PassQueen
a.k.a dumpsqueen.com

HashiCorp Certified: Vault Associate

HashiCorp VA-002-P

Version Demo

Total Demo Questions: 24

Total Premium Questions: 242

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Vault Architecture	25
Topic 2, Vault Concepts	8
Topic 3, Vault Policies	21
Topic 4, Vault Tokens	20
Topic 5, Vault Secrets Engines	34
Topic 6, Vault Authentication Methods	6
Topic 7, Vault Audit Devices	1
Topic 8, Vault Operations	27
Topic 9, Mix Questions	100
Total	242

QUESTION NO: 1

Which of the following secrets engine can generate dynamic credentials? (select three)

A. Azure

B. database

C. key/value

D. Transit

E. AWS

ANSWER: A B E

Explanation:

Azure, database, and AWS are Vault secrets engines that can generate dynamic credentials on demand. Rather than storing a long-lived credential for an application to retrieve, Vault authenticates to the target platform using configured administrative or privileged access and creates a credential specifically for the requesting client. The generated credential is issued with a Vault lease and a defined time-to-live, allowing Vault to revoke it automatically when the lease expires or explicitly when it is no longer needed. This approach reduces the operational risk of static, shared credentials and enables credential rotation without applications having to manage the lifecycle directly.

The AWS secrets engine can generate IAM credentials, including access keys or temporary security credentials, according to configured roles. The Azure secrets engine dynamically creates Azure service principals and assigns the configured roles. The database secrets engine creates database users and passwords based on database roles, with permissions and lifetimes controlled through Vault policies and role configuration. These are core examples of Vault's dynamic-secret workflow. HashiCorp documents the AWS, Azure, and database engines as dynamic-secret integrations in its secrets engines documentation; database-specific credential generation is described in the database secrets engine documentation.

QUESTION NO: 2

As opposed to service tokens, batch tokens are ideal for what type of action?

A. generating dynamic credentials

B. configuring Vault features

C. renewing tokens

D. issuing snapshots

E. encrypting data

F. writing secrets

ANSWER: E

Explanation:

encrypting data is correct because Vault batch tokens are designed for high-volume, performance-sensitive use cases where a client needs a lightweight token to make authorized API requests. Unlike service tokens, batch tokens are not persisted in Vault's storage and do not require a storage lookup for every validation. Instead, they contain the information Vault needs to validate them cryptographically. This substantially reduces storage and replication overhead, which makes batch tokens well suited to workloads that perform many short-lived, repeated operations.

A common example is an application using Vault's Transit secrets engine to encrypt or decrypt data at scale. The application can authenticate and receive a batch token with the narrowly scoped policy needed to call the relevant Transit endpoint. Vault can then process those frequent cryptographic requests efficiently while still enforcing the token's policies, TTL, and other embedded restrictions. Batch tokens therefore support scalable encryption workflows while preserving Vault's access-

control model. HashiCorp specifically identifies high-volume operations, such as encrypting data, as an appropriate use for batch tokens. See the Vault token concepts documentation and the Transit secrets engine documentation.

QUESTION NO: 3

Which of the following settings are configured using the configuration file? (select three)

A. Cluster Name

B. Replication

C. Seal Type

D. Auth Methods

E. Namespaces

F. Storage Backend

G. Audit Devices

ANSWER: A C F

Explanation:

Vault's server configuration file defines foundational settings that must be known when the Vault server process starts. **Cluster Name** is configured with the `cluster_name` parameter, which assigns a human-readable name to the Vault cluster. This name is used to help identify the cluster in operational contexts, including replication-related status and monitoring output.

Seal Type is configured through a `seal` stanza in the server configuration. The seal stanza determines how Vault's master key is protected and accessed, such as by using a cloud key-management service or Hardware Security Module for auto-unseal. Because this affects Vault initialization and unsealing behavior, it is a server-startup configuration.

Storage Backend is configured in a `storage` stanza. The selected backend provides Vault's durable physical storage for encrypted data, including secrets, policies, and core cluster state. Examples include integrated storage (Raft), Consul, and cloud storage implementations. Vault requires this storage configuration before it can run and persist data. HashiCorp documents these server-side settings in the Vault configuration reference, including the storage stanza documentation.

QUESTION NO: 4

In regards to using a K/V v2 secrets engine, select the three correct statements below: (select three)

A. issuing a vault kv destroy statement permanently deletes a single version of a secret

B. issuing a vault kv destroy statement deletes all versions of a secret

C. issuing a vault kv delete statement permanently deletes the secret

D. issuing a vault kv metadata delete statement permanently deletes the secret

E. issuing a vault kv delete statement performs a soft delete

ANSWER: A D E

Explanation:

For a KV version 2 secrets engine, `vault kv delete` performs a soft delete of the current version by marking that version as deleted. The version's underlying data is retained, enabling an authorized user to recover it with `vault kv undelete`. This supports version-aware lifecycle management and protects against accidental removal.

`vault kv destroy` permanently removes the data for explicitly specified secret version numbers. A destroyed version cannot be recovered through the KV recovery commands, while other versions of the same key can remain available. This makes destruction appropriate when a particular historical version must be irreversibly removed.

`vault kv metadata delete` removes the key's metadata and all versioned data associated with that key. Therefore, it permanently deletes the secret as a whole rather than only marking one version deleted. Vault documents these separate operations to distinguish recoverable deletion, permanent removal of selected versions, and complete removal of a versioned key. See the Vault KV delete command documentation and the Vault KV metadata command documentation.

QUESTION NO: 5

When configuring a remote backend in Terraform, it might be a good idea to purposely omit some of the required arguments to ensure secrets and other relevant data are not inadvertently shared with others. What are the ways the remaining configuration can be added to Terraform so it can initialize and communicate with the backend? (select three)

A. directly querying HashiCorp Vault for the secrets

B. command-line key/value pairs

C. use the -backend-config=PATH to specify a separate config file

D. interactively on the command line

ANSWER: B C D

Explanation:

Terraform calls a backend definition with omitted required settings a partial configuration. This pattern lets teams keep sensitive backend values, such as credentials or access keys, out of version-controlled Terraform configuration while still supplying them when Terraform initializes the working directory. Terraform supports completing that configuration through command-line key/value pairs, such as `-backend-config="key=value"`, passed to `terraform init`. This is useful for automation, although sensitive values passed on a command line can be exposed through shell history or process inspection.

A separate configuration file can also be supplied with `-backend-config=PATH`. The file contains backend argument assignments and may be generated or retrieved securely by a deployment process before initialization. Finally, when interactive input is enabled, Terraform prompts for omitted required backend arguments during `terraform init`. This can be appropriate for local, manually run workflows where the needed values can be entered at the prompt. These are the documented mechanisms for completing a partial backend configuration. See HashiCorp's partial backend configuration documentation and the terraform init command reference.

QUESTION NO: 6

In regards to deploying resources in multi-cloud environments, what are some of the benefits of using Terraform rather than a provider's native tooling? (select three)

A. Terraform simplifies management and orchestration, helping operators build large-scale, multi-cloud infrastructure

B. Terraform can help businesses deploy applications on multiple clouds and on-premises infrastructure

C. Terraform can manage cross-cloud dependencies

D. Terraform is not cloud-agnostic and can be used to deploy resources across a single public cloud

ANSWER: A B C

Explanation:

Terraform simplifies management and orchestration, helping operators build large-scale, multi-cloud infrastructure because its workflow uses a consistent declarative configuration language and execution model across providers. Teams can define,

review, version, and apply infrastructure changes through one common tool rather than adopting a separate native provisioning workflow for every platform. HashiCorp describes Terraform as supporting infrastructure across cloud providers, SaaS providers, and on-premises environments through providers.

Terraform can help businesses deploy applications on multiple clouds and on-premises infrastructure because a single Terraform configuration can use multiple providers. This enables a common infrastructure-as-code practice for resources hosted in different public clouds, private environments, and supporting services.

Terraform can manage cross-cloud dependencies through its dependency graph. When one resource references an attribute produced by another resource, Terraform derives the required ordering and creates, updates, or destroys resources in a dependency-aware sequence. This applies even when dependent resources are managed through different providers, which is especially useful for coordinated multi-cloud deployments.

For further detail, see HashiCorp's Terraform introduction and its documentation on the depends_on meta-argument and dependency handling.

QUESTION NO: 7

The command vault lease revoke -prefix aws/ will revoke all leases associated with the secret engine mounted at aws/

A. False

B. True

ANSWER: B

Explanation:

`vault lease revoke -prefix aws/` is correct because the `-prefix` flag revokes every lease whose lease ID begins with the supplied path prefix. For a secrets engine mounted at `aws/`, dynamically generated credentials normally receive lease IDs beneath that mount path, such as paths generated by AWS roles under `aws/creds/...`. Supplying `aws/` therefore targets the full hierarchy of leases created through that mount and causes Vault to revoke them. Revocation is significant for dynamic secrets: Vault performs the configured revocation action for each lease, allowing the AWS secrets engine to clean up the corresponding generated IAM credentials where applicable. This is useful when retiring a mount, responding to an incident, or deliberately invalidating all active dynamic credentials issued from that engine. The command operates on leases identified by their path prefix; the trailing slash precisely scopes the request to the `aws/` mount hierarchy. HashiCorp documents that `vault lease revoke` supports prefix-based revocation and that the AWS secrets engine generates dynamic AWS credentials with leases. See the Vault lease revoke command documentation and the AWS secrets engine documentation.

QUESTION NO: 8

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

1. resource "aws_instance" "db" {

2. ami = "ami-123456"

3. instance_type = "t2.micro"

4. }

A. "db"

B. resource

C. "aws_instance"

D. instance_type

ANSWER: B

Explanation:

The correct answer is **resource**. Terraform configuration uses the HashiCorp Configuration Language (HCL), where a block begins with a block type followed by zero or more labels and then a body enclosed in braces. In the declaration `resource "aws_instance" "db" { ... }`, `resource` is therefore the block type. It tells Terraform that the block declares an infrastructure resource to be managed. The following quoted strings, `"aws_instance"` and `"db"`, are labels that further identify the resource: the first specifies the resource type and the second provides the local name used to reference that resource elsewhere in the configuration. The lines inside the braces, such as `ami` and `instance_type`, are arguments that configure the declared resource. Terraform's language documentation describes this general structure as a block type followed by labels and a body, while the resource-block reference specifically shows `resource` as the keyword that begins a resource declaration. See the Terraform configuration syntax documentation and the resource block syntax reference.

QUESTION NO: 9

What are some of the features of Terraform state? (select three)

A. inspection of cloud resources

B. increased performance

C. mapping configuration to real-world resources

D. determining the correct order to destroy resources

ANSWER: B C D

Explanation:

Terraform state provides a persistent record that connects resource addresses in Terraform configuration with the actual remote objects that Terraform manages. Therefore, *mapping configuration to real-world resources* is a core purpose of state: it lets Terraform know, for example, which remote object corresponds to a particular resource block and instance address.

Increased performance is also a benefit because state acts as a local cache of remote object attributes. Without state, Terraform would need to query provider APIs for every managed object during planning and other operations. Using the state snapshot reduces the amount of remote API discovery required, which is especially valuable for larger infrastructures.

Determining the correct order to destroy resources is supported by state because Terraform records metadata about managed resources, including dependency information. This enables Terraform to retain knowledge needed to sequence operations safely when configuration has changed or resources are being removed. State is therefore not merely an output artifact; it is a fundamental component Terraform uses to reconcile configuration with infrastructure and to plan lifecycle actions. HashiCorp documents these roles in its Purpose of Terraform State guide and further describes state behavior in the Terraform State documentation.

QUESTION NO: 10

An application is trying to use a secret in which the lease has expired. What can be done in order for the application to successfully request data from Vault?

A. request a new secret and associated lease

B. try the expired secret in hopes it hasn't been deleted yet

C. request the TTL be extended for the secret

D. perform a lease renewal

ANSWER: A

Explanation:

“request a new secret and associated lease” is correct because Vault leases are time-bound grants associated with dynamically generated secrets or other leased data. When a lease reaches its expiration time without being renewed, Vault automatically revokes it. Revocation invalidates the associated credentials where the secrets engine supports revocation, so the application can no longer rely on that secret to authenticate to or access the downstream system.

Expiration is final for that lease: renewal must occur while the lease is still valid. To resume access after expiration, the application must make a new request to the appropriate Vault secrets engine endpoint. Vault then generates or returns a new secret, along with a new lease ID and lease duration when the returned secret is renewable or otherwise leased. Applications should normally renew renewable leases before their TTL elapses, or use a Vault Agent or client-library renewal mechanism to manage this lifecycle automatically. HashiCorp documents that expired leases are automatically revoked and that a new secret must be requested after expiration. See the Vault lease concepts documentation and the lease renewal documentation.

QUESTION NO: 11

What happens when a terraform plan is executed?

A. the backend is initialized and the working directory is prepped

B. creates an execution plan and determines what changes are required to achieve the desired state in the configuration files.

C. applies the changes required in the target infrastructure in order to reach the desired configuration

D. reconciles the state Terraform knows about with the real-world infrastructure

ANSWER: B

Explanation:

The correct answer is “creates an execution plan and determines what changes are required to achieve the desired state in the configuration files.” The `terraform plan` command evaluates the current Terraform configuration, the state Terraform has recorded, and—under its normal refresh behavior—the current condition of managed remote objects. It then proposes the actions Terraform would take to make the real infrastructure conform to the declared configuration. These proposed actions can include creating, updating, replacing, or destroying managed resources, but the plan operation itself is a preview and does not make those infrastructure changes.

This planning step is central to Terraform’s workflow because it gives operators an opportunity to inspect the intended result before authorizing execution. Terraform can also save a generated plan to a file, allowing that exact plan to be reviewed and later passed to `terraform apply`. HashiCorp documents that planning is an execution-preview operation: Terraform reads the configuration and current state, calculates the difference, and presents the changes necessary to reach the requested target state. See the official Terraform plan command documentation and the Terraform core workflow documentation.

QUESTION NO: 12

Which of the following storage backends are supported by HashiCorp technical support?

(select four)

A. Filesystem

B. Consul

C. In-Memory

D. Raft

E. DynamoDB

F. MySQL

ANSWER: A B C D

Explanation:

The supported selections are **Filesystem**, **Consul**, **In-Memory**, and **Raft**. In Vault terminology, Raft is Vault's integrated storage backend. HashiCorp documents integrated storage as a built-in, highly available storage option that uses the Raft consensus protocol, making it a primary supported operational choice for Vault deployments. Consul is also a long-standing supported Vault storage backend and can provide highly available storage when deployed appropriately.

Filesystem storage is a supported backend, generally suited to single-node or non-production-oriented use cases because it does not itself provide the distributed coordination required for a highly available Vault cluster. Likewise, the in-memory backend is supported for development and testing workflows: its data exists only in memory and is lost when the Vault process stops, so its support status does not make it appropriate for persistent production data.

The key distinction in the question is between backends implemented by Vault and those covered by HashiCorp technical support. Vault's storage documentation identifies the available backend types and describes the intended operational role of integrated storage, Consul, file, and in-memory storage. See the Vault storage backend documentation and the integrated storage (Raft) documentation.

QUESTION NO: 13

Which two characters can be used when writing a policy to reflect a wildcard or path segment? (select two)

A. @

B. $

C. &

D. *

E. +

ANSWER: D E

Explanation:

The correct policy-path characters are `*` and `+`. In Vault ACL policies, the glob character `*` provides wildcard matching. It is permitted only as the final character of a path and matches any remaining characters in that path. For example, a path such as `secret/data/team/*` can apply capabilities to matching paths below that prefix. This allows policy authors to grant access to an entire set of paths without enumerating every individual secret.

The plus character `+` is a segment wildcard. It matches exactly one path segment when it appears as an entire segment of the policy path. For example, `secret/+/team` matches a single segment between `secret` and `team`. Segment matching is useful where a policy needs to accommodate one variable component while retaining a defined path structure. Vault evaluates policy paths using these documented matching rules, so these characters support concise policies while retaining deliberate access boundaries. See the Vault policy glob-pattern documentation and the Vault policies overview.

QUESTION NO: 14

Which is not a benefit of running HashiCorp Vault in your environment?

A. Integrate with your code repository to pull secrets when deploying your applications

B. Consolidate static, long-lived passwords used throughout your organization

C. Act as root or intermediate certificate authority to automate the generation of PKI certificates

D. The ability to generate dynamic secrets for applications and resource access

ANSWER: A

Explanation:

Integrate with your code repository to pull secrets when deploying your applications is not, by itself, a core benefit or responsibility provided by Vault. Vault is a secrets-management platform: it securely stores and delivers secrets, encrypts data, issues dynamic credentials and certificates, and enforces authenticated, policy-based access to those resources. It does not function as a version-control system, inspect repositories, check out source code, or natively orchestrate an application deployment from a code repository. A deployment platform, CI/CD system, or application can authenticate to Vault and request the secrets it needs at deployment or runtime, but that integration is implemented by the deployment tooling rather than by Vault acting as a code-repository integration service. For example, Vault supports GitHub as an authentication method, allowing users to authenticate based on GitHub organization or team membership; this does not give Vault a source-code checkout or repository-reading role. Vault's documented workflow is for authenticated clients to read secrets or obtain generated credentials through its APIs and supported auth methods. See the Vault GitHub authentication documentation and the Vault overview.

QUESTION NO: 15

The userpass auth method has the ability to access external services in order to provide authentication to Vault.

A. FALSE

B. TRUE

ANSWER: A

Explanation:

FALSE is correct. Vault's userpass auth method authenticates a client by validating a username and password against credentials configured and stored within Vault's own storage backend. It is a locally managed authentication method: administrators create users at the auth mount and assign policies directly to those users. During a login request, userpass does not delegate credential validation to, query, or otherwise depend on an external identity provider or service. This makes it useful for straightforward Vault-local accounts, including initial administrative or break-glass access scenarios, but identity lifecycle and password management remain the responsibility of Vault administrators.

Vault supports other auth methods when authentication must be backed by an external system, such as LDAP, JWT/OIDC, cloud-provider identity services, or Kubernetes. Those methods are designed to validate identities using their respective external platforms. The defining behavior of userpass, however, is that the username/password data is configured locally at the userpass auth mount. HashiCorp's userpass API documentation describes creating users by writing a username and password to the mount, and the auth-method documentation characterizes userpass as username/password authentication managed in Vault. See the Vault userpass authentication documentation and the userpass API documentation.

QUESTION NO: 16

From the answers below, select the advantages of using Infrastructure as Code. (select four)

A. Easily integrate with application workflows (GitLab Actions, Azure DevOps, CI/CD tools)

B. Safely test modifications using a "dry run" before applying any actual changes

C. Provide reusable modules for easy sharing and collaboration

D. Easily change and update existing infrastructure

E. Provide a codified workflow to develop customer-facing applications

ANSWER: A B C D

Explanation:

Infrastructure as Code makes infrastructure definitions repeatable, reviewable, and automatable by expressing desired infrastructure configuration in version-controlled files. “Easily integrate with application workflows (GitLab Actions, Azure DevOps, CI/CD tools)” is an advantage because infrastructure changes can be incorporated into the same automated delivery processes used for software, including validation, review, and controlled deployment stages. “Safely test modifications using a "dry run" before applying any actual changes” reflects Terraform’s plan workflow: Terraform compares configuration with the current state and presents proposed actions before changes are made, supporting safer operational decisions.

“Provide reusable modules for easy sharing and collaboration” is also a core Infrastructure as Code benefit. Terraform modules package related resources behind a reusable interface, allowing teams to standardize common patterns and use them consistently across environments and projects. Finally, “Easily change and update existing infrastructure” is correct because a declarative configuration can be revised and reapplied; Terraform determines the actions required to move managed infrastructure toward the updated desired state. Together, these practices improve consistency, collaboration, and change control. HashiCorp describes these core Infrastructure as Code workflows in its Terraform introduction, with details on proposed changes in Terraform plan documentation.

QUESTION NO: 17

What is the purpose of using the local-exec provisioner? (select two)

A. ensures that the resource is only executed in the local infrastructure where Terraform is deployed

B. to execute one or more commands on the machine running Terraform

C. to invoke a local executable

D. executes a command on the resource to invoke an update to the Terraform state

ANSWER: B C

Explanation:

The local-exec provisioner is used to invoke a process on the machine where Terraform is running. Therefore, “to execute one or more commands on the machine running Terraform” accurately describes its execution context: Terraform starts the specified command locally, rather than connecting to and running it inside the resource being created. This can be useful for local integration tasks, such as writing generated connection details to a file, calling a local script, or notifying an external system after Terraform completes resource creation.

“to invoke a local executable” is also correct because this is Terraform’s direct definition of the provisioner. The `command` argument supplies the command or executable to run, and Terraform executes it after the associated resource is created by default. Terraform also provides configuration such as `working_dir`, `interpreter`, and environment variables to control that local process. Although provisioners should generally be used only when there is no better declarative alternative, local-exec is specifically intended for actions performed from the Terraform runner rather than on the managed infrastructure. See the official local-exec provisioner documentation and HashiCorp’s guidance on provisioner syntax and behavior.

QUESTION NO: 18

After a client has authenticated, what security feature is used to make subsequent calls?

A. key shard

B. ldap

C. pgp

D. token

E. listener

F. path

ANSWER: D

Explanation:

After a client successfully authenticates to Vault through any supported authentication method, Vault returns a client token. This token is the security feature presented with subsequent API, CLI, or UI requests to prove the client's authenticated identity. Vault evaluates the token's associated policies on every request to determine whether the requested operation—such as reading a secret, writing data, or listing paths—is authorized. Tokens can also carry important lifecycle and security properties, including a time-to-live, renewable status, use limits, explicit maximum TTL, and attachment to one or more policies. This model separates authentication, which establishes who or what the client is, from authorization, which determines what that authenticated client may do. Clients normally provide the token in the `X-Vault-Token` HTTP header, while the Vault CLI manages it through its configured token helper or the `VAULT_TOKEN` environment variable. Therefore, **token** is the correct answer because it is the credential Vault issues after authentication and requires for subsequent authenticated calls. See the official Vault token concepts documentation and the Vault authentication documentation.

QUESTION NO: 19

While Terraform is generally written using the HashiCorp Configuration Language (HCL), what another syntax can Terraform be expressed in?

A. JSON

B. XML

C. TypeScript

D. YAML

ANSWER: A

Explanation:

JSON is correct because Terraform supports a JSON-based variant of its native configuration language in addition to the conventional HCL syntax. A configuration file written in this syntax normally uses the `.tf.json` filename extension, whereas standard HCL configuration uses `.tf`. The JSON representation expresses the same Terraform language constructs: resources, data sources, variables, outputs, modules, provider requirements, expressions, and nested configuration blocks. Terraform parses these files as Terraform configuration, so JSON files can be used alongside HCL files in the same working directory when appropriate.

The main reason to select JSON is interoperability with software. JSON is widely supported by programming languages, libraries, templates, and serialization tools, making it practical when configuration is generated programmatically rather than primarily written and maintained by people. HashiCorp notes that the JSON syntax is intentionally more verbose and less convenient for direct human authoring than native HCL, but it remains a fully supported representation of Terraform configuration. Terraform users should therefore recognize JSON as an alternative syntax, not merely a data format used by external tooling. See the official Terraform JSON configuration syntax documentation and the Terraform configuration syntax reference.

QUESTION NO: 20

After executing a terraform apply, you notice that a resource has a tilde (~) next to it. What does this infer?

A. the resource will be destroyed and recreated

B. the resource will be created

C. Terraform can't determine how to proceed due to a problem with the state file

D. the resource will be updated in place

ANSWER: D

Explanation:

The resource will be updated in place. In Terraform's plan and apply output, the tilde (~) is the action symbol for an in-place update. Terraform has detected that one or more configured attribute values differ from the values currently recorded for the existing managed object, and the provider supports applying those particular changes without replacing the resource. During `terraform apply`, Terraform sends the required update operation to the provider and then refreshes the state with the resulting resource values.

This distinction matters because an in-place update preserves the resource's identity: Terraform does not first delete it and then create a replacement. The exact operational impact still depends on the provider and changed attribute; for example, an update might briefly affect service behavior even though the resource is retained. Terraform's execution-plan documentation defines ~ as an update to an existing resource, while a delete-and-recreate operation is represented differently. Reviewing the plan before approval lets an operator identify these changes and confirm that the intended existing infrastructure will be modified. See HashiCorp's Terraform plan command documentation and its Terraform apply command documentation.

QUESTION NO: 21

What are some of the problems of how infrastructure was traditionally managed before

Infrastructure as Code? (select three)

A. Requests for infrastructure or hardware required a ticket, increasing the time required to deploy applications

B. Traditional deployment methods are not able to meet the demands of the modern business where resources tend to live days to weeks, rather than months to years

C. Traditionally managed infrastructure can't keep up with cyclic or elastic applications

D. Pointing and clicking in a management console is a scalable approach and reduces human error as businesses are moving to a multi-cloud deployment model

ANSWER: A B C

Explanation:

Requests for infrastructure or hardware required a ticket, increasing the time required to deploy applications is correct because traditional provisioning commonly depended on manual service requests and handoffs between teams. That workflow introduces queues and delays that are incompatible with teams that need to create, modify, and remove environments quickly and repeatedly. Infrastructure as code instead lets teams define infrastructure in version-controlled configuration and provision it through automated workflows.

Traditional deployment methods are not able to meet the demands of the modern business where resources tend to live days to weeks, rather than months to years is also correct. Modern cloud workloads can be short-lived and frequently changed, so the operational model must support rapid, consistent lifecycle management rather than processes designed around long-lived, relatively static servers.

Traditionally managed infrastructure can't keep up with cyclic or elastic applications is correct because elastic systems require infrastructure capacity to scale in response to changing demand. Declarative infrastructure tooling enables repeatable creation and updates through APIs, making it practical to manage dynamic cloud resources at scale. HashiCorp describes infrastructure as code as defining and provisioning infrastructure using configuration files, while Terraform uses providers to interact with APIs for cloud and other services. See Terraform introduction and Terraform configuration language documentation.

QUESTION NO: 22

What is a downside to using a Terraform provider, such as the Vault provider, to interact with sensitive data, such as reading secrets from Vault?

A. Terraform and Vault must be running on the same physical host

B. Terraform and Vault must be running on the same version

C. Terraform requires a unique auth method to work with Vault

D. Secrets are persisted to the state file and plans

ANSWER: D

Explanation:

Secrets are persisted to the state file and plans. When Terraform reads a secret through the Vault provider or writes a secret value as part of managed infrastructure, Terraform may retain that value in its state so it can track resource and data-source values across runs. Sensitive values can also be present in generated plan files. Marking an argument or output as sensitive reduces its display in normal CLI output, but it does not prevent Terraform from storing the value in state. Therefore, state files and saved plan files associated with configurations that handle Vault secrets must be treated as highly sensitive artifacts. Store remote state in a secured backend, restrict access through least-privilege controls, encrypt storage where appropriate, and avoid sharing plan files or exposing them in logs and build artifacts. HashiCorp specifically notes that using Vault with Terraform causes secrets read or written by Terraform to be persisted in Terraform state and any generated plan files. See the Vault documentation for general Vault deployment guidance and HashiCorp's Terraform sensitive-data documentation for details on state and plan handling.

QUESTION NO: 23

Complete the following sentence:

The terraform state command can be used to ____

A. view the entire state file

B. modify the current state, such as removing items

C. refresh the existing state

D. there is no such command

ANSWER: B

Explanation:

The terraform state command can be used to modify the current state, such as removing items. It is Terraform's command group for advanced state management operations, allowing operators to work with resource-address bindings stored in the state without manually editing the state file. For example, `terraform state rm` removes the binding for a specified resource from state. This tells Terraform to stop tracking that existing remote object while leaving the actual infrastructure object untouched. On a later plan, Terraform can propose creating a replacement if that resource remains in the configuration, or the object can be brought back under management with `terraform import` when appropriate.

State subcommands are especially useful during refactoring and operational migrations, such as moving a resource address, removing an obsolete binding, or replacing provider references. Because state records Terraform's mapping between configuration resource addresses and real infrastructure objects, these operations should be performed carefully, preferably with remote-state locking and a reviewed backup process. HashiCorp documents the `terraform state` command as the interface for advanced state management and provides dedicated documentation for state removal behavior. See the Terraform state command documentation and the terraform state rm reference.

QUESTION NO: 24

Which configuration constraints can be set on a Vault PKI role? (select three)

A. allowed_domains

B. allow_subdomains

C. max_ttl

D. storage backend type

E. unseal key threshold

ANSWER: A B C

Explanation:

A PKI role defines the certificate issuance constraints that Vault applies when clients request certificates. `allowed_domains` restricts the domain names for which the role can issue certificates. `allow_subdomains` controls whether names below an allowed domain are permitted. `max_ttl` limits the maximum lifetime of certificates issued through the role.

These controls help ensure that a client can request only certificates appropriate for its workload and only for an approved duration. A PKI role does not configure the Vault storage backend or determine the number of unseal key shares. See the Vault PKI secrets engine documentation and the PKI role API documentation.