DUMPSARENA

HashiCorp Certified: Terraform Associate (004) (HCTA0-004)

HashiCorp Terraform-Associate-004

Version Demo

Total Demo Questions: 39

Total Premium Questions: 395

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Infrastructure as Code (IaC) Concepts	25
Topic 2, Terraform Fundamentals	25
Topic 3, Terraform Workflow	93
Topic 4, Terraform Configuration	77
Topic 5, Terraform Modules	49
Topic 6, Terraform State Management	93
Topic 7, Terraform Cloud and Enterprise	33
Total	395

QUESTION NO: 1

Which of these are features of Terraform Cloud? Choose two correct answers.

A. A web-based user interface (UI)

B. Automated infrastructure deployment visualization

C. Automatic backups

D. Remote state storage

ANSWER: A D

Explanation:

Terraform Cloud provides a web-based user interface (UI) for centrally managing Terraform workspaces and their associated operations. From the UI, teams can create and configure workspaces, connect configurations to version control repositories, review run details, inspect plan and apply output, manage variables, and administer organization-level settings. This gives collaborators a shared operational view of infrastructure automation without requiring every activity to be performed solely through a local command-line workflow.

Remote state storage is also a core Terraform Cloud capability. Each Terraform Cloud workspace maintains its Terraform state remotely, allowing authorized team members and automated runs to use a consistent source of truth for managed infrastructure. Terraform Cloud coordinates state access and locks state while operations are in progress, helping prevent concurrent changes from producing conflicting state updates. Remote state can also be shared between workspaces through explicitly configured outputs, supporting modular infrastructure designs while retaining access controls. HashiCorp documents the Terraform Cloud user interface and workspace workflow at Terraform Cloud workspaces and explains remote state behavior at Remote state.

QUESTION NO: 2

What is the purpose of the terraform.lock.hcl file in Terraform?

A. There is no such file.

B. Storing references to workspaces, which are locked.

C. Preventing Terraform runs from occurring.

D. Tracking specific provider dependencies.

ANSWER: D

Explanation:

The `terraform.lock.hcl` dependency lock file records the provider selections Terraform has made for a configuration. In particular, it records the exact provider versions selected under the configuration's version constraints, along with checksums for the provider packages. Terraform automatically creates or updates this file during `terraform init` when provider dependencies are installed or changed.

Committing this file to version control enables team members and automated pipelines to use the same provider versions and to verify that downloaded provider packages match the expected checksums. This makes Terraform behavior more reproducible across different machines and over time, even if a newer provider version later becomes available that would otherwise satisfy the configuration's version constraint. The lock file applies to provider dependencies; it is not Terraform state and does not hold workspace references.

HashiCorp recommends including the dependency lock file in version control so that provider dependency selections are shared with collaborators. For additional platform checksum coverage, teams can also use `terraform providers lock` to pre-populate checksums for provider packages on target platforms.

See the Terraform dependency lock file documentation and the terraform providers lock command reference.

QUESTION NO: 3

How does the use of Infrastructure as Code (IaC) enhance the reliability of your infrastructure?

Pick the two correct responses below.

A. Proposed changes can be reviewed before being applied.

B. Infrastructure is automatically scaled to meet demand.

C. Incorrect configurations cannot be deployed.

D. Updates are deployed with zero downtime.

E. Configuration drift is reduced with declarative definitions.

ANSWER: A E

Explanation:

“Proposed changes can be reviewed before being applied.” is correct because Terraform creates an execution plan before it makes infrastructure changes. Running `terraform plan` compares the configuration with the current state and shows the actions Terraform proposes to take. Teams can use that output in local workflows or automated CI/CD processes to inspect, validate, and approve changes before running `terraform apply`. This reviewable, repeatable workflow helps prevent unintended changes and supports safer operational practices.

“Configuration drift is reduced with declarative definitions.” is also correct because Terraform configurations express the intended, desired infrastructure state in version-controlled code. Terraform can refresh its understanding of remote objects during planning and identify differences between the declared configuration and actual infrastructure. This makes drift visible and enables teams to bring managed resources back into alignment through a controlled change process. Maintaining infrastructure definitions as code also promotes consistency when environments are created, modified, or rebuilt, rather than relying on undocumented manual actions. Terraform does not continuously reconcile infrastructure on its own; instead, drift detection and correction occur when teams run Terraform operations or automate those operations in their delivery workflow.

See HashiCorp’s documentation on terraform plan and on refreshing Terraform state.

QUESTION NO: 4

Which of these actions are forbidden when the Terraform state file is locked? (Pick the 3 correct responses)

A. terraform apply

B. terraform state list

C. terraform destroy

D. terraform fmt

ANSWER: A C

Explanation:

`terraform apply` and `terraform destroy` are the state-changing operations in this list. Terraform automatically acquires a state lock before operations that can write updated state, and it will stop an operation if it cannot acquire that lock. This protects the state from concurrent changes, such as two users applying infrastructure updates at the same time, which could otherwise cause lost updates or inconsistent resource tracking. An apply can create, update, or delete managed infrastructure and then persist the resulting resource data to state. Destroy similarly removes managed infrastructure and records those removals in state. Therefore, both commands require exclusive state access and are blocked while another operation holds the lock. Terraform's locking behavior depends on backend support, but supported remote backends enforce this coordination automatically for write-capable operations. HashiCorp documents that Terraform locks state for operations that may write it and recommends resolving locks only when it is certain that no active operation owns them. See the

Terraform state locking documentation and the terraform apply command reference. Only two listed actions are forbidden, so the requested count of three is not technically valid.

QUESTION NO: 5

Which of these are benefits of using Sentinel with HCP Terraform/Terraform Cloud? (Pick the 3 correct responses)

A. You can enforce a list of approved AWS AMIs.

B. Sentinel Policies can be written in HashiCorp Configuration Language (HCL).

C. You can check out and check in cloud access keys.

D. Policy-as-code can enforce security best practices.

ANSWER: A D

Explanation:

Sentinel provides policy as code for HCP Terraform, allowing organizations to automatically evaluate Terraform plans against governance requirements before infrastructure changes are applied. “You can enforce a list of approved AWS AMIs.” is a valid use case because a Sentinel policy can inspect planned resource values and require that an AMI identifier is present in an organization-approved allowlist. This helps ensure deployed compute instances use images that have been reviewed for security, configuration, and compliance requirements.

“Policy-as-code can enforce security best practices.” is also correct. Sentinel policies can evaluate Terraform plan data and organization context to enforce guardrails such as mandatory resource tags, permitted instance types, required encryption settings, and restrictions on public network exposure. Policies are version-controlled code and can be applied consistently across workspaces, which supports repeatable governance while retaining an infrastructure-as-code workflow. HCP Terraform evaluates Sentinel policies during runs and can use enforcement levels to determine whether policy results are advisory or blocking. See the HCP Terraform Sentinel policy enforcement documentation and the Sentinel documentation for policy language and enforcement details.

QUESTION NO: 6

A developer launched a VM outside of the Terraform workflow and ended up with two servers with the same name. They are unsure which VM is managed with Terraform, but they do have a list of all active VM IDs. Which method could you use to determine which instance Terraform manages?

A. Modify the Terraform configuration to add an import block for both of the virtual machines.

B. Run a terraform apply -refresh to identify the virtual machine IDs that are already managed by Terraform.

C. Run terraform state rm on both VMs, then terraform apply to recreate the correct one.

D. Run terraform state list to find the names of all VMs, then run terraform state show for each of them to find which VM ID Terraform manages.

ANSWER: D

Explanation:

Run terraform state list to find the names of all VMs, then run terraform state show for each of them to find which VM ID Terraform manages. Terraform state records the bindings between resource addresses in configuration and the real remote objects that Terraform manages. The `terraform state list` command displays the resource addresses currently tracked in the selected state, allowing the developer to identify the relevant virtual-machine resources. Next, `terraform state show <resource-address>` displays the stored attributes for a specific tracked resource. Those attributes normally include the provider’s unique remote object identifier, such as the VM ID. Comparing that ID with the known list of active VM IDs identifies exactly which identically named server is under Terraform management. This approach is read-only with respect to managed infrastructure and directly inspects Terraform’s authoritative state mapping, making it appropriate

for resolving ambiguity caused by manually created resources. HashiCorp documents `terraform state list` as listing resources in the state and `terraform state show` as showing the attributes of a single resource in state. See the terraform state list documentation and the terraform state show documentation.

QUESTION NO: 7

What functionality do providers offer in Terraform?(Pick 3 correct responses)

A. Interact with cloud provider APIs.

B. Provision resources for on-premises infrastructure services.

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

D. Provision resources for public cloud infrastructure services.

E. Enforce security and compliance policies.

ANSWER: A B D

Explanation:

Terraform providers are plugins that implement the integration between Terraform and external systems. They expose resource types and data sources, then translate Terraform operations into the relevant remote-system API calls. Therefore, *Interact with cloud provider APIs.* is correct: a provider enables Terraform to authenticate to and manage services through that provider's API. Providers are not limited to cloud platforms; they can also manage systems deployed in private environments. Consequently, *Provision resources for on-premises infrastructure services.* is correct, because providers exist for services and platforms commonly used in on-premises environments, allowing Terraform configurations to create, update, read, and delete their managed objects. Finally, *Provision resources for public cloud infrastructure services.* is correct because major public-cloud providers supply Terraform providers that manage cloud resources such as compute instances, networking, storage, and managed services. Terraform's provider model is intentionally extensible, so the same workflow and configuration language can manage infrastructure and services across public cloud, private infrastructure, SaaS platforms, and other APIs. Provider configuration also commonly supplies connection details, such as endpoints, regions, and credentials, needed for Terraform to communicate with the target platform. See the Terraform provider documentation and Terraform providers overview.

QUESTION NO: 8

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

A. When you change a Terraform-managed resource via the Azure Cloud Console, Terraform updates the state file to reflect the change during the next plan or apply

B. Changing resources via the Azure Cloud Console records the change in the current state file

C. When you change a resource via the Azure Cloud Console, Terraform records the changes in a new state file

D. Changing resources via the Azure Cloud Console does not update current state file

ANSWER: A D

Explanation:

Changing a Terraform-managed Azure resource directly in the Azure portal is an out-of-band change: Azure changes the real remote object, but it does not directly write to Terraform's state file. Terraform state is maintained by Terraform in its configured backend, rather than by the cloud provider's console, CLI, or API. Therefore, immediately after a portal change, the current stored state can no longer accurately represent the resource's actual remote attributes.

Before Terraform calculates the proposed infrastructure changes for a normal plan or apply operation, it refreshes its knowledge of managed remote objects by reading them from the provider. This allows Terraform to detect drift between the

previously recorded state, the actual Azure resource, and the declared configuration. Terraform can then update its state information with the refreshed remote values and present an execution plan that reconciles the configuration with the observed infrastructure. Reviewing that plan is important because Terraform may propose changes to restore the configuration's intended values. For more detail, see HashiCorp's Purpose of Terraform State documentation and its terraform plan command reference.

QUESTION NO: 9

Which of the following does HCP Terraform perform during a health assessment for a workspace?

Pick the 2 correct responses below:

A. Terraform test execution

B. Check block validation

C. Estimate infrastructure cost

D. Resource drift detection

E. Sentinel policy checks

ANSWER: B D

Explanation:

HCP Terraform health assessments are part of continuous validation for a workspace. They validate deployed infrastructure without requiring a normal plan-and-apply workflow that changes resources. During an assessment, HCP Terraform performs **Check block validation** by evaluating the configuration's check blocks against the current infrastructure. Check blocks let authors define assertions about operational conditions, such as whether a service endpoint responds successfully or a resource has an expected attribute value. This provides ongoing visibility into conditions that may become unhealthy after deployment.

Health assessments also perform **Resource drift detection**. HCP Terraform compares the real remote objects with the values recorded in Terraform state to identify changes made outside Terraform. Detecting this drift helps teams recognize when managed infrastructure no longer matches the declared configuration and decide whether to reconcile it through Terraform or update configuration intentionally. Together, check validation and drift detection allow HCP Terraform to monitor both explicit operational assertions and consistency between state and real infrastructure. See HashiCorp's workspace health assessments documentation and its check block reference.

QUESTION NO: 10

What type of information can be found on the Terraform Registry when using published modules?

A. Required input variables.

B. Outputs.

C. Optional input variables and default values.

D. All of the above.

ANSWER: D

Explanation:

All of the above is correct because a published module's Terraform Registry page automatically presents documentation derived from the module's Terraform configuration. Its Inputs section lists the module's input variables, indicating whether each variable is required and, when a default is declared, displaying that default value. This lets consumers determine which settings they must provide and which settings they can optionally customize.

The same module page also includes an Outputs section. Outputs expose values produced by the module so that a calling configuration can reference useful results, such as resource identifiers, addresses, or other attributes created by the module. Registry documentation therefore gives module users the key interface details needed to configure and consume a module: its required inputs, optional inputs and defaults, and its returned outputs.

HashiCorp documents that the Registry displays module documentation and usage information, while the standard module documentation format includes generated input and output details. See the Terraform Registry module publishing documentation and HashiCorp's guidance on developing Terraform modules.

QUESTION NO: 11 - (SIMULATION)

What is the name of the default file where Terraform stores the state?

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

ANSWER: See the explanation for the answer

Explanation:

The default Terraform state file is `terraform.tfstate`.

With the default local backend, Terraform creates this file in the working directory to record the managed infrastructure state.

QUESTION NO: 12

Which of these are features of HCP Terraform/Terraform Cloud? Pick the 2 correct responses below.

A. Automated infrastructure deployment visualization.

B. A web-based user interface (UI).

C. Automatic backups of configuration and state.

D. Remote state storage.

ANSWER: B D

Explanation:

HCP Terraform provides a web-based user interface (UI) for working with Terraform across an organization. The UI lets users create and manage organizations and workspaces, connect workspaces to version-control repositories, review run status and logs, inspect state-related information, and administer team access and governance settings. This centralized interface is a core part of the HCP Terraform workflow, particularly for collaborative infrastructure management.

Remote state storage is also a core HCP Terraform capability. Each workspace maintains Terraform state remotely, rather than requiring collaborators to share a local state file. HCP Terraform supports state locking during operations and preserves state versions, helping teams coordinate changes safely and maintain a reliable record of managed infrastructure. Remote state also enables Terraform CLI users to authenticate to HCP Terraform and operate against the same workspace-managed state used by their team.

These capabilities are documented in the HCP Terraform overview and its workspace/state-management documentation: HCP Terraform documentation and Remote backend documentation.

QUESTION NO: 13

A module block is shown in the Exhibit space of this page. When you use a module block to reference a module from the Terraform Registry such as the one in the example, how do you specify version 1.0.0 of the module?

A. Append ?ref=v1.0.0 argument to the source path.

B. You cannot. Modules stored on the public Terraform Registry do not support versioning.

C. Add a version = " 1.0.0 " attribute to the module block.

D. Nothing. Modules stored on the public Terraform module Registry always default to version 1.0.0.

ANSWER: C

Explanation:

Add a version = " 1.0.0 " attribute to the module block. Terraform modules obtained from a registry support version selection through the `version` meta-argument in the module block. Setting this argument to `"1.0.0"` is an exact version constraint, so Terraform selects release 1.0.0 of the module identified by the `source` address. For example, a registry module declaration can contain `source = "namespace/name/provider"` and `version = "1.0.0"`. During `terraform init`, Terraform resolves the module version and records the selected version in the dependency lock file when applicable, helping produce repeatable installations. Registry module versions use semantic-version constraints, so the same attribute can also express compatible ranges when a configuration intentionally allows upgrades. The version argument is specifically the standard mechanism for Terraform Registry modules and should be declared directly in the module block alongside the source address. See HashiCorp's module block version documentation and the version constraints reference.

QUESTION NO: 14

You are responsible for a set of infrastructure that is managed by two workspaces: example-network and example-compute. The example-compute workspace uses data from output values configured in the example-network workspace and must be deployed afterward. Currently, this is a manual process:

An operator deploys changes to the example-network workspace.

They manually copy the output values from the example-network workspace to input variables configured for the example-compute workspace.

They deploy the example-compute workspace.

Which HCP Terraform features can you use to automate this process?

Pick the two correct responses below.

A. A health check configured on the example-network workspace to create a plan on the example-compute workspace when HCP Terraform applies changes to it.

B. A health check configured on the example-compute workspace to create a plan when HCP Terraform applies changes to the example-network workspace.

C. A tfe_outputs data source configured in the example-compute workspace to automatically load output values from the example-network workspace.

D. A run trigger configured on the example-network workspace to automatically plan changes to the example-compute workspace after every apply.

E. A run trigger configured on the example-compute workspace to automatically plan changes after HCP Terraform applies changes to the example-network workspace.

ANSWER: C E

Explanation:

A `tfe_outputs` data source configured in the example-compute workspace to automatically load output values from the example-network workspace removes the manual handoff of values between workspaces. The `tfe_outputs` data source is supplied by the HCP Terraform/Enterprise provider and reads designated output values from another workspace. This lets the compute configuration reference network outputs directly during its own plan and apply workflow, while output sharing is

governed by the producing workspace's remote-state sharing settings. HashiCorp documents this pattern in its tfe_outputs data source reference.

A run trigger configured on the example-compute workspace to automatically plan changes after HCP Terraform applies changes to the example-network workspace automates dependency sequencing. Run triggers establish an upstream-to-downstream relationship: when a successful apply occurs in the network workspace, HCP Terraform queues a run in the dependent compute workspace. The resulting run refreshes the downstream configuration against the newly available network outputs. If the compute workspace is configured for automatic apply, its approved run can continue through apply automatically; otherwise, the normal approval controls remain in effect. This preserves workspace-level governance while eliminating the need for an operator to initiate the dependent run manually. See HashiCorp's run triggers documentation for the upstream and downstream workspace model.

QUESTION NO: 15

Your configuration uses a provider version constraint of `~> 5.0`, and the dependency lock file currently records version `5.20.0`. A newer compatible provider version is available. Which command updates the selected provider version in the lock file while still honoring the configured constraint?

A. `terraform init -upgrade`

B. `terraform validate -upgrade`

C. `terraform plan -refresh-only`

D. `terraform apply -replace=provider.aws`

ANSWER: A

Explanation:

`terraform init -upgrade` instructs Terraform to ignore previously selected dependency versions in the lock file and select the newest versions that satisfy the configuration's version constraints. Terraform then updates `.terraform.lock.hcl` with the newly selected provider version and its checksums.

A normal `terraform init` uses the locked version when it still satisfies the configured constraint, which helps maintain consistent provider selections across team members and automation runs. Changing the constraint alone does not necessarily select the newest compatible release unless initialization is run with the upgrade option.

QUESTION NO: 16

You want to create a string that is a combination of a generated random_id and a variable, and reuse that string several times in your configuration.

What is the simplest correct way to implement this without repeating the random_id and variable?

A. Use a data source.

B. Use a module.

C. Add a local value.

D. Add an output value.

ANSWER: C

Explanation:

Add a local value. Terraform local values let you assign a name to an expression and then reference that named result repeatedly within the same module. For example, a `locals` block can define `combined_name = "${random_id.example.hex}-${var.name}"`. Any resource argument, nested block, or other expression in that

module can then use `local.combined_name`. Terraform evaluates the local expression from its dependencies, so the generated random ID and the input variable are incorporated consistently wherever the local is referenced.

This approach keeps the configuration DRY (do not repeat yourself), improves readability, and provides one clear location to update the naming expression if its format changes. Local values are intended for intermediate or derived values that are useful only within a module; they do not create infrastructure and do not require exposing the value outside the module. HashiCorp documents locals as named values that can simplify Terraform configuration and be referenced with `local.<NAME>`. See the Terraform local values documentation and the Terraform references documentation.

QUESTION NO: 17

You created infrastructure outside the Terraform workflow that you now want to manage using Terraform. Which command brings the infrastructure into Terraform state?

A. terraform get

B. terraform refresh

C. terraform import

D. terraform init

ANSWER: C

Explanation:

The `terraform import` command brings an existing remote object under Terraform management by associating that object with a resource address in Terraform state. It is used when the infrastructure already exists—whether it was created manually, by another automation system, or outside the current Terraform configuration—and you want Terraform to begin tracking it. A typical command specifies the configured resource address and the provider-specific remote object ID, such as `terraform import aws_instance.example i-abc123`. After importing, Terraform has a state record for the object, but the corresponding resource configuration must still be present and should be reviewed so that future plans accurately reflect the intended infrastructure. Terraform can also define imports declaratively with `import` blocks, which support previewing imports through the normal plan and apply workflow. In either approach, the essential purpose is the same: map an existing infrastructure object into Terraform state so Terraform can manage its lifecycle going forward. See the Terraform import documentation and the Terraform language documentation for import blocks.

QUESTION NO: 18

Which of the following can you do with terraform plan? (Pick 2 correct responses)

A. Schedule Terraform to run at a planned time in the future.

B. View the execution plan and check if the changes match your expectations.

C. Save a generated execution plan to apply later.

D. Execute a plan in a different workspace.

ANSWER: B C

Explanation:

`terraform plan` creates an execution plan by comparing the current state, the configured desired state, and—in normal planning mode—the current state of remote objects refreshed from provider APIs. Its primary purpose is to let you inspect the actions Terraform proposes, such as creating, modifying, or destroying managed infrastructure, before making those changes. This preview supports safe review of planned changes and helps confirm that they match the intended outcome.

The command can also save the generated plan using the `-out` option, for example `terraform plan -out=tfplan`. The saved plan file can subsequently be passed to `terraform apply tfplan`, ensuring that the reviewed plan is the one

applied. This workflow is especially useful in automation and approval processes. A saved plan contains the intended actions along with relevant planning information and should be treated as sensitive because it can include values from configuration and state.

HashiCorp documents plan review and saved-plan workflows in the terraform plan command reference and describes applying a previously saved plan in the terraform apply command reference.

QUESTION NO: 19

You just scaled your VM infrastructure and realize you set the count variable to the wrong value. You correct the value and save your change. What must you do next to make your infrastructure match your configuration?

A. Reinitialize because your configuration has changed.

B. Inspect all Terraform outputs to make sure they are correct.

C. Inspect your Terraform state because you want to change it.

D. Run terraform apply and confirm the planned changes.

ANSWER: D

Explanation:

Run `terraform apply` and confirm the planned changes. Terraform treats configuration files as the desired state of the infrastructure. After correcting the value used by the `count` meta-argument, Terraform must compare that updated desired configuration with its current state and the real infrastructure, calculate the required changes, and then execute them. The `terraform apply` command performs this workflow: it creates an execution plan, presents the proposed resource changes for review in an interactive run, and applies them after confirmation. For a reduced count value, the plan will ordinarily include destroying the instances that are no longer represented by the configuration; for an increased value, it will include creating additional instances. This reconciles the managed infrastructure with the corrected configuration and updates Terraform state to reflect the resulting resources. You can optionally run `terraform plan` first for a review-only preview, but applying the approved plan is the action required to make the actual infrastructure match the configuration. HashiCorp documents the apply workflow at terraform apply command reference and explains how `count` creates multiple resource instances at The count meta-argument.

QUESTION NO: 20

You modified your Terraform configuration to fix a typo in the resource ID by renaming it from photoes to photos. What configuration will you add to update the resource ID in state without destroying the existing resource?

Original configuration:

resource " aws_s3_bucket " " photoes " {

bucket_prefix = " images "

}

Updated configuration:

resource " aws_s3_bucket " " photos " {

bucket_prefix = " images "

}

A. moved {
from = aws_s3_bucket.photoes
to = aws_s3_bucket.photos
}

B. moved {bucket.photoes = aws_s3_bucket.photos}

C. moved {aws_s3_bucket.photoes = aws_s3_bucket.photos}

D. None. Terraform will automatically update the resource ID.

ANSWER: A

Explanation:

The appropriate configuration is `moved { from = aws_s3_bucket.photoes to = aws_s3_bucket.photos }`, formatted with the `from` and `to` arguments on separate lines in normal Terraform style. A resource's local name is part of its Terraform resource address. Changing that name changes the address Terraform uses to identify the object in configuration, even though the intended remote S3 bucket has not changed. A `moved` block explicitly records the address transition from `aws_s3_bucket.photoes` to `aws_s3_bucket.photos`.

When Terraform creates a plan, it uses this declaration to update the object's state address rather than treating the renamed address as a new bucket. The existing remote object therefore remains associated with the configuration under its corrected address, allowing the rename to be applied safely without replacement. Moved blocks are declarative refactoring metadata and should be retained in shared module configurations so users upgrading from prior versions can have their state updated consistently. HashiCorp documents moved blocks and their use for resource-address refactoring in the Terraform moved block reference. The broader state-management behavior is also described in the Terraform state documentation.

QUESTION NO: 21

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

A. True

B. False

ANSWER: B

Explanation:

False is correct. Terraform modules create their own variable scope. An input variable declared in a child module is available to the configuration files within that module, where it can parameterize resources, data sources, locals, and expressions. Declaring the variable does not make its value directly addressable by the parent configuration or by sibling modules. Instead, a calling module supplies a value through the module block's arguments, matching the child module's declared input variable names.

To deliberately make a value from a child module available to its caller, the child module must declare an `output` block. The caller can then reference that exported value using the module namespace, such as `module.network.vpc_id`. Outputs are therefore Terraform's explicit interface for exposing values across a module boundary, while variables define inputs accepted by a module. This isolation supports reusable, encapsulated module designs with clear input and output contracts. HashiCorp documents module input variables and output values at https://developer.hashicorp.com/terraform/language/values/variables and https://developer.hashicorp.com/terraform/language/values/outputs.

QUESTION NO: 22

Where does HashiCorp recommend you store API tokens and other secrets within your team's Terraform workspaces?

Pick the three correct responses below.

A. In a plaintext document on a shared drive.

B. In HashiCorp Vault.

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

D. In an environment variable and referenced with TF_VAR_variablename.

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

ANSWER: B D E

Explanation:

HashiCorp Vault is an appropriate location for API tokens and other secrets because it is purpose-built for centralized secrets management. Vault can securely store static secrets, dynamically generate credentials for supported systems, enforce fine-grained access policies, and create audit records for secret access. Terraform can retrieve values from Vault through its provider or other controlled workflow integrations.

An environment variable and referenced with `TF_VAR_variablename` is also an appropriate method when a secret must be supplied as a Terraform input variable. Environment variables are especially useful in local development and automated pipelines because the secret can be injected at runtime rather than placed in Terraform configuration. Terraform recognizes environment variables with the `TF_VAR_` prefix as input-variable values.

An HCP Terraform variable, with the sensitive option checked, is designed for supplying confidential workspace inputs to remote runs. Marking a variable as sensitive prevents HCP Terraform from displaying its value in the user interface and masks it in run output where applicable. This allows a workspace to receive required credentials or configuration secrets without embedding them in version-controlled configuration. See the HashiCorp Vault documentation and Terraform environment-variable documentation.

QUESTION NO: 23

Where does the Terraform local backend store its state?

A. In the terraform file

B. In the /tmp directory

C. In the terraform.tfstate file

D. In the user's terraform,state file

ANSWER: C

Explanation:

The local backend stores Terraform state in a file named `terraform.tfstate` in the current working directory by default. Terraform uses this state file to map resources declared in configuration to real infrastructure objects, retain resource metadata, and determine what changes are necessary during planning and applying. This default behavior applies when no backend configuration selects another backend and the local backend is in use. The local backend's `path` setting can be configured to use a different state-file path, but the standard default remains `terraform.tfstate`. For workspaces other than the default workspace, Terraform stores local state in workspace-specific locations beneath the working directory's `terraform.tfstate.d` directory; however, the default workspace uses the ordinary `terraform.tfstate` file. Because local state can contain sensitive values and is only directly accessible on the machine where it resides, teams commonly use a remote backend for shared, collaborative infrastructure management. HashiCorp documents the local backend's default state path and its configurable `path` argument in the local backend reference. Additional background on the role and handling of state is available in the Terraform state documentation.

QUESTION NO: 24

Exhibit

A resource block is shown in the Exhibit space of this page. What is the provider for this resource?

A. test

B. vpc

C. aws

D. main

ANSWER: C

Explanation:

The provider is **aws**. Terraform resource blocks use a type identifier followed by a local name, in the form `resource "TYPE" "NAME"`. For provider-managed resources, the type is conventionally composed of a provider type prefix and a resource kind separated by an underscore. Thus, the exhibited resource type `aws_vpc` identifies the AWS provider through its `aws` prefix, while `vpc` identifies the particular resource type that provider manages: an Amazon Virtual Private Cloud.

The second label, such as `main`, is Terraform's local name for the declared resource instance. It is used with the type to form references such as `aws_vpc.main`; it does not identify the provider. Likewise, argument values inside the block configure properties of that resource rather than determine its provider. Terraform uses provider requirements and provider configuration to associate resource types with the relevant provider plugin, and the resource type prefix is the key visual indicator in this declaration. See HashiCorp's resource block syntax documentation and its provider requirements documentation for details.

QUESTION NO: 25

A child module named `network` defines the output shown below.

```
output "vpc_id" {
value = aws_vpc.main.id
}
```

The root module calls this child module with `module "network" { source = "./modules/network" }`. Which expression can the root module use to reference the VPC ID?

A. `module.network.vpc_id`

B. `network.vpc_id`

C. `aws_vpc.main.id`

D. `output.network.vpc_id`

ANSWER: A

Explanation:

`module.network.vpc_id` is correct because child-module outputs are referenced through the module call name followed by the output name. The child module exposes `vpc_id`, and the root module's local name for that module call is `network`.

The root module cannot directly reference `aws_vpc.main.id` because resources declared in a child module are scoped to that module. The child module must expose the value through an output before its caller can consume it.

QUESTION NO: 26

You want to bring an existing database under Terraform management. What information is required to create a new import block for the database?

Pick the two correct responses below.

A. The destination resource address of the block that will manage the database.

B. The path to the .tf file that contains the database resource block.

C. The ID associated with the current database on the cloud provider.

D. The database platform and version that the existing resource is running.

E. The connection string that Terraform will use to connect and manage the database.

ANSWER: A C

Explanation:

An import block declares that Terraform should associate an already-existing remote object with a particular resource instance in the Terraform state. The destination resource address of the block that will manage the database is required because the `to` argument identifies the exact managed resource address that Terraform will place the imported object under. This address can refer to a resource in the root module or in a child module, and it must correspond to the intended resource instance in configuration.

The ID associated with the current database on the cloud provider is also required because the `id` argument tells the configured provider which existing remote object Terraform must retrieve and import. The required identifier format is provider and resource-type specific. For example, a provider may require a database instance identifier, a full resource path, or a composite identifier. Terraform uses provider configuration for authentication and access, then uses this import ID to locate the database.

Together, the resource address and provider-defined ID provide the complete import mapping: the existing object to adopt and the Terraform resource instance that will manage it after the import. See HashiCorp's Import existing infrastructure documentation and the import block reference.

QUESTION NO: 27

You have provisioned some virtual machines (VMs) on Google Cloud Platform (GCP) using the gcloud command line tool. However, you are standardizing with Terraform and want to manage these VMs using Terraform instead. What are the two things you must do to achieve this? Choose two correct answers.

A. Run the terraform Import-gcp command

B. Write Terraform configuration for the existing VMs

C. Use the terraform import command for the existing VMs

D. Provision new VMs using Terraform with the same VM names

ANSWER: B C

Explanation:

To bring pre-existing GCP VMs under Terraform management, first write Terraform resource configuration representing each VM, such as a `google_compute_instance` resource. The resource address in this configuration, including its type and

local name, is the address Terraform will use to track the object in state. The configuration should be reviewed and completed so that its arguments describe the intended ongoing management of the VM.

Next, run `terraform import` for each VM, supplying the configured resource address and the provider-specific import ID. Import associates the remote object with that address in the Terraform state; it does not create or modify the VM merely by performing the import. After importing, run `terraform plan` to compare the configuration with the imported remote object and adjust configuration as needed before applying changes. For instance resources, the Google provider documentation defines the accepted import ID formats. See HashiCorp's Terraform import command documentation and the Google Compute Engine instance import documentation.

QUESTION NO: 28

Which of the following should you put into the required_providers block?

A. version > = 3.1

B. version = ">>= 3.1"

C. version ~ > 3.1

ANSWER: B

Explanation:

`version = ">= 3.1"` is valid Terraform syntax for declaring a provider version constraint within a provider requirement. In a `terraform` block, the `required_providers` map identifies providers used by the configuration and can specify each provider's source address and acceptable version range. The `version` argument must be assigned using Terraform's normal argument syntax: an equals sign followed by a quoted string. Here, `>= 3.1` permits Terraform to select version 3.1 or any later compatible provider release when initializing the working directory, subject to any other applicable dependency constraints. Version constraints are strings because they use Terraform's version-constraint language, which supports operators such as `>=`, `~>`, and exact-version matching. Declaring constraints helps make provider selection predictable and lets teams control which provider releases their configuration is intended to use. In production configurations, HashiCorp also recommends specifying the provider source address alongside the version constraint. See the Terraform provider requirements documentation and the version constraints reference.

QUESTION NO: 29

Which method for sharing Terraform modules fulfills the following criteria:

Keeps the module configurations confidential within your organization.

Supports Terraform ' s semantic version constraints.

Provides a browsable directory of your modules.

A. A Git repository containing your modules.

B. Public Terraform module registry.

C. A subfolder within your workspace.

D. HCP Terraform/Terraform Cloud private registry.

ANSWER: D

Explanation:

HCP Terraform/Terraform Cloud private registry is the appropriate module-sharing method because it provides a private, organization-scoped registry for publishing and consuming Terraform modules. Organizations can control who can discover

and use modules through HCP Terraform access controls, keeping module configurations confidential rather than exposing them through the public registry.

Each module publication is assigned a version, and Terraform configurations can specify version constraints when calling a registry module. This enables consumers to select compatible releases using Terraform's normal version-constraint syntax while module authors safely publish updated versions over time. The private registry also presents a searchable, browsable catalog of approved modules, including module names, providers, versions, documentation, and usage information. This creates a centralized internal source for reusable infrastructure patterns and helps teams discover standardized modules without relying on manual repository links.

HashiCorp documents private registry modules as versioned modules published within an organization and available through the registry interface. See the HCP Terraform private registry documentation and the Terraform version constraints reference.

QUESTION NO: 30

Running terraform fmt without any flags in a directory with Terraform configuration files will check the formatting of those files, but will never change their contents.

A. True

B. False

ANSWER: B

Explanation:

The correct answer is **False**. When run with no flags, `terraform fmt` rewrites Terraform configuration files in the current directory to conform to Terraform's canonical style. This includes changes such as normalizing indentation, spacing, alignment, and other formatting conventions. The command is designed to make configuration consistently formatted and is normally safe to run repeatedly, because once a file follows the canonical format, subsequent runs produce no further changes.

Terraform reports the names of files that it reformats, so users can see which files were changed. This default behavior makes `terraform fmt`terraform fmt -check. The `-check` flag changes the command into a verification mode: it indicates whether formatting changes would be needed and returns a nonzero exit status when files are not properly formatted, which is useful in automated CI checks.

HashiCorp documents that `terraform fmt` "automatically updates configurations in the current directory for readability and consistency," while `-check` checks whether input is formatted without writing changes. See the Terraform fmt command reference and the Terraform configuration syntax documentation.

QUESTION NO: 31

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:

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

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

A. Change the reference value to my-network,outputs,vmet_id

B. Define the attribute vmet_id as a variable in the networking modeule

C. Define the attribute vnet_id as an output in the networking module

D. Change the reference value module.my,network,outputs,vnet_id

ANSWER: C

Explanation:

Define the attribute vnet_id as an output in the networking module. A child module has its own scope, so values created or obtained inside that module are not automatically available to the calling configuration. To intentionally expose a value, the module must contain an `output` block, such as `output "vnet_id" { value = ... }`, where the value expression refers to the relevant resource attribute or other value within the module. The root module can then consume that declared output through the module call using an expression such as `module.my-network.vnet_id`. Declaring the output forms the public interface of the networking module and lets Terraform validate the reference before applying infrastructure changes. Outputs are specifically intended to return useful data from a child module to its parent, including resource IDs that other configuration components need. The output name must match the name used by the calling configuration, so exposing it as `vnet_id` makes that value addressable from the module call. See HashiCorp's documentation on output blocks and accessing module output values.

QUESTION NO: 32

What functionality do providers offer in Terraform? (Pick the 3 correct responses below.)

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

B. Provision resources for on-premises infrastructure services.

C. Provision resources for public cloud infrastructure services.

D. Interact with cloud provider APIs.

E. Enforce security and compliance policies.

ANSWER: B C D

Explanation:

Terraform providers are plugins that extend Terraform by defining resource types and data sources for a particular infrastructure platform or service. They enable Terraform to manage public-cloud infrastructure, including services offered by AWS, Azure, Google Cloud, and many other cloud vendors. A provider translates the resource declarations in Terraform configuration into the API operations needed to create, read, update, and delete infrastructure objects.

Providers are not limited to public clouds. They also support on-premises and self-managed services, such as VMware vSphere, Kubernetes, network appliances, DNS systems, databases, and source-control platforms. In each case, the provider is the integration component that authenticates to and communicates with the service's API, allowing Terraform to manage that service declaratively through its supported resource and data source types.

This plugin model lets the same Terraform workflow work across many infrastructure domains while each provider implements the service-specific API behavior. HashiCorp describes providers as plugins that interact with cloud providers, SaaS providers, and other APIs. See the Terraform provider documentation and the Terraform core workflow documentation.

QUESTION NO: 33

After creating a new Terraform configuration, your configuration passes terraform validate but returns an “Access Denied” error from the cloud provider when running terraform plan.

Why did terraform validate not catch this issue?

A. Variables are only applied and validated during terraform plan, so terraform validate assumed defaults and returned a success message.

B. The working directory was not initialized, so the cloud provider plugin was unavailable when running the terraform validate command.

C. terraform validate only checks whether a configuration is syntactically correct and internally consistent, and it does not communicate with providers.

D. The remote backend was not configured, so terraform validate could not load the state and detect the missing credentials.

ANSWER: C

Explanation:

`terraform validate` only performs static validation of the configuration in the current working directory. It checks that the Terraform files are syntactically valid and that their internal references, argument usage, and type constraints are consistent with the installed provider schemas and module definitions. It is therefore useful for detecting configuration errors before attempting an operation that interacts with infrastructure.

Provider authentication and authorization are different concerns. An “Access Denied” response occurs when Terraform performs `terraform plan` and the provider needs to authenticate to the cloud platform, read remote objects, refresh state, or otherwise call that platform’s API. At that point, the cloud provider evaluates the supplied credentials and their permissions. Because validation does not make provider API calls, it cannot determine whether those credentials are valid or authorized for the requested cloud resources.

This separation lets teams run fast, local configuration checks without requiring active cloud credentials, while planning and applying remain the stages that verify real access to the target environment. HashiCorp documents that validation checks syntax and internal consistency without validating remote services. See Terraform validate command reference and Terraform plan command reference.

QUESTION NO: 34

You have used Terraform lo create an ephemeral development environment in the (loud and are now ready to destroy all the Infrastructure described by your Terraform configuration To be safe, you would like to first see all the infrastructure that Terraform will delete.

Which command should you use to show all of the resources that mil be deleted? Choose two correct answers.

A. Run terraform state rm ‘

B. Run terraform show :destroy

C. Run terraform destroy and it will first output all the resource that will be deleted before prompting for approval

D. Run terraform plan .destory

E. Run terraform plan -destroy

ANSWER: C E

Explanation:

`terraform destroy` is correct because Terraform creates and displays a destroy plan before it performs any destructive action. In an interactive terminal, it lists the managed objects scheduled for deletion, summarizes the planned changes, and then requests confirmation. This provides an opportunity to inspect the complete proposed destruction and cancel rather than approve it.

`terraform plan -destroy` is also correct because it generates a speculative destroy plan only. It uses the current configuration and state to show the resources Terraform proposes to delete, but it does not modify infrastructure and does not require an approval prompt. This is especially useful when a team wants to review the destruction plan separately before later running the destroy operation. As with any Terraform plan, the output is based on the state and provider information available when the command runs; refreshing state first can help ensure the preview reflects the latest known remote objects.

HashiCorp documents that `terraform destroy` is a convenience alias for applying a destroy plan, while the `-destroy` planning mode produces the corresponding preview. See the Terraform destroy command documentation and the Terraform plan command documentation.

QUESTION NO: 35

Where does HashiCorp recommend you store API tokens and other secrets within your team ' s Terraform workspaces?

Pick three correct responses below:

A. In a plaintext document on a shared drive.

B. In HashiCorp Vault.

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

D. In an environment variable and referenced with TF_VAR_variablename.

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

ANSWER: B D E

Explanation:

HashiCorp Vault is an appropriate location for API tokens and other secrets because it is purpose-built for secret storage, access control, auditing, and controlled delivery of credentials to workloads. Terraform can obtain dynamic or static credentials from Vault through its provider and data-source integrations, avoiding hard-coding the secret in Terraform configuration.

An environment variable referenced through the `TF_VAR_` convention is also a supported way to supply an input variable value at run time. This keeps a sensitive value out of `.tf` and version-controlled `.tfvars` files. Teams should still use secure CI/CD secret injection and protect the execution environment, since environment variables are available to the process that runs Terraform.

An HCP Terraform variable with the sensitive setting enabled is suitable for workspace-specific secret inputs. Sensitive workspace variables are concealed in the HCP Terraform user interface and are redacted from run output, allowing a team to manage protected values centrally for remote runs. The value can then be passed to the Terraform configuration as an input without placing it in source control. See HashiCorp guidance on assigning variable values with environment variables and managing HCP Terraform workspace variables.

QUESTION NO: 36

Which of the following ate advantages of using infrastructure as code (laC) instead of provisioning with a graphical user interface (GUI)? Choose two correct answers.

A. Lets you version, reuse, and share infrastructure configuration

B. Provisions the same resources at a lower cost

C. Secures your credentials

D. Reduces risk of operator error

E. Prevents manual modifications to your resources

ANSWER: A D

Explanation:

“Lets you version, reuse, and share infrastructure configuration” is a core advantage of infrastructure as code. Terraform configurations are human-readable files that can be stored in version control alongside application code. This makes infrastructure changes reviewable, traceable, and repeatable, while modules enable teams to package and reuse standard infrastructure patterns across projects and environments. Version-controlled configuration also supports collaborative workflows such as pull-request review and automated validation in CI/CD pipelines.

“Reduces risk of operator error” is also an advantage because declarative configuration specifies the intended end state instead of requiring people to repeatedly perform click-by-click provisioning steps. Terraform creates an execution plan before applying changes, allowing teams to inspect the proposed actions. Applying the same reviewed configuration consistently helps avoid configuration drift caused by inconsistent manual procedures and reduces the likelihood of omissions during repetitive provisioning work. Terraform state additionally records the resources it manages so that Terraform can compare the declared configuration with managed infrastructure when planning updates. HashiCorp

describes these benefits as automation, repeatability, and predictable infrastructure changes. See the Terraform introduction and HashiCorp's guidance on Terraform state.

QUESTION NO: 37

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

A. Plan

B. Import

C. Validate

D. Init

E. apply

ANSWER: D E

Explanation:

`Init` and `apply` are the required workflow steps for provisioning infrastructure from a new or uninitialized Terraform working directory. `terraform init` prepares the directory before Terraform can perform operations against the configuration. During initialization, Terraform installs the providers required by the configuration, initializes the configured backend, and prepares modules used by the root module. This establishes the local working environment and the state-storage connection Terraform needs to manage infrastructure consistently.

`terraform apply` is the operation that performs the infrastructure changes. It evaluates the configuration, creates an execution plan (unless a previously saved plan file is supplied), presents the proposed actions for approval in interactive use, and then calls the relevant provider APIs to create the declared resources. Together, initialization prepares Terraform to work with the configuration and its dependencies, while applying executes the desired resource creation. This aligns with Terraform's standard workflow for deploying managed infrastructure. See HashiCorp's documentation for terraform init and terraform apply.

QUESTION NO: 38

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

A. resource.kubernetes_namespace.example.name

B. kubernetes_namespace.example.name

C. data.kubernetes.namespace.name

D. kubernetes_namespace.test.name

E. kubernetes_namespace.example.metadata[0].name

ANSWER: E

Explanation:

`kubernetes_namespace.example.metadata[0].name` is the appropriate reference for the `name` value shown inside the `metadata` block of a `kubernetes_namespace` resource named `example`. Terraform references a managed resource using its resource type and local name, here `kubernetes_namespace.example`. The namespace name is not a top-level attribute of that resource configuration; it is an attribute within the nested `metadata` block. For this provider schema, `metadata` is represented as a collection of nested objects, so the first configured block is addressed with index `[0]`, followed by `.name`. This produces an expression that Terraform can evaluate wherever an input value or output expression is expected, including an interpolation such as `${kubernetes_namespace.example.metadata[0].name}` when

interpolation syntax is needed within a larger string. Terraform's expression documentation describes traversing resource values and accessing collection elements with index notation. The Kubernetes provider documentation defines the namespace resource's `metadata` block and its `name` field. See Terraform references to values and Kubernetes Namespace resource documentation.

QUESTION NO: 39

You have a saved execution plan containing desired changes for infrastructure managed by Terraform. After running the command terraform apply my.tfplan, you receive the error shown in the exhibit below.

Exhibit:

Error: Saved plan is stale

The given plan file can no longer be applied because the state was changed by another operation after the plan was created.

How can you apply the desired changes? (Choose TWO correct answers)

A. Update the current plan file using the terraform state push command.

B. Refresh the current state data using the -refresh-only flag.

C. Force the apply command by adding the flag -lock=false.

D. Generate a new execution plan file with terraform plan, and apply the new plan.

E. Run terraform apply without the saved execution plan.

ANSWER: D E

Explanation:

Generate a new execution plan file with terraform plan, and apply the new plan is correct because a saved plan is tied to the particular Terraform state snapshot that existed when Terraform created it. Terraform stores state lineage and serial information in a plan, then verifies that information when applying it. Once another operation changes the state, the original saved plan is no longer safe to use. Creating a replacement plan obtains the current state and produces an execution plan based on that current snapshot; that newly generated plan can then be reviewed and applied.

Run terraform apply without the saved execution plan is also correct. When no plan file is supplied, Terraform performs the normal workflow: it refreshes/reads the current state as part of planning, presents a newly calculated plan, and—after approval—applies that plan. This avoids attempting to apply the stale saved artifact and ensures Terraform bases the requested infrastructure changes on the latest state available at apply time. In collaborative environments, state locking should remain enabled so concurrent Terraform operations do not alter state during this process. HashiCorp documents saved-plan behavior and the normal planning/apply workflow in terraform plan command reference and terraform apply command reference.