PassQueen
a.k.a dumpsqueen.com

# HashiCorp Certified: Terraform Associate

## HashiCorp Terraform-Associate

**Version Demo**

**Total Demo Questions: 36**

**Total Premium Questions: 365**

**Buy Premium PDF**

https://passqueen.com

support@passqueen.com

# Topic Break Down

| Topic | No. of Questions |
|---|---|
| Topic 1, Understand Infrastructure as Code (IaC) concepts | 21 |
| Topic 2, Understand Terraform basics | 27 |
| Topic 3, Use Terraform outside of core workflow | 44 |
| Topic 4, Interact with Terraform modules | 34 |
| Topic 5, Use the core Terraform workflow | 53 |
| Topic 6, Implement and maintain state | 73 |
| Topic 7, Read, generate, and modify configuration | 85 |
| Topic 8, Understand Terraform Cloud and Enterprise capabilities | 28 |
| Total | 365 |

QUESTION NO: 1

What features does the hosted service Terraform Cloud provide? (Choose two.)

**A.** Automated infrastructure deployment visualization

**B.** Automatic backups

**C.** Remote state storage

**D.** A web-based user interface (UI)

**ANSWER: C D**

**Explanation:**

Terraform Cloud provides **remote state storage** through its remote backend and workspace model. A workspace stores Terraform state remotely, allowing teams and automation to use a shared, authoritative state location rather than relying on an individual operator's local state file. Terraform Cloud also manages state versions and coordinates operations against that remote state, which supports safer collaboration across users and CI/CD systems.

Terraform Cloud also provides **a web-based user interface (UI)**. The UI is used to create and manage organizations and workspaces, connect version-control repositories, configure variables, review run status and logs, inspect state-related information, and administer access. This browser-based management experience is a core part of Terraform Cloud as a hosted service, alongside its API and command-line workflow support.

These features let teams centrally manage Terraform workflows and state without operating the Terraform Enterprise application themselves. See HashiCorp's Terraform Cloud workspaces documentation and the remote state documentation for details.

**QUESTION NO: 2**

You want to define multiple data disks as nested blocks inside the resource block for a virtual machine. What Terraform feature would help you define the blocks using the values in a variable?

**A.** Local values

**B.** Count arguments

**C.** Collection functions

**D.** Dynamic blocks

**ANSWER: D**

**Explanation:**

**Dynamic blocks** are designed for generating repeated nested configuration blocks from a collection value, such as a list, set, or map supplied through a variable. Within a resource definition for a virtual machine, a `dynamic` block uses the `for_each` argument to iterate over the disk definitions. Terraform then creates one concrete nested data-disk block for each item in that collection. This is useful when the number of disks is variable or when each disk has attributes, such as its name, size, type, or logical unit number, that should be defined in structured input rather than duplicated manually in configuration.

The dynamic block's iterator provides access to the current collection element, allowing the nested block's arguments to be populated from each variable value. Dynamic blocks generate only normal nested blocks supported by the resource schema; they do not generate arbitrary Terraform meta-arguments. This pattern keeps the resource configuration concise while making the virtual machine's disk layout configurable and reusable across environments. HashiCorp documents dynamic blocks and their `for_each`-based expansion at Dynamic Blocks. For guidance on declaring structured variable input, see Input Variables.

## QUESTION NO: 3 - (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:**

**Answer:** `terraform.tfstate`

By default, Terraform stores local state in a file named `terraform.tfstate`.

## QUESTION NO: 4

Which of these commands makes your code more human readable?

**A.** Terraform validate

**B.** Terraform output

**C.** Terraform show

**D.** terraform fmt

## ANSWER: D

**Explanation:**

`terraform fmt` is the command that improves the human readability of Terraform configuration by rewriting files into Terraform's canonical, idiomatic format. It automatically standardizes details such as indentation, whitespace, alignment, and the layout of configuration blocks and arguments. Applying a consistent format makes configuration easier for engineers to scan, review, and maintain, particularly when several people contribute to the same repository.

The command is intentionally opinionated: Terraform defines the formatting conventions rather than requiring teams to maintain a separate formatting ruleset. It can be run against the current directory or recursively across subdirectories, and it is commonly used before commits or enforced in continuous-integration checks. Using `terraform fmt -check` in automation checks whether files already conform to the canonical format without modifying them. Formatting does not change Terraform resource behavior or create infrastructure; it improves the presentation and consistency of the configuration source code. HashiCorp recommends using this standard formatter as part of normal Terraform development workflows. See the official terraform fmt command documentation and the Terraform style guide for the canonical formatting and style conventions.

## QUESTION NO: 5

Terraform can run on Windows or Linux, but it requires a Server version of the Windows operating system.

**A.** True

**B.** False

## ANSWER: B

**Explanation:**

**False** is correct because Terraform does not require Windows Server. HashiCorp distributes Terraform as a command-line binary for multiple operating systems, including Windows, Linux, and macOS. On Windows, Terraform is installed by

downloading the appropriate Windows archive, extracting `terraform.exe`, and making that executable available through the system `PATH`. This installation process applies to supported Windows environments generally; it is not limited to Windows Server editions.

Terraform is designed as a cross-platform CLI, allowing practitioners to use the same Terraform configuration files and core workflow commands, such as `terraform init`, `terraform plan`, and `terraform apply`, from a Windows workstation or a Linux host. The relevant platform consideration is selecting the correct Terraform binary for the machine architecture and operating system, rather than using a server-class Windows edition. HashiCorp's installation guidance specifically includes Windows as a supported platform and provides the Windows executable installation steps. See the official Terraform installation documentation and the Terraform CLI documentation.

## QUESTION NO: 6

You decide to move a Terraform state file to Amazon S3 from another location. You write the code below into a file called backend.tf.

Which command will migrate your current state file to the new S3 remote backend?

**A.** terraform state

**B.** terraform init

**C.** terraform push

**D.** terraform refresh

## ANSWER: B

**Explanation:**

`terraform init` is the command that initializes a working directory and configures its backend. When Terraform detects that the backend configuration has changed—for example, after adding an Amazon S3 backend configuration in `backend.tf`—initialization includes backend setup. Terraform recognizes that state is currently associated with a different backend and interactively offers to copy, or migrate, that existing state to the newly configured remote location. Confirming the migration moves the state data to the S3 backend so future Terraform operations use that remote state.

In an automated or non-interactive workflow, the same operation can be explicitly approved with `terraform init -migrate-state`, optionally together with suitable backend configuration inputs. The normal interactive `terraform init` command is the appropriate answer because it performs the required backend initialization and prompts for state migration when applicable. Terraform records backend metadata locally after successful initialization, while the authoritative state is stored in the configured remote backend. HashiCorp documents backend initialization and migration behavior in the terraform init command reference and explains remote state backend configuration in the backend documentation.

## QUESTION NO: 7

Which statements about Terraform variable validation rules are true? (Choose two.)

**A.** It is declared inside a variable block.

**B.** It can reject input values that do not meet a condition.

**C.** It automatically converts invalid input values.

**D.** It is declared inside the resource that uses the variable.

**E.** It can only validate string variables.

## ANSWER: A B

**Explanation:**

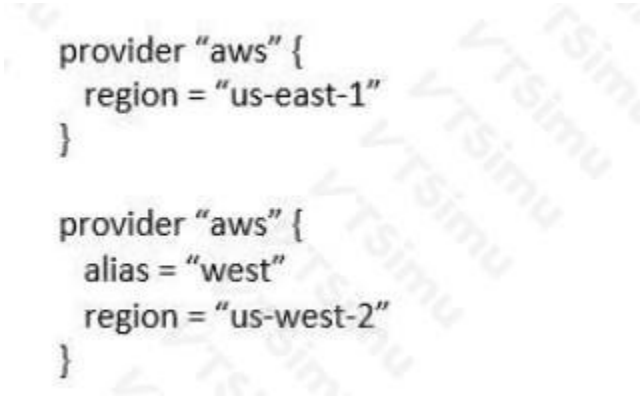
A `validation` block is nested inside an input `variable` block. It allows a module author to define a condition that input values must satisfy and an error message Terraform displays when the condition evaluates to false.

The condition must refer to the variable being validated, using the `var` namespace. For example, a variable named `environment` can validate that its value is one of a permitted set. Validation helps module authors provide earlier and clearer feedback than a provider API error might provide.

Variable validation is not a mechanism for changing an invalid value into a valid one, and it is not declared inside a resource block. See the Terraform variable validation documentation.

## QUESTION NO: 8

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:




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

**A.** alias = west

**B.** provider = west

**C.** provider = aws.west

**D.** alias = aws.west

## ANSWER: C

**Explanation:**

`provider = aws.west` is the required resource meta-argument when the AWS provider configuration for `us-west-2` has the alias `west`. Terraform identifies a provider configuration by its provider local name, `aws`, followed by a period and the alias, `west`. Assigning that provider reference in the resource block explicitly selects the aliased AWS configuration, so Terraform uses its regional settings when planning and creating that resource.

Terraform permits multiple configurations for the same provider in a module. One configuration can be the default, while additional configurations require an `alias`. Resources that should use one of those additional configurations must specify the `provider` meta-argument with the complete provider configuration reference. For example, a resource such as `aws_instance` can include `provider = aws.west`; Terraform will then send AWS API operations for that resource through the credentials and region defined in the aliased `aws` provider configuration. This approach allows a single Terraform configuration to manage infrastructure in multiple AWS regions predictably. See HashiCorp's provider alias documentation and its resource provider meta-argument reference.

## QUESTION NO: 9

Which of the following locations can Terraform use as aprivate sourcefor modules?(Pick 2 correct responses)

**A.** Public repository on GitHub.

**B.** Public Terraform Registry.

**C.** Internally hosted VCS (Version Control System) platform.

**D.** Private repository on GitHub.

**ANSWER: C D**

**Explanation:**

Terraform can install modules directly from version control systems, including private repositories, provided that the Terraform runtime has suitable credentials to authenticate with the source. A private repository on GitHub is therefore a valid private module source. For example, a module source can use a GitHub Git URL, and Terraform can authenticate using SSH credentials, HTTPS credentials, or other Git-supported authentication mechanisms available in the execution environment.

An internally hosted VCS (Version Control System) platform is also a valid private source. Terraform supports Git-based module sources through generic Git URLs, allowing organizations to retrieve modules from self-managed services such as GitHub Enterprise, GitLab, Bitbucket Server, or another internally accessible Git server. This approach lets teams retain module code within their own network and access-control boundaries while continuing to use normal Terraform module installation workflows. The source address identifies the repository and can optionally select a subdirectory and revision. HashiCorp documents Git repositories as supported module sources and notes that Terraform relies on the Git client and its configured authentication for these installations. See Terraform module source documentation and Git repository module sources.

**QUESTION NO: 10**

Which statements about the `prevent_destroy` lifecycle argument are true? (Choose two.)

**A.** It causes Terraform to reject plans that destroy the resource.

**B.** It is configured in a resource lifecycle block.

**C.** It prevents deletion through a cloud provider console.

**D.** It automatically creates a replacement resource before destruction.

**E.** It prevents terraform state rm from removing the resource from state.

**ANSWER: A B**

**Explanation:**

`prevent_destroy = true` causes Terraform to reject a plan that would destroy the resource instance. This can help protect infrastructure that must not be removed accidentally, such as a production database or a critical storage bucket.

The lifecycle setting is declared inside a resource block. It applies to the resource instances managed by that resource configuration and is evaluated when Terraform creates a plan. If a configuration change requires replacement, Terraform must destroy the existing object as part of that replacement, so the setting also prevents that plan unless the protection is removed from the configuration first.

`prevent_destroy` does not prevent deletion performed outside Terraform, and it does not prevent Terraform from removing an object from state with `terraform state rm`. See the Terraform lifecycle meta-argument documentation.

**QUESTION NO: 11**

What is the purpose of the .terraform directory in a Terraform workspace?

**A.** The directory is where Terraform creates and maintains the state file to track the underlying resources it creates and manages.

**B.** The directory is used to convert and store Terraform configuration files into API calls to communicate with the targeted platform.

**C.** The directory contains the provider credentials and the .tfvars files to prevent them from being committed to version control by accident.

**D.** The directory contains plugins and modules that Terraform downloads during initialization, along with other important information.

**ANSWER: D**

**Explanation:**

The `.terraform` directory contains local working data that Terraform creates when you run `terraform init`. Its main purpose is to cache and record the dependencies and initialization metadata needed for Terraform to operate in that working directory. This includes downloaded provider plugins, installed child modules, and information about the configured backend. Keeping these artifacts locally allows later Terraform commands, such as `terraform plan` and `terraform apply`, to use the initialized providers and modules consistently without downloading them again for every command. Depending on the backend configuration, Terraform may also store local metadata related to backend initialization in this directory; however, it is not generally the location of the authoritative state for remote backends. The directory is generated content rather than source configuration, so HashiCorp recommends excluding it from version control. Terraform can recreate it by running `terraform init` again, provided the configuration and required dependencies remain available. For more detail, see HashiCorp's documentation on terraform init and the Terraform directory structure.

## QUESTION NO: 12

What is one disadvantage of using dynamic blocks in Terraform?

**A.** They cannot be used to loop through a list of values

**B.** Dynamic blocks can construct repeatable nested blocks

**C.** They make configuration harder to read and understand

**D.** Terraform will run more slowly

**ANSWER: C**

**Explanation:**

They make configuration harder to read and understand is correct because dynamic blocks generate repeated nested configuration programmatically, rather than showing each nested block explicitly in the resource definition. Although this is useful when the nested blocks are genuinely repetitive and driven by a collection, it can obscure the final structure of a resource. A reader must mentally evaluate the `for_each` expression, iterator values, and `content` block to determine which nested blocks Terraform will generate. This added indirection can make reviews, troubleshooting, and future maintenance more difficult, particularly when a dynamic block is used for only a small amount of repetition or contains complex expressions. HashiCorp explicitly recommends using dynamic blocks only when needed, noting that overuse can make a configuration difficult to read and maintain. The practical best practice is to prefer directly written nested blocks when their number and content are known and reasonably small, and to use dynamic blocks when they clearly reduce substantial duplication while retaining understandable input data. See HashiCorp's guidance on dynamic blocks and its Terraform style guide.

## QUESTION NO: 13

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

**E.** terraform plan

**ANSWER: A C E**

**Explanation:**

`terraform apply`, `terraform destroy`, and `terraform plan` require Terraform to acquire the workspace state lock before proceeding. State locking prevents concurrent Terraform operations from using and potentially updating the same state at the same time, which protects the state from conflicting changes and corruption. An existing lock causes these commands to stop rather than proceed, unless the lock is properly released or an operator deliberately uses the locking override only when it is safe to do so. `terraform apply` and `terraform destroy` perform infrastructure changes and update state, while `terraform plan` normally refreshes resource information and uses locking by default as part of its state-related workflow. Terraform documents that it automatically locks state for operations that can write state, and both the plan and apply command references document the `-lock=false` option, whose default is to retain locking behavior. See the Terraform state locking documentation and the terraform plan command reference.

## QUESTION NO: 14

Your team is using version 3.1.4 of a module from the public Terraform Registry, and they are worried about possible breaking changes in future versions of the module. Which version argument should you add to the module block to prevent newer versions from being used?

**A.** version = " < 3.2 "

**B.** version = " > = 3.1.5 "

**C.** version = " 3.1.4 "

**D.** version = " ~ > 3.1.4 "

**ANSWER: C**

**Explanation:**

`version = "3.1.4"` is the appropriate module version constraint when the requirement is to prevent Terraform from selecting any newer module release. A version constraint containing only a specific version is an exact-version constraint: Terraform must install precisely version 3.1.4 of the module. This makes module selection deterministic across runs and protects the configuration from behavior changes introduced by later patch, minor, or major releases. The constraint belongs in the module block alongside the module `source` argument, for example: `module "example" { source = "namespace/name/provider" version = "3.1.4" }`. Terraform records the selected module version in the dependency lock file when applicable, but declaring the exact constraint in configuration communicates and enforces the intended compatibility boundary directly. If the team later validates a newer release, it can deliberately update the version constraint and rerun initialization to adopt that release. HashiCorp documents module version constraints and their placement in module blocks in the module block syntax reference. The exact-version constraint behavior follows Terraform's general version constraints specification.

## QUESTION NO: 15

Which statements about sensitive output values are true? (Choose two.)

**A.** They are redacted in normal CLI output.

**B.** They can still be stored in Terraform state.

**C.** They are automatically encrypted by the Terraform CLI.

**D.** They cannot be referenced by a parent module.

**E.** They prevent the provider from receiving the value.

**ANSWER: A B**

**Explanation:**

Setting `sensitive = true` on an output value causes Terraform to redact that value in normal CLI output. This helps reduce accidental exposure when a plan or apply displays outputs.

Sensitivity does not encrypt the value or remove it from state. Terraform state can still contain sensitive values when they are needed to manage infrastructure, so state storage and access must be protected. Terraform also allows a sensitive output to be consumed by another module expression. See HashiCorp's documentation for sensitive output values and sensitive data in state.

## QUESTION NO: 16

You want to use API tokens and other secrets within your team ' s Terraform workspaces. Where does HashiCorp recommend you store these sensitive values? (Pick the 3 correct responses)

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

**B.** In HashiCorp Vault.

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

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

**E.** In a plaintext document on a shared drive.

**ANSWER: A B C**

**Explanation:**

In an HCP Terraform/Terraform Cloud variable, with the sensitive option checked; in HashiCorp Vault; and in a terraform.tfvars file, securely managed and shared with your team are appropriate ways to handle values such as API tokens when the corresponding security controls are applied. HCP Terraform sensitive variables are designed for workspace-specific secret input: their values are redacted in the user interface and are not displayed in run output. This lets teams provide credentials to Terraform runs without hard-coding them in configuration. HashiCorp Vault is purpose-built for centralized secret storage, access control, auditing, and, where supported, dynamic credentials; Terraform can retrieve secrets from Vault through its provider or integrations. A terraform.tfvars file can also supply sensitive input values when it is treated as a protected operational artifact, access is limited to authorized team members and automation, and it is managed outside ordinary source-controlled Terraform configuration. Terraform’s guidance specifically emphasizes not committing sensitive variable files to version control and using secure mechanisms to provide those values. See the HCP Terraform guidance on managing workspace variables and the Terraform documentation for variable definition files.

## QUESTION NO: 17

Which of these is true about Terraform ' s plugin-based architecture?

**A.** Terraform can only source providers from the internet

**B.** Every provider in a configuration has its own state file for its resources

**C.** You can create a provider for your API if none exists

**D.** All providers are part of the Terraform core binary

**ANSWER: C**

**Explanation:**

You can create a provider for your API if none exists. Terraform uses a plugin-based architecture in which providers are separate plugins that implement interactions with remote systems, including cloud platforms, SaaS products, and custom APIs. This extensibility means an organization is not limited to the providers already available in the public Terraform Registry. When a required API does not have an existing suitable provider, developers can implement one themselves and distribute it for their own use or publish it for others. HashiCorp supports provider development through the Terraform Plugin Framework, which is the recommended modern approach for building providers, as well as the Terraform Plugin SDK for existing and certain new development use cases. A custom provider defines resource and data source behavior, translates Terraform's requests into API calls, and returns the resulting state data for Terraform to manage. This provider model is a core benefit of Terraform's plugin architecture because it allows Terraform workflows and the Terraform language to be extended to services beyond those supported out of the box. See the Terraform plugin development documentation and the Terraform Plugin Framework documentation for implementation guidance.

## QUESTION NO: 18

Which of these statements about Terraform Cloud workspaces is false?

**A.** They have role-based access controls

**B.** You must use the CLI to switch between workspaces

**C.** Plans and applies can be triggered via version control system integrations

**D.** They can securely store cloud credentials

**ANSWER: B**

**Explanation:**

**You must use the CLI to switch between workspaces** is the false statement. Terraform Cloud workspaces are managed primarily through the Terraform Cloud web interface, where users can browse an organization's workspaces and open the desired workspace directly. They can also be managed and accessed programmatically through the Terraform Cloud API. Therefore, switching context does not require use of the Terraform CLI.

The CLI can be connected to Terraform Cloud using the `cloud` integration block, and it can select a workspace through configured workspace settings or a workspace name/prefix mapping. This is a supported workflow for local CLI-driven runs, but it is not an exclusive requirement for navigating or using Terraform Cloud workspaces. Terraform Cloud workspaces are independent managed environments that organize configuration, state, variables, runs, and permissions; users commonly interact with them in the web application, particularly for remote execution and VCS-driven workflows.

HashiCorp's workspace documentation describes workspaces as the core unit for managing infrastructure in Terraform Cloud and documents their web-based management capabilities. See Terraform Cloud workspaces and Terraform CLI cloud settings.

## QUESTION NO: 19

Which features do HCP Terraform workspaces provide that are not available in Terraform Community Edition? (Pick the 3 correct responses below.)

**A.** State versions and run history.

**B.** Automatic detection of common security issues.

**C.** Store Terraform and environment variables in variable sets.

**D.** Remote execution of Terraform operations.

**E.** Store your configuration in a Version Control System (VCS).

**F.** Support for multiple cloud providers.

**ANSWER: A C D**

**Explanation:**

**State versions and run history** are HCP Terraform workspace capabilities because HCP Terraform stores successive state versions and records the plan-and-apply run activity associated with a workspace. This provides a centrally available operational record and enables users to inspect previous state versions from the HCP Terraform interface or API. HashiCorp documents workspace state versioning at HCP Terraform state versions.

**Store Terraform and environment variables in variable sets** is also correct. Variable sets let an organization define Terraform input variables and environment variables once, then apply them to multiple workspaces. This supports consistent shared configuration and centralized management of values such as credentials, region settings, and organizational defaults. HCP Terraform documents this workspace-sharing mechanism at Managing HCP Terraform variable sets.

**Remote execution of Terraform operations** is a core HCP Terraform workflow feature. A workspace can use the remote execution mode, in which HCP Terraform performs Terraform plan and apply operations in its managed execution environment. This centralizes execution, captures run output, and associates operations with the workspace's remote state and run history.

## QUESTION NO: 20

What does Terraform not reference when running a terraform apply -refresh-only ?

**A.** State file

**B.** Credentials

**C.** Cloud provider

**D.** Terraform resource definitions in configuration files

**ANSWER: D**

**Explanation:**

`Terraform resource definitions in configuration files` is correct. Refresh-only mode is intended to reconcile Terraform's recorded state with the actual objects returned by the provider APIs, without proposing infrastructure changes based on the desired resource configuration. Terraform refreshes the remote-object data associated with resources already tracked in state and then presents any resulting state updates for approval. This is useful after out-of-band changes, such as a manually modified cloud resource, because it lets the state reflect the remote system without attempting to restore the prior declared configuration.

In this mode, Terraform uses the existing state as the basis for the refresh operation and communicates with the relevant provider using configured authentication so it can read the current remote resource values. The important distinction is that resource definitions are not evaluated as desired changes to apply: refresh-only planning deliberately disregards configuration changes and limits the operation to state and root-output updates derived from the remote system. HashiCorp documents this behavior in its Terraform planning modes documentation and in the terraform apply command reference.

## QUESTION NO: 21

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 declaratively tells Terraform to associate an object that already exists in a provider's remote system with a resource instance in Terraform state. The destination resource address of the block that will manage the database is required as the `to` argument. This address identifies the exact managed resource instance in the configuration that Terraform should link to the existing database, including any module path or instance key when applicable.

The ID associated with the current database on the cloud provider is also required as the `id` argument. This value must use the import identifier format implemented by the specific provider and resource type. For example, the required value might be a database identifier, a full resource path, or a composite identifier. Terraform passes that identifier to the provider so it can locate the remote database and read its current attributes into state.

After defining the import block, running `terraform plan` shows the proposed import and any configuration changes needed to align the resource configuration with the imported object. HashiCorp documents import blocks and their required `to` and `id` arguments at Terraform import blocks. Provider-specific identifier formats are documented in each resource's import section, as described in Terraform import.

**QUESTION NO: 22**

You add a new provider to your configuration and immediately run terraform apply in the CD using the local backend. Why does the apply fail?

**A.** The Terraform CD needs you to log into Terraform Cloud first

**B.** Terraform requires you to manually run terraform plan first

**C.** Terraform needs to install the necessary plugins first

**D.** Terraform needs you to format your code according to best practices first

**ANSWER: C**

**Explanation:**

Terraform needs to install the necessary plugins first. Providers are distributed as plugins, and Terraform must initialize the working directory to download the provider packages required by the configuration. Running `terraform init` performs this initialization: it installs required provider plugins, records selected provider versions and checksums in the dependency lock file when applicable, and prepares the current directory for Terraform operations. With a local backend, initialization also prepares the local working directory and local state handling, but it does not eliminate the provider-installation requirement. Therefore, when a new provider is added, the CI/CD job must run `terraform init` before `terraform apply` so the newly required provider is available. A typical automated workflow runs initialization non-interactively, such as `terraform init -input=false`, followed by planning and applying as appropriate for the pipeline. Terraform can generate a plan as part of `terraform apply`; a separately executed plan is not inherently required for provider installation. See HashiCorp's terraform init command documentation and its provider requirements documentation.

**QUESTION NO: 23**

Exhibit:

module " web_stack " {

source = " ./modules/web_stack "

}

Your configuration defines the module block shown in the exhibit. The web_stack module accepts an input variable named servers. Which of the following changes to the module block sets the servers variable to the value of 3?

**A.** module " web_stack " {source = " ./modules/web_stack " var.servers = 3}

**B.** module " web_stack " {source = " ./modules/web_stack " inputs = { servers = 3 }}

**C.** module " web_stack " {source = " ./modules/web_stack " servers = 3}

**D.** module " web_stack " {source = " ./modules/web_stack " inputs.servers = 3}

**ANSWER: C**

**Explanation:**

The module argument `servers = 3` correctly supplies the value `3` to the child module's input variable named `servers`. In Terraform, a module call uses argument names directly to assign values to the input variables declared by the referenced child module. Therefore, when the `web_stack` module declares a variable block such as `variable "servers" { ... }`, placing `servers = 3` inside the calling `module "web_stack"` block passes that numeric value into the module for use in its resources, expressions, and outputs.

The `source` argument continues to identify the local module directory, while `servers` is a separate module input argument. Terraform evaluates module input values in the context of the calling (parent) module and makes the resulting value available inside the child module through `var.servers`. This direct argument syntax is the standard Terraform mechanism for configuring reusable modules. For details, see the HashiCorp documentation on module block syntax and input variables.

**QUESTION NO: 24**

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

**A.** Defined in Environment variables

**B.** Inside the backend block within the Terraform configuration

**C.** Defined in a connection configuration outside of Terraform

**D.** None of above

**ANSWER: A**

**Explanation:**

**Defined in Environment variables** is the most secure choice for supplying credentials used by a Terraform remote backend. Terraform backend configuration can require sensitive values such as cloud access keys, tokens, or client secrets. HashiCorp recommends providing these values through environment variables (or other external credential mechanisms supported by the backend) rather than placing them directly in Terraform configuration.

This approach keeps credentials out of version-controlled `.tf` files and avoids embedding secrets in backend configuration. It also supports established secret-management and CI/CD practices: a local shell, pipeline runner, or workload identity can inject credentials only for the duration of a Terraform operation. For example, cloud-provider SDK environment variables can be used by compatible backend implementations, while Terraform Cloud authentication can use the `TF_TOKEN_app_terraform_io` environment variable.

HashiCorp specifically cautions that sensitive backend settings may be stored in the local `.terraform` directory and included in plan files if supplied directly as backend configuration. Using environment variables minimizes this exposure while allowing Terraform to authenticate during `terraform init` and subsequent backend operations. See the Terraform backend credentials and sensitive data documentation and the Terraform CLI environment-variable credentials documentation.

## QUESTION NO: 25

What are some benefits of using Sentinel with Terraform Cloud/Terraform Cloud? Choose three correct answers.

**A.** You can restrict specific resource configurations, such as disallowing the use of CIDR=0.0.0.0/0.

**B.** You can check out and check in cloud access keys

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

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

**E.** You can enforce a list of approved AWS AMIs

## ANSWER: A D E

**Explanation:**

Sentinel provides policy as code for Terraform Cloud, enabling organizations to evaluate policy rules against Terraform run data before infrastructure changes are applied. “You can restrict specific resource configurations, such as disallowing the use of CIDR=0.0.0.0/0.” is a practical governance benefit: policies can inspect planned resources and prevent deployments that violate an organization’s network-security requirements. This lets teams consistently block unsafe configurations at the point where infrastructure is provisioned.

“Policy-as-code can enforce security best practices” is also a core Sentinel use case. Teams can version, review, test, and centrally enforce organization-wide controls, including requirements relating to tagging, encryption, approved regions, or resource settings. Depending on the enforcement level selected, a failing policy can stop a run or require an authorized override.

“You can enforce a list of approved AWS AMIs” is a specific example of this same capability. A policy can examine an AWS instance’s planned AMI identifier and require it to match an approved allowlist, helping ensure workloads use vetted and supported base images. Terraform Cloud’s policy checks are designed to apply these controls consistently across workspaces and runs. See the Terraform Cloud policy enforcement documentation and the Sentinel introduction.

## QUESTION NO: 26

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 for managing Terraform workspaces and reviewing infrastructure automation. Through the UI, teams can configure workspace settings, connect version control repositories, inspect plans and applies, review run history, manage variables, and collaborate on runs. This centralized interface is a core part of the HCP Terraform workflow and makes operational information accessible without requiring every user to interact solely through the Terraform CLI.

Remote state storage is also a core HCP Terraform capability. Each workspace has managed remote state, allowing Terraform runs and collaborating users to work from a consistent record of deployed infrastructure. HCP Terraform securely stores state versions and coordinates access through its remote backend and workspace model. This supports collaboration by ensuring that the current state is available to authorized users and automation rather than relying on a state file stored on an individual machine. HashiCorp documents the platform’s workspace-based state management and remote operations in

its HCP Terraform workspaces documentation and explains how the remote backend integrates Terraform CLI workflows with HCP Terraform.

## QUESTION NO: 27

Which statements about the Terraform dependency lock file are true? (Choose two.)

**A.** It records selected provider versions.

**B.** It includes checksums for provider packages.

**C.** It stores the current resource state.

**D.** It replaces provider version constraints in required_providers.

**E.** It should never be committed to version control.

## ANSWER: A B

**Explanation:**

The dependency lock file, normally named `.terraform.lock.hcl`, records the provider versions Terraform selected for a configuration. It also includes checksums for the selected provider packages, allowing Terraform to verify that installed packages match the expected provider distributions.

The lock file is intended to be committed to version control for most shared configurations. This helps team members and automation use the same provider selections. It does not store Terraform state and does not replace the `required_providers` constraints in configuration. See HashiCorp's documentation on dependency lock files.

## QUESTION NO: 28

Which methods can provide a value for a Terraform input variable named `region`? (Choose two.)

**A.** A terraform.tfvars file

**B.** The -var command-line option

**C.** An output block named region

**D.** A provider block named region

**E.** A resource tag named TF_VAR_region

## ANSWER: A B

**Explanation:**

An input variable can be assigned using a variable definition file, such as `terraform.tfvars` or a file passed with `-var-file`. It can also be assigned using the `-var` command-line option, such as `terraform plan -var='region=us-east-1'`.

Terraform also supports environment variables named using the `TF_VAR_` prefix, such as `TF_VAR_region`. A provider block does not supply an input-variable value by itself, and output blocks expose values rather than define inputs. See HashiCorp's documentation for assigning values to root module variables.

## QUESTION NO: 29

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 describe the desired infrastructure state. Teams can store these files in version-control systems, review changes through established code-review workflows, reuse configuration through modules, and consistently apply the same patterns across environments. This makes infrastructure changes more traceable, collaborative, and repeatable than point-and-click provisioning.

"Reduces risk of operator error" is also correct because Terraform automates the creation and modification of infrastructure from a declared configuration. Rather than requiring an operator to repeatedly perform potentially inconsistent manual GUI steps, Terraform generates an execution plan and applies the intended changes in a consistent workflow. The plan enables review before changes are made, while the configuration provides a durable record of the intended infrastructure. HashiCorp identifies automation, repeatability, versioning, and collaboration as key benefits of managing infrastructure with Terraform. See the Terraform introduction and HashiCorp's Build infrastructure tutorial.

**QUESTION NO: 30**

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 recommends using secret-management mechanisms that keep sensitive values out of Terraform configuration and avoid exposing them in normal command output. HashiCorp Vault is designed to centrally store, control access to, and dynamically generate sensitive credentials. Terraform can retrieve values from Vault through its provider, allowing credentials to be managed separately from infrastructure source code. See the HashiCorp Vault documentation.

Environment variables named with the `TF_VAR_` prefix are a supported way to supply Terraform input variables at runtime. This is useful for local or automated execution when the environment is securely managed and the secret is not embedded in configuration files. Terraform documents this variable-definition mechanism in its environment variables reference.

An HCP Terraform variable marked sensitive is an appropriate workspace-specific location for secret input values. HCP Terraform stores sensitive variable values securely and redacts them from the user interface and run output. This enables a workspace to receive needed credentials or tokens while limiting their visibility to authorized users and processes.

## QUESTION NO: 31

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 allow Terraform to integrate with external systems and services. They define the resource types and data sources available in a configuration, and they contain the implementation needed to perform lifecycle operations against the relevant service interfaces. Consequently, *Provision resources for public cloud infrastructure services.* is correct: providers such as AWS, AzureRM, and Google Cloud expose resources that Terraform can create, read, update, and delete in those platforms. *Provision resources for on-premises infrastructure services.* is also correct because Terraform providers support many non-public-cloud systems, including infrastructure and platform services that may run in private environments. Finally, *Interact with cloud provider APIs.* is correct because API interaction is a central provider responsibility; providers translate Terraform's desired resource state into requests to the remote service and use returned information to refresh Terraform state. Although the wording refers specifically to cloud-provider APIs, this describes the same provider integration model used for other service APIs as well. Terraform configurations declare which providers are required, while provider plugins supply the service-specific behavior. See the official Terraform provider documentation and Terraform Plugin SDK documentation for details.

## QUESTION NO: 32

You run `terraform apply` after changing only an output value in the root module. No managed resources require changes.

What can Terraform update?

**A.** The output values stored in state

**B.** Only provider configuration

**C.** All resources in the configuration

**D.** Only the dependency lock file

## ANSWER: A

**Explanation:**

Terraform can update the output values stored in state without changing managed infrastructure. Output values are part of Terraform's recorded state and can change when their expressions change, even if the provider-managed resources remain unchanged.

For example, changing an output expression from one resource attribute to another can cause Terraform to show an output update during planning. Applying that plan records the new output value in state. Terraform does not need to call a provider to create, modify, or delete resources solely because an output expression changed.

See the Terraform output values documentation and the Terraform state documentation.

## QUESTION NO: 33

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

**A.** Using the destroy command with auto-approve.

**B.** Passing -destroy to terraform plan.

**C.** Running terraform destroy from the correct directory and then typing yes when prompted in the CLI.

## ANSWER: B

**Explanation:**

Passing `-destroy` to `terraform plan` is not, by itself, a way to trigger a destroy operation. This flag changes the planning mode so Terraform calculates and displays a plan that would destroy all managed infrastructure. It is useful for reviewing the expected deletions, assessing dependencies, and optionally saving the resulting plan file for later use. However, planning does not modify remote infrastructure or state-managed resources. Terraform only performs the proposed deletions when an apply operation executes that destroy plan, such as `terraform apply tfplan` after creating it with `terraform plan -destroy -out=tfplan`. HashiCorp documents `-destroy` as a planning-mode option and notes that it creates a destroy plan rather than carrying out the destruction itself. This distinction between generating a proposed change and applying it is fundamental to Terraform's workflow and helps teams review destructive actions before resources are removed. See the Terraform plan command documentation and the Terraform destroy command documentation.

## QUESTION NO: 34

Which actions can `terraform state rm` perform? (Choose two.)

**A.** Remove a resource instance from Terraform state.

**B.** Stop Terraform from tracking a remote object at that address.

**C.** Destroy the remote object through its provider.

**D.** Import an existing remote object into state.

## ANSWER: A B

**Explanation:**

`terraform state rm` removes one or more resource instances from Terraform state. After removal, Terraform no longer tracks the specified remote object at that resource address, but the command does not destroy the corresponding remote infrastructure.

Because the object remains in the provider environment, a subsequent `terraform plan` may propose creating a replacement if the configuration still declares that resource address. Practitioners commonly use this command when they intend to stop managing an object with Terraform or before associating it with a different address through an appropriate migration workflow.

The command does not delete objects from the cloud provider and does not add existing objects to state. See the terraform state rm command reference.

## QUESTION NO: 35

Terraform validate reports syntax check errors from which of the following scenarios?

**A.** Code contains tabs indentation instead of spaces

**B.** There is missing value for a variable

**C.** The state files does not match the current infrastructure

**D.** None of the above

**ANSWER: D**

**Explanation:**

**None of the above** is correct. The `terraform validate` command checks whether Terraform configuration is syntactically valid and internally consistent. Its validation occurs locally against the configuration files and provider schemas available to Terraform; it does not require Terraform to contact remote infrastructure or compare the current state file to real resources. Tabs used for indentation are valid whitespace in Terraform's HCL syntax, so their use is not itself a syntax error. Likewise, declaring an input variable without a default value is valid configuration: a value may be supplied later through a variable file, environment variable, command-line argument, or an interactive prompt during commands that need the value. Validation does not treat the absence of such a runtime input value as a configuration syntax error. A mismatch between state and deployed infrastructure is also outside the scope of local configuration validation. Therefore, none of the listed scenarios represents a syntax-check error reported by `terraform validate`. See the HashiCorp documentation for terraform validate and the input variables documentation.

**QUESTION NO: 36**

HashiCorp Configuration Language (HCL) supports user-denned functions.

**A.** True

**B.** False

**ANSWER: B**

**Explanation:**

**False** is correct. In Terraform configurations, HCL expressions can call Terraform's built-in functions, such as `max()`, `merge()`, `length()`, and `try()`, but configuration authors cannot declare their own reusable functions with parameters and a function body. Function calls use a function name followed by comma-separated arguments in parentheses, for example `max(5, 12, 9)`. Terraform's available functions are defined by the Terraform language and documented as built-in functions.

When a configuration needs reusable logic, Terraform provides other language constructs suited to that purpose. A `locals` block can assign a name to a computed expression so it can be referenced repeatedly within a module. Modules can package and reuse groups of resources and configuration logic, with input variables providing parameterization. These mechanisms support maintainable reuse, but neither creates a user-defined HCL function. See the official Terraform documentation for built-in functions and guidance on local values.