a variable, Terraform will use the default value defined within the module (if specified). If no default is set and the variable is required, Terraform will prompt you to provide the value.

- The behavior described in the statement would only apply if the module variable has a default value. Otherwise, the variable must be explicitly assigned either via a variable file or as part of the module block in the parent module.

Question 15 Incorrect

When you use a **remote backend** that needs authentication, HashiCorp recommends that you:

Write the authentication credentials in the Terraform configuration files

Explanation

Writing authentication credentials directly in the Terraform configuration files is not recommended by HashiCorp. This approach can lead to potential security vulnerabilities, as the credentials may be exposed in plain text within the codebase. It is important to keep sensitive information separate from the configuration files.

Your answer is incorrect

Keep the Terraform configuration files in a secret store

Explanation

Keeping Terraform configuration files in a secret store is a good practice for managing sensitive information, but it does not specifically address the issue of securely handling authentication credentials for a remote backend. While using a secret store can help protect other sensitive data, it is still important to follow HashiCorp's recommendation of using partial configuration for authentication credentials.

Correct answer

Use partial configuration to load the authentication credentials outside of the Terraform code

Explanation

Using partial configuration to load authentication credentials outside of the Terraform code is recommended by HashiCorp to enhance security. This approach helps to separate sensitive information from the actual Terraform configuration, reducing the risk of exposing credentials in version-controlled files.

Push your Terraform configuration to an encrypted git repository

Explanation

Pushing Terraform configuration to an encrypted git repository may provide some level of security for the code itself, but it does not address the issue of securely managing authentication credentials. Storing credentials in version-controlled files, even in an encrypted repository, can still pose a security risk.

Overall explanation

Correct Answer: Use partial configuration to load the authentication credentials outside of the Terraform code

Explanation:

HashiCorp recommends using partial configuration to load authentication credentials outside of the Terraform code, typically through environment variables or external systems like HashiCorp Vault. This approach avoids embedding sensitive credentials directly in the Terraform configuration files, which can lead to accidental exposure in version control or logs.

- Writing credentials directly in the Terraform configuration files is a security risk because it exposes sensitive information in plain text.
- Storing configuration files in an encrypted Git repository may offer some level of protection, but it still involves placing sensitive information in the codebase.

- Keeping the configuration in a secret store is good for sensitive data but does not address the secure loading of credentials for backend authentication.

By using external configuration, such as environment variables or a secret management tool like Vault, the sensitive credentials are not exposed in the code itself, ensuring better security practices.

Resources
Remote backend

Question 16 **Incorrect**
Please fill the blank field(s) in the statement with the right words.

Terraform provisioners that require authentication can use the __ block.

Your answer is incorrect
provision
Correct answer
connection
Explanation
Correct Answer: connection

Explanation:

The `connection` block in Terraform provisioners is used to specify authentication details required for connecting to a resource, such as a virtual machine. Within the `connection` block, you can define parameters like `type`, `user`, `password`, `host`, and `private_key`, allowing Terraform to securely access the resource for configuration or provisioning actions.

Resources
Provisioner Connection Settings

Question 17 **Correct**
You have created a **`main.tf`** Terraform configuration consisting of an application server, a database, and a load balancer.

You ran `terraform apply` and all resources were created successfully. Now you realize that you do not actually need the load balancer so you run `terraform destroy` without any flags What will happen?

Terraform will immediately destroy all the infrastructure
Explanation
Terraform will not immediately destroy all the infrastructure when running terraform destroy without any flags. Instead, it will prompt you to confirm the destruction of all resources, giving you the opportunity to review and confirm before proceeding with the destroy operation.

Terraform will destroy the application server because it is listed first in the code
Explanation
The order of resources in the Terraform configuration file does not determine the order in which resources are destroyed. Terraform follows dependencies and relationships between resources to determine the correct destroy order, so the application server will not be destroyed just because it is listed first in the code.

Terraform will destroy the main.tf file
Explanation
Terraform does not destroy the main.tf file when running terraform destroy. The main.tf file is the configuration file that defines the infrastructure, and it is not affected by the destroy operation.

Terraform will prompt you to pick which resource you want to destroy
Explanation
Terraform does not prompt you to pick which specific resource you want to destroy when running terraform destroy without any flags. The destroy operation will remove all the resources that were created by the Terraform configuration.

Your answer is correct
Terraform will prompt you to confirm that you want to destroy all the infrastructure
Explanation

When running terraform destroy without any flags, Terraform will prompt you to confirm that you want to destroy all the infrastructure. This is a safety measure to prevent accidental destruction of resources and allows you to review and confirm the destruction before proceeding.

Overall explanation

Correct Answer: Terraform will prompt you to confirm that you want to destroy all the infrastructure.

Explanation:

When you run `terraform destroy` without specifying any flags or resources, Terraform will attempt to destroy all the infrastructure that was created by the Terraform configuration. It will prompt you for confirmation before proceeding with the destruction of all the resources. This safeguard ensures that destructive actions are intentional and that the user is aware of the consequences.

- Terraform does not destroy resources based on the order in which they are listed in the code. Therefore, Terraform will not destroy the application server just because it's listed first.
- Terraform does not delete the `.tf` files. It manages infrastructure resources but not the configuration files themselves.
- Terraform does not prompt you to pick specific resources to destroy unless flags like `-target` are used to specify individual resources.
- Terraform will always prompt for confirmation before destroying all resources unless you use the `-auto-approve` flag to bypass this confirmation step.

Question 18 **Correct**

Terraform plan updates your state file.

True

Explanation

The statement implies that the 'terraform plan' command modifies the state file, which is not accurate. The 'terraform plan' command is used to create an execution plan and to show what actions will be performed on the infrastructure, but it does not make any changes to the state file itself.

Your answer is correct

False

Explanation

This statement is correct because the 'terraform plan' command does not update the state file. The state file is only updated when changes are actually applied to the infrastructure using the 'terraform apply' command.

Overall explanation

Correct Answer: False

Explanation:

The `terraform plan` command does not update the state file. Instead, it generates an execution plan by comparing the current state of resources in the state file with the desired state defined in the configuration files. This allows users to preview the changes that would occur upon running `terraform apply` without making any actual modifications to infrastructure or updating the state.

Only commands like `terraform apply` or `terraform refresh` actually modify the state file to reflect the current state of resources or apply changes to achieve the desired configuration.

Resources

Terraform plan

Question 19 **Correct**

You just upgraded the version of a provider in an existing Terraform project. What do you need to do to install the new provider?

Run terraform apply -upgrade

Explanation

Running `terraform apply -upgrade` is not the correct command to install a new provider after upgrading the version in an existing Terraform project. The `apply` command is used to apply changes to the infrastructure, not to install providers.

Run terraform refresh

Explanation

Running `terraform refresh` is not the correct command to install a new provider after upgrading the version in an existing Terraform project. The `refresh` command is used to update the state file with the real-world infrastructure, not to install providers.

Your answer is correct

Run terraform init -upgrade

Explanation

Running `terraform init -upgrade` is the correct command to install the new provider after upgrading the version in an existing Terraform project. The `init` command initializes the Terraform working directory and downloads the necessary plugins, including the updated provider.

Upgrade your version of Terraform

Explanation

Upgrading your version of Terraform is not the necessary step to install a new provider after upgrading the version in an existing Terraform project. While it is important to keep Terraform up to date, running `terraform init -upgrade` is specifically required to install the new provider.

Overall explanation

Correct Answer: Run terraform init -upgrade

Explanation:

When you upgrade the version of a provider in your Terraform configuration, you need to reinitialize your working directory with `terraform init -upgrade`. This command will download and install the new version of the provider specified in your configuration, ensuring that Terraform uses the latest version for subsequent operations.

The other options are incorrect because:

- `terraform apply -upgrade` does not exist and is not relevant for installing or upgrading a provider.
- `terraform refresh` updates the state file with the latest information from the cloud provider, but it doesn't install or upgrade providers.
- Upgrading Terraform itself is not necessary just to update a provider version; `terraform init -upgrade` will handle the provider updates independently of the Terraform binary version.

Resources

Terraform init

Question 20Correct

You have a simple Terraform configuration containing one virtual machine (VM) in a cloud provider. You run terraform apply and the VM is created successfully.

What will happen if you terraform apply again immediately afterwards without changing any Terraform code?

Terraform will create another duplicate VM

Explanation

Terraform is designed to be idempotent, meaning that running the same configuration multiple times should result in the same state. If you run terraform apply again without any changes in the code, Terraform will not create another duplicate VM. It will recognize that the VM already exists in the state file and no action is needed.

Terraform will apply the VM to the state file

Explanation

When you run terraform apply, Terraform compares the current state with the desired state defined in the code and makes any necessary changes to align them. If you run terraform apply again immediately afterwards without changing any Terraform code, Terraform will not apply the VM to the state file because the VM is already in the desired state.

Your answer is correct

Nothing

Explanation

If you run terraform apply again immediately after successfully creating a VM without making any changes to the Terraform code, Terraform will detect that there are no changes to be made. Therefore, it will not take any action and the output will indicate that nothing has changed.

Terraform will terminate and recreate the VM

Explanation

Terraform uses the state file to track the resources it manages. If you run terraform apply again without changing any Terraform code, Terraform will compare the current state with the desired state defined in the code. Since the VM already exists in the state file and no changes are made in the code, Terraform will not terminate and recreate the VM.

Overall explanation

Correct Answer: Nothing

Explanation:

When you run `terraform apply` immediately after successfully creating the VM, Terraform will compare the current state of the infrastructure (as reflected in the state file) with the desired configuration (from the Terraform code). Since no changes have been made to the configuration, Terraform will determine that the infrastructure is already in the desired state and will not make any modifications.

- **Terminating and recreating the VM** is incorrect because Terraform only recreates resources if there is a change in the configuration or if the resource has been manually deleted or tainted.
- **Creating a duplicate VM** is also incorrect because Terraform detects that the resource already exists and does not create a new one.
- **Applying the VM to the state file** is not correct because the state file is already updated with the VM's details, so no changes are needed.

In this scenario, Terraform simply acknowledges that the configuration is already applied as expected, and no changes are made.

Question 21 **Incorrect**

Which of the following is not an action performed by `terraform init`?

Initialize a configured backend

Explanation

Terraform init initializes a configured backend, which includes setting up the backend configuration for storing state files remotely or locally.

Your answer is incorrect

Retrieve the source code for all referenced modules

Explanation

Terraform init retrieves the source code for all referenced modules, ensuring that the necessary module dependencies are available for the configuration.

Correct answer

Create a sample main.tf file

Explanation

Terraform init does not create a sample main.tf file. It initializes the working directory, but it does not generate any sample configuration files.

Load required provider plugins

Explanation

Terraform init loads the required provider plugins, which are necessary for interacting with the infrastructure providers specified in the Terraform configuration.

Overall explanation

Correct Answer: Create a sample main.tf file

Explanation:

The `terraform init` command is responsible for initializing the working directory for Terraform operations. It performs several important tasks, but it does not create files like `main.tf` or any other Terraform configuration files.

- **Initializing a configured backend**: This action is performed to ensure that the state is stored in the specified backend (e.g., local, S3, Terraform Cloud).
- **Retrieving the source code for all referenced modules**: `terraform init` downloads any modules defined in your configuration files so they are available for use.
- **Loading required provider plugins**: `terraform init` automatically downloads and installs the necessary provider plugins that are specified in the configuration.

Since `terraform init` does not create or modify configuration files (like `main.tf`), that is why "Create a sample main.tf file" is the correct answer.

Question 22 **Correct**

Which of the following is allowed as a Terraform variable name?

count

Explanation

The variable name "count" is allowed in Terraform as it is a valid identifier that can be used to define a variable in Terraform configurations.

source

Explanation

The variable name "source" is allowed in Terraform as it is a valid identifier that can be used to define a variable in Terraform configurations.

Your answer is correct

name

Explanation

The variable name "name" is allowed in Terraform as it is a valid identifier that can be used to define a variable in Terraform configurations.

version

Explanation

The variable name "version" is allowed in Terraform as it is a valid identifier that can be used to define a variable in Terraform configurations.

Overall explanation

Correct Answer: name

Explanation:

In Terraform, certain words are reserved and cannot be used as variable names because they have specific meanings in the configuration language. Reserved keywords include `count`, `source`, and `version` because they are used for specific Terraform functionalities:

- **`count`** is used to specify the number of instances to create of a resource or module.
- **`source`** specifies the source location for modules.
- **`version`** is used to define the version constraints, particularly for providers and modules.

However, **`name`** is not a reserved keyword in Terraform and is therefore a valid variable name.

Resources

Input variables

Question 23 **Correct**

If a module declares a variable with a default value, must the variable also be explicitly defined within the module?

True

Explanation

No, if a module declares a variable with a default value, it does not need to be defined within the module. The default value will be used if no other value is provided when the module is called.

Your answer is correct

False

Explanation

False. If a module declares a variable with a default value, it is not mandatory to define that variable within the module. The default value will be used if the variable is not explicitly set when calling the module. This flexibility allows for easier reuse of modules and customization based on specific requirements.

Overall explanation

Correct Answer: False

Explanation:

If a module declares a variable with a default value, it doesn't need to be explicitly defined when calling the module. The default value will be used if no value is provided by the calling module.

Example:

Imagine you have a module that provisions an EC2 instance, and it declares a variable `instance_type` with a default value:

Module (`ec2-instance/main.tf`):

```
variable "instance_type" {
 description = "The instance type to use for the EC2 instance"
 default     = "t2.micro"
}

resource "aws_instance" "example" {
 ami           = "ami-0c55b159cbfafe1f0" # Example AMI ID
 instance_type = var.instance_type
}
```

Calling the module:

```
module "ec2_instance" {
 source = "./ec2-instance"
}
```

In this case:

- If the `instance_type` variable is **not set** when calling the module, it will default to `t2.micro`.
- If you override it, for example:

```
module "ec2_instance" {
 source        = "./ec2-instance"
 instance_type = "t3.medium"
}
```

- Terraform will use the overridden value (`t3.medium`).

This demonstrates how the default value ensures flexibility while still allowing customization.

Question 24 Correct

When you initialize Terraform, where does it cache modules from the public Terraform Module Registry?

Your answer is correct

On disk in the .terraform sub-directory

Explanation

The correct choice. Terraform caches modules from the public Terraform Module Registry on disk in the .terraform sub-directory within the working directory. This allows for easy access and reuse of modules during Terraform operations without the need to re-download them each time.

In memory

Explanation

Caching modules in memory is not a common approach in Terraform initialization. Storing modules in memory would not provide persistent storage and could lead to performance issues when working with large modules or multiple projects.

They are not cached

Explanation
Modules from the public Terraform Module Registry are indeed cached on disk in the .terraform sub-directory. They are not re-downloaded every time Terraform is initialized, which helps in speeding up the process and reducing unnecessary network requests.

On disk in the /tmp directory
Explanation
Storing Terraform modules in the /tmp directory on disk is not the standard practice for caching modules from the public Terraform Module Registry. The modules are typically cached in a different location for better management and accessibility.

Overall explanation
Correct Answer: On disk in the .terraform sub-directory

Explanation: When Terraform initializes (`terraform init`), it caches modules downloaded from the public Terraform Module Registry in a directory called `.terraform` within the working directory. This cache ensures that Terraform can reuse downloaded modules for future operations without needing to re-download them unless there is a version update.

- Caching in `.terraform` is efficient and allows for faster re-runs of Terraform commands.
- The `/tmp` directory or memory is not used for caching modules, and Terraform always maintains a local cache, so "not cached" is also incorrect.

Resources
Initializing Working Directories

Question 25Incorrect
`terraform validate` reports syntax check errors from which of the following scenarios?

Correct answer
None of the above
Explanation
The correct choice is "None of the above" because Terraform validate specifically checks for syntax errors in the Terraform configuration files and does not cover scenarios such as tabs indentation, missing variable values, or state file mismatches.

The state files does not match the current infrastructure
Explanation
Terraform validate is used to check the syntax and structure of the Terraform configuration files, not to compare the state files with the current infrastructure. State file inconsistencies are typically identified during the plan or apply phases of Terraform operations.

Your answer is incorrect
There is missing value for a variable
Explanation
Terraform validate focuses on validating the syntax and structure of the Terraform configuration files. While missing values for variables can lead to runtime errors during the execution of Terraform commands, it is not typically caught by the validate command.

Code contains tabs indentation instead of spaces
Explanation
Terraform validate checks for syntax errors in the Terraform configuration files, such as incorrect syntax, missing brackets, or invalid configuration settings. It does not specifically flag tabs indentation instead of spaces as a syntax error.

Overall explanation
Correct Answer: None of the above

Explanation:

The `terraform validate` command checks the syntax and internal consistency of the Terraform configuration files without interacting with any remote services or state files. It verifies whether the configuration is syntactically correct, including proper HCL (HashiCorp Configuration Language) structure, valid resource and provider blocks, and compliant syntax. However, it does not check for issues like missing variable values, incorrect tab/space usage, or mismatches between the state file and actual infrastructure.

- **Tabs vs. spaces** (option A) are generally irrelevant to Terraform syntax as long as the structure is valid. `terraform fmt` is the command that handles formatting but not validation.
- **Missing variable values** (option B) would not be detected by `terraform validate` but might be flagged during `terraform plan` or `terraform apply`.
- **State mismatches** (option C) are identified through `terraform plan` or `terraform refresh`, not during the `validate` process.

`terraform validate` focuses solely on the correctness of the configuration syntax, not the operational state or runtime values.

Resources
Terraform validate

Question 26 Correct
What does **state locking** accomplish?

Copies the state file from memory to disk
Explanation
State locking does not involve copying the state file from memory to disk. It is a mechanism used to prevent concurrent modifications to the state file by multiple Terraform commands or users.

Your answer is correct
Blocks Terraform commands from modifying the state file
Explanation
State locking is designed to block Terraform commands from modifying the state file concurrently. This ensures that only one command can make changes to the state file at a time, preventing potential conflicts and data corruption.

Encrypts any credentials stored within the state file
Explanation
State locking does not specifically focus on encrypting credentials stored within the state file. While security is important, state locking primarily aims to prevent conflicts and inconsistencies in the state file.

Prevents accidental deletion of the state file
Explanation
State locking does not primarily focus on preventing accidental deletion of the state file. Its main purpose is to prevent concurrent modifications to the state file to maintain consistency and integrity during Terraform operations.

Overall explanation
Correct Answer: Blocks Terraform commands from modifying the state file

Explanation:

State locking is used to prevent multiple Terraform processes from modifying the state file simultaneously, which could result in conflicts or corruption. It ensures that only one Terraform operation can access and modify the state file at a time. This is especially useful in collaborative or automated environments where multiple users or systems might attempt to apply changes concurrently.

State locking does not handle encryption of credentials, copying the state file, or preventing deletion of the state file. Its primary purpose is to maintain the integrity of the state by restricting concurrent modifications.

Resources
State Locking

Question 27 Correct
You write a new Terraform configuration and immediately run terraform apply in the CLI using the local backend.

Why will the apply fail?

Terraform requires you to manually run terraform plan first
Explanation
While it is a best practice to run terraform plan before applying changes to your infrastructure, it is not a mandatory step for the apply to succeed with the local backend. Running terraform plan helps you preview the changes that Terraform will make, but it is not a prerequisite for the apply command to function with the local backend.

Terraform needs you to format your code according to best practices first
Explanation

Formatting your code according to best practices is not a prerequisite for running terraform apply with the local backend. While it is recommended to follow best practices for readability and maintainability, it is not a reason for the apply to fail in this scenario.

Your answer is correct

Terraform needs to install the necessary plugins first

Explanation

When running terraform apply, Terraform needs to download and install the necessary plugins to interact with the resources defined in your configuration. If the plugins are not installed or if there are any issues with plugin installation, the apply process will fail. This is a common reason for failure when running terraform apply with the local backend.

The Terraform CLI needs you to log into Terraform cloud first

Explanation

Logging into Terraform Cloud is not required when running terraform apply with the local backend. Terraform Cloud is a separate service provided by HashiCorp for remote state management and collaboration, and it is not directly related to the local backend apply process.

Overall explanation

Correct Answer: Terraform needs to install the necessary plugins first.

Explanation:

When you write a new Terraform configuration and immediately run `terraform apply`, the command will fail if the required provider plugins are not installed. Terraform needs these plugins to interact with the specified resources in the configuration. The first step, before running `terraform apply`, is to initialize the working directory using `terraform init`. This command downloads the necessary provider plugins and sets up the backend configuration (in this case, the local backend) to store state files.

- Running `terraform init` ensures that all dependencies are properly set up.
- Formatting or manually running `terraform plan` is not required but recommended as a best practice.
- Logging into Terraform Cloud is only necessary if using the Terraform Cloud backend, not the local backend.

Question 28Correct

You have declared an input variable called **environment** in your parent module.

What must you do to pass the value to a child module in the configuration?

Your answer is correct

Declare the variable in a terraform.tfvars file

Explanation

Declaring the variable in a terraform.tfvars file allows you to set the value of the input variable "environment" and pass it to the child module in the configuration. This is the correct way to pass values from the parent module to the child module.

Nothing, child modules inherit variables of parent module

Explanation

Child modules do not automatically inherit variables from the parent module. To pass the value of the input variable "environment" to the child module, you need to explicitly declare and set the variable in the child module configuration.

Add node_count = var.node_count

Explanation

Adding "node_count = var.node_count" in the configuration does not pass the value of the input variable "environment" to the child module. This line of code specifically assigns the value of the variable "node_count" to the child module, not the "environment" variable.

Declare a node_count input variable for child module

Explanation

Declaring a "node_count" input variable for the child module is not necessary to pass the value of the input variable "environment" from the parent module to the child module. The child module should reference the input variable "environment" directly.

Overall explanation

Correct Answer: Declare the variable in a terraform.tfvars file

Explanation:

In Terraform, input variables declared in the parent module need to be explicitly passed to the child module, and this is typically done via a `module` block in the parent configuration. If you're passing a variable from the parent module to the child, you should define the value either within the `module` block or by using a `terraform.tfvars` file, which holds the variable values for Terraform to use.

- The `terraform.tfvars` file is where you can define values for input variables across your entire Terraform configuration, and it is the recommended way to manage variable values.
- The option "Declare the variable in a terraform.tfvars file" is correct because it's a method of assigning values to input variables.
- "Declare the variable in the child module" is incorrect because the child module only needs to declare variables as input, but it doesn't automatically inherit values from the parent unless you pass them explicitly.
- "Add node_count = var.node_count" could be relevant if you were inside the parent module and wanted to pass the variable to a specific resource or output, but doesn't address the process of passing it to a child module.

Resources
terraform.tfvars

Question 29Correct
You've used Terraform to deploy a virtual machine and a database. You want to replace this virtual machine instance with an identical one without affecting the database. What is the best way to achieve this using Terraform?

Use the terraform state rm command to remove the VM from state file
Explanation
Using the terraform state rm command to remove the VM from the state file will remove the VM from Terraform's state management, but it will not actually replace the VM with an identical one. This approach may lead to inconsistencies in the infrastructure and does not ensure the replacement of the VM without affecting the database.

Use the terraform apply command targeting the VM resources only
Explanation
Using the terraform apply command targeting the VM resources only will not specifically indicate that the VM needs to be replaced with an identical one. This approach may not guarantee that the replacement of the VM will be done correctly without affecting the database.

Your answer is correct
Use the terraform taint command targeting the VMs then run terraform plan and terraform apply
Explanation
Using the terraform taint command targeting the VMs marks the resources as tainted, indicating that they need to be replaced. By running terraform plan and terraform apply after tainting the VMs, Terraform will recreate the VM instances while keeping the database intact. This approach ensures that the VM is replaced without affecting the database.

Delete the Terraform VM resources from your Terraform code then run terraform plan and terraform apply
Explanation
Deleting the Terraform VM resources from your Terraform code and then running terraform plan and terraform apply may lead to the deletion of the existing VM instance without ensuring that an identical replacement is created. This approach may result in the loss of the VM instance and its associated configurations without properly replacing it.

Overall explanation
Correct Answer: Use the terraform taint command targeting the VM then run terraform plan and terraform apply

Explanation:

Using the `terraform taint` command allows you to mark the virtual machine (VM) resource for recreation. When you run `terraform apply` afterward, Terraform will destroy and recreate only the targeted VM, leaving other resources (like the database) unaffected. This approach is ideal for cases where you want to replace a specific resource without impacting others in the configuration.

- `terraform state rm` is incorrect because it removes the VM from the state file entirely, making Terraform unaware of its existence. This would require importing the VM back into the state or reconfiguring it, which could lead to an inconsistent setup.
- `terraform apply` **targeting the VM resources only** could work, but it does not guarantee that the VM will be replaced. It simply attempts to apply any updates to the VM as per the configuration.
- **Deleting the VM resources from the code** and then running `terraform apply` would destroy the VM but also permanently remove it from the configuration, which is not the goal here since the VM should be recreated, not removed.

Resources

Forcing Re-creation of Resources

Question 30 **Correct**

A Terraform backend determines how Terraform loads state and stores updates when you execute _____.

destroy

Explanation

The `destroy` command in Terraform is used to destroy all the resources defined in the Terraform configuration. Similar to the `apply` command, the backend configuration plays a crucial role in determining how state is loaded and stored during the destroy process.

apply

Explanation

When you execute the `apply` command in Terraform, the backend determines how Terraform loads the state and stores any updates made during the apply process. The backend configuration specifies where the state file is stored and how it is accessed.

taint

Explanation

The `taint` command in Terraform is used to mark a resource for recreation during the next apply. While the backend does not directly determine how state is loaded or stored during the `taint` command execution, it is still an important part of Terraform operations.

None of the above

Explanation

The choice E is incorrect because all of the mentioned commands (`apply`, `taint`, and `destroy`) in Terraform trigger state loading and storage updates, which are influenced by the backend configuration. Therefore, the statement "None of the above" is not valid in this context.

Your answer is correct

All of the above

Explanation

The correct choice is D because all of the above commands (`apply`, `taint`, and `destroy`) in Terraform trigger state loading and storage updates, which are determined by the backend configuration. The backend configuration specifies where the state file is stored, how it is accessed, and how updates are managed.

Overall explanation

Correct Answer: All of the above

Explanation:

A Terraform backend is crucial for determining where and how Terraform loads the state file and how updates to the state are stored. This applies to all major Terraform commands that interact with infrastructure, including `apply`, `taint`, and `destroy`.

- **terraform apply**: Applies changes to the infrastructure and updates the state file with the new resource information.
- **terraform taint**: Marks a resource as tainted, indicating it should be recreated, and updates the state file.
- **terraform destroy**: Destroys infrastructure and updates the state file to reflect the deletion of resources.

In all these cases, Terraform relies on the backend to manage the state file, whether it's stored locally or remotely. The backend ensures that changes are appropriately tracked and the state is consistently updated.

Question 31 **Correct**

What type of block is used to construct a collection of nested configuration blocks?

Your answer is correct

dynamic

Explanation

The 'dynamic' block in Terraform is used to construct a collection of nested configuration blocks based on dynamic values or conditions. It allows for the creation of multiple nested blocks within a single resource or module based on the input provided, making it the correct choice for constructing nested configuration blocks.

repeated

Explanation

There is no specific 'repeated' block in Terraform. This term does not correspond to any valid Terraform configuration block or syntax for constructing nested configuration blocks.

for_each

Explanation

The 'for_each' block is used to iterate over a collection of objects or elements and create multiple instances of a resource or module based on the elements in the collection. It is not specifically used to construct nested configuration blocks within a single resource or module.

nesting

Explanation

'Nesting' is a general concept in programming and configuration management, but it is not a specific block type in Terraform. While nesting is common in Terraform for organizing and structuring configuration, it is not a dedicated block type for constructing nested configuration blocks.

Overall explanation

Correct Answer: dynamic

Explanation:

The `dynamic` block in Terraform is used to programmatically construct a collection of nested configuration blocks. This feature is particularly useful when the number of nested blocks is variable or when certain nested blocks need to be conditionally included based on values. By using `dynamic`, you can loop over a list or map, generating multiple configuration blocks as required.

- **`for_each`** is a keyword used to iterate over collections in Terraform but does not create nested blocks by itself.
- **`repeated`** and **`nesting`** are not valid Terraform keywords. They don't exist as block types in Terraform syntax.

Resources

Dynamic Blocks

Question 32Incorrect

Which statement describes a goal of infrastructure as code?

An abstraction from vendor specific APIs

Explanation

Infrastructure as code aims to provide an abstraction layer from vendor-specific APIs, allowing users to define infrastructure in a vendor-agnostic way. This abstraction simplifies the management and provisioning of resources across different cloud providers.

A pipeline process to test and deliver software

Explanation

A pipeline process to test and deliver software refers to continuous integration and continuous delivery (CI/CD) practices, which are essential for automating software development processes. While infrastructure as code can be integrated into CI/CD pipelines, it is not the primary goal of infrastructure as code itself.

Correct answer

The programmatic configuration of resources

Explanation

The goal of infrastructure as code is to programmatically define and manage infrastructure resources using code. This approach allows for the automation of infrastructure provisioning, configuration, and management, leading to increased efficiency, consistency, and scalability in managing infrastructure.

Your answer is incorrect

Write once, run anywhere

Explanation

The concept of "write once, run anywhere" is more closely related to cross-platform compatibility in software development. While infrastructure as code does promote consistency and repeatability in provisioning infrastructure, it is not specifically focused on running code across different platforms.

Overall explanation

Correct Answer: The programmatic configuration of resources

Explanation:

Infrastructure as Code (IaC) is the practice of defining and managing infrastructure through code, rather than manually configuring resources. This allows for the programmatic configuration, provisioning, and management of cloud or on-premises infrastructure. The goal is to automate the creation and management of infrastructure resources, enabling consistency, repeatability, and version control.

- "An abstraction from vendor specific APIs" is not the primary goal of IaC, though tools like Terraform may help abstract some of the underlying APIs.
- "Write once, run anywhere" is a general principle of software development but does not specifically relate to infrastructure configuration.
- "A pipeline process to test and deliver software" refers to Continuous Integration and Continuous Delivery (CI/CD) practices, not directly to IaC.

Question 33 Correct

Which backend does the Terraform CLI use by default?

Terraform Cloud

Explanation

Terraform Cloud is a remote backend option that allows for collaboration, state storage, and additional features like remote runs and workspaces. It is not the default backend used by the Terraform CLI.

Remote

Explanation

Remote backend is a generic term for any backend that is not stored locally. While it is a valid option for storing Terraform state, it is not the default backend used by the Terraform CLI.

Consul

Explanation

Consul is a backend option that can be used for storing Terraform state, but it is not the default backend used by the Terraform CLI.

Your answer is correct

Local

Explanation

Local backend is the default backend used by the Terraform CLI. It stores the state file on the local disk of the machine running Terraform, making it the simplest and most straightforward option for managing state.

Overall explanation

Correct Answer: Local

Explanation: By default, the Terraform CLI uses the **local** backend, which stores the state file on the local filesystem of the machine where Terraform is run. This backend is ideal for single-user scenarios and testing but lacks collaboration features, so it's generally used only in development or simple setups.

- **Terraform Cloud** and **Consul** are supported backends that provide enhanced features, such as remote state management and locking, but are not used by default.
- **Remote** is a general term that includes various backend options but does not refer to a specific backend by itself.

For collaborative workflows and production environments, a remote backend (like S3, Terraform Cloud, or Consul) is typically recommended over the local backend.

Resources

https://developer.hashicorp.com/terraform/language/backend

Question 34 Correct

You want to know from which paths Terraform is loading providers referenced in your Terraform configuration (*.tf files). You need to enable debug messages to find this out.

Which of the following would achieve this?

Your answer is correct

Set the environment variable TF_LOG=TRACE

Explanation

Setting the environment variable TF_LOG=TRACE will enable debug logging for Terraform, including detailed information about provider loading paths. This will help you identify from which paths Terraform is loading the providers referenced in your configuration files.

Set the environment variable TF_LOG_PATH

Explanation

Setting the environment variable TF_LOG_PATH will not achieve the goal of finding out from which paths Terraform is loading providers. TF_LOG_PATH is used to specify a file path where Terraform log output should be written, not to enable debug messages related to provider loading paths.

Set verbose logging for each provider in your Terraform configuration

Explanation

Setting verbose logging for each provider in your Terraform configuration will not achieve the goal of finding out from which paths Terraform is loading providers. This option focuses on individual provider logging, not the overall provider loading paths.

Set the environment variable TF_VAR_log=TRACE

Explanation

Setting the environment variable TF_VAR_log=TRACE is not the correct way to enable debug messages for Terraform. TF_VAR is used for setting variables, not for logging configurations. This option will not provide the information needed to identify provider loading paths.

Overall explanation

Correct Answer: Set the environment variable TF_LOG=TRACE

Explanation:

Setting the environment variable `TF_LOG=TRACE` enables the most detailed level of logging in Terraform. This includes debug-level information that will show exactly where Terraform is loading providers from in your configuration files. TRACE logging provides detailed insights into internal operations, including the discovery and loading of providers, resource management, and other low-level processes.

- The other options are incorrect because:
  - Terraform does not support verbose logging at the provider level within the configuration itself.
  - `TF_VAR_log` is not a valid environment variable for controlling logging in Terraform.
  - `TF_LOG_PATH` is not a recognized environment variable for controlling logging output in Terraform; instead, `TF_LOG` controls the verbosity level.

Resources

Debugging Terraform

Question 35Correct

You have just developed a new Terraform configuration for two virtual machines with a cloud provider. You would like to create the infrastructure for the first time.

Which Terraform command should you run first?

terraform plan

Explanation

Before applying any changes to the infrastructure, it is crucial to first generate an execution plan using `terraform plan`. This command allows you to preview the changes that Terraform will make to the infrastructure based on the configuration. However, it is not the initial command to run when setting up the infrastructure for the first time.

terraform show

Explanation
The `terraform show` command is used to display the current state or output of the Terraform configuration. While it can be helpful to inspect the current state of the infrastructure, it is not the command to run first when creating the infrastructure for the first time.
terraform apply
Explanation
Running `terraform apply` without initializing the configuration first may lead to errors or unexpected behavior. This command is used to apply the changes defined in the Terraform configuration to the infrastructure, but it should not be the first command to run when setting up the infrastructure for the first time.
Your answer is correct
terraform init
Explanation
The correct command to run first when setting up the infrastructure for the first time is `terraform init`. This command initializes the Terraform configuration, downloads any necessary plugins or modules, and prepares the working directory for Terraform to use. It is a crucial step before applying any changes or generating a plan.
Overall explanation
Correct Answer: terraform init

Explanation:

The `terraform init` command is the first step when starting to work with a Terraform configuration. It initializes the working directory by downloading necessary provider plugins and setting up the backend for state management. Without this step, Terraform won't be able to run any further commands, such as `terraform plan` or `terraform apply`.

- `terraform apply` is used to actually apply the changes to the infrastructure, but you need to initialize your environment first with `terraform init`.
- `terraform plan` generates an execution plan showing the changes Terraform will make, but it also requires the environment to be initialized first.
- `terraform show` is used to display the state or the output of a previous apply or plan, but it doesn't initialize your working environment.

Question 36Correct
What features stops multiple admins from changing the Terraform state at the same time?
Your answer is correct
State locking
Explanation
State locking in Terraform is a feature that prevents multiple administrators from making changes to the Terraform state simultaneously. When state locking is enabled, Terraform locks the state file to ensure that only one admin can make changes at a time, preventing conflicts and data corruption. This helps maintain the integrity and consistency of the infrastructure managed by Terraform.
Provider constraints
Explanation
Provider constraints in Terraform define limitations and requirements for interacting with specific cloud providers or services. While providers play a crucial role in Terraform configurations, they do not provide mechanisms to prevent multiple admins from modifying the state concurrently.
Version control
Explanation
Version control is not directly related to preventing multiple admins from changing the Terraform state at the same time. While version control systems can help manage changes to Terraform configurations, they do not inherently prevent concurrent state modifications.
Backend types
Explanation

Backend types in Terraform determine where state data is stored and how it is accessed, but they do not inherently prevent multiple admins from changing the state simultaneously. Different backend types offer various features and capabilities, but state locking is specifically designed to address concurrent state modifications.

Overall explanation

Correct Answer: State locking

Explanation:

State locking is a feature in Terraform that prevents multiple admins or processes from making concurrent changes to the state file, ensuring consistency and avoiding potential conflicts. When state locking is enabled (usually in remote backends like S3 with DynamoDB for locking, or Terraform Cloud), it locks the state file during operations such as `plan` or `apply`. This prevents simultaneous modifications that could lead to resource mismanagement or corruption of the state file.

- **Version control** (like Git) tracks code changes but does not lock the state file.
- **Backend types** refer to the various options for state storage but do not inherently prevent concurrent changes.
- **Provider constraints** set restrictions on provider versions and compatibility but have no impact on state locking.

Resources

Managing Terraform state

Question 37Correct

You need to deploy resources into two different cloud regions in the same Terraform configuration. To do that, you declare multiple provider configurations as follows:

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

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

What meta-argument do you need to configure in a resource block to deploy the resource to the `us-west-2` AWS region?

provider = west

Explanation

The provider meta-argument in the resource block is used to specify the provider configuration to use when deploying the resource. However, the correct format for specifying the provider configuration is not just the alias "west" but should be in the format "aws.west" to indicate the specific provider configuration for the "us-west-2" AWS region.

alias = west

Explanation

The alias in the resource block is used to specify which provider configuration to use when deploying the resource. In this case, the alias should be set to "west" to indicate that the resource should be deployed to the "us-west-2" AWS region.

alias = aws.west

Explanation

The alias in the resource block should not be set to "aws.west" as it does not correctly specify the provider configuration to use for deploying the resource to the "us-west-2" AWS region. The correct format for specifying the provider configuration is "aws.west" in the provider meta-argument.

Your answer is correct

provider = aws.west

Explanation

The correct way to configure the provider in the resource block to deploy the resource to the "us-west-2" AWS region is by specifying "provider = aws.west". This format ensures that the resource is deployed using the specific provider configuration with the alias "west" for the desired region.

Overall explanation

Correct Answer: provider = aws.west

Explanation:

To specify which provider configuration to use for a resource when you have multiple provider configurations in your Terraform setup, you must use the `provider` meta-argument.

In this case, the provider with the alias `west` is configured to use the `us-west-2` region, so the correct way to refer to it in the resource block is by specifying `provider = aws.west`. This tells Terraform to use the `aws` provider configuration with the alias `west`, which corresponds to the `us-west-2` region.

- The `alias = west` and `alias = aws.west` are incorrect because they refer to how the provider is defined, not how to use it in the resource block.
- The `provider = west` is incorrect because it does not fully specify the provider and its alias. The full reference should include `aws` before the alias, as shown in `aws.west`.

Resources
Provider Configuration

Question 38Correct
Which of the following does `terraform apply` change after you approve the execution plan? (Choose two.)

Your selection is correct
Cloud infrastructure
Explanation
Terraform apply changes the cloud infrastructure by creating, updating, or deleting resources based on the approved execution plan. This is the primary purpose of running terraform apply after reviewing the plan.

Terraform code
Explanation
Terraform code is not changed by running terraform apply. The Terraform code, typically stored in .tf files, defines the desired state of the infrastructure and is not modified by the apply command.

The execution plan
Explanation
The execution plan is not changed by running terraform apply. The execution plan is generated before applying changes to the infrastructure and serves as a preview of the actions that will be taken.

Your selection is correct
State file
Explanation
Terraform apply updates the state file after the changes have been successfully applied to the cloud infrastructure. The state file keeps track of the current state of the infrastructure and is updated to reflect the changes made during the apply process.

The .terraform directory
Explanation
The .terraform directory is not changed by running terraform apply. This directory contains Terraform's internal files and caches, and it is not directly affected by the apply command.

Overall explanation
Correct Answer: Cloud infrastructure, State file

Explanation:

When you run `terraform apply` and approve the execution plan, Terraform performs changes to the infrastructure (cloud resources like virtual machines, databases, etc.) to match the configuration in your Terraform files. This is where your cloud infrastructure is modified or created as per the desired configuration.

Additionally, `terraform apply` updates the state file, which tracks the real-world state of the infrastructure that Terraform manages. This file helps Terraform determine what resources are currently under management and how they need to be adjusted based on the configuration and execution plan.

- The **.terraform directory** is used for storing internal Terraform configurations, such as provider plugins and modules. It is not directly modified by `terraform apply`.
- The **execution plan** is not changed by `terraform apply`. It's simply generated before the apply step and used to apply the planned changes.
- **Terraform code** is not changed by `terraform apply`. The configuration files (e.g., `.tf` files) are not modified by Terraform during the execution unless explicitly updated by the user.

Question 39 **Correct**

You just scaled your VM infrastructure and realized you set the **count** variable to the wrong value. You correct the value and save your change.

What do you do next to make your infrastructure match your configuration?

Your answer is correct

Run an apply and confirm the planned changes

Explanation

Running an apply command is the correct next step after correcting the count variable in your configuration. This command will apply the changes to your infrastructure based on the updated configuration and ensure that your VM infrastructure matches the desired state.

Inspect all Terraform outputs to make sure they are correct

Explanation

Inspecting Terraform outputs is not the immediate next step after correcting the count variable in your configuration. Outputs are used to extract information from your Terraform configuration, but in this situation, you need to apply the corrected configuration to ensure your infrastructure matches the desired state.

Inspect your Terraform state because you want to change it

Explanation

Inspecting the Terraform state is not necessary in this scenario as you have already corrected the count variable in your configuration. The state file keeps track of the current state of your infrastructure compared to the configuration, but in this case, you have made the necessary correction and need to apply the changes.

Reinitialize because your configuration has changed

Explanation

Reinitializing Terraform is not required after correcting the count variable in your configuration. Reinitializing is typically done when there are major changes to the configuration or when setting up Terraform for the first time. In this case, applying the changes is sufficient.

Overall explanation

Correct Answer: Run an apply and confirm the planned changes

Explanation:

When you change the `count` variable for a resource, Terraform needs to adjust the infrastructure to match this updated configuration. After saving the change, running `terraform apply` will allow Terraform to calculate the necessary modifications and present a plan. You can then confirm these changes, which will automatically update your infrastructure to reflect the new `count` value.

- **Inspecting your Terraform state** is unnecessary in this scenario because Terraform automatically manages the state file to reflect your configuration. You generally shouldn't modify the state file directly unless there's a specific issue.
- **Reinitializing** (`terraform init`) is only necessary when there are changes to provider versions or backend configurations. Adjusting a variable value doesn't require reinitialization.
- **Inspecting all Terraform outputs** is unrelated to making your infrastructure match the configuration. Outputs display information, but they do not affect or update the infrastructure itself.

Question 40 **Correct**

HashiCorp Configuration Language (HCL) supports **user-defined** functions.

Your answer is correct

False

Explanation

True. HCL does not have built-in support for user-defined functions. While Terraform itself provides a set of built-in functions for manipulating and working with data in configurations, users cannot define their own custom functions within HCL.

True

Explanation

False. HashiCorp Configuration Language (HCL) does not support user-defined functions. HCL is primarily used for writing infrastructure as code and defining resources in Terraform configurations, but it does not have the capability to define custom functions.

Overall explanation

Correct Answer: False

Explanation:

HashiCorp Configuration Language (HCL) does not support user-defined functions in the same way many programming languages do. While you can use built-in functions in HCL (such as `length()`, `upper()`, etc.), it does not provide a way for users to define their own custom functions within Terraform configurations. Instead, users can structure their configurations using variables, resources, data sources, and built-in functions, but custom function creation is not a feature in HCL.

Resources

Built-in Functions

Question 41 **Incorrect**

You are using a networking module in your Terraform configuration with the name label **my_network**. In your main configuration you have the following code:

```
output: "net_id" {
  value = module.my_network.vnet_id
}
```

When you run `terraform validate`, you get the following error:

Error: Reference to undeclared output value

on main.tf line 12, in output "net_id":

12: value = module.my_network.vnet_id

What must you do to successfully retrieve this value from your networking module?

Correct answer

Define the attribute vnet_id as an output in the networking module

Explanation

Defining the attribute vnet_id as an output in the networking module is the correct approach to make it accessible in the main configuration. Outputs in Terraform modules allow you to expose specific values to be used in other parts of the configuration.

Change the referenced value to module.my_network.outputs.vnet_id

Explanation

Changing the referenced value to module.my_network.outputs.vnet_id is incorrect because outputs are accessed directly from the module, not from a nested outputs object within the module. The correct syntax to access an output value is module.my_network.vnet_id.

Change the referenced value to my_network.outputs.vnet_id

Explanation

Changing the referenced value to my_network.outputs.vnet_id is incorrect because the correct syntax to access an output value from a module is module.my_network.vnet_id. The outputs object is not needed in the reference to the output value.

Your answer is incorrect

Define the attribute vnet_id as a variable in the networking module

Explanation

Defining the attribute vnet_id as a variable in the networking module will not make it accessible as an output in the main configuration. Variables are used to input values into a module, not to output values from it.

Overall explanation
Correct Answer: Define the attribute vnet_id as an output in the networking module

Explanation:

In Terraform, to successfully access a value from a module, it must be explicitly declared as an output within that module. The error you encountered occurs because the value `vnet_id` is not declared as an output in the networking module.

- The correct approach is to define `vnet_id` as an **output** in the networking module itself, so that it can be referenced from the main configuration. This would look something like this in the networking module:

```
output "vnet_id" {
 value = <resource or variable holding vnet_id>
}
```

- **Change the referenced value to module.my_network.outputs.vnet_id** is incorrect because the correct way to reference an output from a module is simply `module.my_network.vnet_id`, not `module.my_network.outputs.vnet_id`.
- **Define the attribute vnet_id as a variable in the networking module** is incorrect because the issue is with accessing an output, not defining a variable. The `vnet_id` should be declared as an output to make it accessible from other configurations.
- **Change the referenced value to my_network.outputs.vnet_id** is incorrect because `my_network` is the module's local name, and it should always be prefixed by `module.` to access its outputs or resources.

The key here is that to retrieve a value from a module, it must be explicitly defined as an output within the module.

Resources
Terraform Outputs

Question 42 Incorrect
Which type of **block** fetches or computes information for use elsewhere in a Terraform configuration?

provider
Explanation
The provider block is used to configure the details of a specific provider, such as AWS or Azure, that Terraform will use to interact with external APIs. It is not responsible for fetching or computing information for use elsewhere in the configuration.

Your answer is incorrect
resource
Explanation
The resource block is used to define a specific infrastructure object, such as an EC2 instance or S3 bucket, that Terraform will manage. It is not primarily used for fetching or computing information for use elsewhere in the configuration.

local
Explanation
The local block is used to define local values or computations within a Terraform configuration. While it can be used to store and compute information locally, it is not specifically designed to fetch or compute information for use elsewhere in the configuration.

Correct answer
data
Explanation
The data block is used to fetch or compute information from external sources, such as APIs or databases, for use elsewhere in a Terraform configuration. It allows Terraform to retrieve data that is needed for resource creation or configuration.

Overall explanation
Correct Answer: data

Explanation:

In Terraform, a `data` block is used to fetch or compute information from external sources that can then be used elsewhere in your Terraform configuration. This allows you to reference existing resources, external data, or system attributes without needing to manage them directly within the configuration.

- The `provider` block is used to configure the provider, which allows Terraform to interact with the relevant APIs or services, but it does not fetch or compute data for use in the configuration itself.
- The `resource` block is used to define and manage infrastructure resources that Terraform will create, modify, or destroy, but it doesn't fetch or compute information from other sources.
- The `local` block is used to define local values within the configuration to simplify expressions, but it doesn't fetch external data. It stores values that are computed within the Terraform configuration.

Resources
Data Sources

Question 43 Correct
Your security team scanned some Terraform workspaces and found secrets stored in a **plaintext** in state files.
How can you protect sensitive data stored in Terraform state files?

Edit your state file to scrub out the sensitive data
Explanation
Editing the state file to scrub out sensitive data is not a reliable method for protecting sensitive information. Manually editing the state file can be error-prone and may not completely remove all instances of sensitive data, leaving potential security vulnerabilities in the infrastructure.

Your answer is correct
Store the state in an encrypted backend
Explanation
Storing the state in an encrypted backend is the correct approach to protect sensitive data stored in Terraform state files. By using an encrypted backend, you can ensure that the data is securely stored and only accessible to authorized users with the necessary encryption keys.

Always store your secrets in a secrets.tfvars file.
Explanation
Storing secrets in a separate secrets.tfvars file is not a recommended practice for protecting sensitive data stored in Terraform state files. While using separate files for secrets can help with organization, it does not address the security concerns related to storing sensitive information in plaintext in state files.

Delete the state file every time you run Terraform
Explanation
Deleting the state file every time you run Terraform is not a recommended solution for protecting sensitive data. This approach can lead to data loss and inconsistency in the state management process, making it difficult to track changes and manage infrastructure effectively.

Overall explanation
Correct Answer: Store the state in an encrypted backend

Explanation:

Storing the state in an encrypted backend is the best practice for protecting sensitive data, as it ensures that the state file is encrypted both at rest and, often, in transit. This approach secures sensitive data like passwords or access keys that may appear in the state file.

Other options are less secure or practical because:

- **Deleting the state file each time** would disrupt Terraform's ability to track resources, making it difficult to manage infrastructure properly.
- **Editing the state file to scrub out sensitive data** is error-prone, manual, and impractical for large setups. Terraform will also often rewrite state files, so any manual changes would be overwritten.
- **Storing secrets in a `secrets.tfvars` file** does not prevent them from appearing in the state file once Terraform processes them; it only organizes them separately from other configuration files.

Using an encrypted backend (e.g., AWS S3 with encryption enabled, Terraform Cloud) ensures that sensitive data is handled securely without compromising Terraform's functionality.

Resources

Sensitive Data in State

Question 44 **Incorrect**

How do you specify a module's version when publishing it to the public Terraform Module Registry?

Terraform Module Registry does not support versioning modules

Explanation

Terraform Module Registry does support versioning modules. Versioning is crucial for managing changes and ensuring compatibility between different versions of the module. Without versioning, it would be challenging to track and control the evolution of the module.

The module's configuration page on the Terraform Module Registry

Explanation

The module's configuration page on the Terraform Module Registry does not directly allow you to specify a module's version when publishing it. It is primarily used for displaying information about the module, such as its documentation, inputs, and outputs.

Correct answer

The release tags in the associated repo

Explanation

The correct way to specify a module's version when publishing it to the public Terraform Module Registry is by using release tags in the associated repository. By tagging specific commits in the repository with version numbers, you can indicate which version of the module is being published to the registry.

Your answer is incorrect

The module's Terraform code

Explanation

The module's Terraform code itself does not contain explicit information about its version when publishing it to the public Terraform Module Registry. While the code may undergo changes and updates, the versioning information is typically managed through release tags in the associated repository.

Overall explanation

Correct Answer: The release tags in the associated repo

Explanation:

When publishing a module to the public Terraform Module Registry, the version is determined by the release tags in the associated repository (such as GitHub). Each tag should follow semantic versioning (e.g., `v1.0.0`, `v1.1.0`), which the registry then uses to track different versions of the module. This allows users to specify a version when they use the module in their configurations, ensuring consistency across deployments.

- **The module's configuration page on the Terraform Module Registry** is incorrect because you don't specify the version directly on the registry's webpage; it is controlled by the repository tags.
- **Terraform Module Registry does not support versioning modules** is incorrect, as the registry indeed supports versioning through repository tags.
- **The module's Terraform code** is incorrect because versioning is not specified in the code itself; it relies on the repository tags to define versions.

Resources

Publishing Modules

Question 45 **Correct**

A junior admin accidentally deleted some of your cloud instances. What does Terraform do when you run terraform apply?

Tear down the entire workspace infrastructure and rebuild it

Explanation

Terraform tearing down the entire workspace infrastructure and rebuilding it would mean recreating all resources, not just the instances that were deleted. This would be an inefficient and time-consuming process, especially if only a few instances were accidentally deleted.

Build a completely brand new set of infrastructure

Explanation

Building a completely brand new set of infrastructure would mean creating all resources from scratch, not just the instances that were accidentally deleted. This would result in unnecessary duplication and potential conflicts with existing resources.

Stop and generate an error message about the missing instances

Explanation

Terraform does not stop and generate an error message about missing instances when running terraform apply. Instead, it evaluates the current state of the infrastructure compared to the desired state defined in the Terraform configuration files and makes the necessary changes to reconcile any differences.

Your answer is correct

Rebuild only the instances that were deleted

Explanation

Terraform's behavior of rebuilding only the instances that were deleted is the correct choice in this scenario. By identifying the missing instances and recreating them, Terraform ensures that only the necessary resources are recreated, minimizing downtime and unnecessary changes to the infrastructure.

Overall explanation

Correct Answer: Rebuild only the instances that were deleted

Explanation:

When you run `terraform apply` after a resource (like an instance) has been manually deleted, Terraform detects that the resource no longer exists in the cloud environment but still exists in the state file. Terraform will consider the missing instances as "drifted" and will attempt to recreate only the deleted instances to match the desired configuration defined in your Terraform code.

- **Building a completely new set of infrastructure** is incorrect because Terraform will not rebuild everything unless the configuration itself has changed or the entire infrastructure is missing from the state.
- **Tearing down the entire workspace infrastructure** is incorrect because Terraform only attempts to recreate the missing resources and leaves the rest of the infrastructure intact.
- **Stopping and generating an error message** is incorrect because Terraform does not generate an error for missing resources; it will attempt to fix the drift by recreating the resources that were deleted.

In this case, Terraform will only rebuild the specific instances that were deleted, keeping the other resources unchanged.

Question 46Incorrect

A module can always refer to all variables declared in its parent module.

Correct answer

False

Explanation

This statement is correct. A module in Terraform can only refer to variables declared in its parent module if those variables are explicitly passed as input variables to the module. Without passing the variables as inputs, the module will not have access to them.

Your answer is incorrect

True

Explanation

This statement is incorrect. A module can only refer to variables declared in its parent module if those variables are explicitly passed as input variables to the module. Without passing the variables as inputs, the module will not have access to them.

Overall explanation

Correct Answer: False

Explanation:

A module in Terraform cannot automatically reference all variables declared in its parent module. The parent module must explicitly pass the values of any variables to the child module. This is done by defining input variables in the child module and passing the values when invoking the module.

If a variable is not explicitly passed, the child module will not be able to access it. This allows for modular and reusable code, where each module has its own scope and dependencies are managed via explicit inputs and outputs.

Question 47Correct

In contrast to Terraform Open Source, when working with Terraform Enterprise and Cloud Workspaces, conceptually you could think about them as completely separate working directories.

Your answer is correct

True

Explanation

True. When working with Terraform Enterprise and Cloud Workspaces, they can be thought of as completely separate working directories. This separation allows for better organization, isolation, and management of infrastructure configurations and state files. Each workspace in Terraform Enterprise or Cloud Workspaces operates independently, with its own state file, variables, and configuration, making them conceptually distinct from each other.

False

Explanation

False. In contrast to Terraform Open Source, Terraform Enterprise and Cloud Workspaces are designed to provide a centralized platform for managing infrastructure configurations and state files. While they may have separate workspaces for different projects or environments, they are all part of the same platform and share common features and capabilities. The concept of separate working directories is not applicable in this context as Terraform Enterprise and Cloud Workspaces offer a more integrated and collaborative approach to infrastructure management.

Overall explanation

Correct Answer: True

Explanation:

In Terraform Enterprise and Terraform Cloud, each workspace functions similarly to an isolated working directory. Each workspace has its own state, configuration, and environment, allowing you to manage different environments or projects separately within the same Terraform organization. Unlike Terraform Open Source, where you often use workspaces within the same directory structure and local state file, Terraform Enterprise and Cloud offer completely separate environments, even allowing for unique variables, secrets, and permissions per workspace.

This separation enhances security and organization for managing multiple environments (such as dev, test, and production) without any overlap or risk of cross-contamination.

Resources

Workspaces

Question 48Incorrect

All modules published on the official Terraform Module Registry have been verified by HashiCorp.

Correct answer

False

Explanation

The correct choice is false because not all modules on the official Terraform Module Registry have been verified by HashiCorp. Users should exercise caution and review the modules themselves to ensure they meet their specific requirements and standards before incorporating them into their Terraform configurations.

Your answer is incorrect

True

Explanation

The statement that all modules published on the official Terraform Module Registry have been verified by HashiCorp is not accurate. While the registry serves as a platform for sharing Terraform modules, it does not guarantee that every module has undergone verification or approval by HashiCorp before being made available to users.

Overall explanation

Correct Answer: False

Explanation:

Not all modules published on the official Terraform Module Registry are verified by HashiCorp. While HashiCorp verifies some modules, particularly those they create or officially endorse, the majority of modules on the registry are submitted by the community. These community-contributed modules may not have undergone the same level of scrutiny or verification as those provided directly by HashiCorp.

It's always important to review the module's code and documentation carefully before use, especially when using modules from the community.

Resources
Publishing Modules

Question 49 **Correct**
You're building a CI/CD (continuous integration/ continuous delivery) pipeline and need to inject sensitive variables into your Terraform run.
How can you do this safely?

Your answer is correct
Pass variables to Terraform with a `-var` flag

Explanation
Passing variables to Terraform with a -var flag is the correct approach to inject sensitive variables safely. This method allows you to provide sensitive information at runtime without exposing them in the Terraform code or storing them in plain text in a repository, ensuring security and confidentiality.

Store the sensitive variables as plain text in a source code repository

Explanation
Storing sensitive variables as plain text in a source code repository is a highly insecure method of injecting sensitive variables into your Terraform run. This approach exposes the sensitive information to anyone with access to the repository, making it vulnerable to security breaches and unauthorized access. It is crucial to avoid storing sensitive data in plain text within source code repositories to maintain data security and integrity.

Store the sensitive variables in a secure_vars.tf file

Explanation
Storing sensitive variables in a secure_vars.tf file may seem like a secure approach, but it is not the best practice for injecting sensitive variables into your Terraform run. This method still involves storing sensitive information in the Terraform codebase, which can pose security risks and compromise the confidentiality of the data.

Copy the sensitive variables into your Terraform code

Explanation
Copying sensitive variables directly into your Terraform code is not a recommended practice for injecting sensitive variables safely. Storing sensitive information in the code itself can lead to security vulnerabilities and expose the information to unauthorized access.

Overall explanation
Correct Answer: Pass variables to Terraform with a `-var` flag

Explanation:

Passing sensitive variables using the `-var` flag allows you to inject them safely at runtime without hardcoding them into the configuration files or version control. This approach keeps sensitive information out of your codebase, ensuring it doesn't get exposed in source control or configuration files.

The other options are unsafe or not ideal because:

- **Copying sensitive variables into Terraform code** makes them visible to anyone with access to the codebase.
- **Storing sensitive variables in a `secure_vars.tf` file** is risky unless the file is strictly excluded from version control, which still poses risks of accidental exposure.
- **Storing sensitive variables as plain text in a source code repository** directly exposes them to anyone with access to the repository, which is highly insecure.

In production CI/CD setups, it is best to use secure systems like environment variables or secret management tools (e.g., AWS Secrets Manager or HashiCorp Vault) to handle sensitive data safely.

Resources

Input Variables

Question 50 **Incorrect**

To check if all code in a Terraform configuration with multiple modules is properly formatted without making changes, what command should be run?

terraform fmt -write-false

Explanation

The command "terraform fmt -write-false" is not a valid Terraform command. The "-write-false" flag is not recognized as a valid option for the "terraform fmt" command.

terraform fmt -list -recursive

Explanation

The command "terraform fmt -list -recursive" is not a valid Terraform command. The correct flag to use for checking multiple modules recursively is "-recursive" instead of "-list".

Correct answer

terraform fmt -check -recursive

Explanation

The command "terraform fmt -check -recursive" is the correct command to check if all code in a Terraform configuration with multiple modules is properly formatted without making changes. The "-check" flag ensures that the command only checks the formatting without modifying the files, and the "-recursive" flag checks all modules within the configuration recursively.

Your answer is incorrect

terraform fmt -check

Explanation

The command "terraform fmt -check" is used to check if all code in a Terraform configuration is properly formatted without making any changes. However, this command does not include the option to check multiple modules within the configuration.

Overall explanation

Correct Answer: `terraform fmt -check -recursive`

Explanation:

The `terraform fmt -check -recursive` command is used to verify that all Terraform files in a configuration (including files in nested module directories) are formatted according to Terraform's standard style, without making any changes. The `-check` flag performs the check without altering files, and the `-recursive` flag ensures the command applies to all directories, which is particularly useful when working with configurations that include multiple modules.

- `terraform fmt -check` only checks the formatting in the current directory, not recursively in all module directories.
- `terraform fmt -write=false` is not a valid option.
- `terraform fmt --list -recursive` is also not a valid option.

Question 51 **Incorrect**

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

Most Terraform providers interact with __ to communicate with cloud providers.

Your answer is incorrect

public registry

Correct answer

API

Explanation

Correct Answer: API

Explanation: Most Terraform providers interact with an API (Application Programming Interface) to communicate with the cloud service or platform they are managing. This allows Terraform to create, update, and manage resources programmatically. Providers use APIs to make requests to external systems like AWS, Azure, Google Cloud, etc.

Resources

Terraform language providers

Question 52 **Correct**

Which of the following arguments are required when declaring a Terraform output?

sensitive

Explanation

The "sensitive" argument in Terraform outputs is optional and is used to mark the output value as sensitive, meaning it will be redacted from any Terraform command output and state files. It is not a required argument for declaring a Terraform output.

default

Explanation

The "default" argument in Terraform outputs is optional and is used to provide a default value for the output in case the value is not explicitly set. It is not a required argument for declaring a Terraform output, as outputs can be declared without specifying a default value.

Your answer is correct

value

Explanation

The "value" argument in Terraform outputs is required when declaring a Terraform output. It is used to specify the actual value that will be exposed as an output from the Terraform configuration. Without specifying a value, the output declaration would be incomplete and result in an error.

description

Explanation

The "description" argument in Terraform outputs is optional and is used to provide a description or explanation of the output value. While it is a good practice to include descriptions for clarity and documentation purposes, it is not a required argument for declaring a Terraform output.

Overall explanation

Correct Answer: value

Explanation:

When declaring a Terraform output, the **value** argument is required. The value represents the data that you want to expose from your Terraform configuration after it has been applied. This could be the result of a resource, an expression, or the output of a module. Without specifying a value, the output would have no content to display.

- **sensitive**: This argument is optional. It is used to mark the output as sensitive, which ensures that the value is not displayed in the output logs or state files unless explicitly requested.
- **description**: This is also an optional argument. It allows you to provide a human-readable description of what the output represents. It helps others understand the purpose of the output when reviewing your Terraform configuration.
- **default**: This argument is not valid for Terraform outputs. The concept of a "default" value is typically associated with input variables, not outputs. Outputs are meant to represent dynamic data generated by the resources, not default values.

Thus, the **value** argument is the only required one when declaring a Terraform output.

Resources

Output Values

Question 53 **Correct**

Your risk management organization requires that new AWS S3 buckets must be private and encrypted at rest.

How can Terraform Enterprise automatically and proactively enforce this security control?

With an S3 module with proper settings for buckets

Explanation

While creating an S3 module with proper settings for buckets can help in standardizing the configuration of S3 buckets, it does not provide an automated and proactive way to enforce security controls. Sentinel policies are the recommended approach for automatically enforcing security controls in Terraform Enterprise.

Your answer is correct

With a Sentinel policy, which runs before every apply

Explanation

With a Sentinel policy, Terraform Enterprise can automatically enforce security controls by running the policy before every apply operation. This ensures that new AWS S3 buckets are always created as private and encrypted at rest, in line with the risk management organization's requirements.

By adding variables to each TFE workspace to ensure these settings are always enabled

Explanation

Adding variables to each TFE workspace may help in setting default values for certain configurations, but it does not provide an automated and proactive way to enforce security controls like ensuring new AWS S3 buckets are private and encrypted at rest. Sentinel policies are specifically designed for this purpose.

Auditing cloud storage buckets with a vulnerability scanning tool

Explanation

Auditing cloud storage buckets with a vulnerability scanning tool is a post-deployment activity and does not provide a proactive way to enforce security controls during the infrastructure provisioning process. To automatically enforce security controls like ensuring new AWS S3 buckets are private and encrypted at rest, a solution like Sentinel policies in Terraform Enterprise is more appropriate.

Overall explanation

Correct Answer: With a Sentinel policy, which runs before every apply

Explanation: Sentinel is HashiCorp's policy-as-code framework that allows you to enforce policies across your Terraform configurations. It enables you to define rules and checks, such as ensuring that all AWS S3 buckets must be private and encrypted, before any changes are applied to infrastructure. This proactive enforcement prevents non-compliant resources from being deployed.

- The other choices are not fully automated solutions for enforcing security controls.
    - Storing variables in workspaces could ensure correct settings but doesn't proactively enforce compliance.
    - Using an S3 module with proper settings can define the right defaults, but it's not an automated enforcement mechanism for all cases.
    - Auditing cloud storage buckets with a vulnerability scanning tool is a reactive measure, not a proactive enforcement step.

Resources

AWS S3 Security Best Practices

Question 54Incorrect

A fellow developer on your team is asking for some help in refactoring their Terraform code. As part of their application's architecture, they are going to tear down an existing deployment managed by Terraform and deploy new. However, there is a server resource named aws_instance.ubuntu[1] they would like to keep to perform some additional analysis.

What command should be used to tell Terraform to no longer manage the resource?

terraform plan rm aws_instance.ubuntu[1]

Explanation

The command "terraform plan rm aws_instance.ubuntu[1]" is not the correct command to remove a resource from Terraform management. The "plan" command is used to show the execution plan for changes, not to remove resources from Terraform.

terraform delete aws_instance.ubuntu[1]

Explanation

The command "terraform delete aws_instance.ubuntu[1]" is not the correct command to tell Terraform to no longer manage the resource. The "delete" command is not a valid Terraform command for removing a resource from Terraform management.

Correct answer

terraform state rm aws_instance.ubuntu[1]

Explanation

The correct command to tell Terraform to no longer manage the resource is "terraform state rm aws_instance.ubuntu[1]". This command removes the resource from the Terraform state file, effectively telling Terraform to forget about managing that specific resource.

Your answer is incorrect

terraform apply rm aws_instance.ubuntu[1]

Explanation

The command "terraform apply rm aws_instance.ubuntu[1]" is not the correct command to tell Terraform to no longer manage the resource. The "apply" command is used to apply changes to the infrastructure, not to remove a specific resource from Terraform management.

Overall explanation

Correct Answer: terraform state rm aws_instance.ubuntu[1]

Explanation:

The command `terraform state rm` is used to remove a specific resource from Terraform's state without affecting the actual infrastructure. By running `terraform state rm aws_instance.ubuntu[1]`, Terraform will stop managing the resource `aws_instance.ubuntu[1]` in its state file, but it will not delete the resource in the cloud provider. This is useful when you want to perform an operation on resources independently of Terraform's management, as in this case where the server needs to be kept for analysis.

- **terraform apply rm** and **terraform plan rm** are incorrect because they are not valid Terraform commands for modifying the state file.
- **terraform delete** is also incorrect because Terraform does not use this command, and it would not affect the state management for a specific resource.

Resources

Command: state rm

Question 55Incorrect

Your DevOps team is currently using the **local backend** for your Terraform configuration. You would like to move to a remote backend to begin storing the state file in a central location.

Which of the following backends would not work?

Amazon S3

Explanation

Amazon S3 is a valid option for a remote backend in Terraform. It allows you to store your state file securely in an object storage service provided by AWS, enabling collaboration and consistency among team members.

Your answer is incorrect

Artifactory

Explanation

Artifactory is a valid option for a remote backend in Terraform. It provides a secure and centralized location to store your state file, ensuring that it is easily accessible and managed by your DevOps team.

Correct answer

Git

Explanation

Git is not a suitable option for a remote backend in Terraform. While Git is commonly used for version control, it is not designed to handle the state file management and locking mechanisms required for Terraform operations. Using Git as a backend can lead to issues with state consistency and concurrency.

Terraform Cloud

Explanation

Terraform Cloud is a valid option for a remote backend in Terraform. It is a managed service provided by HashiCorp that offers collaboration, state management, and other features to streamline Terraform workflows. It is specifically designed to work seamlessly with Terraform configurations.

Overall explanation

Correct Answer: Git

Explanation: While backends like Amazon S3, Artifactory, and Terraform Cloud are valid options for remote backends, Git is not a supported backend for Terraform state storage. Remote backends are specifically designed to store, lock, and manage state files, enabling collaboration and state consistency. Git repositories, however, are not suited for this purpose because they lack native support for locking mechanisms and secure state management, which are essential to prevent conflicts and protect sensitive information in state files.

- **Amazon S3** is a supported backend and can store the state file in an S3 bucket with optional encryption and versioning.
- **Artifactory** is also a supported backend that allows centralized state storage.
- **Terraform Cloud** is a recommended backend, offering native support for secure and managed state storage, collaboration, and locking.

Git does not provide these state management features, making it unsuitable as a backend for Terraform.

Resources
Local Backend

Question 56Correct
Please fill the blank field(s) in the statement with the right words.

A terraform __ command is used to import infrastructure into Terraform's state file.

Your answer is correct
import
Explanation
Correct Answer: import

Explanation:

The `terraform import` command is used to bring existing infrastructure into Terraform's state file. This allows Terraform to manage resources that were created manually or by other tools, linking them to their corresponding resource definitions in the configuration. It does not create new resources but maps existing ones to Terraform's state.

Example:

To import an AWS S3 bucket named `my-existing-bucket`:

1. Define it in `main.tf`:
```
resource "aws_s3_bucket" "example" {
 bucket = "my-existing-bucket"
}
```
2. Run:
```
terraform import aws_s3_bucket.example my-existing-bucket
```

Resources
Terraform Import command

Question 57Correct
What is the name assigned by Terraform to reference this resource?

```
mainresource “google_compute_instance” “main” {
      name = “test”
}
```

compute_instance

Explanation

The name "compute_instance" is not the standard naming convention used by Terraform to reference resources. It does not follow the typical format and may cause confusion when managing resources within Terraform configurations.

Your answer is correct

main

Explanation

The name "main" is the default name assigned by Terraform to reference the main configuration file in a Terraform project. It is a standard convention and helps to identify the primary configuration file within the project.

google

Explanation

The name "google" does not align with the typical naming conventions used by Terraform to reference resources. It does not provide any meaningful information about the resource being referenced and may lead to ambiguity in the Terraform configuration.

teat

Explanation

The name "teat" is not a standard or recognized naming convention used by Terraform to reference resources. It does not follow the typical format and may cause confusion when trying to identify and manage resources within Terraform configurations.

Overall explanation

Correct Answer: main

Explanation:

Terraform references each resource by the **local name** defined within the resource block. In this example, the resource block is defined as `google_compute_instance "main"`, where `"main"` is the specific name assigned by Terraform to reference this instance. To reference this resource in the configuration, you would use `google_compute_instance.main`.

The other options are incorrect because:

- **compute_instance** and **google** are not part of the naming convention used to reference this resource directly.
- **test** is the name assigned within the resource for the instance itself but is not the local reference name used by Terraform.