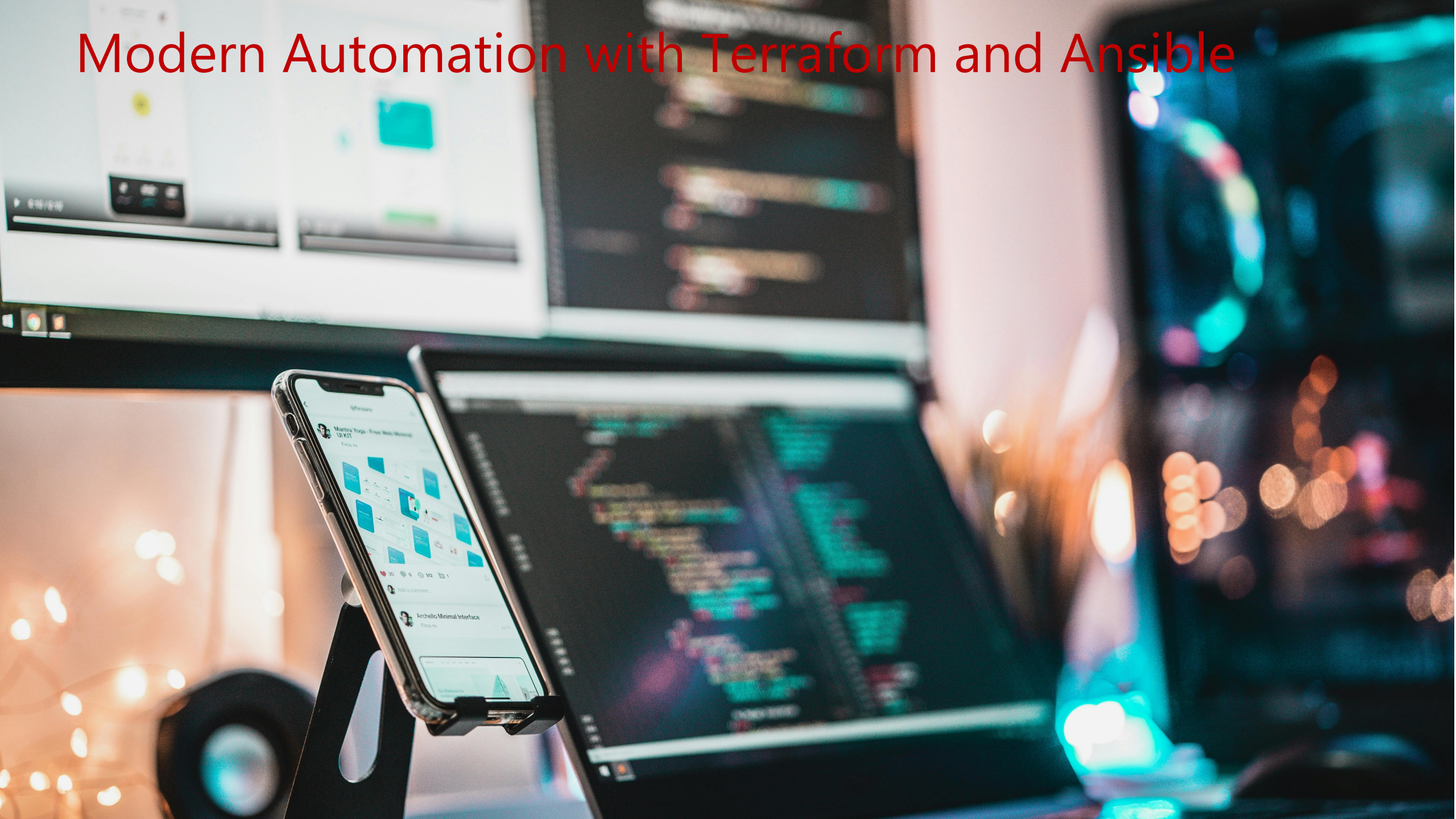
# Modern Automation with Terraform and Ansible



TEKsystems Global Services

# WORKFORCE DEVELOPMENT

# Infrastructure Drift Prevention

Drift prevention is the practice of implementing controls and processes to stop infrastructure changes from happening outside of your IaC tool. It involves setting up guardrails, access controls, and automated monitoring to ensure all infrastructure changes go through your approved Terraform workflows.

- Comprehensive tagging policies - Ensure all resources have consistent tags
- Resource locking - Prevent manual modifications to critical resources
- Regular monitoring - Set up alerts for unexpected changes
- Access controls - Limit who can modify infrastructure Outside of Terraform

# Using HashiCorp Sentinel for Drift Prevention


Sentinel is HashiCorp's policy-as-code framework that provides automated policy enforcement for Terraform deployments. It acts as a gatekeeper that evaluates infrastructure changes against predefined organizational policies before they can be applied.

Sentinel policies can enforce security standards, compliance requirements, and operational best practices by analyzing Terraform plans and configurations. When policies fail, they can either block the deployment entirely or provide warnings, depending on the enforcement level configured.

# Using Terratest for Drift Prevention


Terratest is a Go library that provides automated testing for infrastructure code, allowing you to write comprehensive tests that validate your Terraform configurations work correctly in real environments. It can test infrastructure deployments, validate resource configurations, and ensure compliance with organizational standards.

Terratest runs tests against actual deployed infrastructure, providing real-world validation that your configurations meet requirements and work as expected. The framework can test resource attributes, security configurations, and compliance standards by making actual API calls to cloud providers.

# HCP Terraform Workspace Best Practices

This section covers essential best practices for organizing and managing HCP Terraform workspaces effectively, based on HashiCorp's recommended workflow for collaborative infrastructure as code. We'll explore workspace structure, organizational delegation, and the fundamental challenges in provisioning at scale.

The topics include addressing technical and organizational complexity through proper workspace organization, understanding the four main personas for infrastructure management, and implementing the recommended workspace structure that supports effective delegation and parallel development.

# One Workspace Per Environment Per Terraform Configuration

```
applications/
├── billing-app-dev
├── billing-app-stage
├── billing-app-prod
├── user-portal-dev
├── user-portal-stage
└── user-portal-prod

infrastructure/
├── networking-dev
├── networking-stage
├── networking-prod
├── security-dev
└── security-prod
```

HCP Terraform workspaces are designed to delegate control effectively, which means their structure should align with your organizational permissions structure. The recommended approach is to use one workspace for each environment of a given infrastructure component, following the formula: Terraform configurations × environments = workspaces.

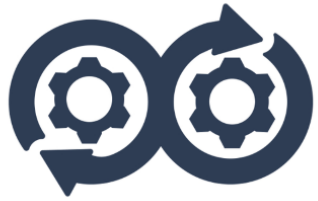
This approach differs from other tools' environment management by avoiding single workspaces that manage entire production or staging environments. Instead, create smaller, focused workspaces that are easier to delegate and manage. This flexibility also means not every configuration needs the same environments - if a UAT environment doesn't make sense for security infrastructure, you aren't forced to create one.

# Delegating Workspaces

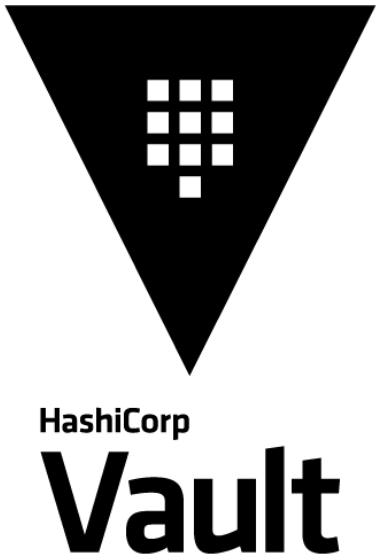

HCP Terraform's workspace structure enables effective delegation of infrastructure ownership through per-workspace access controls. Since each workspace represents one environment of one infrastructure component, you can precisely control who has access to specific components and environments.

- Teams that help manage a component can start Terraform runs and edit variables in dev or staging
- The owners or senior contributors of a component can start Terraform runs in production, after reviewing other contributors' work
- Central IT and organization architects can administer permissions on all workspaces, to ensure everyone has what they need to work
- Teams that have no role managing a given component don't have access to its workspaces

# Lab: Drift Prevention and Debugging



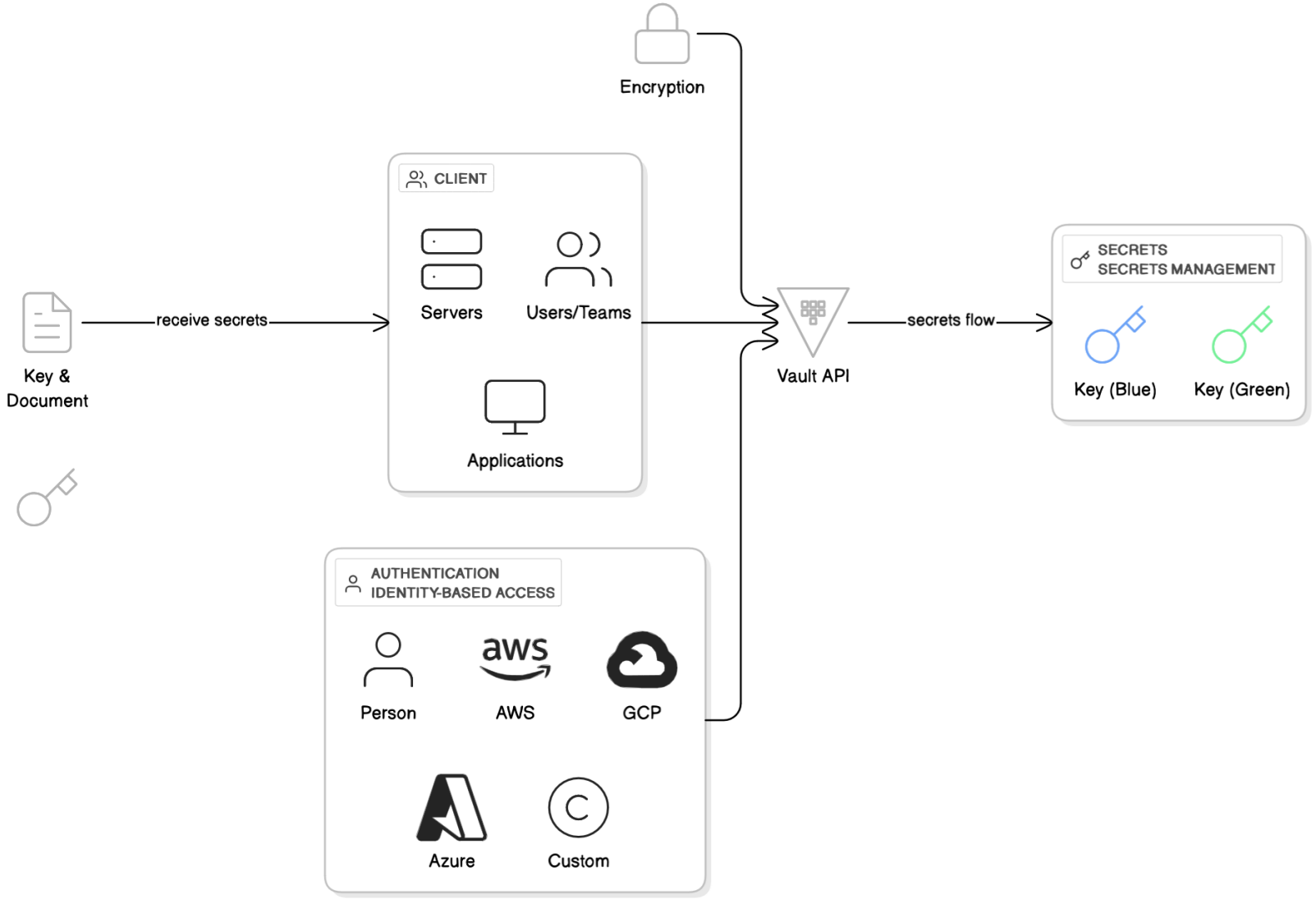
# What is HashiCorp Vault?



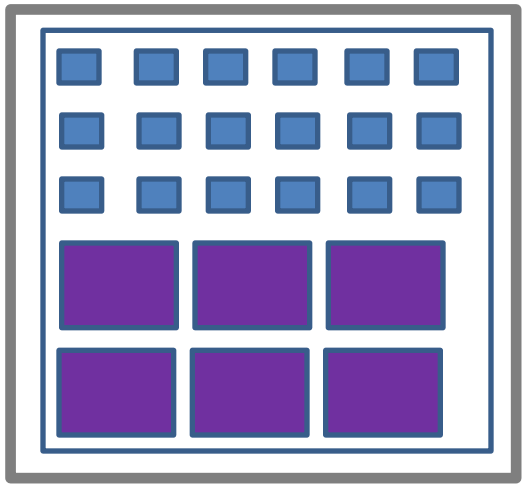

HashiCorp Vault provides centralized, well-audited privileged access and secret management for mission-critical data across on-premises, cloud, and hybrid environments. With a modular design based around a growing plugin ecosystem, Vault integrates with existing systems and customizes application workflows. Modern software relies on secrets - sensitive information like credentials, encryption keys, authentication certificates, and other critical data that applications need to run consistently and securely. Vault helps harden applications by centralizing secret management.

Vault enables organizations to manage static secrets, certificates, identities and authentication, third-party secrets, sensitive data, and support regulatory compliance requirements.

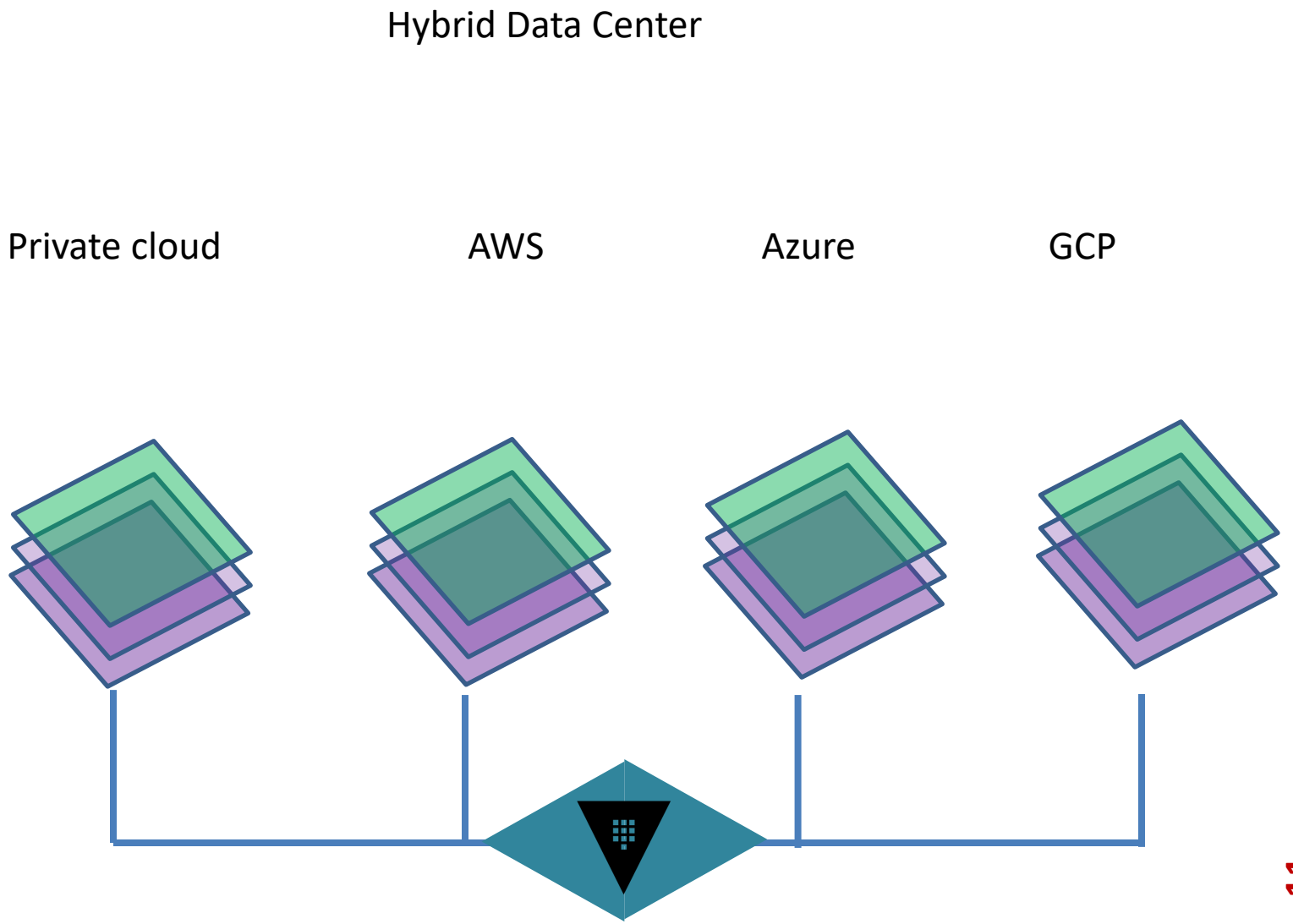
# HashiCorp Vault Architecture



# Secure Infrastructure using Vault

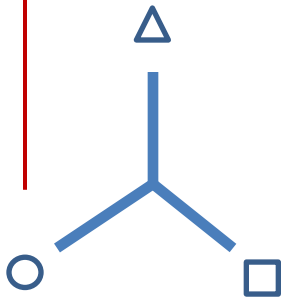


With each new cloud, network topologies become more complex.



# Vault Objectives

- Provide single source of secrets for humans and machines .
- Scale to meet security needs of largest organizations.
- Allow for complete secret lifecycle management.



Eliminate Secret Sprawl

Securely Store any Secret

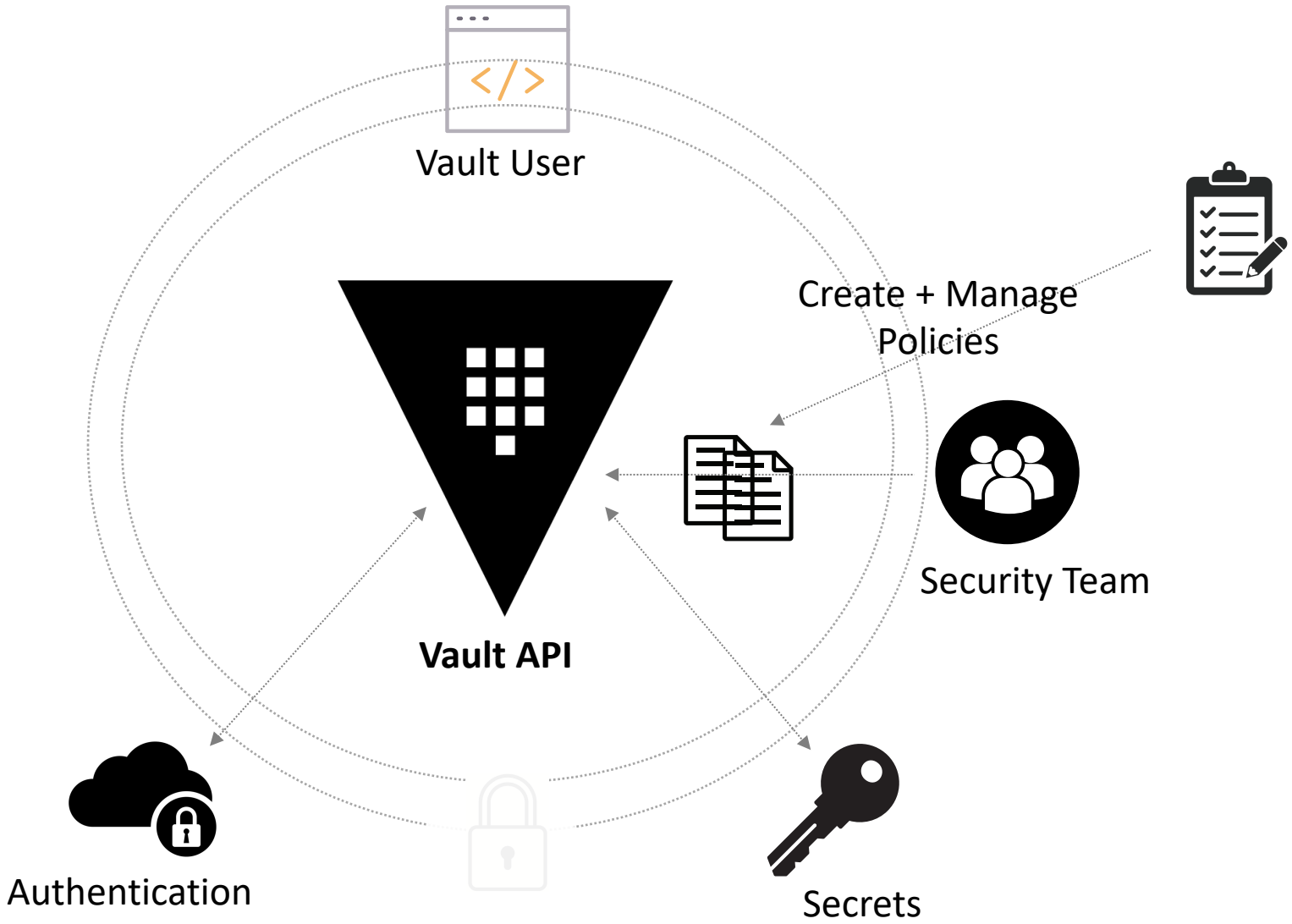
Secret Governance

# Use Cases

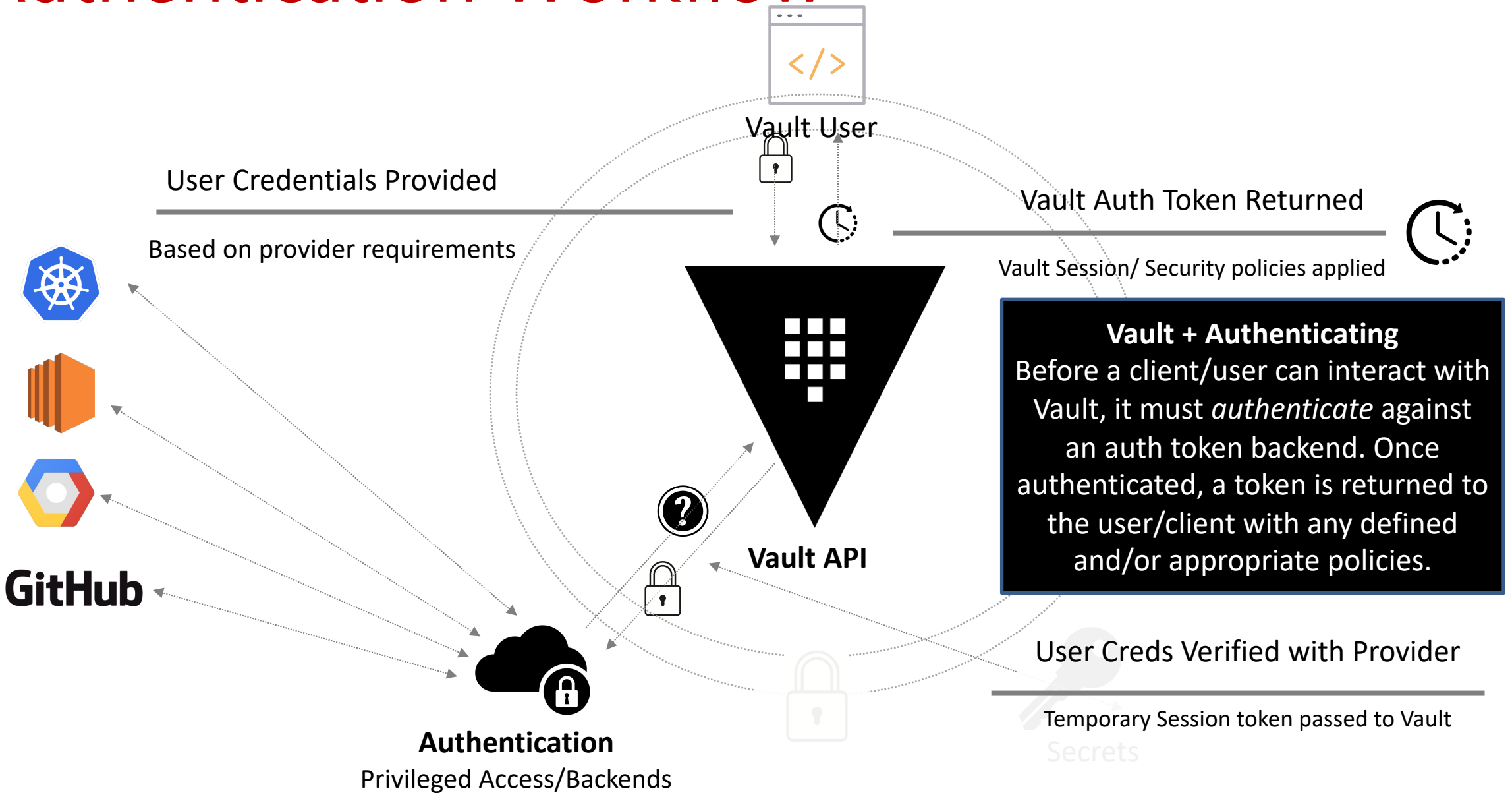
| Secrets Management | Encryption as a Service | Identity Access Management |
| --- | --- | --- |
| Secrets, identity, and access policy management workflow to secure any infrastructure and application resources. | One workflow to create and control the keys used to encrypt your data | Empower developers and operators to securely make application and infrastructure changes. |

# Vault - Security Policies

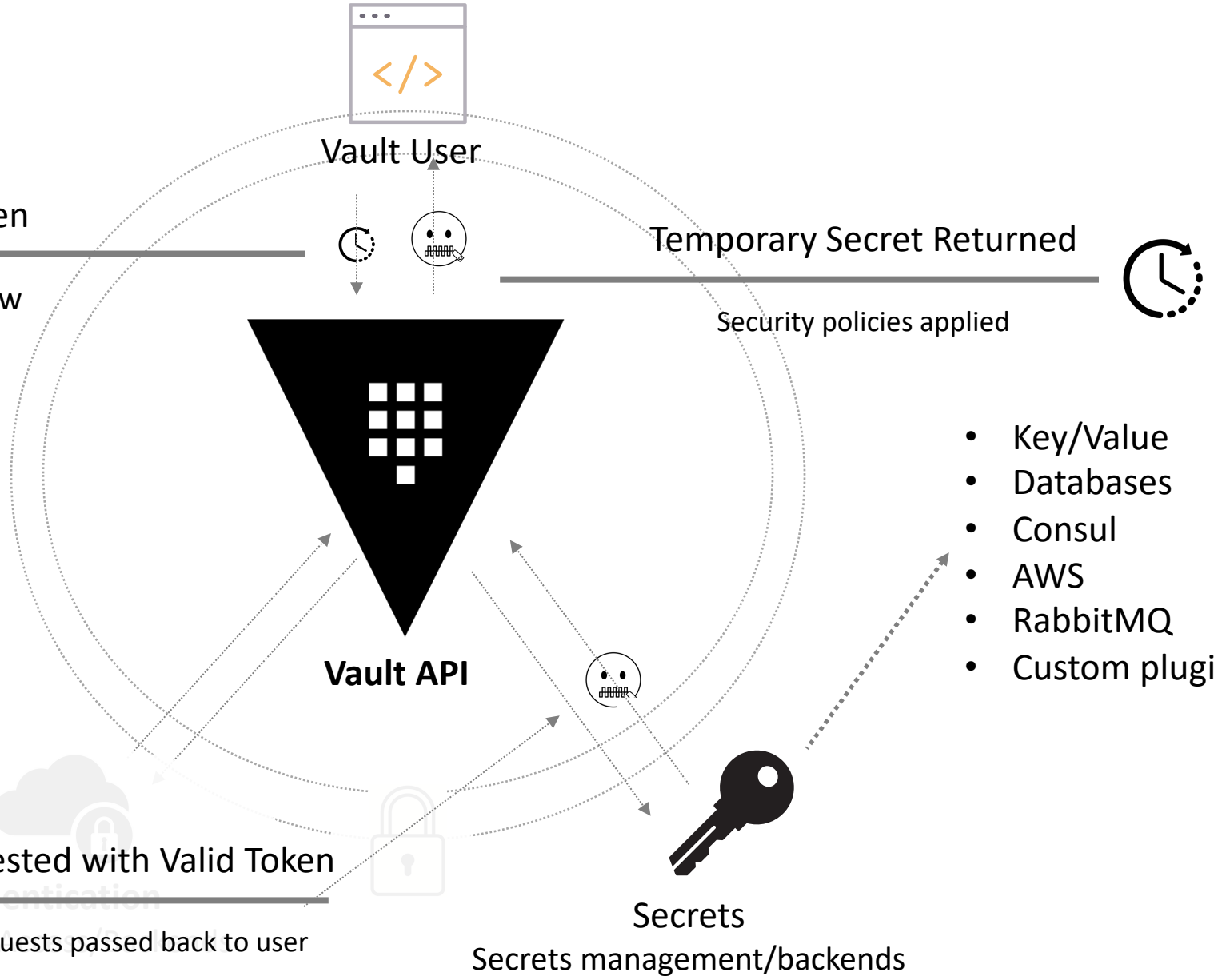
**Policies with Vault**
Vault uses policies to manage and safeguard access and secret distribution to applications and infrastructure. Policies provide a declarative way to **grant** and **deny** access to operations and paths.



# Authentication Workflow



# Secrets Workflow

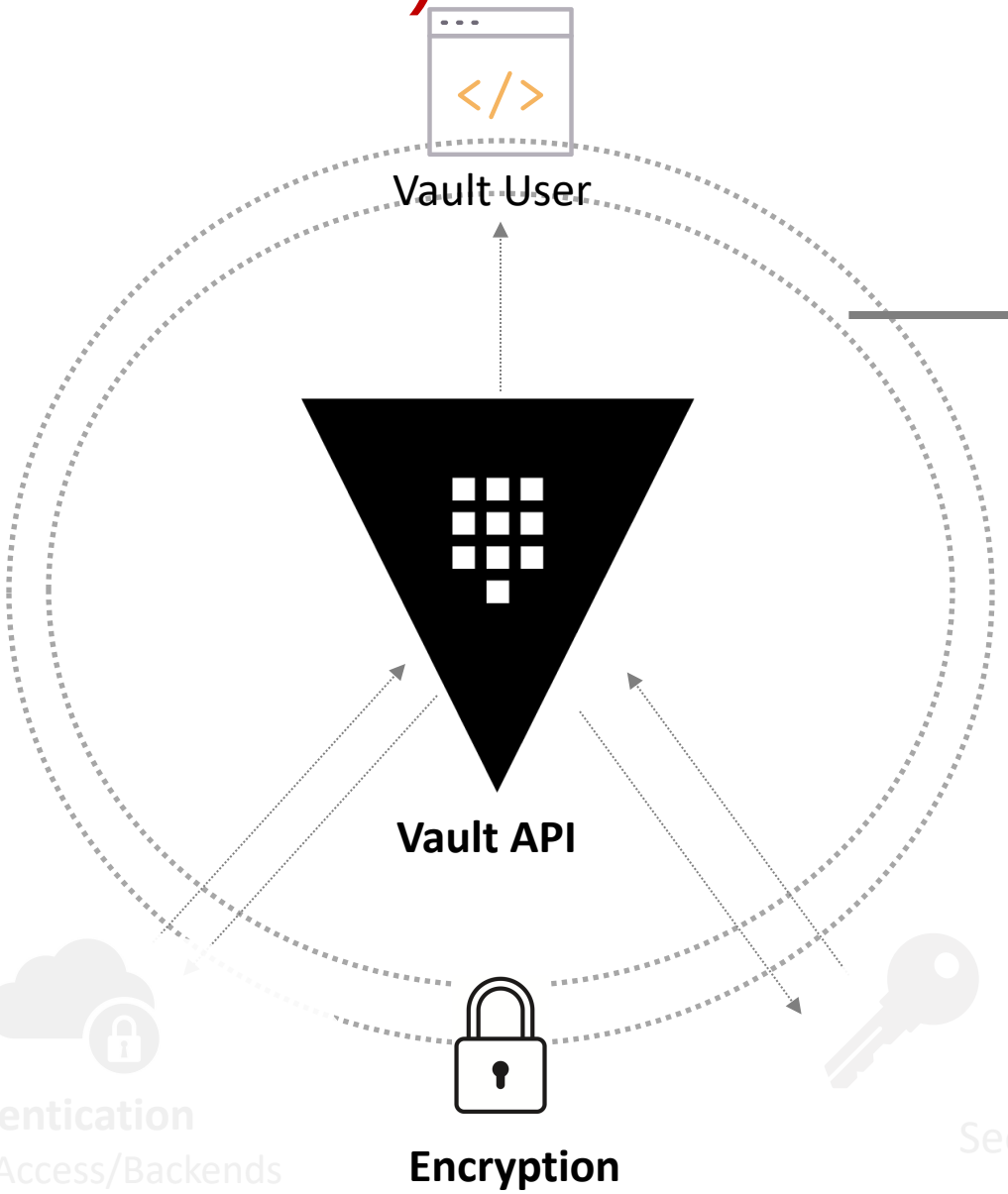



**Vault + Secrets**

*Authentication Token Required*

Secrets can be stored and generated. Some secrets can be generated dynamically, while others are verbatim. Secrets are returned to the user/client with any defined and/or appropriate policies.

# Encryption(as a Service)




**Encrypt everything**
Vault believes that everything should always be encrypted. Vault uses ciphertext wrapping to encrypt all data at rest and in-flight. This minimizes exposure of secrets and sensitive information.

**Full Data Encryption**

Data is encrypted in-flight and at rest.

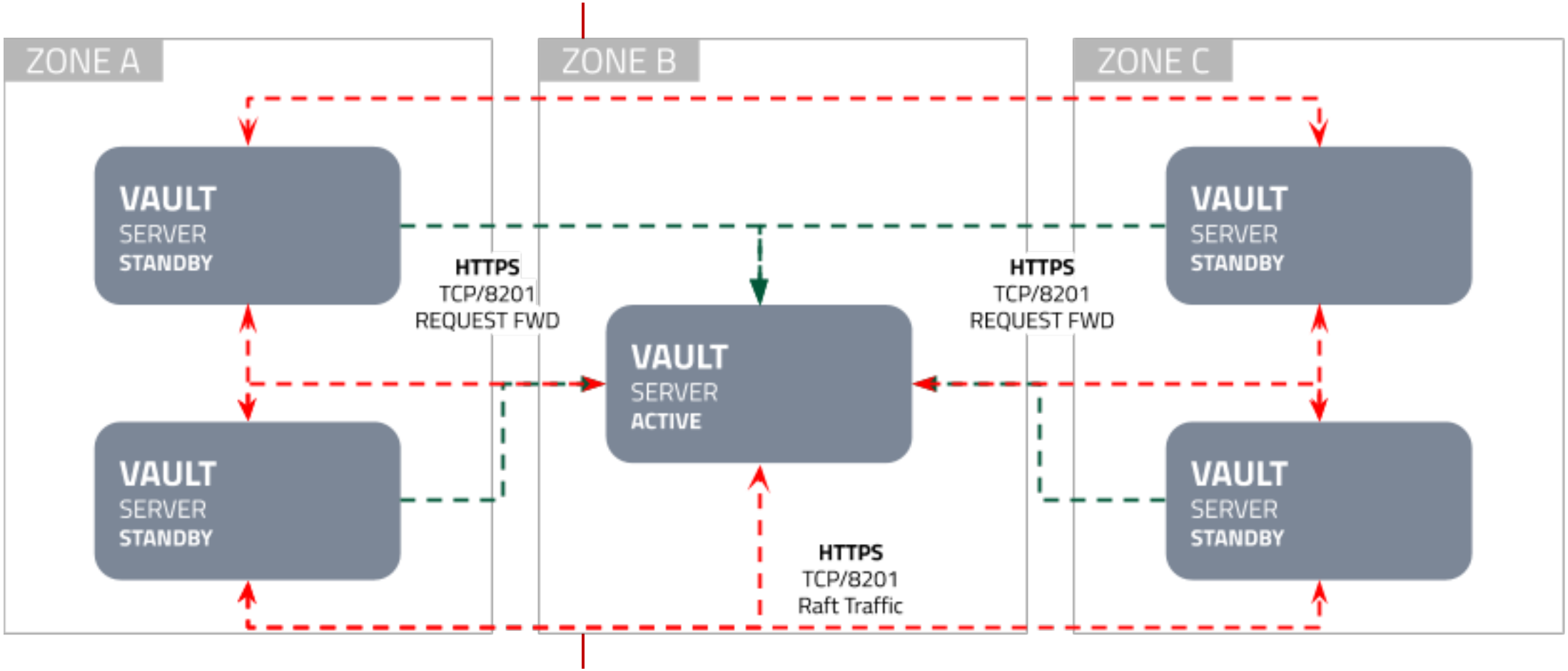
# Server

- Vault server provides an HTTP API which clients interact with and manages the interaction between all backends, ACL enforcement, and secret lease revocation.
- Vault server requires a storage backend so that data is available across restarts.
- Vault server starts the HTTP API exposed on start so that clients can interact.

# Seal/Unseal

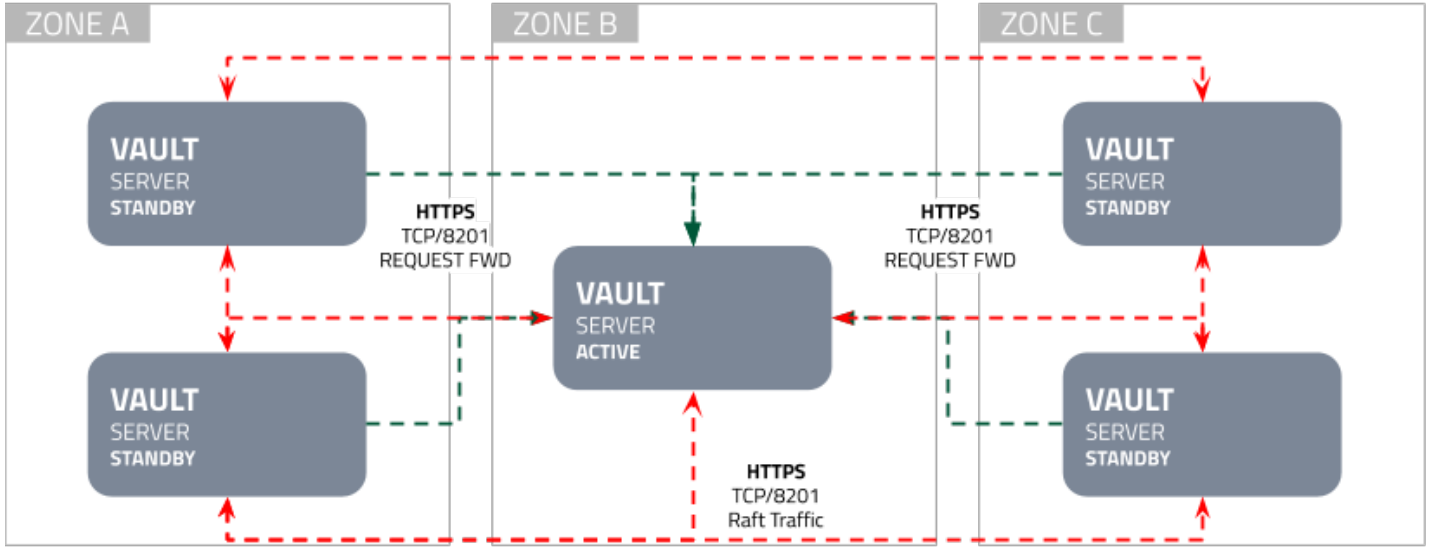
- When a Vault server is started, it starts in sealed state. It must be unsealed by passing the unseal keys before it can decrypt data.
- Unsealing is the process of constructing the master key necessary to read the decryption key to decrypt the data.
- The data stored by Vault is encrypted with an encryption key
- The encryption key is encrypted by a master key

# Architecture - Network connectivity



# Architecture - Network connectivity

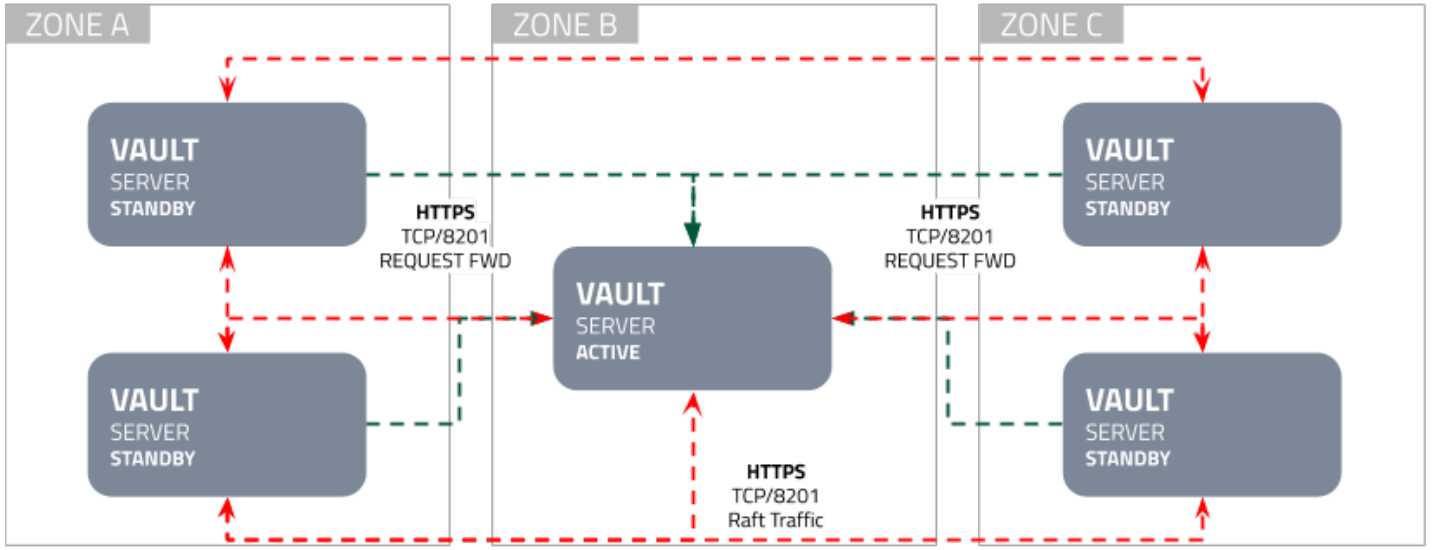
- This diagram shows the network flow between components in Vault cluster



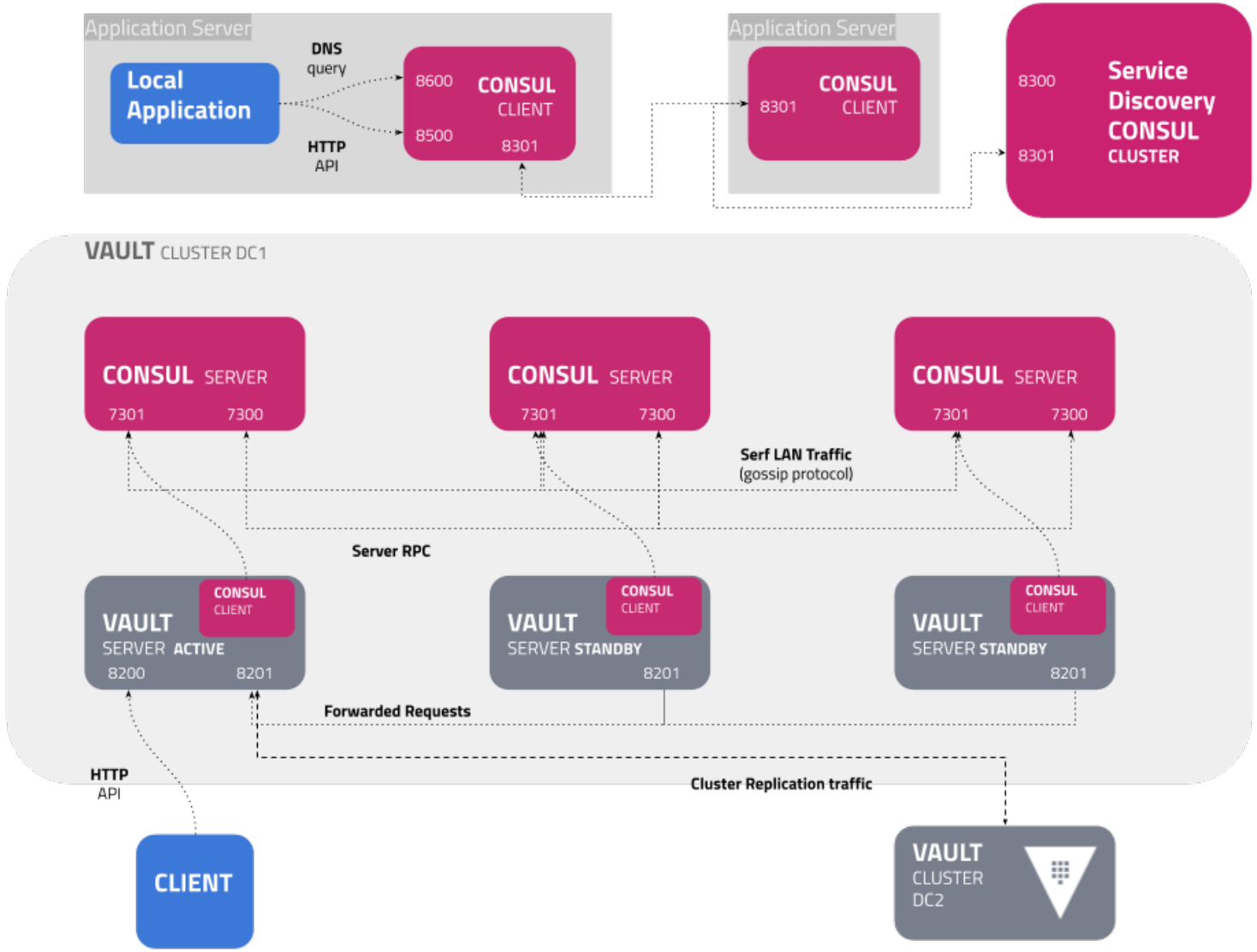

| Source | Destination | port | protocol | Direction | Purpose |
|---|---|---|---|---|---|
| Vault clients | Vault servers | 8200 | TCP | incoming | Vault API |
| Vault clients | Vault servers | 8201 | TCP | bidirectional | Vault replication traffic, request forwarding |
| Vault clients | Vault servers | 8201 | TCP | bidirectional | Raft gossip traffic |

# Communication: Gossip

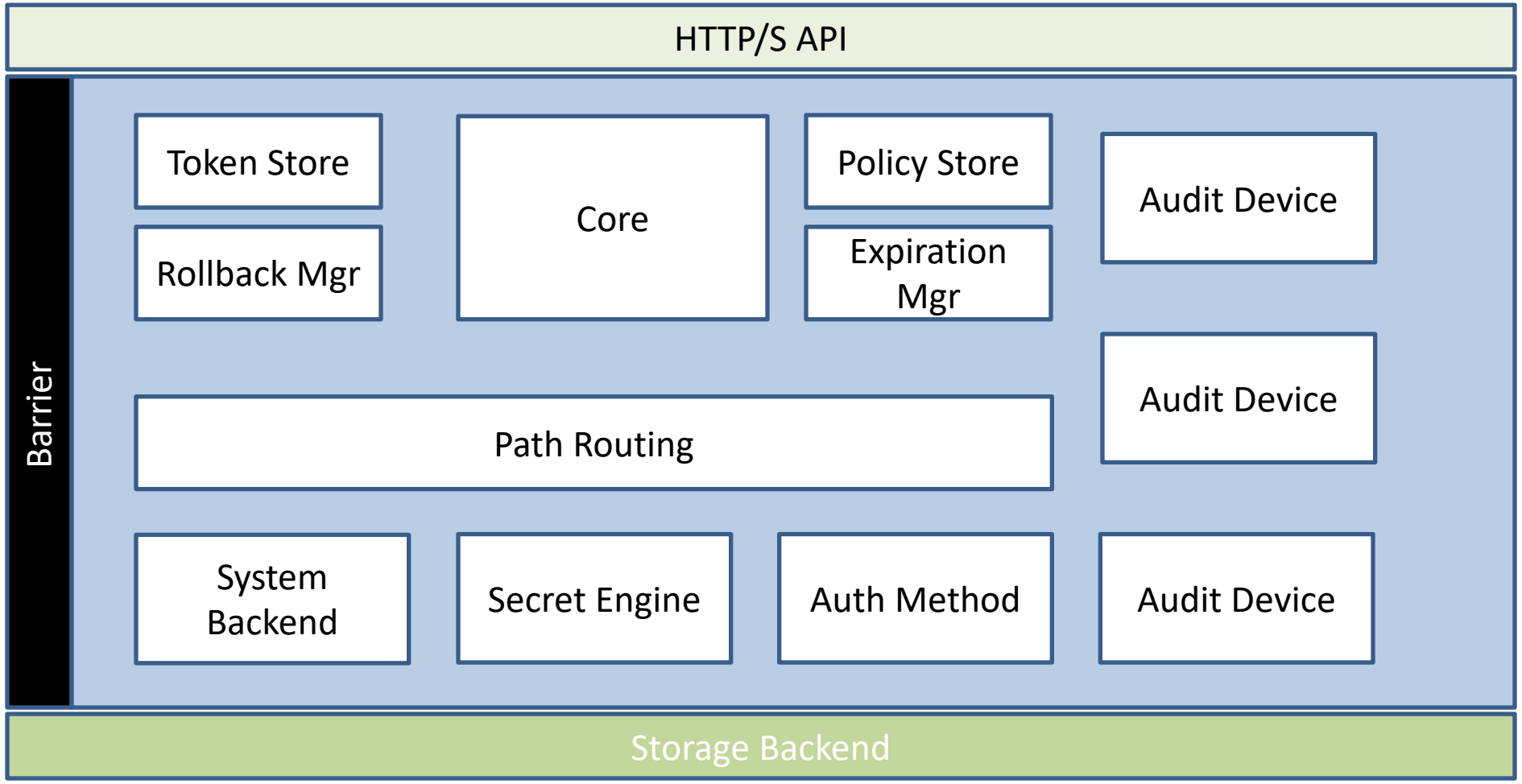
- Gossip protocol is used for communication between Vault servers. It is designed to be simple.
- Each node updates the nodes around it, and this continues until all nodes are in sync.



# Architecture - Consul

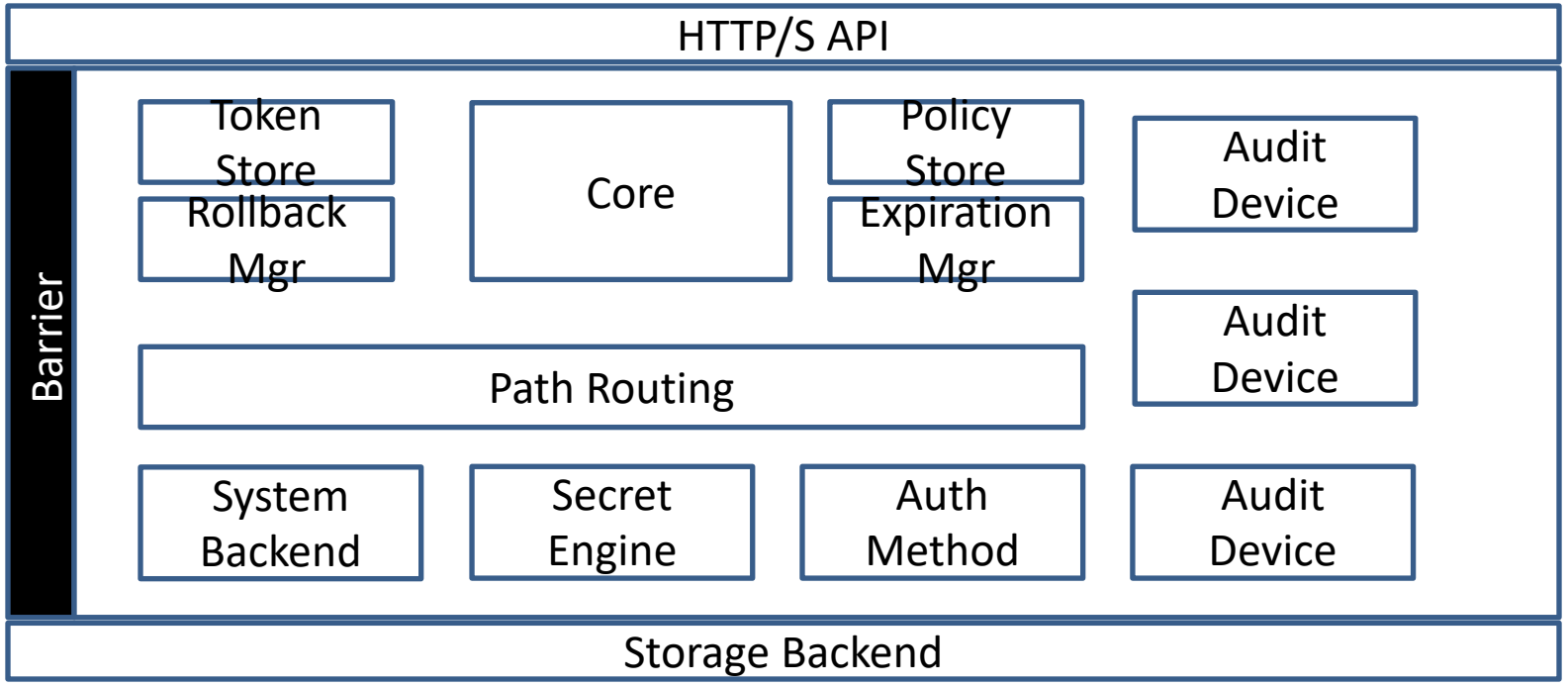


# Architecture - Internal



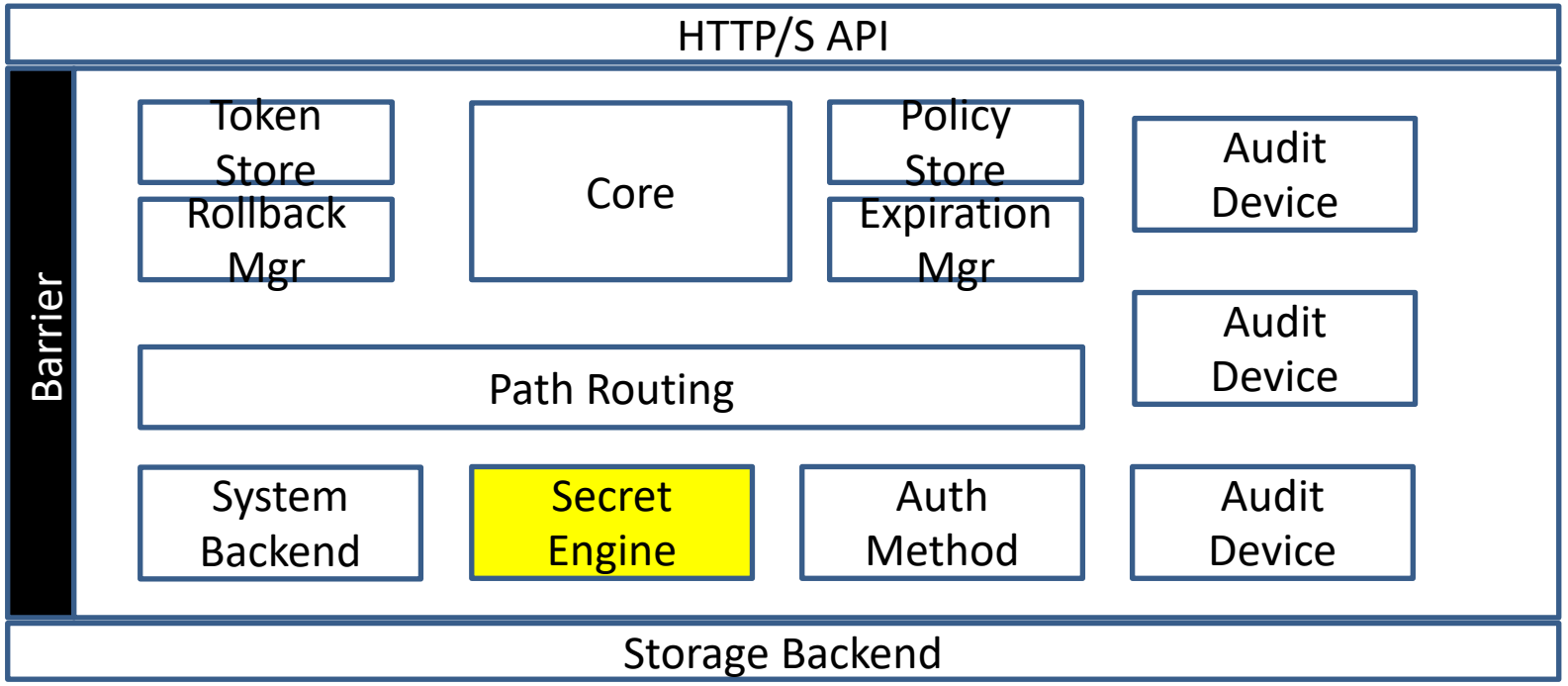
# Barrier

- The **barrier** is a cryptographic seal around the Vault
- All data that flows between Vault and the storage backend passes through the barrier.



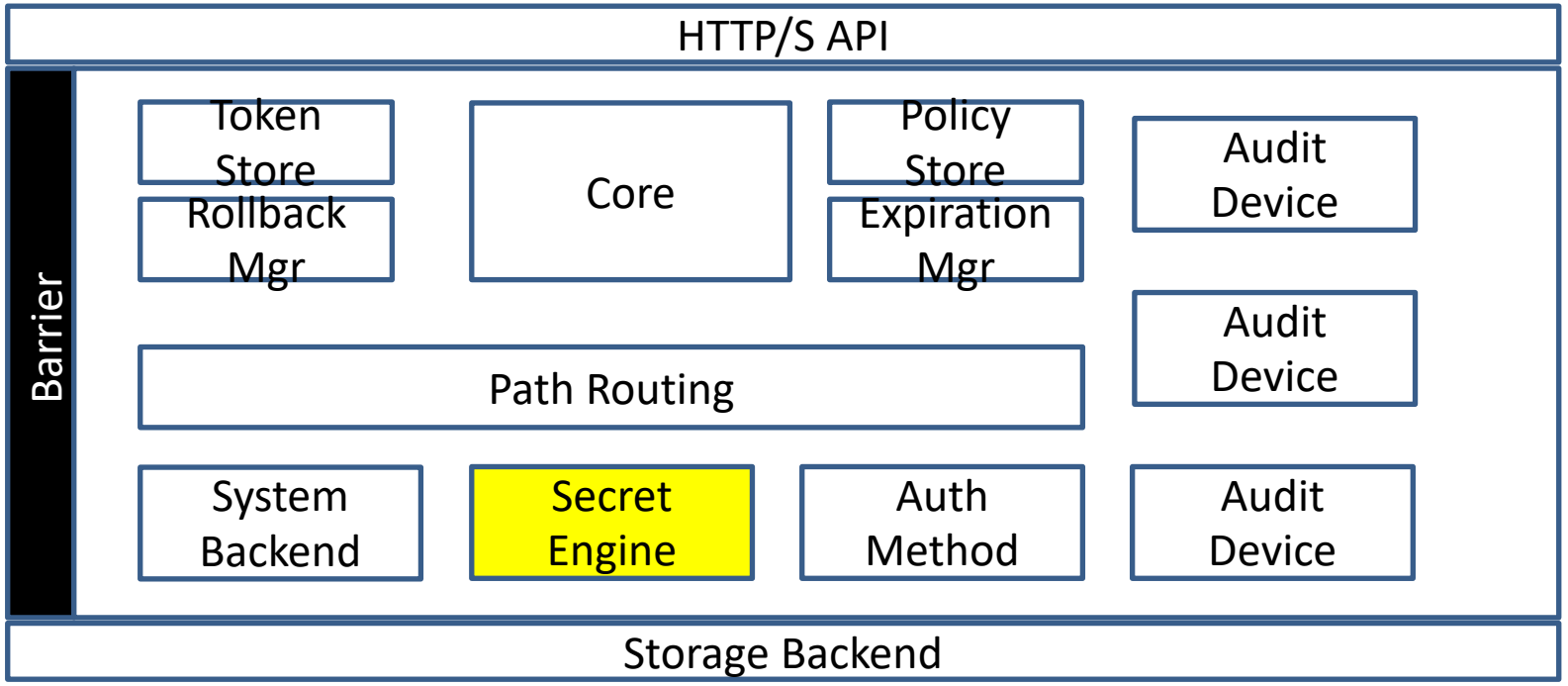
# Secret Engine

- A **secret engine** stores, generates, or encrypts data
  - responsible for managing secrets
- Some secret engines behave like encrypted key-value stores while others dynamically generate secrets when queried.



# Secret Engine

- A Vault cluster can have **multiple secret engines**
  - Consul
  - AWS
  - Key-Value
  - Custom

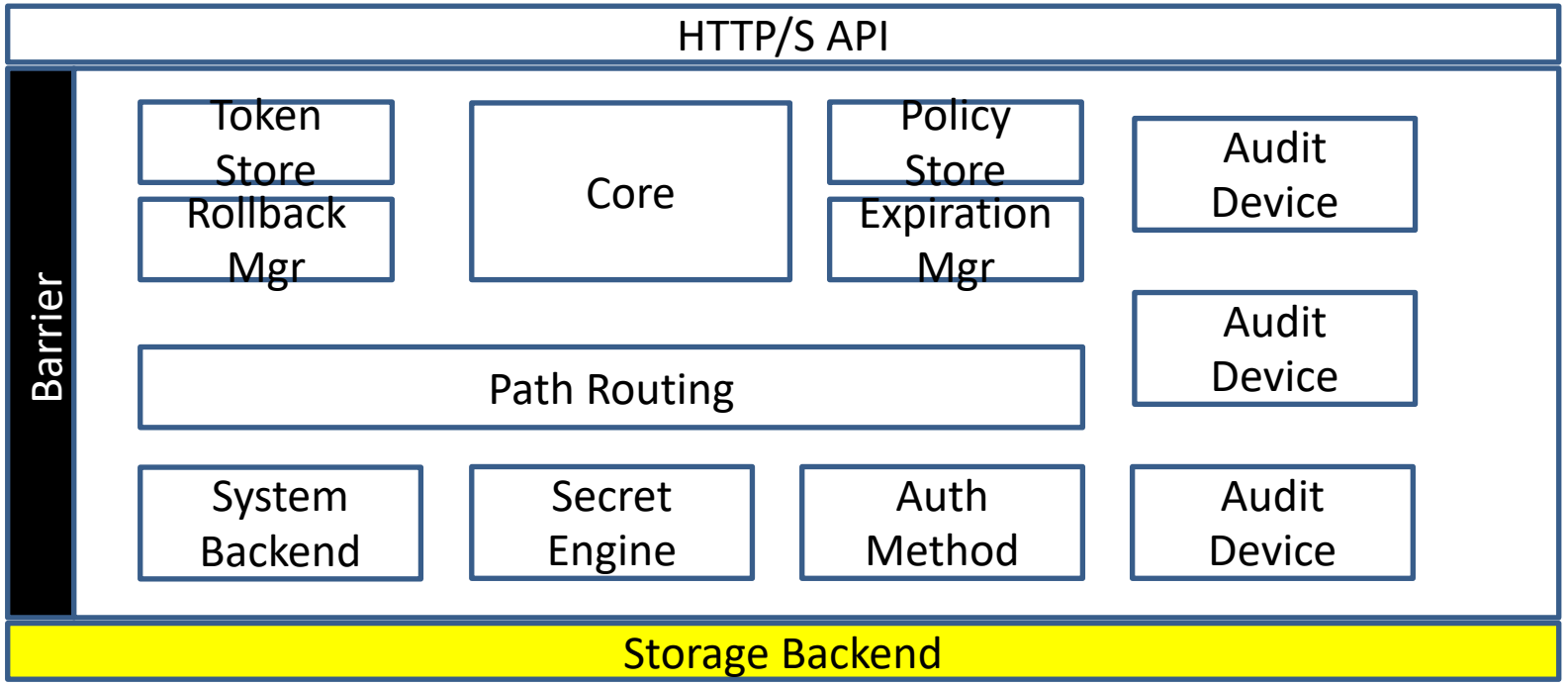


# Secret Engine

- A **secret** is:
  - Any sensitive data that is acquired by unauthorized party would cause political, financial, or appearance harm to an organization
  - Anything stored or returned by Vault that contains confidential or cryptographic material
  - Secrets have an associated lease

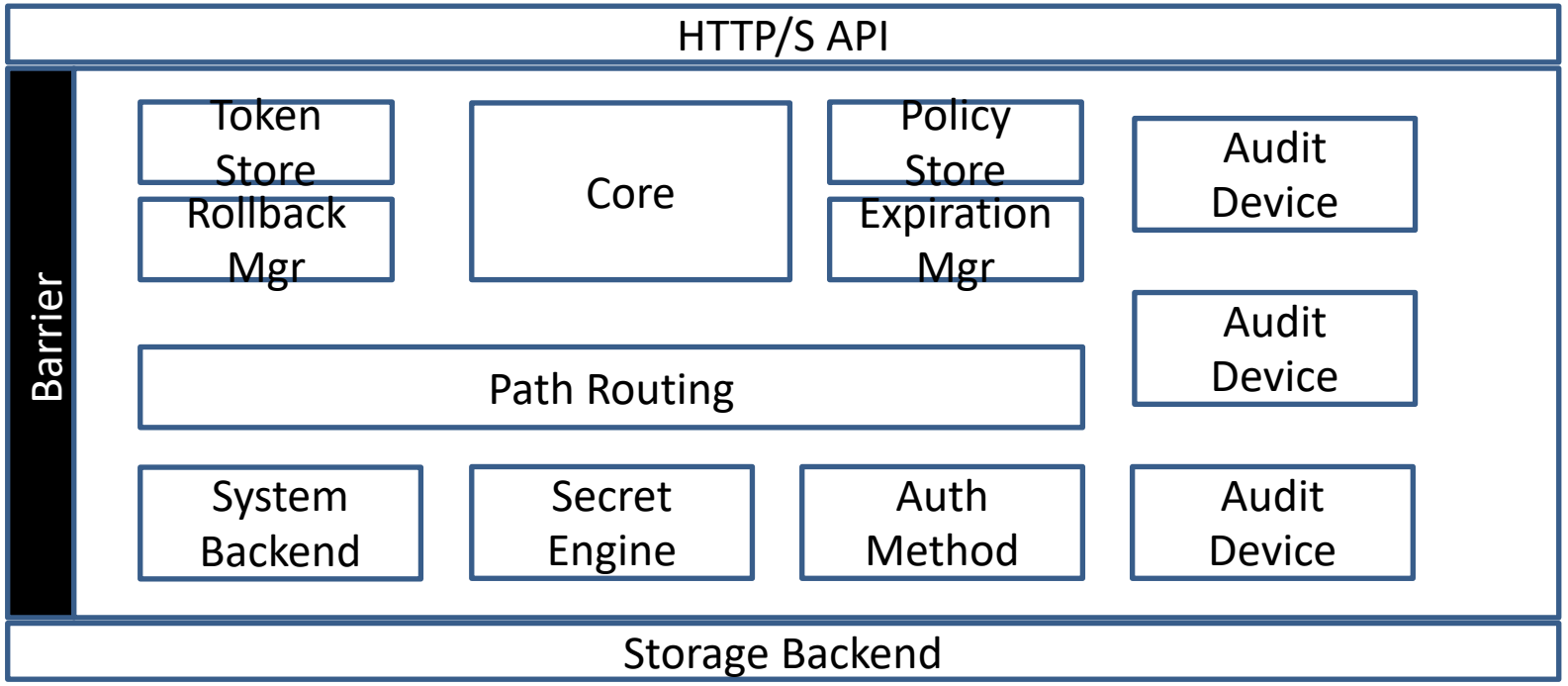
# Storage Backend

- The storage backend is responsible for durable storage of encrypted data
  - There is only one storage backend per Vault cluster
- Data is encrypted in-transit and at-rest with 256 bit AES
- Storage backend examples:
- Consule
- Amazon S3
- MySQL
- in-memory



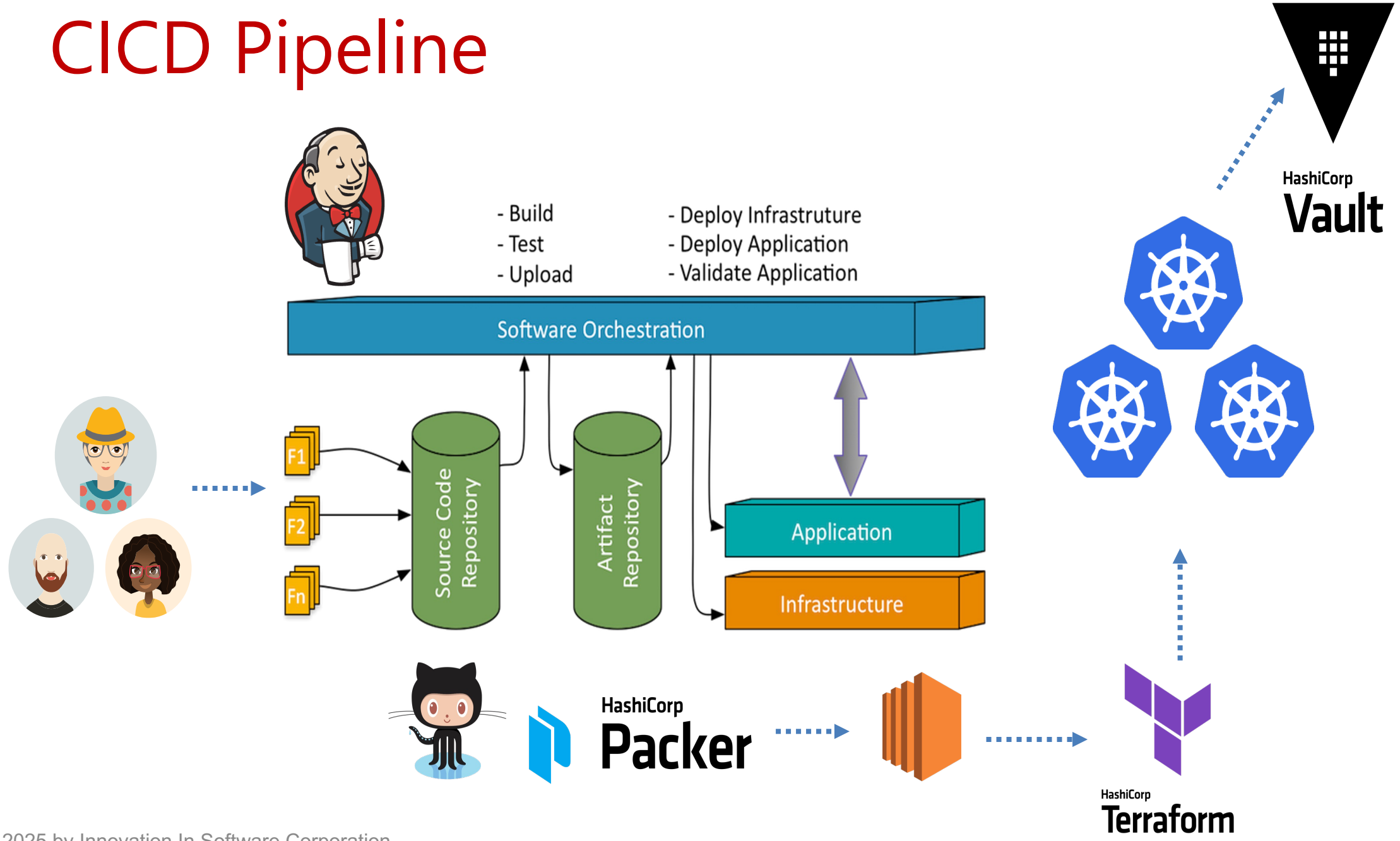
# Vault Resources

- Everything in Vault is path based
- Example:
  - auth/ldap/...
  - aws/roles/...
  - nomad/config/access
  - database/config
  - sys/health
  - etc.

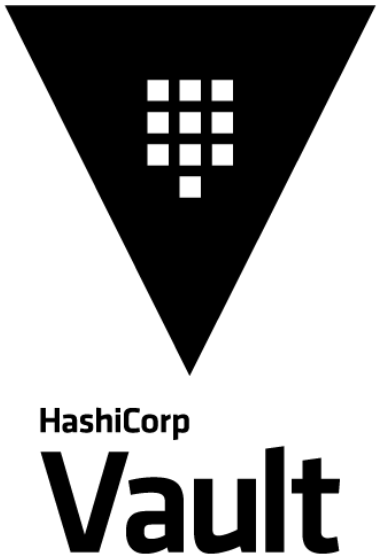


# Putting it all together!

# CICD Pipeline



# Why HCP Vault Dedicated?

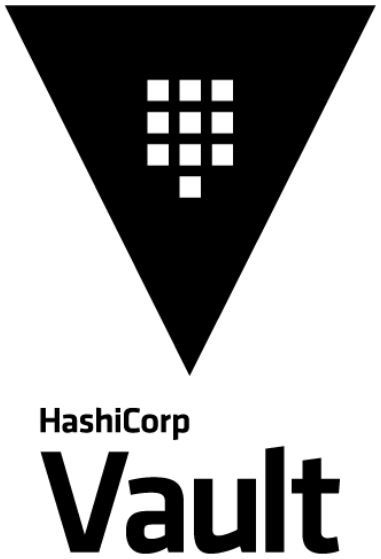



HCP Vault Dedicated provides significant benefits over self-managed Vault deployments. It reduces operational overhead through push-button deployment, fully managed upgrades, and automated backups.

Benefits include:

- Reduce operational overhead with push-button deployment and managed upgrades
- Increase security across clouds and machines through unified control
- Control costs by centralizing secrets management
- Day zero readiness for modern cloud security
- Reliability backed by HashiCorp's enterprise expertise
- Ease of use for quick onboarding of applications and teams

# HCP Vault Dedicated vs Self-Managed

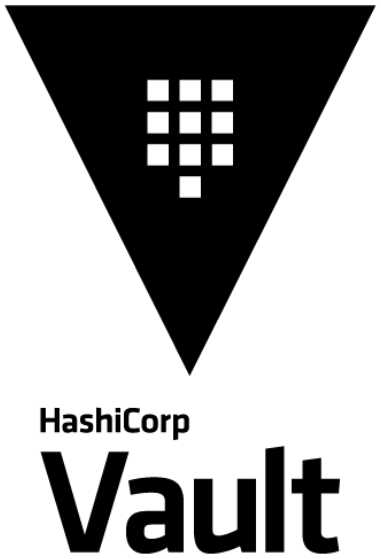



HCP Vault Dedicated provides significant advantages over self-managed Vault deployments while maintaining feature parity for most enterprise capabilities. The service automatically handles critical operational tasks that require manual effort in self-managed environments.

Operational differences:

- Automatic vs manual version upgrades
- Built-in vs manual audit logging configuration
- Integrated storage vs manual storage backend management
- Auto-unseal with KMS vs manual unseal key management
- Dynamic scaling vs no built-in scaling features
- Portal-generated admin tokens vs manual initialization
- Built-in HA/DR vs manual setup requirements

# HCP Vault Dedicated Security



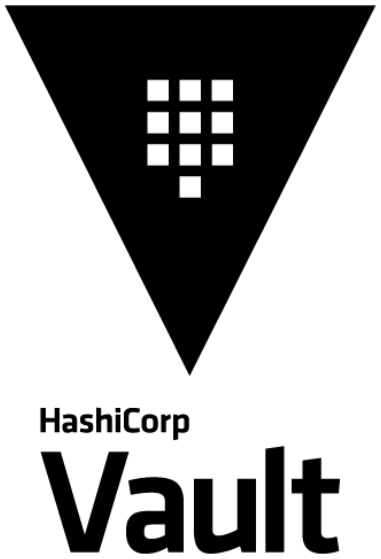

HCP Vault Dedicated implements enterprise-grade security with unique KMS keys per cluster, end-to-end TLS encryption, and single-tenant dedicated clusters.

The service provides automated security features including daily snapshots, comprehensive audit logging, and production hardening guidelines. All data is encrypted and stored in customer-specific storage with full audit trails.

Security benefits:

- Unique KMS keys with full audit trails
- End-to-end TLS with auto-rotation
- Single-tenant dedicated clusters
- Daily automated snapshots
- Comprehensive audit logging

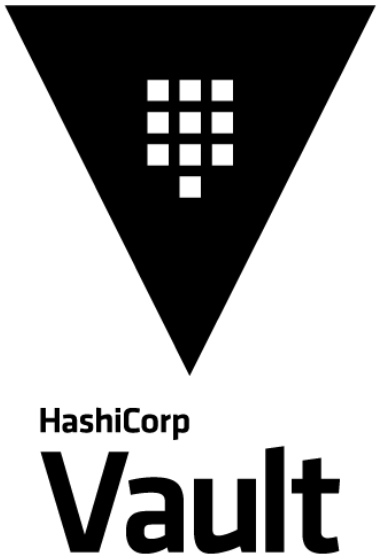
# HCP Vault Dedicated Integrations




HCP Vault Dedicated supports comprehensive integrations with enterprise systems and cloud platforms. The service includes all major HashiCorp official plugins for authentication methods, secrets engines, and database connections.

The platform supports integrations with major cloud providers including AWS, Azure, and GCP, along with enterprise authentication methods like LDAP, OIDC, and Kubernetes. Database integrations cover popular systems including PostgreSQL, MySQL, Oracle, and Microsoft SQL Server.

# HCP Vault Dedicated High Availability




HCP Vault Dedicated provides enterprise-grade high availability and disaster recovery capabilities through its highly-available, single-tenant data plane architecture. All production-tier clusters consist of 3 highly available Vault nodes leveraging Vault Enterprise's performance standby capability.

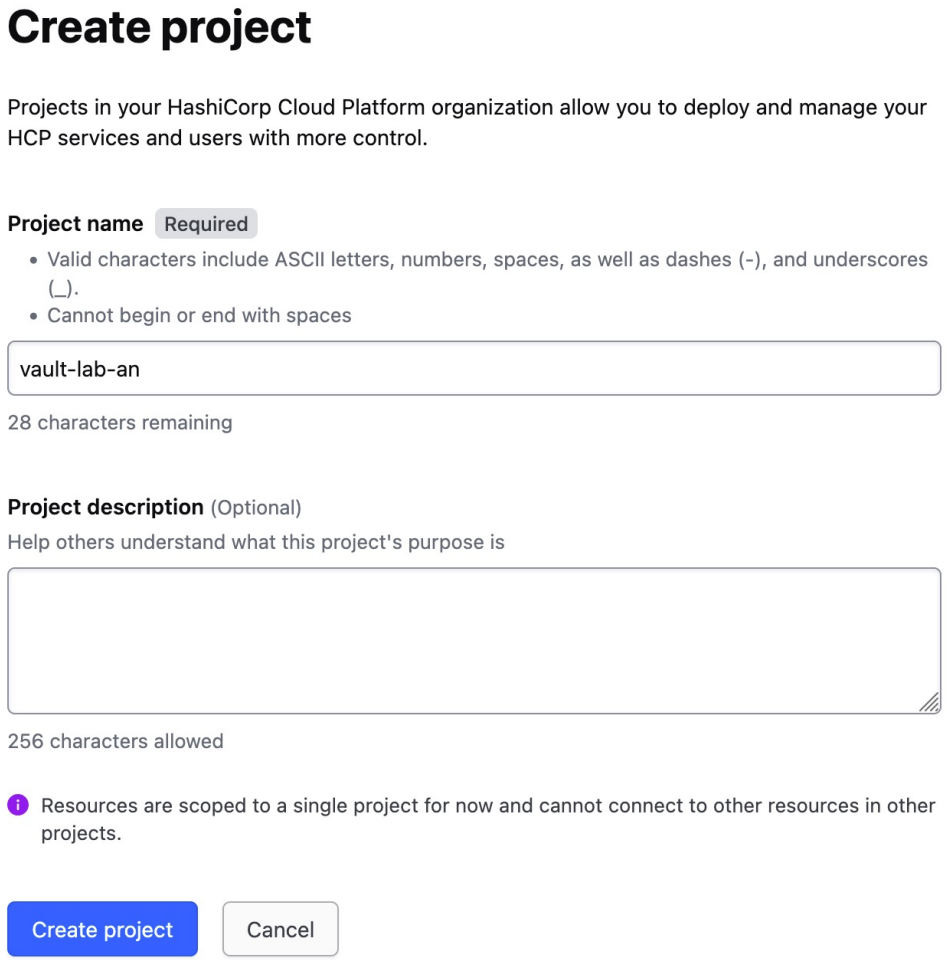
Key HA/DR features:

- 3-node highly available clusters
- Cross-region disaster recovery
- Automatic failover capabilities
- 30-day snapshot retention
- Platform outage resilience

# Understanding HCP Projects

Before creating a Vault cluster, you need to understand what a project is in HCP. A project is a logical container that groups related resources together, providing organization and isolation for your infrastructure components.

Think of a project like a folder that keeps all your related resources organized - similar to having separate folders for different environments or teams. Projects help you manage access control, billing, and resource organization within your HCP organization.

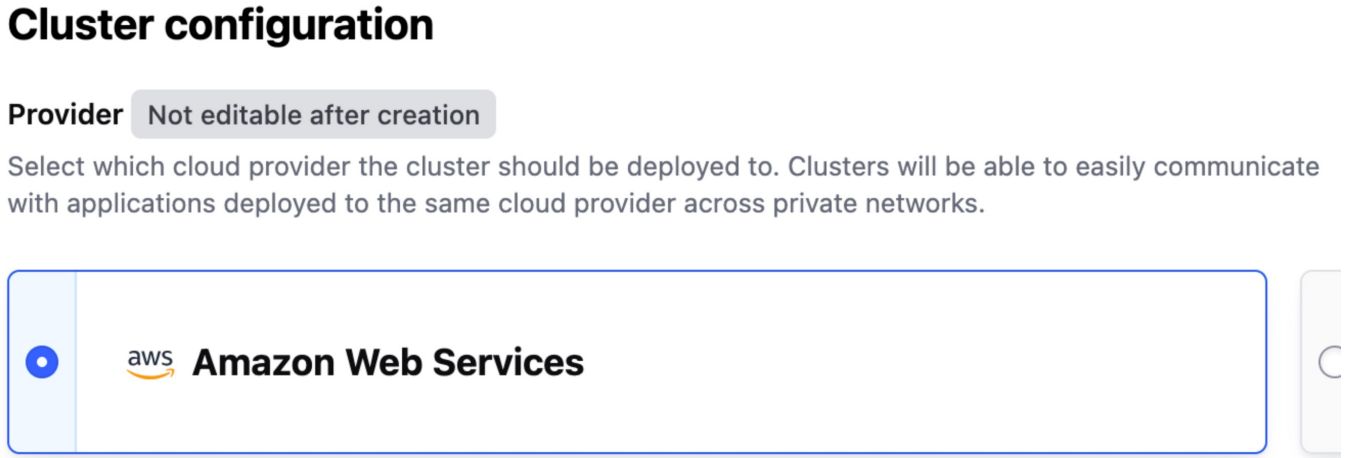


# Creating an HCP Vault Cluster

The cluster creation wizard guides you through selecting your preferred cloud provider from AWS, Azure, or GCP, then configuring cluster settings including tier selection, sizing, and network configuration.

Key configuration options:

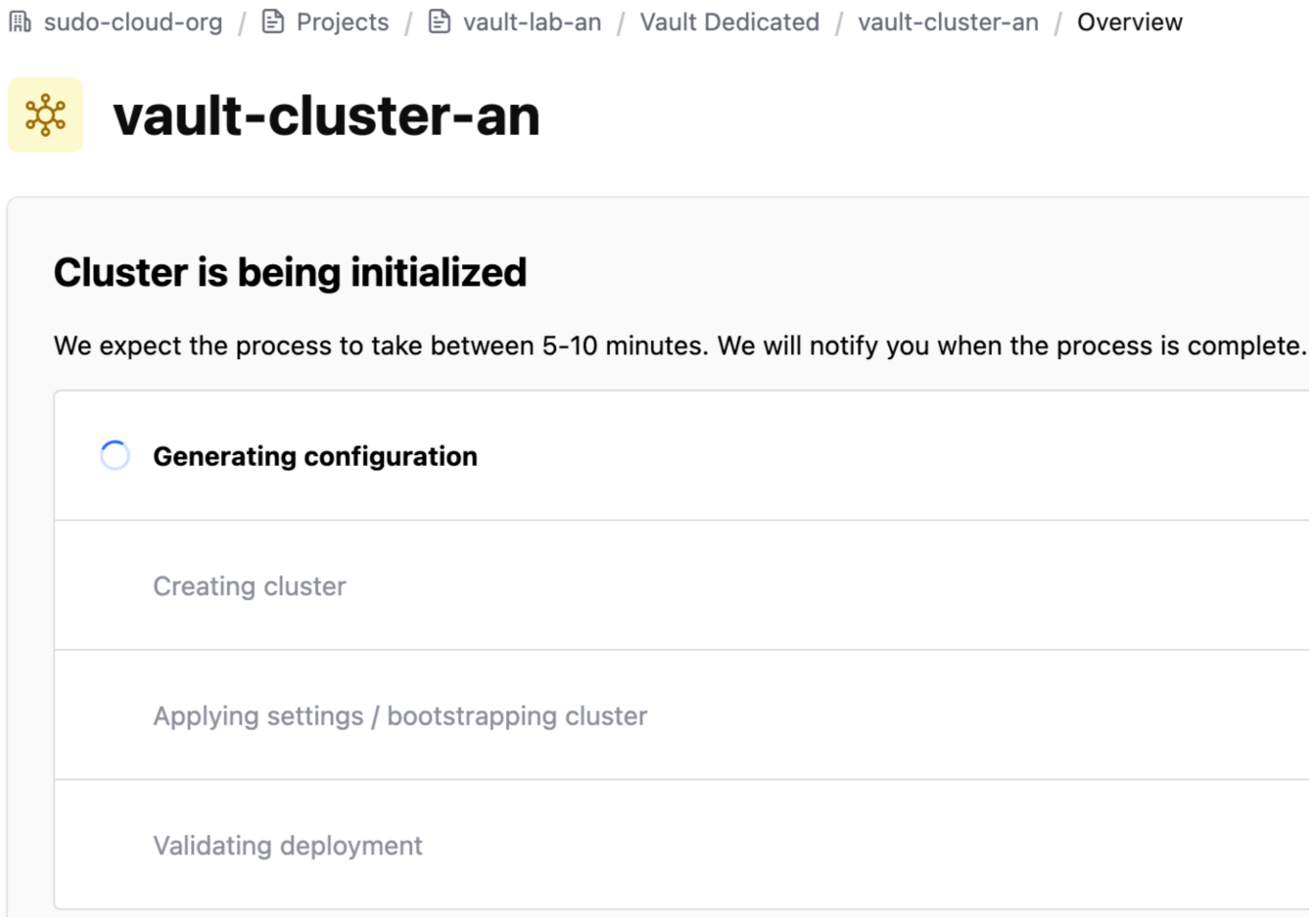
- Cloud provider selection (AWS, Azure, GCP)
- Cluster tier and sizing options
- Network ID configuration with initials
- Region and CIDR block settings



# Cluster Initialization Process

The cluster initialization process typically takes 5-10 minutes as HCP automatically handles all the underlying infrastructure setup and Vault configuration. During this time, HCP provisions the necessary compute resources, configures networking and security settings, and initializes the Vault cluster.

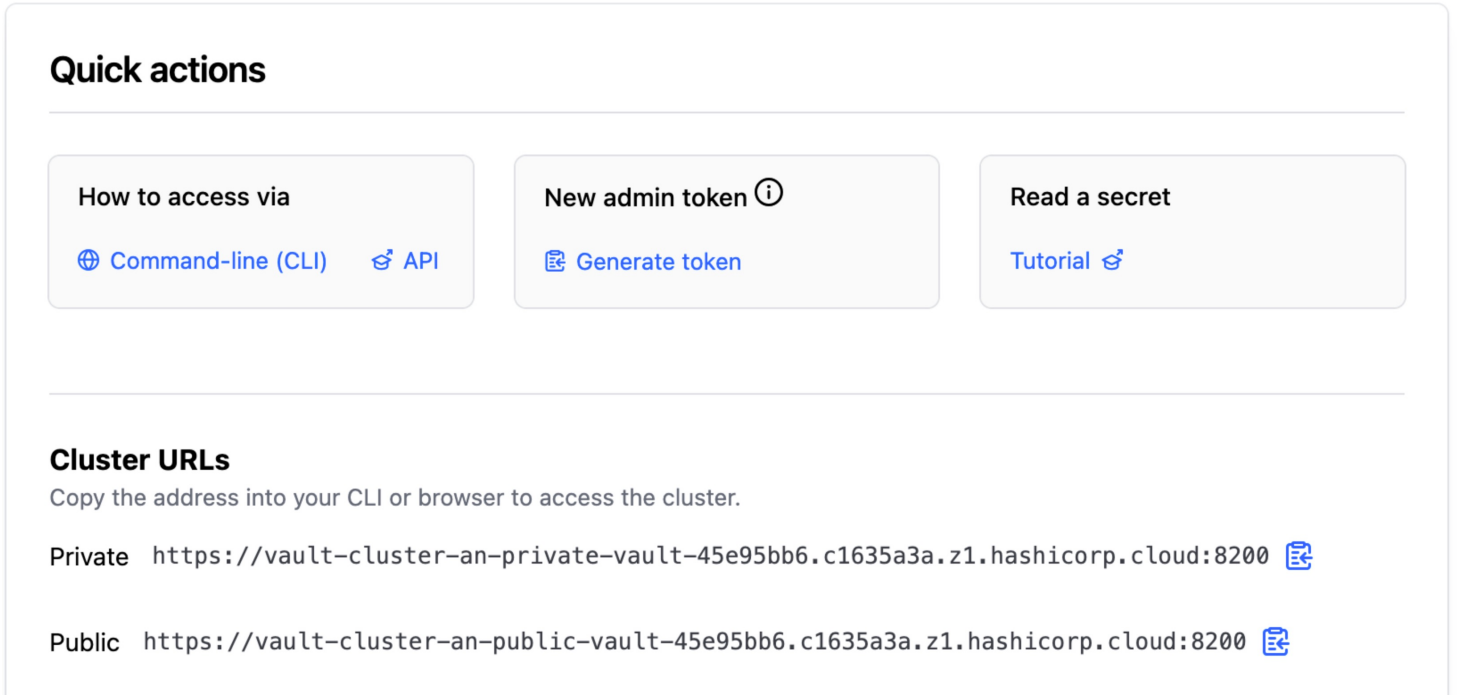
You can monitor the progress on the cluster overview page, which provides real-time status updates throughout the deployment process.



# Cluster Access Methods

HCP Vault Dedicated clusters can be configured with either public or private access depending on your security requirements. Public access allows you to connect to Vault from any internet-connected device, while private access requires connection through cloud provider networking.

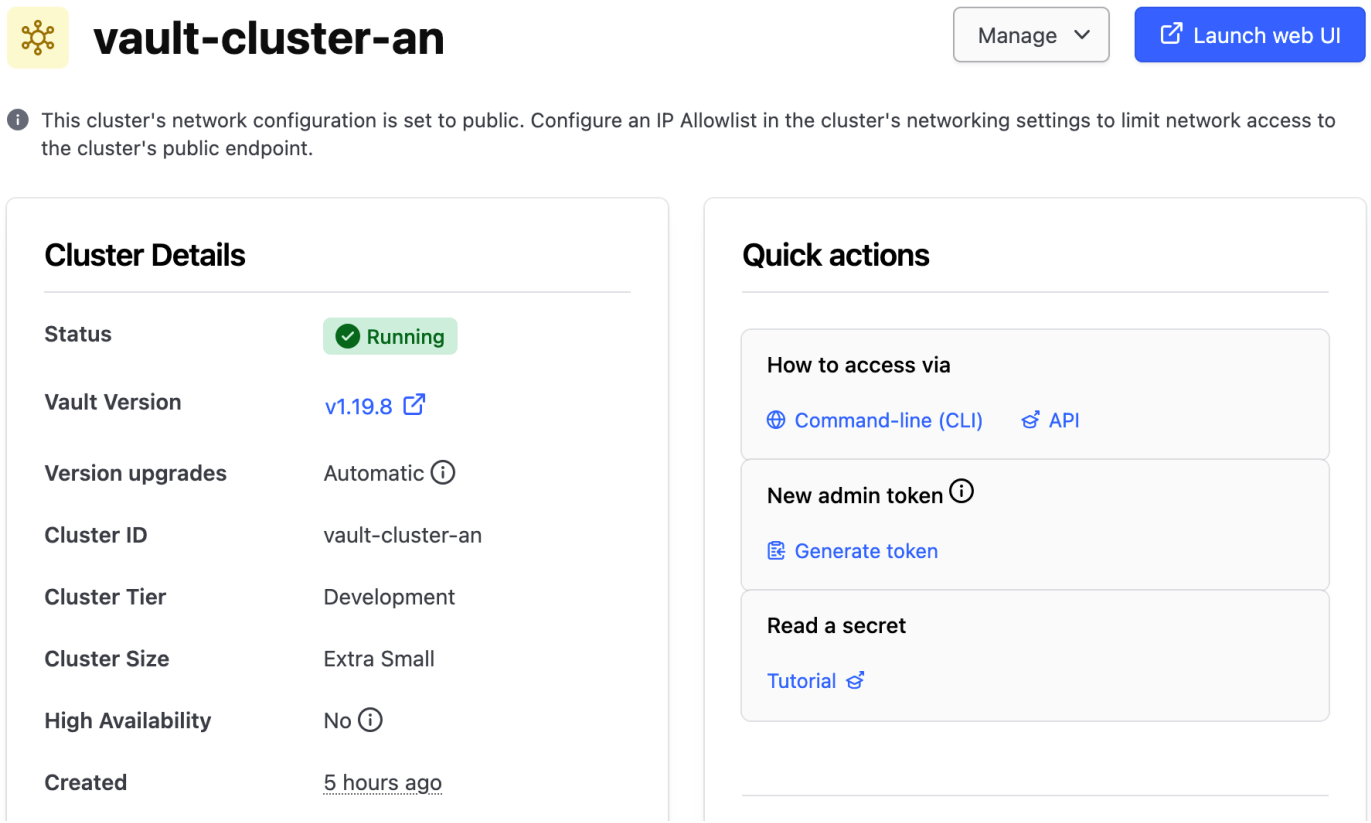
For public access, you can configure an IP allow list to limit which IPv4 addresses or CIDR ranges can connect to Vault. Private access requires VPC peering connections, transit gateways, or virtual network peering depending on your cloud provider.



# Web UI Access

The primary method for accessing your HCP Vault cluster is through the web UI, which provides a user-friendly interface for managing Vault operations. This interface offers intuitive navigation and visual management of secrets, authentication methods, and policies.

To access the web UI, navigate to the Overview page of your cluster and click Generate token in the New admin token card. Copy the generated token, then click Launch web UI to open the Vault interface in a new tab.

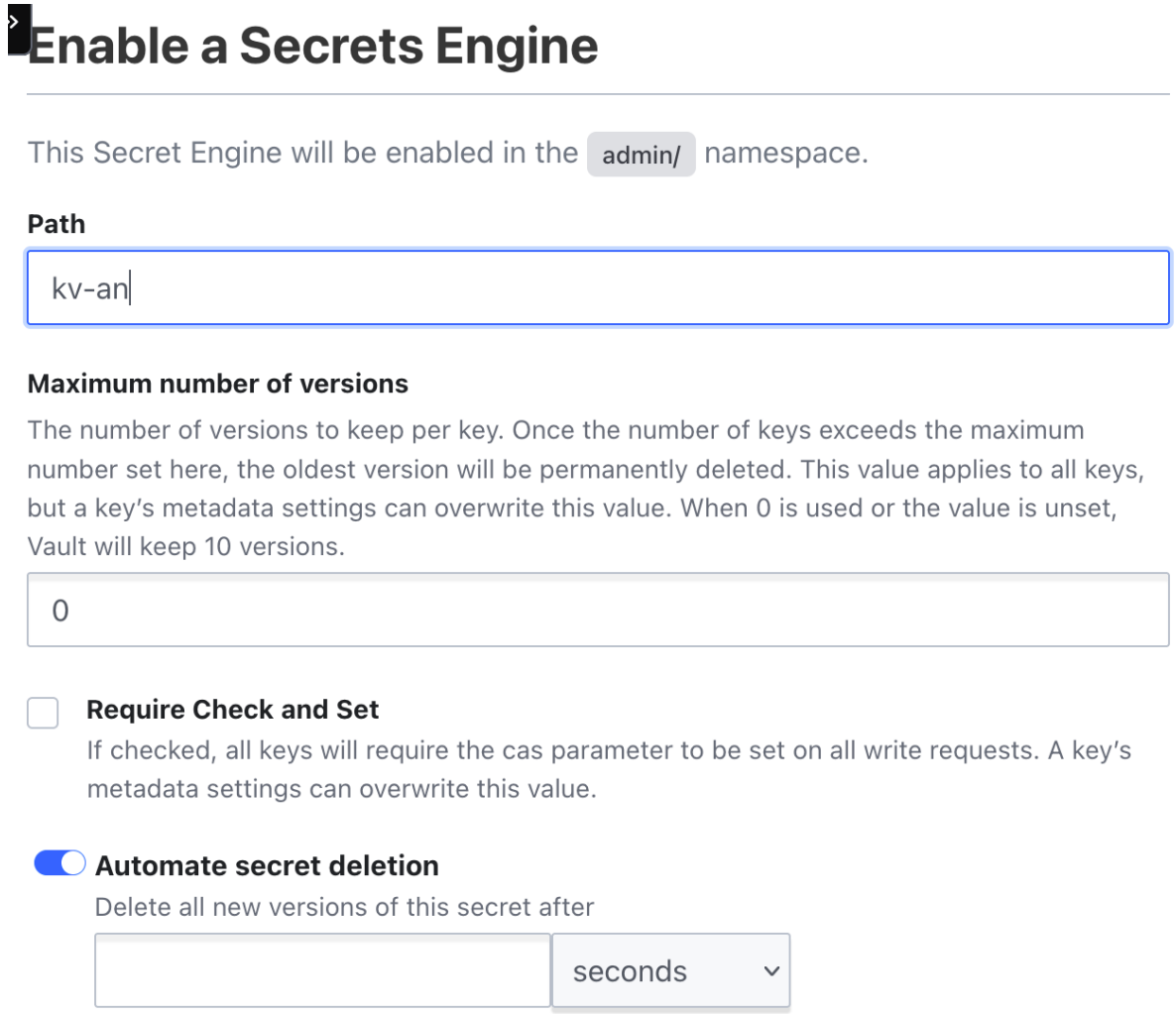


# Enabling KV Secrets Engine

The KV engine configuration involves setting a custom path using initials for uniqueness and selecting version 2 for enhanced features. The engine provides secure encryption at rest, audit logging integration, and flexible access control policies.

Configuration options:

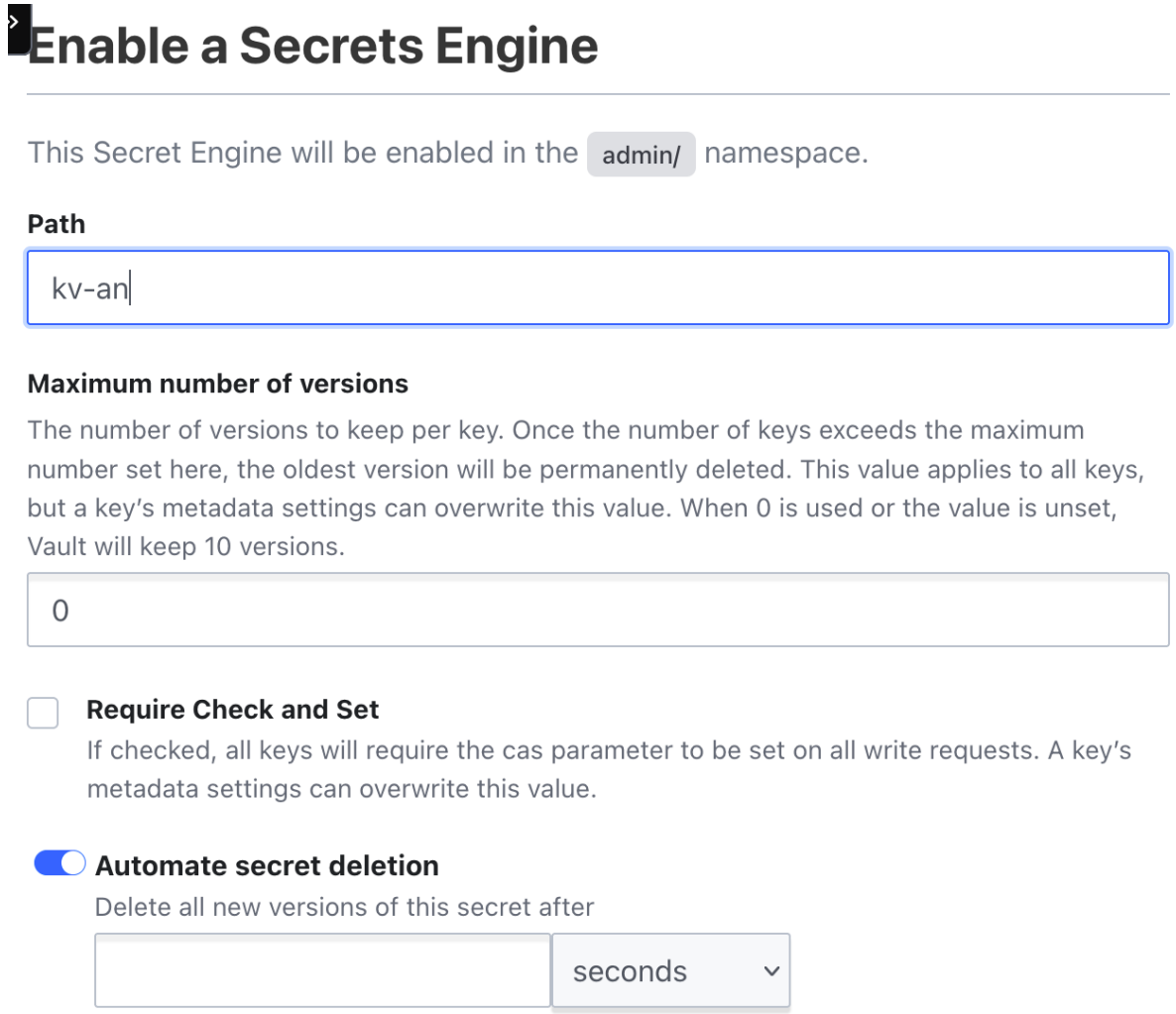
- Custom path with initials for uniqueness
- Maximum number of versions per key
- Require Check and Set parameter
- Default and maximum lease TTL settings
- Audit logging exclusions and headers



# Storing AWS Credentials

Once the KV secrets engine is enabled, you can begin storing sensitive data like AWS credentials securely. The KV engine allows you to create structured secrets with multiple key-value pairs within a single path, providing organized storage for related credentials.

To store AWS credentials, navigate to your KV engine and create secrets using the path aws/credentials. Store the access key ID and secret access key as separate keys within this path, allowing for structured access to credential components.



# HCP Terraform Integration Setup

Integrating HCP Vault with Terraform requires configuring the Vault provider in your Terraform configuration. This integration enables Terraform to retrieve secrets from Vault during infrastructure deployment, eliminating the need to hardcode sensitive data in your Terraform files.

The integration process involves adding the Vault provider to your required_providers block and configuring the provider with your cluster URL and authentication token.

```
# Update the existing required_providers block in main.tf
required_providers {
  aws = {
    source = "hashicorp/aws"
    version = "~> 3.76.0"
  }
  vault = {
    source  = "hashicorp/vault"
    version = "~> 4.0"
  }
}

# Configure the Vault provider
provider "vault" {
  address = "https://your-actual-vault-cluster-url"
  token   = var.vault_token
}
```

# Retrieving Secrets from Vault

Once the Vault provider is configured, Terraform can retrieve stored secrets using data sources. This approach replaces hardcoded credentials with dynamic secret retrieval from your HCP Vault cluster.

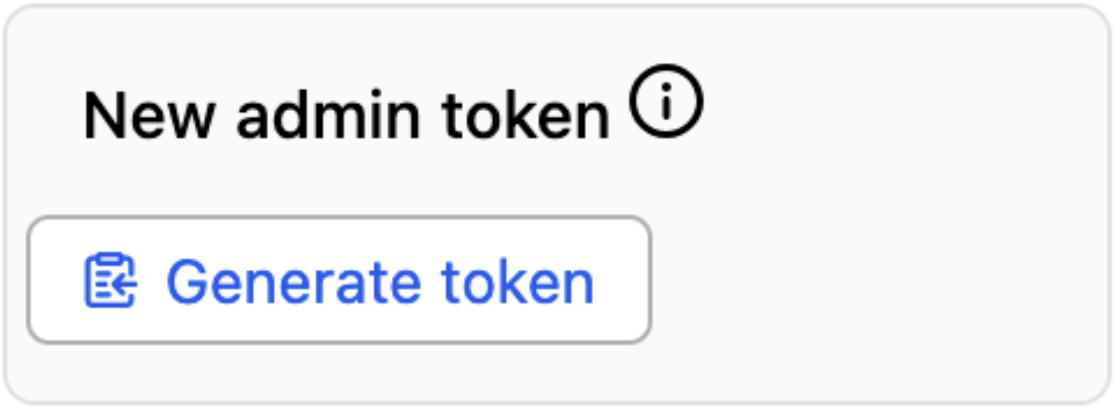
The vault_kv_secret_v2 data source connects to your KV secrets engine and retrieves the stored AWS credentials. Terraform then uses these credentials to authenticate with AWS, eliminating the need to store sensitive data in your Terraform files or workspace variables.

```
data "vault_kv_secret_v2" "aws_creds" {
  mount = "kv-{your-initials}"
  name  = "aws/credentials"
}

provider "aws" {
  region     = "us-west-1"
  access_key = data.vault_kv_secret_v2.aws_creds.data["access_key_id"]
  secret_key = data.vault_kv_secret_v2.aws_creds.data["secret_access_key"]
}
```

# Generating Vault Authentication Token

To enable Terraform to authenticate with your HCP Vault cluster, you need to generate an authentication token. This token provides the necessary permissions for Terraform to access secrets stored in your Vault cluster.

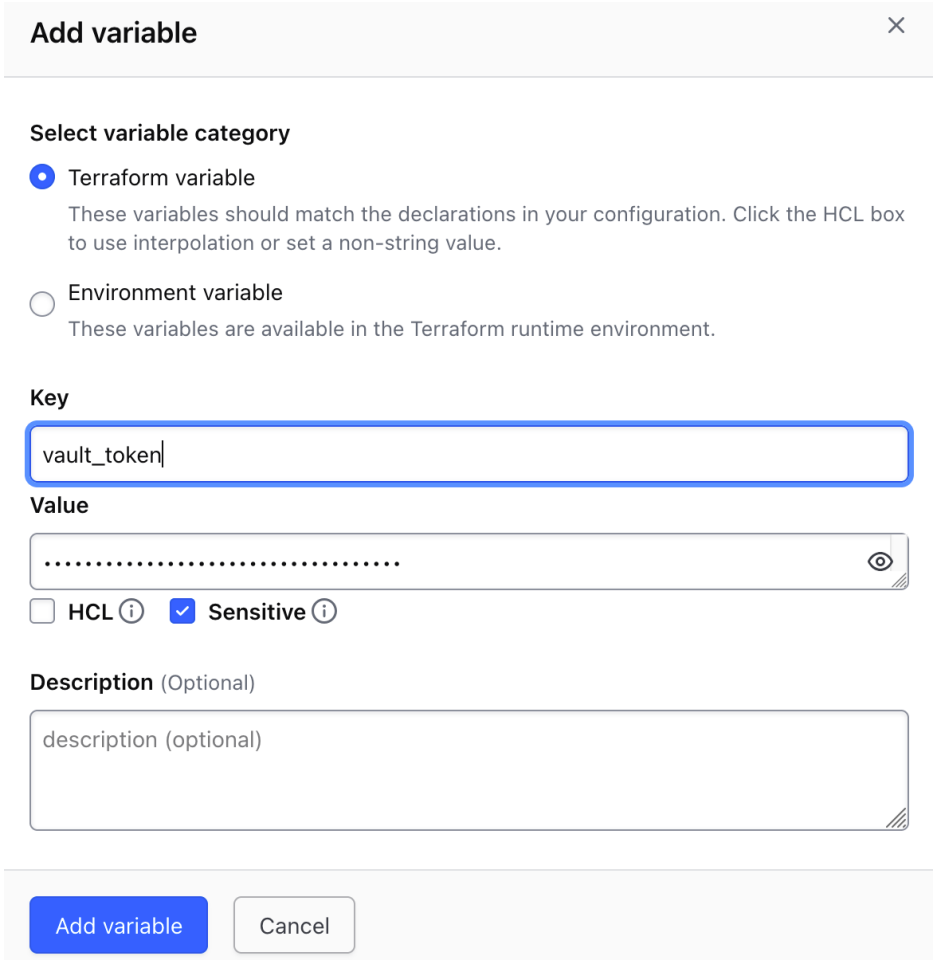
The token generation process is straightforward and can be completed through the HCP Vault web interface. The generated admin token grants full access to your Vault cluster, allowing Terraform to read secrets from the KV secrets engine.



# Updating HCP Terraform Workspace Variables

After generating your Vault authentication token, you need to update your HCP Terraform workspace variables to include the token and remove any existing hardcoded AWS credentials. This step ensures Terraform can authenticate with Vault while maintaining security best practices.

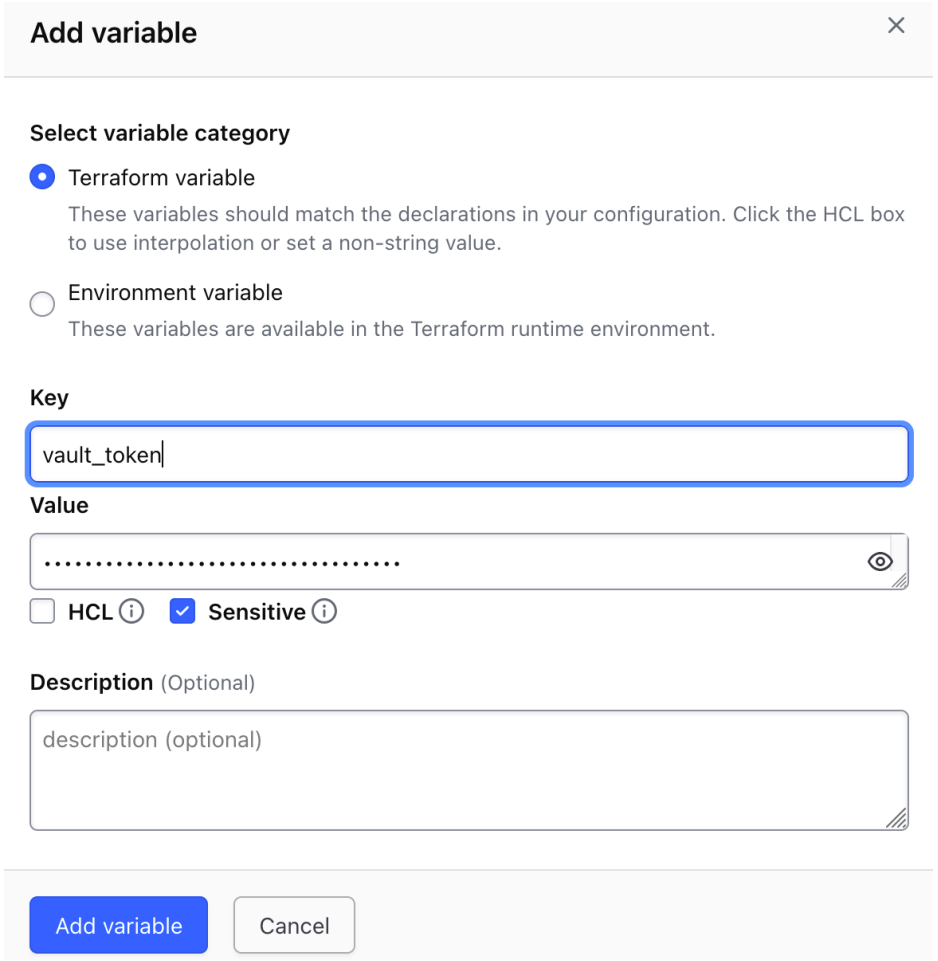
The workspace variables page allows you to manage sensitive configuration data outside of your Terraform code.



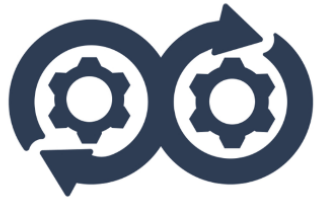
# Deploying with HCP Terraform

With all configuration complete, you can now deploy your infrastructure using HCP Terraform's VCS integration. The deployment process leverages your existing Git workflow, automatically triggering Terraform runs when you commit and push changes to your repository.

HCP Terraform will use the Vault token from your workspace variables to authenticate with your HCP Vault cluster and retrieve the AWS credentials needed for deployment.



# Lab: HCP Vault & HCP Terraform Integration



# Ansible Architecture

# Ansible Architecture

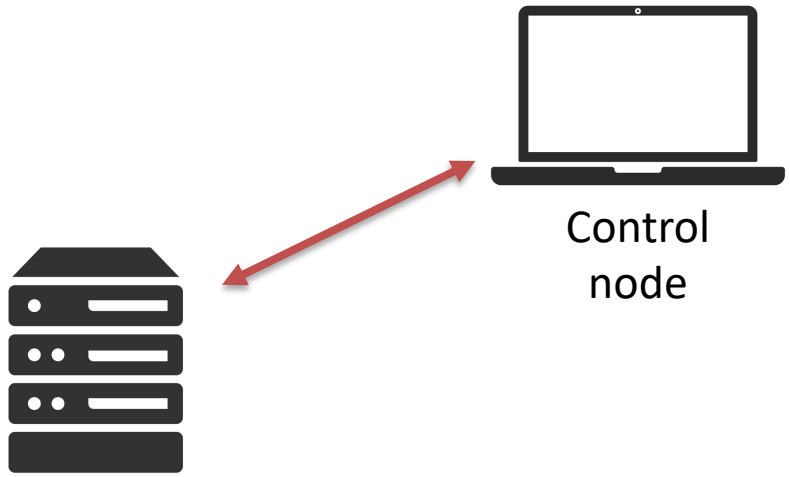
Control
node

# Ansible Control Node

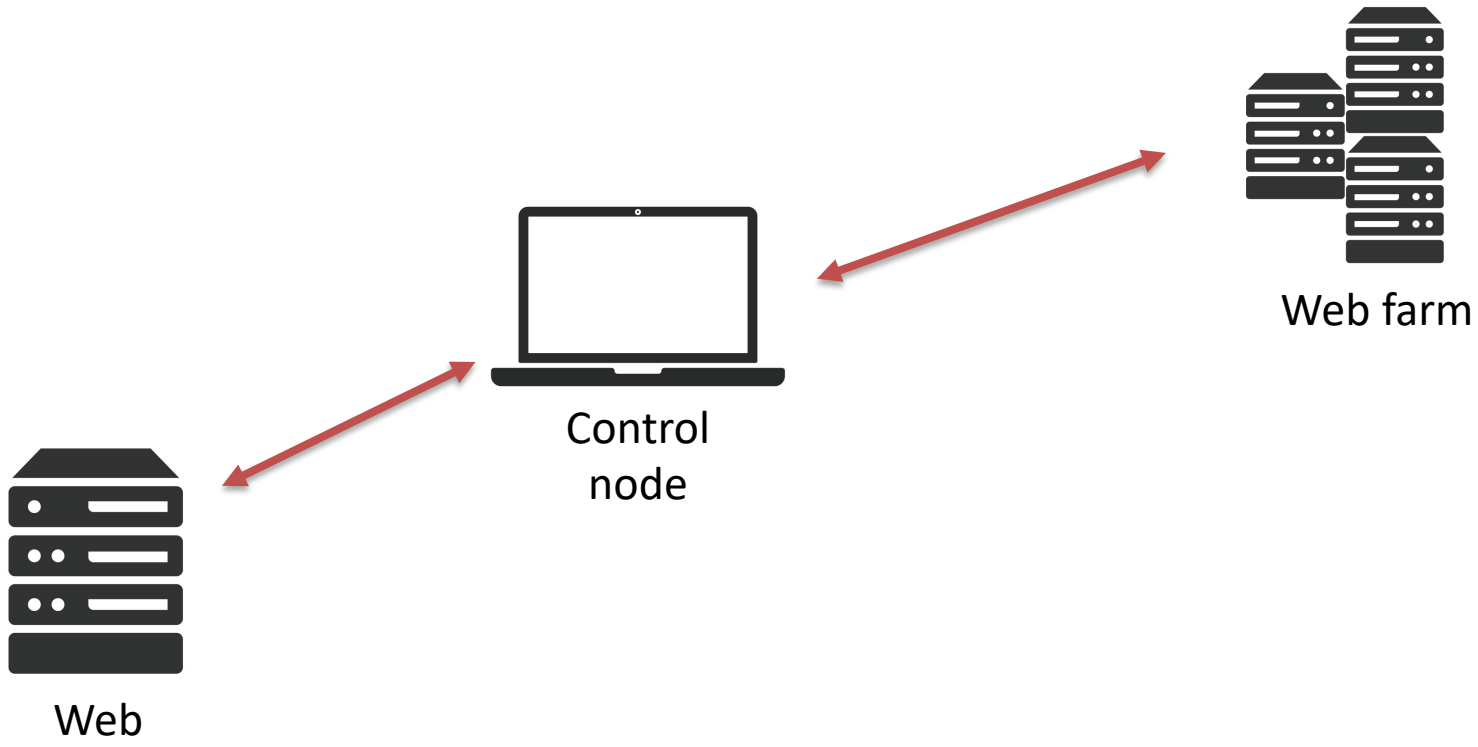
The machine from which you run the Ansible CLI tools (`ansible-playbook` , `ansible`, `ansible-vault` and others).

You can use any computer that meets the software requirements as a control node - laptops, shared desktops, and servers can all run Ansible. Multiple control nodes are possible, but Ansible itself does not coordinate across them, see AAP for such features.

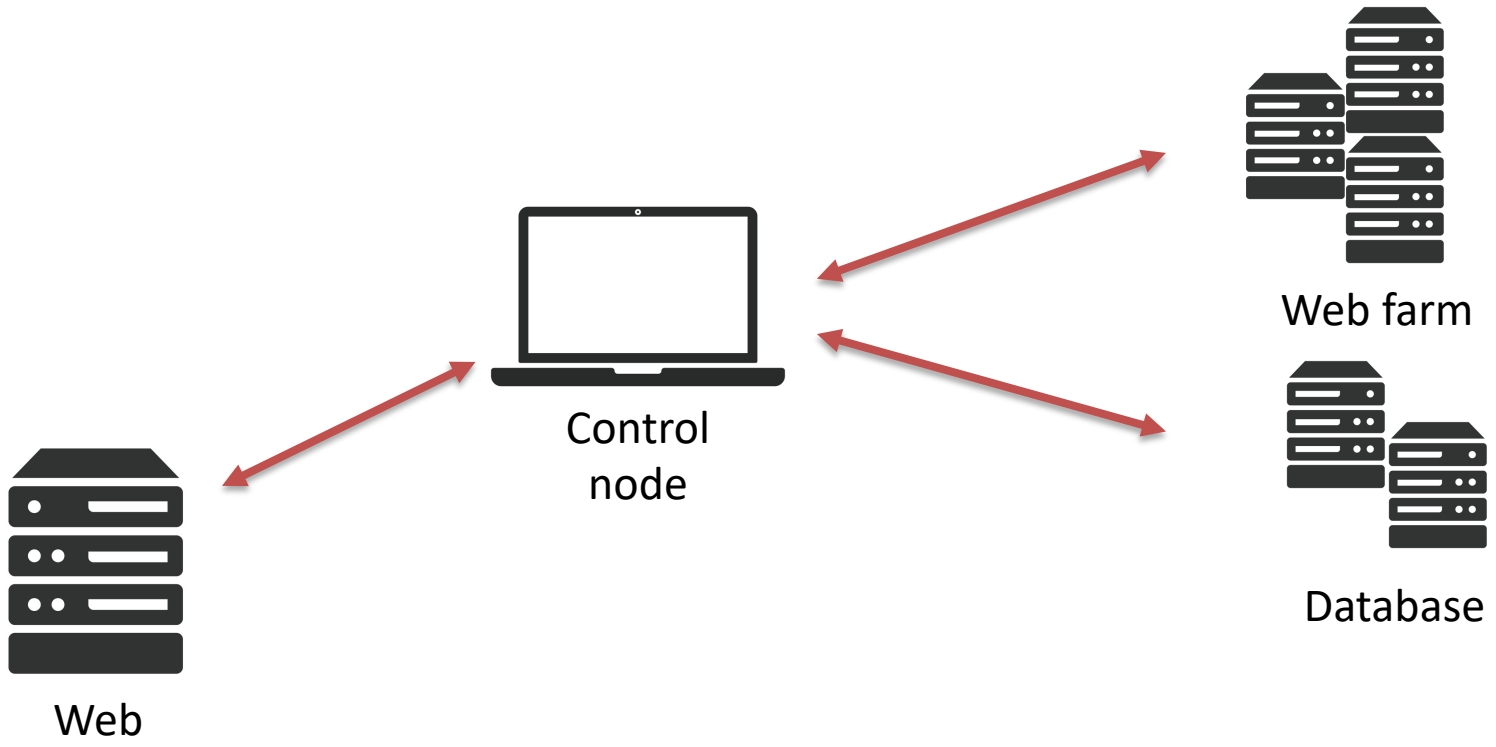
# Ansible Architecture



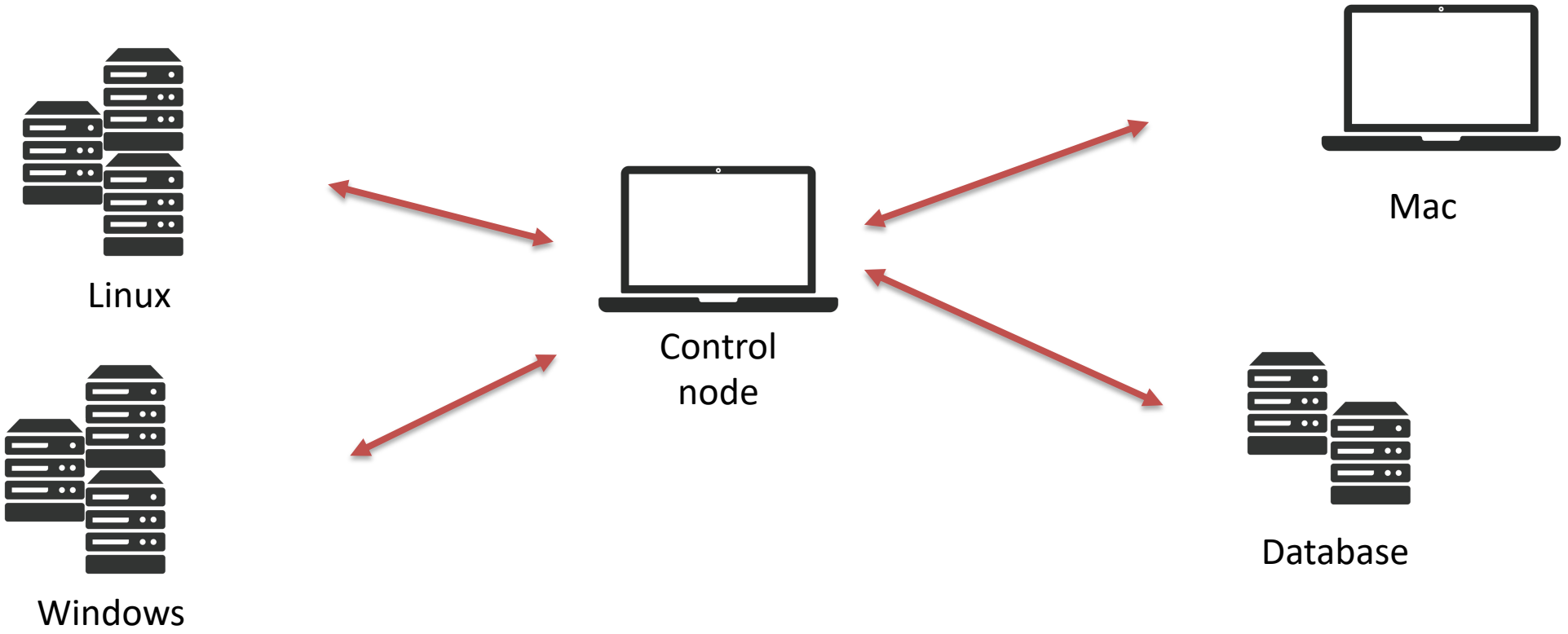
# Ansible Architecture



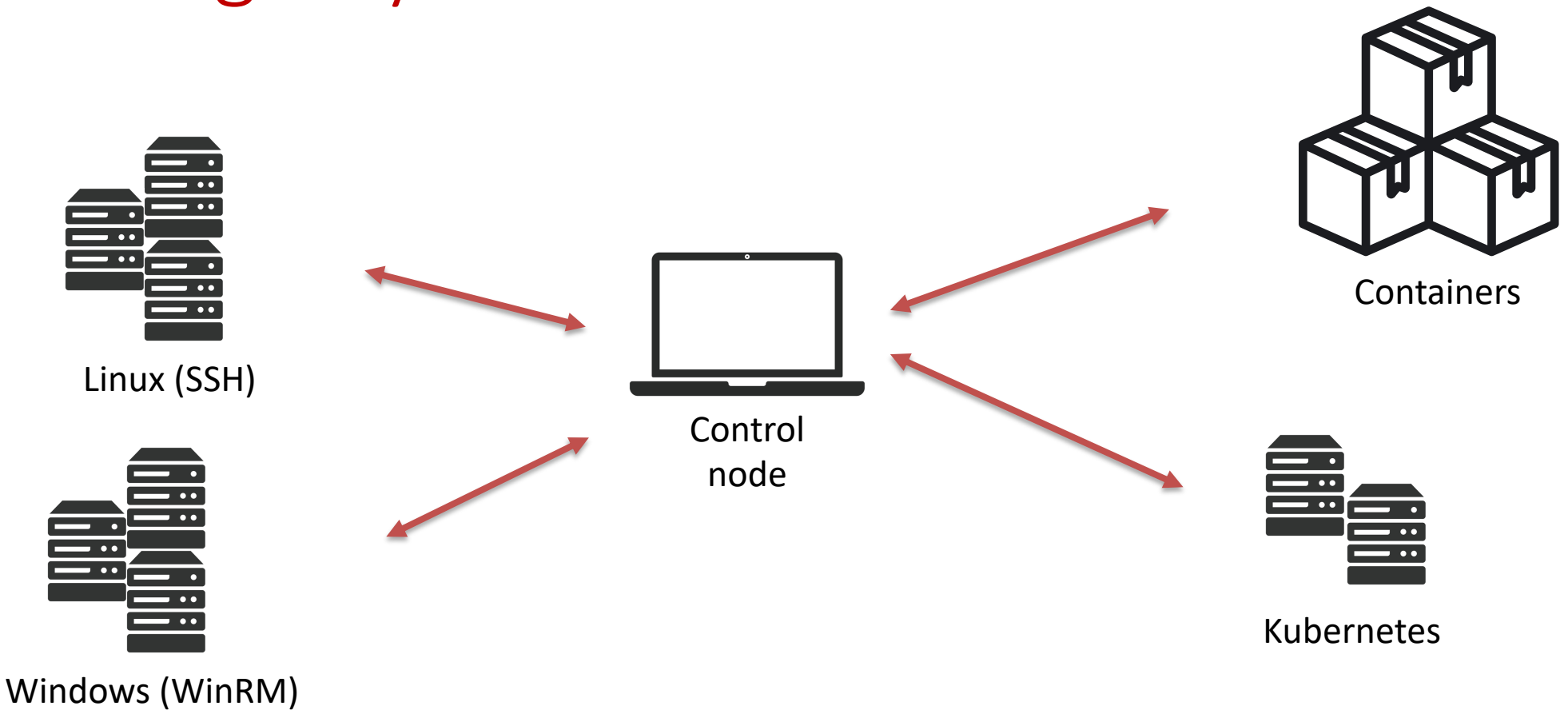
# Ansible Architecture



# Connecting Anywhere



# Connecting Anywhere



# Managed Node

Also referred to as ‘hosts’, these are the target devices (servers, network appliances or any computer) you aim to manage with Ansible.

Ansible is not normally installed on managed nodes, unless you are using `ansible-pull`, but this is rare and not the recommended setup.

# Inventory

Ansible works against multiple managed nodes or “hosts” in your infrastructure at the same time, using a list or group of lists known as inventory.

Once your inventory is defined, you use patterns to select the hosts or groups you want Ansible to run against.

# Inventory

The default location for inventory is a file called

- `/etc/ansible/hosts`

You can specify another inventory file/directory at the command-line using the `-i <path>` option.

You can also use multiple inventory files at the same time aand/or pull inventory from dynamic or cloud sources.

# Inventory

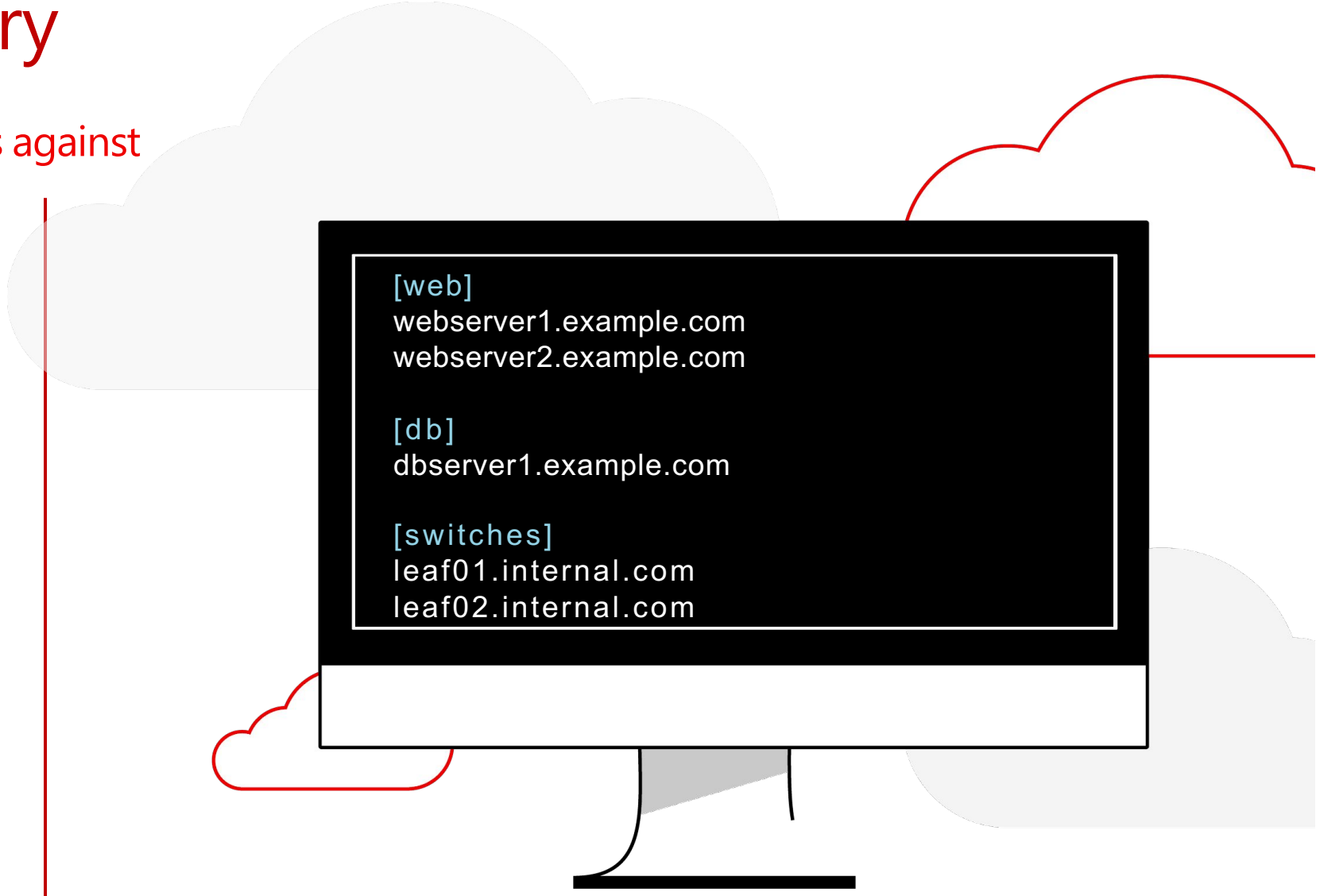
- Ansible works against multiple systems in an inventory
- Inventory is usually file based
- Can have multiple groups
- Can have variables for each group or even host

# Ansible Inventory

The systems that a playbook runs against

## What are they?

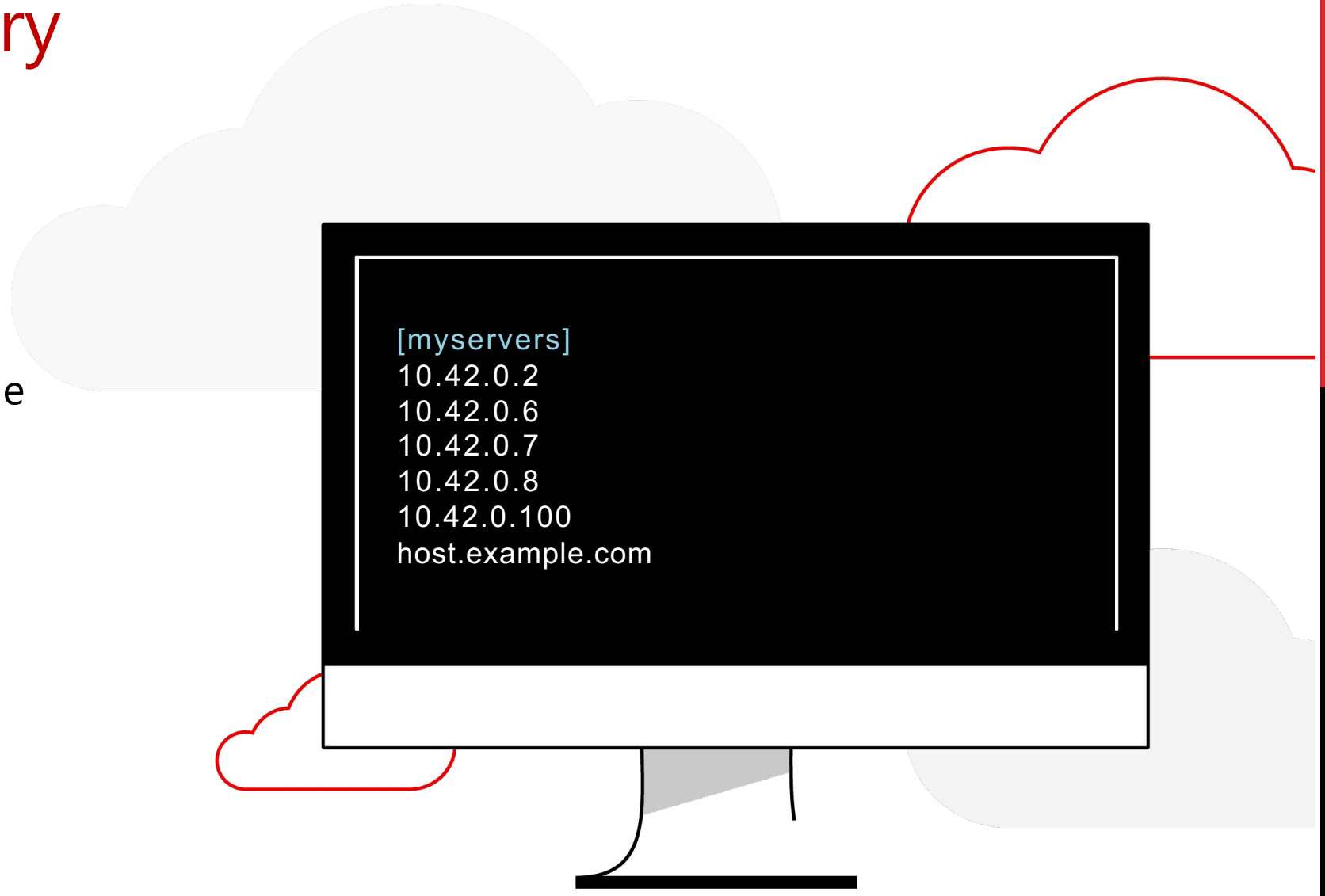
List of systems in your infrastructure that automation is executed against



# Ansible Inventory

## The Basics

An example of a static Ansible inventory including systems with IP addresses as well as fully qualified domain name (FQDN)



# Inventory Variables



```
[app1srv]
appserver01 ansible_host=10.42.0.2
appserver02  ansible_host=10.42.0.3

[web]
node-[1:30]  ansible_host=10.42.0.[31:60]

[web:vars]
apache_listen_port=8080
apache_root_path=/var/www/mywebdocs/

[all:vars]
ansible_user=kev
ansible_ssh_private_key_file=/home/kev/.ssh/id_rsa
```

# Ansible Inventory Formats

Ansible inventory can be expressed in `'INI'` or `'YAML'` format.

Example (INI):

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

# Ansible Inventory Formats

Here's the same basic inventory file in YAML format:

Example (YAML):

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
```

# Inventory Groups



```
[nashville]
appserver01 ansible_host=10.42.0.2
appserver02  ansible_host=10.42.0.3

[atlanta]
node-[1:30]  ansible_host=10.42.0.[31:60]

[south:children]
Atlanta
Nashville
hsvapp05
```

# Ansible Inventory Groups

There are two default groups:

- all: Contains every host
- ungrouped: Contains all hosts that don't have another group aside from all.

Every host will always belong to at least 2 groups (all and ungrouped or all and some other group).

Though all and ungrouped are always present, they can be implicit and not appear in group listings.

# Ansible Inventory Groups

You can (and probably will) put each host in more than one group. For example, a production webserver in a datacenter in Atlanta might look like this:

```
all:
  children:
    prod:
      hosts:
        web1.example.com:
    atlanta:
      hosts:
        web1.example.com
    webservers:
      hosts:
        web1.example.com
```

# Ansible Inventory Ranges

If you have a lot of hosts with a similar pattern, you can add them as a range rather than listing each hostname separately:
Example (INI):

```
[webservers]
www[01:50].example.com
```

```
...
  webservers:
    hosts:
      www[01:50].example.com:
```

# Ansible Dynamic Inventory

If your Ansible inventory fluctuates over time, with hosts spinning up and shutting down in response to business demands you may need to track hosts from multiple sources: cloud providers, LDAP, Cobbler, and/or enterprise CMDB systems.

Ansible integrates all of these options through a dynamic inventory system. Ansible uses inventory plugins to keep track of managed hosts.

# Ansible Dynamic Inventory

Dynamic inventory supports many solutions including Cloud Providers.
Example (AWS):

```
plugin: aws_ec2
regions:
  - us-west-1
filters:
  instance-state-name: running
keyed_groups:
  - key: tags['role']
    prefix: tag_role
hostnames:
  - ip-address
```

# POP QUIZ: DISCUSSION

Where is the default inventory location?

# POP QUIZ: DISCUSSION

Where is the default inventory location?

- `/etc/ansible/hosts`

# POP QUIZ: DISCUSSION

Which formats are valid for an Ansible inventory?

# POP QUIZ: DISCUSSION

Which formats are valid for an Ansible inventory?

- INI
- YAML

# Lab: Setup Ansible

# Lab: Ansible inventory

# Ansible Ad-hoc

An Ansible ad hoc command uses the `/usr/bin/ansible` command-line tool to automate a single task on one or more managed nodes. ad hoc commands are quick and easy, but they are not reusable.

Why learn about ad hoc commands first? ad hoc commands demonstrate the simplicity and power of Ansible. The concepts you learn will port over directly to the playbook language.

# POP QUIZ: DISCUSSION

What are some use-cases for ad-hoc mode?

# POP QUIZ: DISCUSSION

What are some use-cases for ad-hoc mode?

- Copy files

# POP QUIZ: DISCUSSION

What are some use-cases for ad-hoc mode?

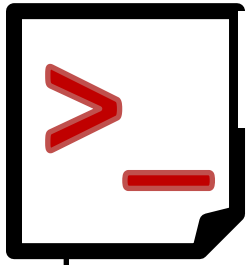
- Copy files
- Manage packages, users, groups

# POP QUIZ: DISCUSSION

What are some use-cases for ad-hoc mode?

- Copy files
- Manage packages, users, groups
- Reboot servers

# Ansible ad-hoc



## Scripted Ad-hoc

```
ansible -m copy -a "src=master.gitconfig dest=~/.gitconfig" localhost

ansible -m homebrew -a "name=bat state=latest" localhost

ansible -i hosts all -m ping -u ubuntu
```

# Ansible ad-hoc

ad hoc commands are great for tasks you repeat rarely. For example, if you want to power off all the machines in your lab for Christmas vacation, you could execute a quick one-liner in Ansible without writing a playbook. An ad hoc command looks like this:

```
$ ansible [pattern] -m [module] -a "[module options]"
```

Example of installing packages on nodes in webservers group.

```
$ ansible webservers -m yum -a "name=apache state=present"
```

# Lab: Ansible ad-hoc

# YAML

Playbooks are written in YAML.
YAML is used because it is easier for humans to read and write than other data formats like XML and JSON.
It is a format widely used by many tools (Kubernetes, Docker compose, Machine Learning, etc.)

# YAML Basics

For Ansible, nearly every YAML file starts with a list. Each item in the list is a list of key/value pairs, commonly called a “hash” or a “dictionary”. So, we need to know how to write lists and dictionaries in YAML.

There’s another small quirk to YAML. All YAML can optionally begin with --- and end with …. This is part of the YAML format and indicates the start and end of a document.

# YAML Basics

All members of a list are lines beginning at the same indentation level starting with a "- " (a dash and a space):

Example:

```
---
# A list of tasty fruits
- Apple
- Orange
- Strawberry
- Mango
...
```

# YAML Basics

A dictionary is represented in a simple key: value form (the colon must be followed by a space):

Example:

```
---
# An employee record
martin:
  name: Martin D'vloper
  job: Developer
  skill: Elite
```

# YAML Basics

More complicated data structures are possible, such as lists of dictionaries, dictionaries whose values are lists or a mix of both:

Example:

```
---
# Employee records
- martin:
    name: Martin D'vloper
    job: Developer
    skills:
      - python
      - perl
      - pascal
- tabitha:
    name: Tabitha Bitumen
    job: Developer
    skills:
      - lisp
      - fortran
      - erlang
```

# YAML Basics

Values can span multiple lines using | or >.

Spanning multiple lines using a “Literal Block Scalar” | will include the newlines and any trailing spaces.

Using a “Folded Block Scalar” > will fold newlines to spaces; it’s used to make what would otherwise be a very long line easier to read and edit.

In either case the indentation will be ignored.

# YAML Basics

Example:

```
include_newlines: |
        exactly as you see
        will appear these three
        lines of poetry

fold_newlines: >
        this is really a
        single line of text
        despite appearances
```

# Ansible Playbook

Ansible Playbooks offer a repeatable, reusable, simple configuration management and multi-machine deployment system, one that is well suited to deploying complex applications.

If you need to execute a task with Ansible more than once

- Write a playbook
- Put it under source control.

Then you can use the playbook to push out new configurations or confirm the configuration of remote systems.

# Ansible Playbook

Playbooks can:

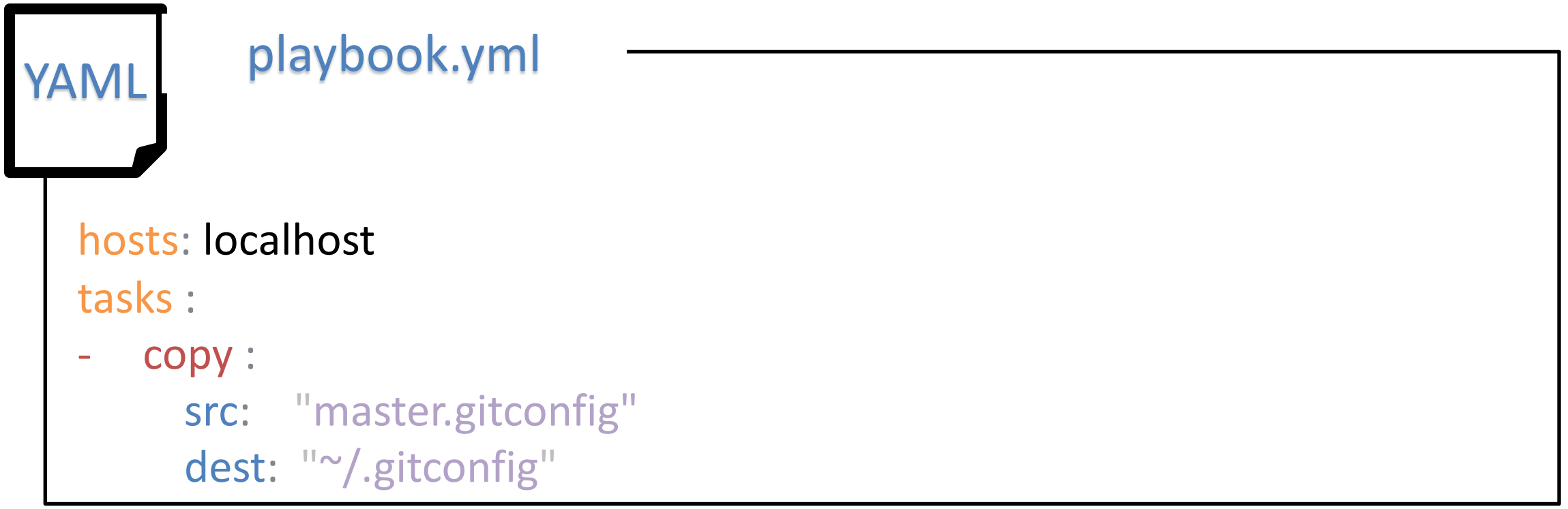
- Declare configurations
- Orchestrate steps of any manual ordered process, on multiple sets of machines, in a defined order
- Launch tasks synchronously or asynchronously

# Ansible Playbook

A playbook is composed of one or more ‘plays’ in an ordered list.

The terms ‘playbook’ and ‘play’ are sports analogies. Each play executes part of the overall goal of the playbook, running one or more tasks. Each task calls an Ansible module.

# Playbook



# Playbook


playbook.yml

```
hosts: localhost
tasks :
-    copy :
        src:    "master.gitconfig"
        dest:  "~/.gitconfig"
```

```
ansible-playbook playbook.yml
```

# Ansible Playbook Execution

A playbook runs in order from top to bottom. Within each play, tasks also run in order from top to bottom.

Playbooks with multiple ‘plays’ can orchestrate multi-machine deployments, running one play on your webservers, then another play on your database servers, then the third play on your network infrastructure, and so on.

# Ansible Playbook Tips

At a minimum, each play defines two things:

- The managed nodes to target, using a pattern
- At least one task to execute

- Ansible creates a <playbook>.retry playbook for hosts where it failed. You can execute the <playbook>.retry playbook and it will try to run it ONLY on the hosts that failed.
- Limit: Used to run Ansible playbook only on hosts you specify. Great for testing on one host.
- Whitespace: Ansible is like Python, it requires correct indentation. (ansible lint, syntax-check, etc.)

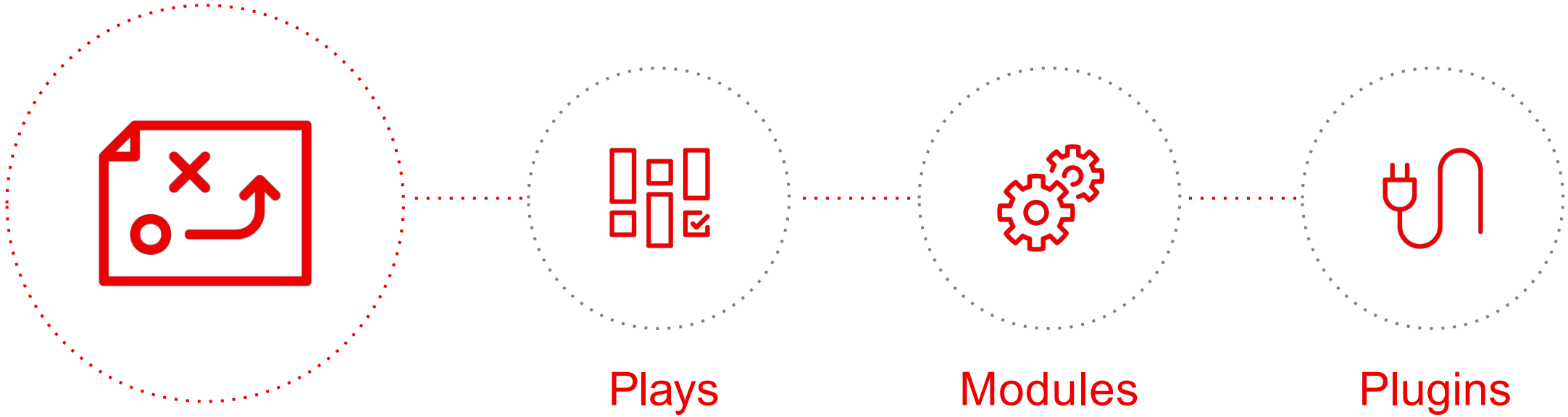
# Playbooks

Simple playbook to install and configure Apache web server

```
---
- name: install and start apache
  hosts: web
  become: yes

  tasks:
    - name: httpd package is present
      yum:
        name: httpd
        state: latest

    - name: latest index.html file is present
      template:
        src: files/index.html
        dest: /var/www/html/

    - name: httpd is started service:
        name: httpd state: started
```

# Playbooks

## What makes up an Ansible playbook?



# Ansible Plays

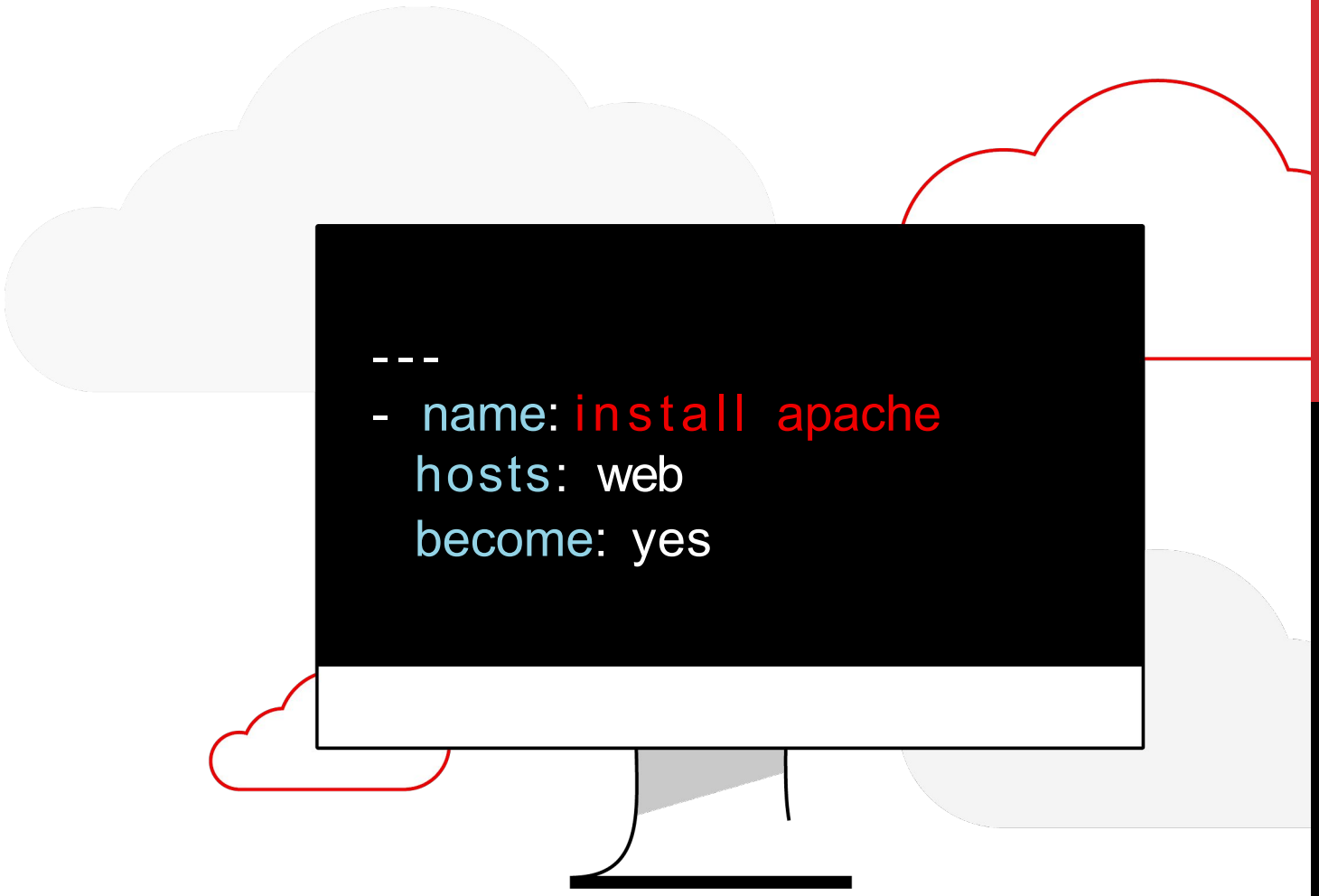
## What am I automating?

### What are they?

Top level specification for a group of tasks. Will tell that play which hosts it will execute on and control behavior such as fact gathering or privilege level.

### Building blocks for playbooks

Multiple plays can exist within an Ansible playbook that execute on different hosts.



# Ansible Modules

## The “tools in the toolkit”

### What are they?

Parametrized components with internal logic, representing a single step to be done.

The modules “do” things in Ansible.

### Language

Usually Python, or Powershell for Windows setups. But can be of any language.

# Ansible Plugins

## The “extra bits”

### What are they?

Plugins are pieces of code that augment Ansible’s core functionality. Ansible uses a plugin architecture to enable a rich, flexible, and expandable feature set.

# More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Name of Play

# More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Hosts to run Play on

# More Complete Playbook

```
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

User to run Play as

# More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Define what to do after package is installed.

# More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Call the defined handler

# More Complete Playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Tasks to run
- install Docker

# More Complete Playbook

```
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Create user account

# Ansible Variables

Ansible uses variables to manage differences between systems. With Ansible, you can execute tasks and playbooks on multiple different systems with a single command.

- Create variables with YAML syntax, including lists and dictionaries
- Define these variables

# Ansible Variables

Variables can be defined in many locations:

- Playbooks
- Inventory
- re-usable files or roles
- command line.

You can also create variables during a playbook run by registering the return value or values of a task as a new variable.

# Ansible Variable Valid Names

A variable name can only contain:

- Letters
- Numbers
- Underscores

A variable name cannot begin with a number, but can begin with an underscore.

# POP QUIZ: DISCUSSION

Are these valid variable names?

- foo

# POP QUIZ: DISCUSSION

Are these valid variable names?

- foo - valid

# POP QUIZ: DISCUSSION

Are these valid variable names?

- app.port

# POP QUIZ: DISCUSSION

Are these valid variable names?

- app.port - invalid

# POP QUIZ: DISCUSSION

Are these valid variable names?

- *ssl_key

# POP QUIZ: DISCUSSION

Are these valid variable names?

- *ssl_key - invalid

# POP QUIZ: DISCUSSION

Are these valid variable names?

- _web_root

# POP QUIZ: DISCUSSION

Are these valid variable names?

- _web_root - valid

# Simple Variable

Defining simple variables

```
remote_install_path: /opt/my_app_config
```

After you define a variable, use Jinja2 syntax to reference it. Jinja2 variables use double curly braces.

```
ansible.builtin.template:
  src: foo.cfg.j2
  dest: '{{ remote_install_path }}/foo.cfg'
```

# Variable Quotation

If you start a value with {{ foo }}, you must quote the whole expression to create valid YAML syntax. If you do not quote the whole expression, the YAML parser cannot interpret the syntax - it might be a variable or it might be the start of a YAML dictionary.

This example will error.

```
- hosts: app_servers
  vars:
    app_path: {{ base_path }}/22
```

ERROR! Syntax Error while loading YAML

# Variable Quotation

If you start a value with {{ foo }}, you must quote the whole expression to create valid YAML syntax. If you do not quote the whole expression, the YAML parser cannot interpret the syntax - it might be a variable or it might be the start of a YAML dictionary.

Fix the issue by quoting the entire expression.

```
- hosts: app_servers
  vars:
    app_path: "{{ base_path }}/22"
```

# Ansible Modules

Modules are the main building blocks of Ansible playbooks. Although we do not generally speak of “module plugins”, a module is a type of plugin.

Common modules:

- Working with files: copy, archive, unarchive, get_url
- user, group
- ping
- service
- yum, apt, package
- template

# Ansible Modules

Common modules:

- Lineinfile
  - Manipulate text in files
    - Add alias for hosts
    - Supports regex
    - Idempotent
- Shell/command
- Script module
- Debug module

# Lab: Ansible playbook fundamentals