



HashiCorp Vault

Comprehensive guide



By DevOps Shack

- ◆ **1. Introduction to HashiCorp Vault**

What is Vault?

HashiCorp Vault is a powerful, flexible, identity-aware secrets management and data protection system. It enables secure storage and access to sensitive information such as:

- **API keys**
- **Passwords**
- **Tokens**
- **Certificates**
- **Encryption keys**

Vault acts as a **central secrets management hub**, offering a consistent API for securely accessing, storing, and distributing secrets — regardless of whether they are static or dynamic. Vault is identity-aware, logs every action, and can provide fine-grained access control down to the specific secret or path level.

Why Vault Exists

In modern cloud-native environments and DevOps pipelines, secrets like database passwords, API tokens, and SSH keys are often:

- Hardcoded in configuration files or scripts
- Shared across environments or teams
- Stored in plain text
- Difficult to rotate without downtime

These practices introduce significant **security risks**, such as data leaks, unauthorized access, and breaches. Vault addresses these challenges by offering:

Centralization

Instead of secrets scattered in configs, Vault centralizes them in an encrypted, access-controlled vault.

Fine-Grained Access Control

Vault policies define exactly **who** can access **what**, and **how**.

Secret Lifecycle Management

Secrets can be automatically **rotated**, **revoked**, and **leased** with defined TTL (time-to-live) to minimize exposure.

🔧 Real-World Example

A company uses multiple microservices that all connect to a PostgreSQL database. Traditionally, every service might use the **same hardcoded password**, posing a risk.

With Vault:

- Each microservice gets **dynamic database credentials**
- These credentials have a **short TTL**
- If compromised, they **expire quickly**
- Access is **audited** and **policy-controlled**

 Result: No static passwords, automatic revocation, centralized control, and complete audit logs.

◆ 2. Why Use a Secrets Manager?

🔑 Password Manager vs. Secrets Manager

Password Manager	Secrets Manager
Meant for human use	Meant for machine-to-machine
Manual copy-paste usage	Programmatic access via API
Browser extensions	CLI, SDK, or HTTP API
No versioning or TTL	Supports TTL, revocation

Secrets managers like Vault are specifically designed to **automate secrets management** in dynamic, cloud-native, and microservice-based environments.

Core Benefits of a Secrets Manager

1. Centralized Management

- All secrets stored securely in one place.
- Easy management of access and auditability.

2. Encryption at Rest and Transit

- All secrets are encrypted using strong algorithms (AES-256).
- Vault ensures secure handling even in memory.

3. Fine-Grained Access Control

- Policies define access per role, path, or method.
- Example: A service can read only its DB password, not others.

4. Dynamic Secrets

- Secrets are created on-demand.
- Example: Vault generates a temporary AWS IAM key that expires in 1 hour.

5. Lease and Revocation

- All secrets have TTL.
 - Vault can revoke any secret instantly if compromise is suspected.
-



Real-World Scenario

Imagine a DevOps team manually rotates database passwords every 30 days. This leads to:

- Downtime during password updates
- Forgotten rotations
- Static secrets in playbooks or CI/CD pipelines

With Vault:

- Every build job requests a temporary credential from Vault
- Vault generates and revokes passwords automatically
- No need to manage static passwords anymore

◆ 3. Vault Architecture

HashiCorp Vault follows a robust and secure client-server architecture. It is designed with high scalability, pluggability, and security in mind.

Core Components of Vault Architecture

1. Storage Backend

Vault needs persistent storage to save encrypted secrets and metadata. This is called the **storage backend**.

Popular options:

- **Integrated Storage (Raft)** – Recommended for production use since Vault 1.4+
- **Consul** – Reliable and widely used for service discovery + HA coordination
- **DynamoDB** – Useful in AWS-based deployments
- **Google Cloud Storage / Azure Blob** – Useful in respective clouds
- **Filesystem** – Only for dev/test use

 Vault encrypts all data before it writes to storage. The storage backend never sees plaintext secrets.

2. Seal/Unseal Mechanism

Vault encrypts all data with a master key. This master key is encrypted with a **root key**, which is split into multiple **unseal keys** using [Shamir's Secret Sharing](#).

- Default is **5 key shares, 3 required to unseal**
- You must provide unseal keys every time Vault starts (unless auto-unseal is enabled)

Auto-Unseal (Cloud KMS support):

- AWS KMS
- GCP KMS
- Azure Key Vault

3. Vault API Server

The **Vault server** exposes a RESTful HTTP API on port 8200. All client interactions — whether CLI, GUI, or SDK — are done over this API.

- All requests are authenticated
- All access is logged (if auditing is enabled)
- All responses are encrypted

4. Audit Devices

Audit logs are essential for production security. Vault provides pluggable audit backends:

- File
- Syslog
- Socket
- Kafka (via plugins)

```
vault audit enable file file_path=/var/log/vault_audit.log
```

Each API request is logged, including:

- Who made it
- What was requested
- When it was done
- What was returned

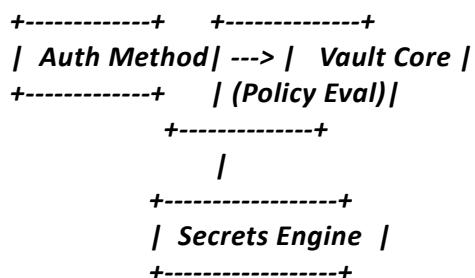
 Logs are critical for forensics and access compliance.

5. Auth Methods & Secrets Engines

Vault supports a **plugin system** for both:

- **Authentication backends:** How clients authenticate (LDAP, AppRole, Kubernetes, etc.)
- **Secrets engines:** What kind of secrets are being managed (KV, database, AWS, etc.)

Logical Flow (Simplified Diagram)



-
1. A client sends a token or credential to **auth method**
 2. Vault evaluates the **policy**
 3. If permitted, Vault returns secrets from the **secrets engine**
-

High-Level Lifecycle of a Vault Request

1. **Authenticate** – via AppRole, AWS IAM, etc.
 2. **Policy Evaluation** – "Is this action allowed?"
 3. **Execution** – Read/write to secrets engine
 4. **Lease Assignment** – If the secret has a TTL
 5. **Audit Logging** – Log the request and response
-

Key Takeaways

- Vault separates responsibilities using modular plugins.
- All secrets are encrypted before hitting disk.
- Vault has **no concept of users**, only **identities and tokens**.
- The architecture is built for high security and observability.

◆ 4. Vault Core Concepts

HashiCorp Vault is built around a few core concepts that form the foundation of its security model and functionality. Understanding these concepts is key to using Vault effectively in real-world, production-grade environments.

1. Seal & Unseal

What does "sealed" mean in Vault?

When Vault is sealed:

- The master encryption key is not present in memory.
- No operations can be performed (read/write/list/etc.)
- The server is effectively in lockdown.

How unsealing works:

- The master key is encrypted with a "root key."
- That root key is split into **key shares** using Shamir's Secret Sharing.
- A **minimum threshold** of key shares is required to unseal the Vault.

```
vault operator unseal <key1>
```

```
vault operator unseal <key2>
```

```
vault operator unseal <key3> # Now Vault is unsealed
```

Auto-Unseal (cloud-based): Vault can unseal itself automatically using a cloud KMS:

- AWS KMS
- Azure Key Vault
- GCP Cloud KMS

```
seal "awskms" {
  region    = "us-west-2"
  kms_key_id = "abcd-1234-5678"
}
```

2. Tokens

Vault does **not** use usernames and passwords directly (except with the userpass method). Instead, it uses **tokens** for access.

Types of Tokens:

- Root Token:

- Created during initialization.
 - Has full access (should be used rarely).
 - Cannot be revoked unless Vault is reinitialized.
- **Child Tokens:**
 - Scoped with specific policies.
 - Can have TTL, renewable, and can be revoked.

Token Capabilities:

- **Policies** define what each token can do.
- Tokens are **revocable** at any time.
- Tokens can be **orphaned** (not tied to parent token).

```
vault token create -policy="dev-team" -ttl=2h
```

```
vault token revoke <token_id>
```

✓ 3. Policies

Policies are written in **HCL (HashiCorp Configuration Language)** or **JSON**. They define what actions are allowed on which paths.

Example:

```
path "kv/data/dev/*" {  
    capabilities = ["create", "update", "read", "delete", "list"]  
}
```

- **Capabilities:**
 - read, create, update, delete, list, sudo

Policies are attached to:

- Tokens
- AppRoles
- Authenticated identities

📌 Best Practice: **Principle of Least Privilege** — Grant only what's needed.

✓ 4. Auth Methods

Vault supports many **authentication backends**, each tailored for a specific use case.

Auth Method	Use Case
userpass	Simple user login (dev/test only)
approle	Machines and CI/CD pipelines
github	GitHub team access
ldap	Enterprise directory authentication
kubernetes	Pods authenticating using service account
aws	EC2 IAM role authentication
jwt	Generic OIDC/JWT auth flows

Example: Enabling AppRole Auth Method

```
vault auth enable approle
```

```
vault write auth/approle/role/webapp \
  token_policies="webapp" \
  secret_id_ttl=60m \
  token_ttl=30m \
  token_max_ttl=120m
```

Then retrieve the role_id and secret_id:

```
vault read auth/approle/role/webapp/role-id
```

```
vault write -f auth/approle/role/webapp/secret-id
```

These values are used by clients (e.g., CI jobs, apps) to get a token.

Summary

- Vault is sealed by default and must be unsealed before use.
- Everything revolves around tokens — there are no "users."
- Policies define what each token can do.
- Auth methods map real-world identities to policies and tokens.

◆ 5. Vault Deployment Options

HashiCorp Vault is versatile when it comes to how it can be deployed. Whether you're experimenting on your laptop or rolling it out across a multi-region enterprise-grade architecture, Vault provides tailored modes to support different environments.

1. Dev Mode (Development Server)

This is the simplest mode and is great for:

- Learning
- Local testing
- Demos

```
vault server -dev
```

Features:

- Runs entirely in memory (no persistent storage)
- Automatically unsealed on start
- Generates a root token automatically
- Uses default kv secrets engine

 **Not for production** — secrets disappear on restart, and security is minimal.

2. Production Mode

In production, you want full encryption, persistence, and high availability.

Key components for production:

- Persistent storage backend
- TLS encryption
- Unseal configuration (manual or auto-unseal)
- Audit logging
- Secure ACL policies

```
storage "raft" {
  path = "/opt/vault/data"
}
```

```
listener "tcp" {
  address = "0.0.0.0:8200"
```

```
tls_cert_file = "/etc/vault/tls/cert.pem"
tls_key_file = "/etc/vault/tls/key.pem"
}
```

```
seal "awskms" {
  region  = "us-west-2"
  kms_key_id = "abcd-1234"
}
```

Considerations:

- Use **firewall rules** to restrict API access
- Enable **audit devices**
- Rotate keys and tokens regularly

3. High Availability (HA)

Vault can be run in a **HA configuration** to ensure uptime, availability, and failover.

Supported Storage Backends for HA:

- **Consul** (external coordination)
- **Integrated Raft storage** (recommended since Vault 1.4+)

How HA works:

- One node is **active**
- Others are **standby**
- Standbys automatically become active if the leader fails

```
storage "raft" {
  path   = "/opt/vault/data"
  node_id = "vault-node-1"
}
```

```
ha_enabled = true
```

4. Deployment Topologies

Dev Lab (non-HA):

[Vault (dev mode)]

Basic Prod (1 node, TLS, Raft storage):

[Vault (active)] --> [TLS] --> [Users / Apps]
|
[Raft Storage]

Full HA with Auto-Unseal:

```
[ Vault Node 1 (active) ]  
[ Vault Node 2 (standby) ]  
[ Vault Node 3 (standby) ]  
|     |     |  
[ Raft Integrated Storage ]  
|  
[ AWS KMS for Auto-Unseal ]
```

Key Best Practices

- Always enable TLS with **valid certs**
- Prefer **Raft over Consul** unless you're already using Consul
- Enable **audit logging**
- Use **load balancers** in front of Vault cluster
- Use **cloud KMS** for unseal automation

◆ 6. Authentication Methods

Authentication in Vault determines **who or what** is accessing Vault and **what they are allowed to do** (via policies). Vault provides multiple pluggable auth methods to support a wide range of identity systems — from static credentials to cloud-based identity verification.

Vault Authentication Overview

- Vault **does not have users** in the traditional sense.
- Vault issues **tokens** upon successful authentication.
- Each auth method maps an external identity to Vault's internal identity system (token + policy).
- You can enable multiple auth methods at once.

Common Auth Methods

Auth Method	Best Use Case
token	Basic auth for automation/scripts
userpass	Manual testing, small teams
ldap	Enterprises with centralized directory access
github	Developers in GitHub orgs
approle	CI/CD pipelines, automation
aws	IAM-role-based auth for EC2, Lambda
kubernetes	Pod-to-Vault authentication
jwt/oidc	App or user access via identity provider

AppRole Authentication (Best for CI/CD)

AppRole is a **programmatic login** method ideal for:

- Jenkins
- GitLab CI
- Automation scripts

It uses:

- role_id (public)
- secret_id (private)

Enable and Configure AppRole:

```
vault auth enable approle
```

```
vault write auth/approle/role/webapp \
token_policies="webapp" \
secret_id_ttl=60m \
token_ttl=30m \
token_max_ttl=60m
```

Retrieve credentials:

```
vault read auth/approle/role/webapp/role-id
```

```
vault write -f auth/approle/role/webapp/secret-id
```

Login using AppRole:

```
vault write auth/approle/login \
role_id="..." \
secret_id="..."
```

Kubernetes Authentication

Kubernetes Auth method allows Vault to authenticate Pods using their **ServiceAccount JWT tokens**.

Use Cases:

- Inject secrets into running containers
- Fetch DB credentials from inside pods

Setup Steps:

1. Enable Kubernetes auth:

```
vault auth enable kubernetes
```

2. Configure Kubernetes connection:

```
vault write auth/kubernetes/config \
token_reviewer_jwt "<JWT>" \
kubernetes_host="https://<K8S-API-ENDPOINT>" \
kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

3. Create Vault role:

```
vault write auth/kubernetes/role/webapp \
  bound_service_account_names="vault-auth" \
  bound_service_account_namespaces="default" \
  policies="webapp" \
  ttl="24h"
```

AWS IAM Authentication

This method uses the **identity of an EC2 instance or Lambda function** to authenticate to Vault.

Steps:

1. Enable AWS auth:

```
vault auth enable aws
```

2. Create a Vault role bound to an IAM role:

```
vault write auth/aws/role/dev-role \
  auth_type=iam \
  bound_iam_principal_arn="arn:aws:iam::123456789:role/vault-role" \
  policies="dev-policy"
```

3. On EC2 instance:

```
vault write auth/aws/login \
  role="dev-role" \
  iam_http_request_method="POST" \
  iam_request_url="$URL" \
  iam_request_body="$BODY" \
  iam_request_headers="$HEADERS"
```

GitHub Authentication

Allows users in a GitHub organization to authenticate.

```
vault auth enable github
```

```
vault write auth/github/config organization="devopsteam"
```

```
vault write auth/github/map/teams/devops value="dev-policy"
```

Users log in with:

```
vault login -method=github token=<your_github_token>
```

LDAP Authentication

For organizations with Active Directory or OpenLDAP.

```
vault auth enable ldap
```

```
vault write auth/ldap/config \  
 url="ldap://ldap.example.com" \  
 userdn="ou=users,dc=example,dc=com" \  
 groupdn="ou=groups,dc=example,dc=com" \  
 binddn="cn=admin,dc=example,dc=com" \  
 bindpass='admin_password' \  
 userattr="uid" \  
 groupfilter="(objectClass=groupOfNames)"
```

Map LDAP groups to policies:

```
vault write auth/ldap/groups/dev-team policies="dev-policy"
```

Best Practices

- Use **AppRole** for automation.
- Use **Kubernetes Auth** for internal workloads.
- Use **OIDC or GitHub** for developer-facing auth.
- Always limit token TTL and renewals.
- Regularly rotate secret_id if using AppRole.

◆ 7. Access Control (Policies)

HashiCorp Vault enforces security through **policies**. Policies define what a user, application, or identity can and cannot do inside Vault.

Vault's policies are written using **HCL (HashiCorp Configuration Language)** or JSON, and they are applied to identities via **tokens, roles, or auth methods**.

Why Policies Matter

- Vault doesn't rely on usernames or roles natively.
 - Access is entirely governed by **capabilities on paths**.
 - You assign **read/write/delete/etc.** privileges to specific paths in Vault.
-

Anatomy of a Vault Policy

A Vault policy is a mapping of **path** to **capabilities**:

```
path "kv/data/dev/*" {  
    capabilities = ["create", "update", "read"]  
}
```

Capabilities include:

- create – Write new secrets
 - read – Read existing secrets
 - update – Overwrite secrets
 - delete – Delete secrets
 - list – List secrets in a path
 - sudo – Bypass normal policy restrictions
-

Example: Read-Only Policy for Dev Secrets

```
path "kv/data/dev/*" {  
    capabilities = ["read", "list"]  
}
```

This allows read-only access to **any key** under the path `kv/data/dev/`.

💡 Example: Admin Policy for All Paths

```
path "*" {  
    capabilities = ["create", "read", "update", "delete", "list", "sudo"]  
}
```

Use with **caution** — this grants **full access** to everything.

📌 Attaching Policies to Tokens or Roles

When a token is created (manually or via login), it can have one or more policies attached:

```
vault token create -policy="dev-policy" -ttl=1h
```

If you're using AppRole or Kubernetes auth, assign the policy to the **role**, not the token directly:

```
vault write auth/approle/role/myapp \  
    token_policies="dev-policy"
```

📁 Real-World Scenario

Let's say you want to configure access like this:

Role	Path Prefix	Capabilities
Developer	kv/data/dev/*	read, list, update
CI/CD Pipeline	database/creds/ci	read
Admin	*	All

You would create 3 policies:

- dev-policy.hcl
- cicd-policy.hcl
- admin-policy.hcl

Then apply them to roles, auth backends, or tokens.

Listing and Inspecting Policies

List all policies:

```
vault policy list
```

View a specific policy:

```
vault policy read dev-policy
```

Delete a policy:

```
vault policy delete dev-policy
```

Vault Default Policies

- root: Full access — only the root token should use this.
- default: Automatically assigned if no policy is provided.

Best practice: **Never use root token in daily operations.** Use specific policies for each use case.

Pro Tips

- Structure secrets using logical paths (kv/data/dev, kv/data/prod)
- Grant minimal capabilities — **principle of least privilege**
- Use HCL format for better readability
- Store policies in version-controlled Git repos
- Create readonly, admin, ci, developer templates

◆ 8. Secrets Engines

🔑 What is a Secrets Engine?

Secrets Engines are **pluggable components** in Vault that handle secrets. They are responsible for **storing, generating, encrypting, or managing access** to secrets like:

- Static key-value secrets
- Dynamic database credentials
- AWS IAM tokens
- Encryption-as-a-service (EaaS)

Each secrets engine is **mounted** at a path (like kv/, database/, or transit/) and can have its own configuration, versioning, and access controls.

✳️ Types of Secrets Engines

Engine	Use Case
kv	Static secrets (API keys, passwords)
transit	Encryption as a service (EaaS)
database	Dynamic, lease-based DB credentials
aws	On-demand IAM credentials
azure	Azure service principal generation
gcp	GCP service accounts
cubbyhole	Private, token-bound temporary secrets
identity	OIDC and identity mapping
pki	Generate and manage TLS certificates

✓ 1. KV (Key-Value) Secrets Engine

Used for **static secrets** like tokens, passwords, or config settings.

Enable KV:

```
vault secrets enable -path=kv kv
```

Store a secret:

```
vault kv put kv/myapp username=admin password=secret123
```

Retrieve it:

```
vault kv get kv/myapp
```

Delete it:

```
vault kv delete kv/myapp
```

2. Transit Engine (Encryption-as-a-Service)

Transit does not store data. It simply **encrypts/decrypts** data using Vault-managed keys.

Great for:

- Tokenizing PII data
- Application-layer encryption
- PCI/GDPR-compliant workflows

Enable transit:

```
vault secrets enable transit
```

Create encryption key:

```
vault write -f transit/keys/customer-data
```

Encrypt data:

```
vault write transit/encrypt/customer-data plaintext=$(base64 <<< "sensitive-info")
```

Decrypt data:

```
vault write transit/decrypt/customer-data ciphertext="vault:v1:..."
```

3. Database Secrets Engine

Generates **dynamic, time-bound DB credentials** on-the-fly.

Supported DBs:

- PostgreSQL
- MySQL/MariaDB
- MSSQL
- Oracle (via plugins)

Enable and configure:

```
vault secrets enable database
```

```
vault write database/config/my-postgresql \
  plugin_name=postgresql-database-plugin \
  allowed_roles="readonly" \
  connection_url="postgres://vaultuser:password@db.example.com:5432/mydb"
```

Create a role:

```
vault write database/roles/readonly \
  db_name=my-postgresql \
  creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{password}}'
VALID UNTIL '{{expiration}}';" \
  default_ttl="1h" \
  max_ttl="24h"
```

Fetch dynamic creds:

```
vault read database/creds/readonly
```

 **4. AWS Secrets Engine**

Dynamically generates **short-lived AWS credentials** with custom IAM permissions.

Enable:

```
vault secrets enable aws
```

Configure root credentials:

```
vault write aws/config/root \
  access_key=AKIA... \
  secret_key=...
```

Create a Vault role:

```
vault write aws/roles/my-role \
  credential_type=iam_user \
  policy_document=-<<EOF
{
  "Version": "2012-10-17",
  "Statement": [{
```

```
"Effect": "Allow",
"Action": "ec2.*",
"Resource": "*"
}]
}
EOF
```

Get IAM credentials:

```
vault read aws/creds/my-role
```

5. Cubbyhole Secrets Engine

- Bound to a **single token**
- Secret is deleted when the token expires
- Used for **one-time secret sharing or short-lived bootstrap secrets**

```
vault write cubbyhole/mysecret secret_value="hello"
```

```
vault read cubbyhole/mysecret
```

Managing Secret Engines

- **List all engines:**

```
vault secrets list
```

- **Disable an engine:**

```
vault secrets disable kv
```

- **Enable at custom path:**

```
vault secrets enable -path=secrets kv
```

Security Tips

- Mount secrets engines under logical paths: kv/dev/, kv/prod/, etc.
- Apply least privilege: limit who can access each secrets engine
- Use TTLs and leases where applicable
- Combine engines (e.g., Transit + Database) for advanced workflows

◆ 9. KV v1 vs KV v2 (Key-Value Secrets Engine Versions)

The **Key-Value (KV)** secrets engine is the most commonly used secrets engine in Vault. It allows you to store **static secrets** (such as tokens, passwords, config values) as key-value pairs.

Vault provides **two versions** of the KV engine:

- **KV v1** – Simple and flat
 - **KV v2** – Adds versioning and metadata
-

KV v1: Flat Key-Value Store

Features:

- Supports basic get, put, delete operations
- No support for versioning
- Quick and simple to use

Usage:

```
vault secrets enable -path=kv kv # default is v1
```

```
vault kv put kv/myapp username="admin" password="pass123"
```

```
vault kv get kv/myapp
```

```
vault kv delete kv/myapp
```

 KV v1 is straightforward but not suited for sensitive workflows that require auditability, version rollback, or change tracking.

KV v2: Versioned Key-Value Store

Introduced in Vault 0.10.0

Features:

- Built-in **version history**
- Soft delete with **delete & undelete**
- Optional **destroy** (permanently remove)
- Metadata management
- Protection against overwriting unintended versions

Enable KV v2:

```
vault secrets enable -path=secret kv-v2
```

Write secret (auto creates version 1):

```
vault kv put secret/myapp username="admin" password="vault123"
```

Get latest version:

```
vault kv get secret/myapp
```

Get specific version:

```
vault kv get -version=2 secret/myapp
```

Deletion Semantics (KV v2)

Soft Delete:

```
vault kv delete secret/myapp
```

Marks the secret as deleted, but it can still be **recovered**.

Undelete:

```
vault kv undelete -versions=2 secret/myapp
```

Destroy (permanently):

```
vault kv destroy -versions=2 secret/myapp
```

Metadata Commands

- List metadata for a key:

```
vault kv metadata get secret/myapp
```

- Delete metadata (and all versions):

```
vault kv metadata delete secret/myapp
```

Comparison: KV v1 vs KV v2

Feature	KV v1	KV v2
Basic CRUD	✓ Yes	✓ Yes
Versioning	✗ No	✓ Yes
Soft Delete	✗ No	✓ Yes
Undelete support	✗ No	✓ Yes

Feature	KV v1	KV v2
Destroy a version	 No	 Yes
Metadata operations	 No	 Yes

Best Practices

- Use **KV v2** if:
 - You need rollback or version tracking
 - Your secrets are sensitive
 - You want fine-grained deletion control
 - Use **KV v1** if:
 - Simplicity and speed matter more
 - You manage secrets via automation and versioning isn't needed
-

Real-World Use Case

A development team wants to store deployment credentials:

- In dev: they use KV v1 because rollback isn't needed.
- In prod: they use KV v2 so that accidental overwrites can be undone.

This strategy balances simplicity with safety.

◆ 10. Secure Introduction of Clients

Before any application or service can access secrets from Vault, it must **authenticate securely**. This process of securely providing a client with access credentials — without manual intervention — is known as **secure introduction** or **bootstrap authentication**.

Problem with Static Tokens

You might be tempted to hardcode a token in your app or store it in a config file. But:

- If the token leaks, anyone can access Vault
- You lose visibility into *who* is using Vault
- Revoking a token requires app redeloys

 Hardcoding Vault tokens is a huge **security anti-pattern**.

Secure Introduction Methods

HashiCorp Vault provides two **secure**, dynamic methods for introducing new clients:

1. AppRole with Vault Agent (Pull model)

AppRole is ideal for non-human entities (like services or CI/CD jobs). Vault Agent is a lightweight client that runs beside your app.

How it works:

- You assign an **AppRole** with limited permissions.
- Vault Agent fetches the secret_id and uses it with role_id to login.
- The token is **cached and automatically renewed**.

Sample Setup:

```
auto_auth {  
    method "approle" {  
        mount_path = "auth/approle"  
        config = {  
            role_id_file_path = "./role_id"  
            secret_id_file_path = "./secret_id"  
        }  
    }  
  
    sink "file" {  
        config = {  
            path = "/home/myapp/.vault-token"  
        }  
    }  
}
```

```
    }  
}  
}
```

Then run Vault Agent:

```
vault agent -config=agent-config.hcl
```

2. Kubernetes Auth Method (Push model)

For apps running inside Kubernetes, Vault can trust a Pod's **ServiceAccount JWT** and issue it a Vault token.

How it works:

- Pod starts with a ServiceAccount attached
- It calls Vault /auth/kubernetes/login
- Vault verifies the JWT and issues a scoped token
- The Pod uses this token to read secrets

Setup:

1. Enable Kubernetes auth:

```
vault auth enable kubernetes
```

2. Configure Vault to talk to Kubernetes API:

```
vault write auth/kubernetes/config \  
  kubernetes_host="https://<k8s-api>" \  
  kubernetes_ca_cert=@ca.crt \  
  token_reviewer_jwt=@token.jwt
```

3. Create a role:

```
vault write auth/kubernetes/role/my-app \  
  bound_service_account_names="vault-app" \  
  bound_service_account_namespaces="default" \  
  policies="myapp" \  
  ttl="24h"
```

4. Pod logs in:

```
vault write auth/kubernetes/login \  
  role="my-app" \  
  jwt=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
```

Additional Options

Method	Best For
Vault Agent + AppRole	VMs, CI/CD, bare-metal apps
Kubernetes Auth	Apps running in K8s
AWS IAM Auth	EC2 / Lambda authentication
OIDC	User-interactive auth

Security Best Practices

- **Use TTLs** for tokens and credentials
- **Use Vault Agent** to auto-renew tokens
- Never **hardcode Vault tokens** in apps
- Leverage **policies** to restrict what a client can access
- Use **dynamic secrets** to minimize risk even if a token is leaked

Real-World Use Case

A CI/CD pipeline uses AppRole to fetch short-lived credentials for deploying apps. The token lasts only 30 minutes and is rotated for each build.

A Kubernetes-based app uses Vault Agent Injector to fetch TLS certs from Vault and store them in a mounted volume at runtime.

◆ 11. Leasing & Renewals

In Vault, most secrets are not static. They are **leased**, meaning they have a **Time-To-Live (TTL)** and expire automatically after a period.

This concept is central to Vault's design and provides massive advantages over traditional secrets management systems.

What Is a Lease?

A **lease** is a time-bound association between a client and a secret.

- When a secret is issued, it comes with a **lease ID** and a **TTL**.
- Vault tracks the lease internally and can **revoke** it when:
 - It expires
 - You revoke it manually
 - The parent token is revoked

Every lease can be **renewed** or **revoked** before it expires.

How Leasing Works (Behind the Scenes)

Let's say you generate dynamic PostgreSQL credentials:

```
vault read database/creds/readonly
```

Vault will return:

```
{  
  "data": {  
    "username": "v-root-abc1",  
    "password": "random-pass"  
  },  
  "lease_id": "database/creds/readonly/abcxyz",  
  "lease_duration": 3600,  
  "renewable": true  
}
```

- The lease is good for **1 hour**.
- Vault will **revoke** the credentials when the lease expires.

-
- You can **renew** the lease if needed.
-

Useful Lease Commands

Check lease info:

```
vault lease lookup database/creds/readonly/abcxyz
```

Renew a lease:

```
vault lease renew database/creds/readonly/abcxyz
```

You can also **specify a new TTL** (within allowed range):

```
vault lease renew -increment=3600 <lease_id>
```

Revoke a lease:

```
vault lease revoke database/creds/readonly/abcxyz
```

Leases Are Everywhere

Leases apply to:

- **Dynamic secrets** (like database, AWS, GCP)
 - **Tokens**
 - **Temporary access to encryption keys**
 - **PKI certificates**
-

Token Renewal & Expiration

Tokens are also **leased**.

When you log in via AppRole, Kubernetes, etc., your Vault token has a TTL:

```
vault token lookup
```

Output:

```
{  
  "ttl": 3600,  
  "renewable": true,  
  "expire_time": "2025-04-11T20:00:00Z"  
}
```

Renew the token:

```
vault token renew
```

TTL Hierarchy

Vault calculates the **least** of:

- The **token TTL**
- The **secret TTL**
- The **policy max TTL**
- The **lease's default TTL**

Example: If your policy TTL is 1 hour but the secret engine's TTL is 24h, the secret expires in 1 hour.

You can configure default and max TTLs per backend:

```
vault write database/roles/readonly \
  default_ttl="1h" \
  max_ttl="24h"
```



Real-World Scenario

You generate short-lived AWS credentials for deployment:

```
vault read aws/creds/dev-role
```

You use the credentials to run a deployment job.

The lease is revoked after 15 minutes, making those credentials useless — even if compromised.



Vault Agent Auto-Renew

If you're using **Vault Agent**, it can automatically renew leases and tokens behind the scenes.

```
cache {
  use_auto_auth_token = true
}

auto_auth {
  ...
}
```

```
listener "tcp" {  
    ...  
}
```

Summary

- **Every secret** in Vault can have a TTL.
- Vault uses **leases** to track secret lifecycles.
- You can renew or revoke leases using simple CLI/API calls.
- Vault Agent makes renewal automatic.

◆ 12. Audit Devices

Why Auditing Matters

Vault is a **security-critical system**. Every interaction with Vault should be traceable.

Audit devices record all requests and responses to Vault, including:

- Who made the request
- When it was made
- What was requested
- What Vault returned

Audits **do not** log sensitive data (like full secrets), but they log metadata.

Enabling an Audit Device

Enable file-based audit:

```
vault audit enable file file_path=/var/log/vault_audit.log
```

Enable syslog:

```
vault audit enable syslog
```

Disable an audit:

```
vault audit disable file
```

 It's recommended to enable **at least one** audit device in production.

Audit Log Example (Truncated):

```
{  
  "time": "2025-04-11T12:00:00.000Z",  
  "auth": { "client_token": "hmac-sha256", "display_name": "approle" },  
  "request": { "path": "kv/data/creds", "operation": "read" },  
  "response": { "status": 200 }  
}
```

◆ 13. Vault Agent

The Vault Agent is a lightweight daemon that:

- Handles **authentication** (auto-auth)
- Automatically **renews tokens and leases**
- Can **render secrets into templates**

Example Use Case:

Run Vault Agent alongside your app. It logs in using AppRole, renews tokens in the background, and writes secrets to a file.

Vault Agent HCL Config:

```
auto_auth {  
  method "approle" {  
    mount_path = "auth/approle"  
    config = {  
      role_id_file_path = "./role_id"  
      secret_id_file_path = "./secret_id"  
    }  
  }  
  
  sink "file" {  
    config = {  
      path = "/home/app/.vault-token"  
    }  
  }  
}
```

```
}
```

```
}
```

```
}
```

Run agent:

```
vault agent -config=agent.hcl
```

◆ 14. Template Rendering

Vault Agent can render secrets into files using **Go templating**.

Example Template:

config.tpl

```
{  
  "db_username": "{{ with secret \"kv/data/myapp\" }}{{ .Data.data.username }}{{ end }}",  
  "db_password": "{{ with secret \"kv/data/myapp\" }}{{ .Data.data.password }}{{ end }}"  
}
```

Agent config:

```
template {  
  source = "./config.tpl"  
  destination = "/etc/config.json"  
}
```

Run Vault Agent. It'll write the populated config file with secrets to disk.

◆ 15. Vault with Kubernetes (Helm)

Vault can be deployed and integrated with Kubernetes using **Helm charts** and the **Vault Injector**.

Deploy Vault via Helm

```
helm repo add hashicorp https://helm.releases.hashicorp.com
```

```
helm install vault hashicorp/vault
```

Use the **Vault Injector** to auto-inject secrets into Pods as:

- Environment variables

-
- Mounted files
-

Sample Pod Annotation:

annotations:

```
vault.hashicorp.com/agent-inject: "true"  
vault.hashicorp.com/role: "myapp"  
vault.hashicorp.com/agent-inject-secret-db-creds: "kv/data/db"  
vault.hashicorp.com/agent-inject-template-db-creds: |  
{{- with secret "kv/data/db" -}}  
DB_USERNAME={{ .Data.data.username }}  
DB_PASSWORD={{ .Data.data.password }}  
{{- end }}
```

Secrets are injected at container runtime.

◆ 16. HA Setup with Integrated Storage

Vault supports **native Raft storage** for HA — no need for Consul or external DB.

Example config for Raft HA:

```
storage "raft" {  
    path  = "/opt/vault/data"  
    node_id = "vault-1"  
}
```

```
listener "tcp" {  
    address  = "0.0.0.0:8200"  
    tls_disable = 1  
}
```

```
seal "awskms" {  
    region  = "us-west-2"
```

```
kms_key_id = "abc-123"  
}
```

```
api_addr = "http://vault-1:8200"  
cluster_addr = "http://vault-1:8201"
```

All Vault nodes share a **Raft consensus cluster**. One becomes **active**, the rest are **standby**.

-  For high availability, add Vault behind a load balancer.
-

◆ 17. Enterprise Features

Vault Enterprise includes advanced functionality:

◆ Namespaces

- Like multi-tenancy
- Separate auth backends, secrets engines, policies

◆ Replication

- **Performance Replication** (multi-region reads)
- **Disaster Recovery Replication**

◆ Sentinel

- Policy-as-code enforcement
 - Add business rules to Vault operations
-

◆ 18. Dynamic Secrets: Database

Vault can **generate temporary DB credentials** on-demand.

Configure PostgreSQL Plugin

```
vault write database/config/my-postgresql \  
plugin_name=postgresql-database-plugin \  
allowed_roles="readonly" \  
connection_url="postgres://{{username}}:{{password}}@localhost:5432/mydb"
```

Create a role:

```
vault write database/roles/readonly \
```

```
db_name=my-postgresql \
creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{password}}'
VALID UNTIL '{{expiration}}';" \
default_ttl="1h" \
max_ttl="24h"
```

Fetch credentials:

vault read database/creds/readonly

You get:

- A temporary username
 - A temporary password
 - TTL of 1 hour
-

◆ 19. CI/CD Integration

Vault integrates easily with CI/CD tools like:

- GitHub Actions
- GitLab CI
- Jenkins
- ArgoCD

Example: GitLab CI

vault:

script:

- export VAULT_TOKEN=\$(vault login -method=approle ...)
- export DB_PASSWORD=\$(vault kv get -field=password kv/db)
- ./deploy.sh

Use Vault CLI or **Vault Agent Injector** to inject secrets into pipelines.

◆ 20. Real-World Use Cases

1. Replace secrets in Ansible/Playbooks

Use `lookup('hashi_vault', ...)` to pull secrets during provisioning.

2. Auto-Rotate DB Credentials

Dynamic secrets rotate every hour. Vault revokes old creds automatically.

3. Short-Lived Cloud Access

Vault generates AWS/GCP/Azure credentials with TTL = 15 mins. Perfect for CI/CD.

4. Central Secrets Hub in Multi-Cloud

Vault works across:

- AWS
- GCP
- Azure
- On-prem VMs
- Kubernetes

Centralize all secrets in one place.

Final Words

HashiCorp Vault is not just a secrets manager — it's a **dynamic trust broker**, a **central identity-aware secret lifecycle controller**, and a **security automation powerhouse**.

Whether you're securing a single app or managing a global infrastructure, Vault empowers you with:

- Auditability
- Dynamic access control
- Secrets rotation
- Enterprise-grade architecture