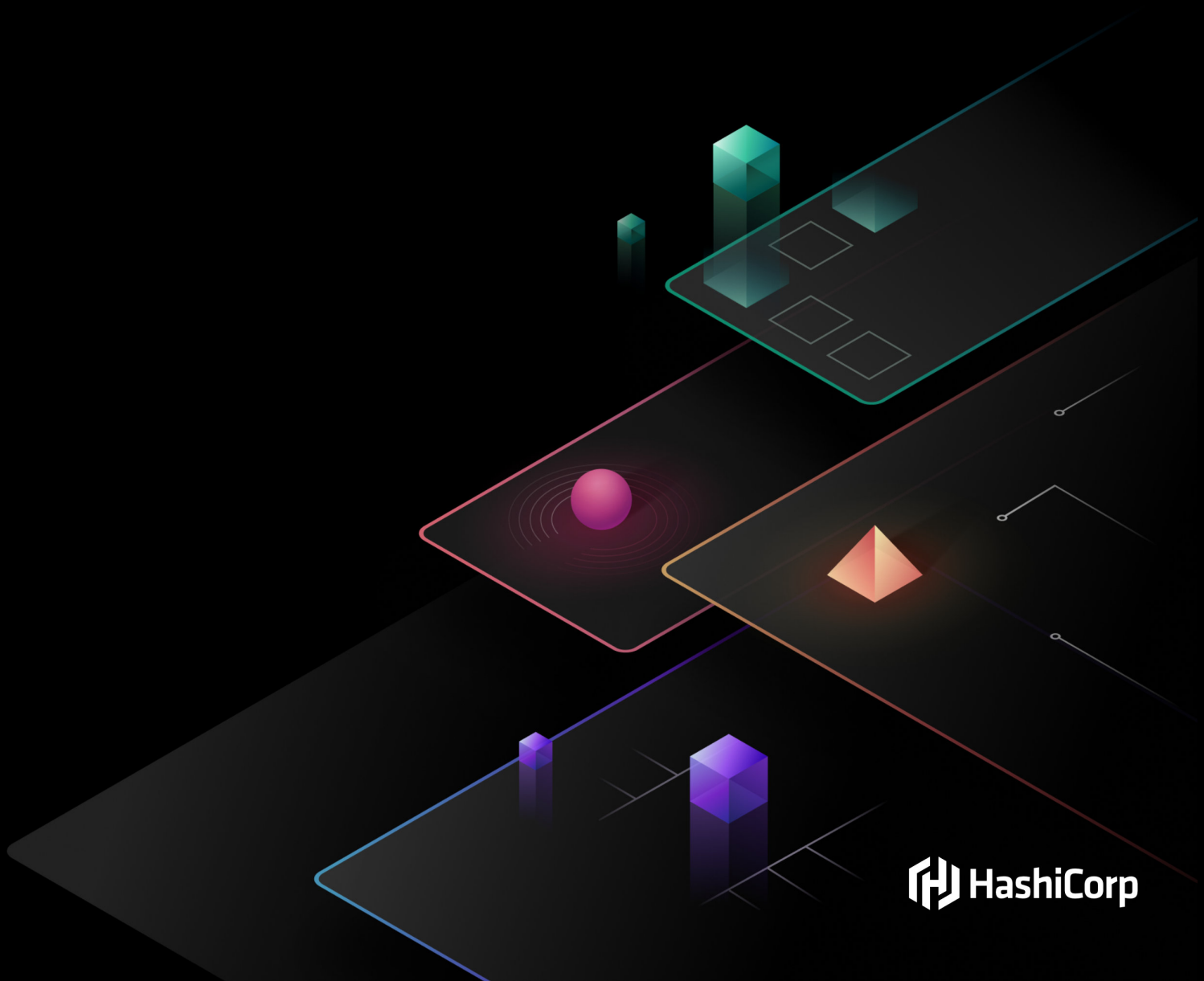
HashiCorp

Terraform: Enterprise Solutions Design Guide

HashiCorp | Validated Designs | Version v1.00.00

27 October 2023



Contents

1 HashiCorp Validated Design

1.1 Terraform Enterprise solution design guide

Notes to the reader:

> This document provides helpful recommendations for implementing Terraform infrastructure-as-code (IaC) as a shared service for your organization. The team responsible for this task may be called different names, such as the "Platform Team" or "Cloud Center of Excellence" (CCoE). Regardless of the name, this guide is designed to assist teams responsible for owning, implementing, and operating Terraform IaC in their organization. For more detailed information on Platform Teams, please refer to the Operating Guide.

> Our objective in providing "HashiCorp Validated Designs" is to offer prescriptive guidance based on our experience partnering with hundreds of organizations. We understand that the industry is complex and that there are many permutations and combinations for implementing solutions. However, we believe that the most important thing is that you are able to safely provision and manage cloud resources *at scale* and experience the business benefits and value that automated Terraform Enterprise workflows provide.

2 Introduction

2.1 Why use HashiCorp Validated Designs?

HashiCorp introduced the Validated Designs program to give enterprise customers and partners a set of recommendations to deliver a resilient, secure, and high-performance deployment of HashiCorp solutions. The purpose of this document is to provide the Platform Team with HashiCorp's validated design for deploying Terraform Enterprise, enabling your organization to embrace and accelerate infrastructure automation practices. By following this approach, you will eliminate ambiguity in deployment options and be able to make project-level decisions with confidence.

> The current version of this guide is focused on deploying Terraform Enterprise in AWS and in VM-based private clouds such as VMware. Future versions will include other major clouds including Azure and GCP.

2.2 Audience

This document is intended for platform engineers, infrastructure architects, DevOps administrators, and cloud operators who want to design, deploy and administer a highly scalable, resilient infrastructure-as-code platform with Terraform Enterprise.

2.3 Document structure

Document section	Purpose
Architecture	An overview of Terraform Enterprise's logical architecture. Terraform Enterprise requirements, components, and deployment modes listed.
People	Description of people skills and access required to accomplish the deployment.
Terraform-based install	Installation of Terraform Enterprise using our validated Terraform modules. Currently we provide a module for AWS and modules for Azure & GCP will be delivered in the future.
Private cloud installation	Manual installation steps that can be used to install Terraform Enterprise in a VM/bare metal private cloud/datacenter environment.
Security hardening	Guidance and recommendation to harden the Terraform Enterprise VM images used in the installation.

Document section	Purpose
Next steps	Configure the artifacts with the data collected from your environment, and use them for installing Terraform Enterprise.

2.4 Supported versions

This guide has been validated with the following versions of Terraform Enterprise:

- Terraform Enterprise FDO v202309-1 and above

2.5 Language and definitions

HashiCorp is an enabler of multi cloud strategies, and as such we take this into account when writing designs. While every attempt has been made to use technology agnostic terminology, we primarily aim to support the three largest CSPs together with however there are some terms which do not translate perfectly between the public cloud and the data center. For the sake of clarity, our definitions for these terms are included below.

Term	Definition
Availability zone	A separate failure domain within a logical datacenter.
Region	A separate logical datacenter.
Public subnet	A network accessible from the public Internet, containing publically-addressable infrastructure.
Private subnet	A network not accessible *from* the public Internet and whose infrastructural objects are either blocked from connecting *to* the public Internet or do through a NAT gateway.

3 Terraform Enterprise architecture

Terraform Enterprise follows the Tao of HashiCorp's principle of "Simple, Modular, Composable." It automates infrastructure provisioning and lifecycle management. It also integrates with the ecosystem to offer additional features like single sign-on for user authentication, version control, observability, and artifact management.

3.1 Design attributes

Most decision points in this document require the evaluation of non-functional requirements to find a recommendation that is acceptable within the bounds of a typical customer environment. Where possible, decisions will be accompanied by associated narrative allowing the reader to understand our rationale.

- Availability: Represents design points that minimize the impact of subsystem failures on the solution uptime. *An example of this is introducing multiple load-balanced application instances.*
- Operational complexity: Occasionally it is necessary to introduce some complexity during the initial deployment in order to provide a reduced effort for operations teams to manage the solution. *An example of this is introducing a machine imaging process with Packer that permits immutable operations for scaling up/down and application updates.*
- Scalability: Some design choices may work at a small scale but introduce overhead as the solution scales. *An example of this is choosing to onboard teams via a manual process in the Terraform Enterprise UI vs creating an automated onboarding workflow to complete the necessary setup.*
- Security: Indicates how a decision changes the security posture of the overall solution. *An example of this is taking advantage of workload identity so that workspaces authenticate against their API targets using JWT based workflows rather than using long-lived credentials as workspace variables.*

3.2 Terraform Enterprise components

Terraform Enterprise consists of several components, each with specific responsibilities in the overall architecture. This section explains the function of each component and the technologies they support.

- **Terraform Enterprise application**: This is the Terraform Enterprise container that Hashicorp provides as the core application.
- **Terraform Cloud agent (optional)**: A fleet of isolated execution environments that perform Terraform runs on behalf of Terraform Enterprise users. The Terraform Cloud agent (which is also usable with Terraform Enterprise) can consume APIs in isolated network segments to provide quality of service to specific teams in your organization and/or distributes load away from the core Terraform Enterprise service. While use of Terraform Cloud agents is highly recommended, it is not required. Terraform Enterprise comes with built-in agents for job execution.
- **PostgreSQL database**: Serves as the primary store of Terraform Enterprise's application data such as workspace settings and user settings. You can review the supported versions and requirements on our Terraform Enterprise PostgreSQL page.
  - For AWS we recommend a starting instance of `db.r6g.xlarge`.
- **Redis cache**: Used for Rails caching and coordination between Terraform Enterprise core's web and background workers. In order to have multiple, simultaneously active application nodes, an external cache is required. Terraform Enterprise is validated to work with the native Redis services from AWS, Azure, and GCP, and Redis is used internally on single-node VMware-based deployments.
  - *Terraform Enterprise is not compatible with using Redis Cluster or Redis Sentinel.*
  - *Redis is a required as an external service in private cloud deployments consisting of multiple compute nodes.*
  - Redis 6.x and 7.x are the fully tested and supported versions of Redis for Terraform Enterprise.
  - Redis 7.x is the recommended version.
  - For AWS we recommend a starting instance of `cache.m5.large`.
- **Object Storage**: Used for storage of Terraform state files, plan files, configuration, output logs and other such objects. All objects persisted to blob storage are symmetrically encrypted prior to being written. Each object is encrypted with a unique encryption key. Objects are encrypted using 128 bit AES in CTR mode. The key material is processed through the internal HashiCorp Vault transit secret engine, which uses the default transit encryption cipher (AES-GCM with a 256-bit AES key and a 96-bit nonce), and stored alongside the object. This pattern is called envelope encryption. Any S3-compatible object storage service, GCP Cloud Storage or Azure

blob storage meets Terraform Enterprise's object storage requirements. You must create an object store instance for Terraform Enterprise to use, and specify its connection string details during installation. We recommend you ensure the object store is in the same region as the Terraform Enterprise instance.

- For AWS, we recommend using EBS gp3 volumes in your EC2 instance configuration because they provide the best performance and scale with a baseline performance of 3000 IOPS.

3.3 Deployment topologies

In this section, we review the supported deployment topologies for Terraform Enterprise.

From v202309-1, Terraform Enterprise now offers Flexible Deployment Options (FDO) across a variety of platforms including Kubernetes. Our official documentation on installation on this is here. We recommend using this for all private cloud deployments. Deployments which require the Replicated software distribution platform will be deprecated and removed in due course.

Deployment topology	Characteristics	Description
Mounted Disk	All Terraform Enterprise services are installed on a single node.	Most suitable for functional testing, training, and gaining familiarity. Should not be used for production.
Single Instance + External Services	External database, external object storage, stateless application installed on a single node with internal cache with optional, remote Terraform agents.	Most suitable for organizations who are concerned with system performance, but less so with system resilience.

Deployment topology	Characteristics	Description
Active/Active	Replicated database and object storage, external caching layer and multiple, stateless application nodes in redundant availability zones. Optionally, multiple remote Terraform agents.	Most suitable for a business-wide service that requires the highest levels of resilience and performance.

3.3.1 Mounted disk

In a *Mounted Disk* deployment, an organization deploys Terraform Enterprise in a singular compute environment without failover or active/active capabilities. In this deployment model, all services (including PostgreSQL DB and other storage requirements) are self-contained within the Docker environment, eliminating external dependencies. The application data is self-managed and uses localized Docker disk volumes for caching and data operations.

A *Mounted Disk* deployment is well-suited for development, functional testing, and acceptance testing scenarios, however we do not recommend that you use this for any production environments. For production deployments only the *Single instance + External Services* or *Active/Active* deployment topologies are recommended.

The following diagram illustrates the logical architecture of a *Mounted Disk* Terraform Enterprise deployment:

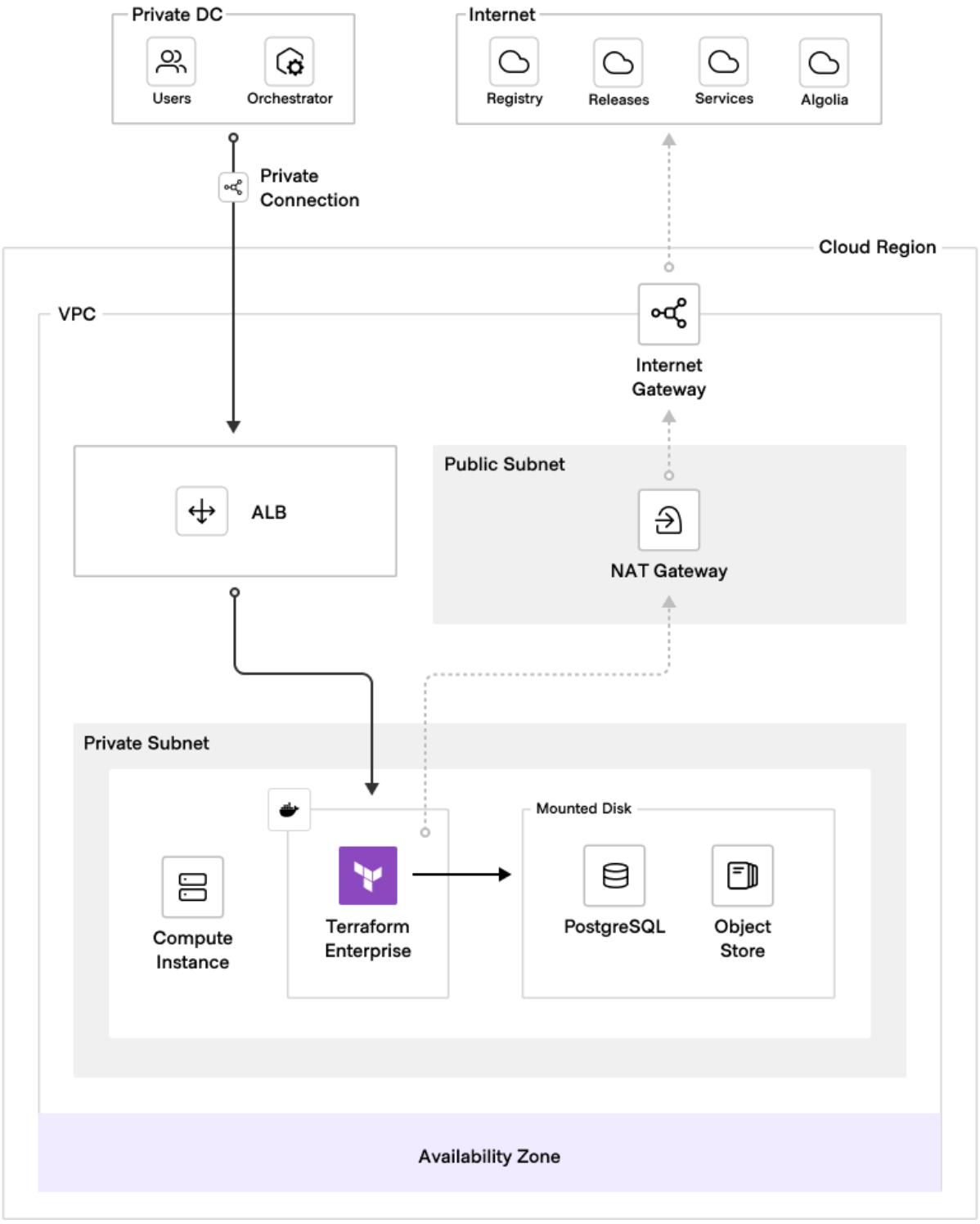


Figure 1: Terraform Mounted Disk deployment topology

The benefits of this deployment model include minimal resource requirements, reduced reliance on specialized expertise, and rapid deployment. However, it lacks enterprise-grade resiliency for unexpected downtime and cannot scale performance without interrupting services.

3.3.2 Single Instance + External Services

In order to move to a stateless application front-end, the External Services topology takes the application database and object storage and moves them to dedicated infrastructure. The cache remains internal but represents transient caching only.

Each of the public clouds provides native services for a PostgreSQL database and object storage that supports this deployment pattern. The configuration of these external services decouples the application infrastructure, eliminating single points of vulnerability and improving availability. While this deployment model may not offer performance scalability (although AWS Aurora storage can be scaled and minimizing downtime is possible when scaling the DB instance through instance failover), it enhances the deployment resilience by distributing the core components.

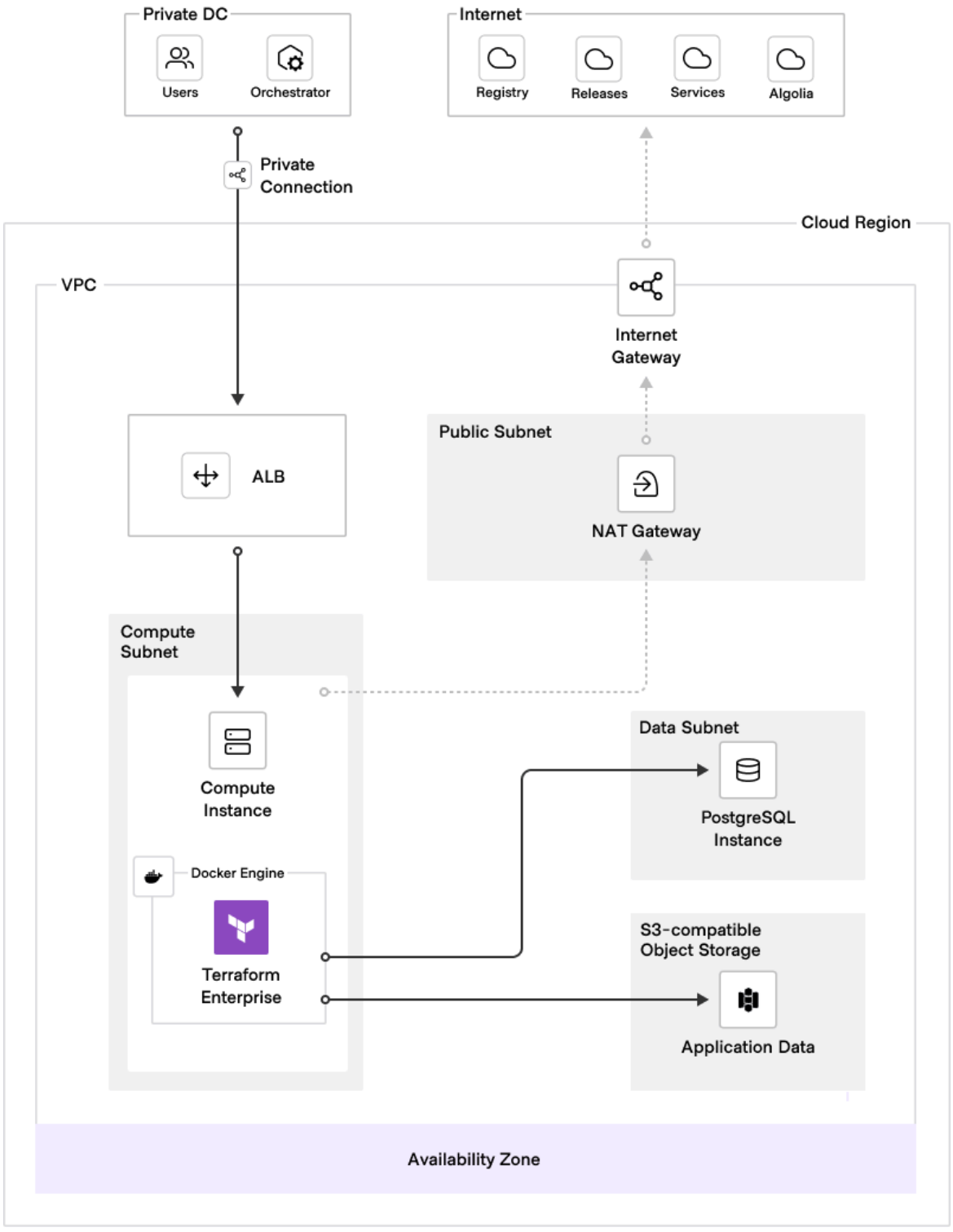


Figure 2: Single Instance + External Services deployment topology

The model is an initial step towards a fully resilient and scalable deployment. It relies on one compute environment for a Docker engine. This means that this single instance of the Terraform Enterprise application becomes a vulnerable single point of failure for an enterprise-grade operation. While this model could be partially mitigated by automated deployment (such as to a compute scaling group of one node to 'self-heal'), cross-region replication for the object store and a distributed database cluster, if your target architecture requires zero downtime from expected failures such as instance or availability zone failure, we recommend Active/Active below.

3.3.3 Active/Active (multiple instances)

When your organization requires increased availability or performance from Terraform Enterprise that your current single application instance cannot provide, it is time to scale to the Active/Active topology.

Before scaling to Active/Active, you should weigh its benefits against the increased operational complexity. Specifically, consider the following aspects of the Active/Active topology:

- A completely automated TFE deployment process is a hard requirement
- When monitoring multiple TFE instances, it is important to consider the impact on your monitoring dashboards and alerts.
- To effectively manage the lifecycle of application nodes, custom automation is necessary.

Terraform Enterprise supports an active/active deployment model, which involves combining multiple instances of Terraform Enterprise with external services. In this model, the primary application services are hosted in separate Docker engines (on VMs or by an orchestrator such as Kubernetes), spread across at least three availability zones within the same cloud region. These instances connect to centralized services that provide the application database, shared object storage and durable cache, all managed by the cloud service provider.

Using an active/active model has a key advantage: It eliminates potential service failure points. By distributing multiple application instances, the service becomes more resilient and available. The deployment includes redundant elements that improve reliability, increase resilience, and enhance performance.

The following diagram illustrates the recommended architecture for an active/active deployment. It provides a visual representation of data flow, user access, key connections, and infrastructure components.

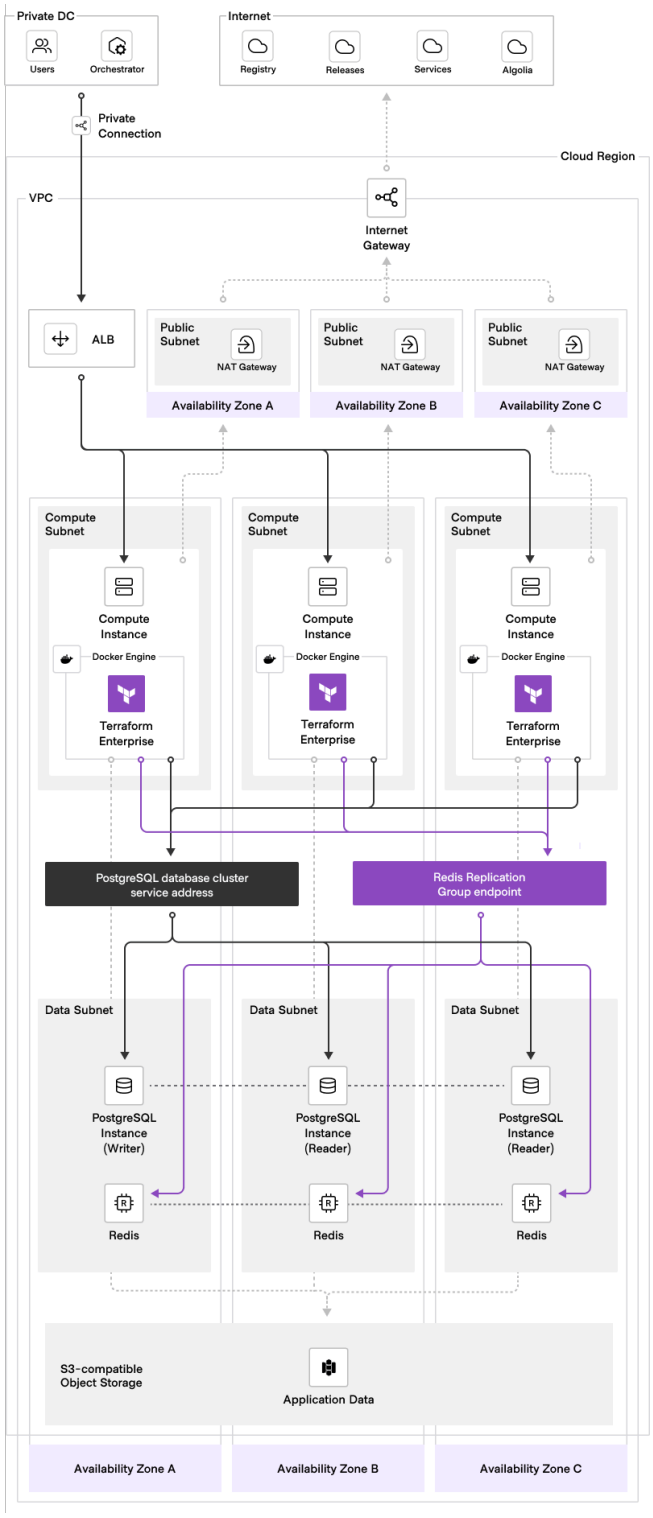


Figure 3: Active/active deployment topology

In addition to improved availability, an active/active deployment offers scalability and supportability benefits. The presence of multiple application instances allows for better workload distribution, alleviating bottlenecks and enhancing overall performance, with redundant application instances and external services.

> Please note that Redis does not have to be externalized when there is a single node of Terraform Enterprise. When using our Terraform-based deployment modules, a separate Redis instance will be automatically deployed when the active/active parameter is set to `true`.

3.4 Deployment topology decision

The Active/Active topology offers crucial benefits to your organization and is our recommendaction because it:

- Ensures service availability and scalability while mitigating potential failure scenarios caused by service interruption or resource starvation.
- Safeguards against revenue loss due to unscheduled service interruptions.
- Ensures data integrity while adhering to your data residency security guidelines.

3.5 Automated versus manual installation

As mentioned above, customers deploying Terraform Enterprise using the Active/Active deployment topology must use automated means, and this requirement will guide your thoughts as to how this needs to occur with your installation approach.

For public cloud customers, we recommend the Terraform modules HashiCorp provide for Single Instance + External Services and Active/Active topologies. Public cloud customers intending to deploy Mounted Disk mode for development and testing are advised to develop an automated deployment which works for you, in order to be able to destroy and recreate instances quickly and easily.

For private cloud customers who have Chef, Puppet, Ansible, Salt or other automated configuration management technologies as their organisational standard, we agree that use of such is the best practice because it provides

- A well-known approach to installation and software management for your organisation and thus
- Lower barrier to success in deployment.
- Adherence to organisational standard likely also means increased chance of regulatory adherence where applicable.

We advise private cloud customers to automate deployment of Terraform Enterprise irrespective of the deployment topology used because

- If you decide to manually install Terraform Enterprise, this will increase the length of time and the amount of human involvement required to upgrade each instance each month. Automation significantly reduces this burden.
- Automated installation means the code used can be versioned and collaborated on.
- Automated installation also means being also to rely on the same deployment pattern every time, which increases maintenance confidence for both architecture teams and services teams responsible for managing the product in production.

4 Personnel and Access

This section defines the roles required from the participants, and includes authority and access needs to the working environment.

4.1 Personnel

Focusing on the deployment activity alone, we recommend enlisting a project leader and a cloud administration team. A project leader coordinates events, facilitates resources and assigns duties to the Cloud Administration Team.

Members of the Cloud Administration Team carry out functional tasks to install Terraform Enterprise. For cloud administrators, we make assumptions about general knowledge of the following:

- Cloud architecture and administration
- Administration-level experience with Linux
- Practical knowledge of Docker
- Practical knowledge of Terraform

In addition, we strongly recommend requesting assistance from a security operations team. The emphasis is on integrating formal security controls required for services hosted in your preferred cloud environment.

It is also essential to specify a role for the production services team who will take over the deployment when the project goes live.

4.2 Access

- Infrastructure access: Significant access to various infrastructure elements will be required depending your type of installation.
  - The installation team requires direct access (including admin) to various resources to install and configure Terraform Enterprise including:
    - Compute/storage instances
    - Network objects such as firewalls, load balancers etc.
    - Certificates: a TLS certificate is required either on the ALB or Terraform Enterprise compute node(s) - also see below.

    - Identity such as AWS IAM, GCP Cloud Identity, AAD etc.
    - Secrets management e.g. AWS Secrets Manager, GCP Secret Manager, Azure Key-Vault or vSphere Native Key Provider
    - Data encryption services e.g. AWS KMS

- License File: To deploy Terraform Enterprise you must obtain a license from HashiCorp.
- Certificate authority:
  - Terraform Enterprise requires a TLS certificate and private key on each node in order to operate securely. This certificate must match the Terraform Enterprise hostname, either by being issued for the FQDN or being a wildcard certificate.
  - The certificate can be signed by a public or private CA, but it must be trusted by all of the services that Terraform Enterprise is expected to interface with; this includes your VCS provider, any CI systems or other tools that call the Terraform Enterprise API, and any services that Terraform Enterprise workspaces might send notifications to (for example: Slack).
  - The key and X.509 TLS certificate must be PEM encoded, and should be provided to the installer as text. Terraform Enterprise validates the certificate to ensure it uses a Subject Alternative Name (SAN) for Domain Names (DN) entries and not just a Common Name (CN) entry.
  - It is possible to put the TLS certificate on the application load balancer, but in this case, the route from the inside of the LB to the compute nodes also requires certificate(s) which we recommend should be official and from the same CA. As such, certificates are still required on each Terraform Enterprise instance.
- DNS: Ensure that a DNS record exists for Terraform Enterprise and that the certificates mentioned above have been configured with the correct DNS name.

5 Terraform based TFE installation

HashiCorp provides validated Terraform modules to install Terraform Enterprise on AWS/GCP & Azure. The initial version of HVD will provide support only for AWS and we intend to add GCP & Azure in subsequent releases of this HVD.

5.1 AWS architecture

Following is our recommended architecture for installing Terraform Enterprise on AWS. The architecture is optimized for High Availability and features the following:

1. Deployment of the Terraform Enterprise Binary on EC2 instances that are part of an autoscaling group spanning 3 availability zones (AZ) and significantly reduces the overhead in managing those services including patching/upgrading etc.
2. Leveraging AWS's managed versions of Postgres DB and Redis Cache in cluster mode, that ensures replicas are distributed in different AZ.
3. Use of a Application Load Balancer to ingress API & Web traffic to the Terraform Enterprise instance. By using ALB, we can automate TLS certificate management and do not have to place certificates within the EC2 instances. [Note: A downside of this approach is that network traffic between the ALB and the EC2 instance is not encrypted. If your company policies require that the traffic between the LB and EC2 is encrypted, you can switch to a Network Load balancer and place the certificates on the EC2].
4. By placing EC2 and external services in multiple AZ, the system can survive failure of 2 AZ or 2 services at a time. However if the entire DC is offline/unavailable then there will be an outage of Terraform Enterprise. At this moment we do not have the ability to support Active/Active Terraform Enterprise configurations spanning multiple DCs.
5. We do not recommend that Terraform Enterprise be exposed to the "Public Internet" for ingress traffic. Users should be on the company network to be able to access the Terraform Enterprise API/UI. However, Terraform Enterprise needs access to the internet to download providers and binaries etc. If you want to run Terraform Enterprise in a fully internet air gapped mode, we provide a "air gapped" installation mode.

If your organization is sensitive to the cost of our proposed HA based architecture, you can reduce instance counts of EC2 and external services. At the minimum as discussed earlier you need 1 EC2 and 1 Postgres DB to operate the system alternatively Terraform Cloud might be a better option.

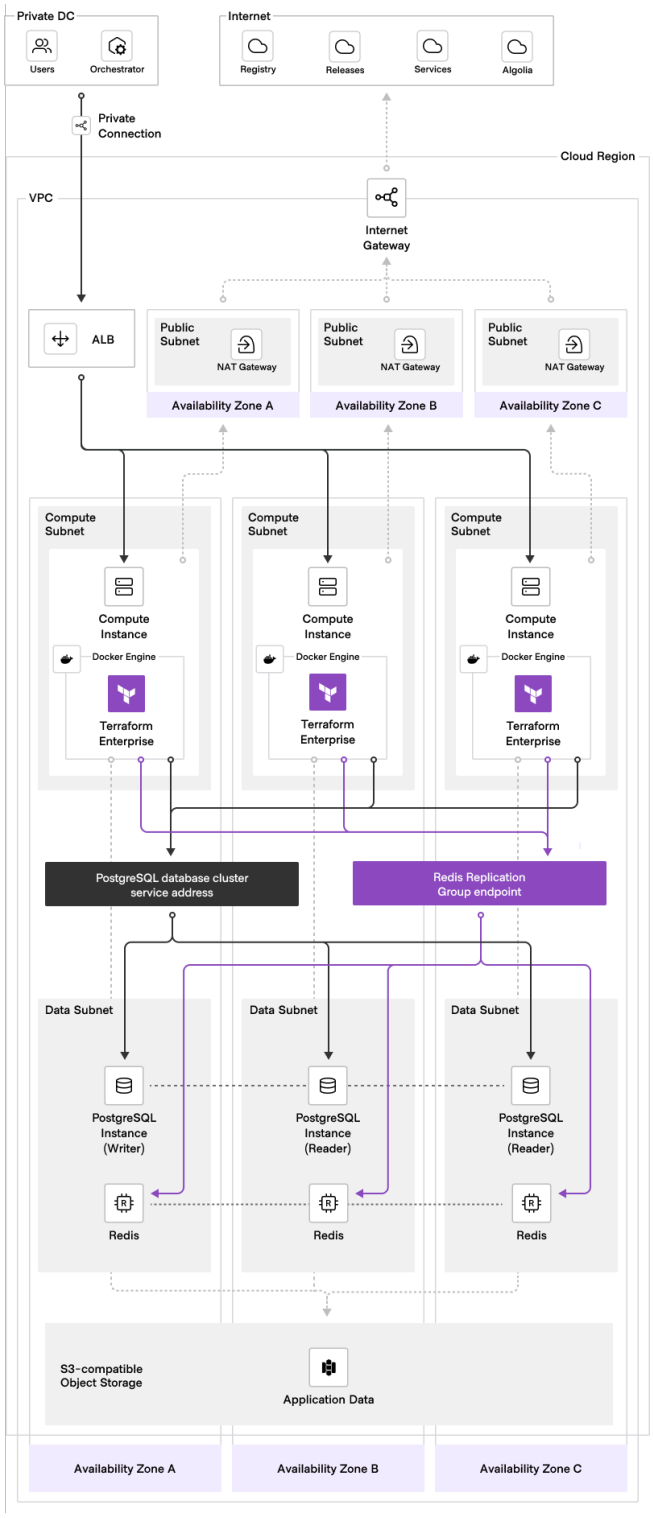


Figure 4: TFE logical architecture

5.2 Terraform modules for installation

The installation code we provide has been used by our partners and HashiCorp professional services to set up best practice Terraform Enterprise. We highly recommend leveraging partner or professional services help to accelerate a robust installation process if possible.

If you will be installing Terraform enterprise yourself we recommend that you follow the following steps:

1. Import the provided Terraform Enterprise code into your VCS Repository.
2. Establish where to store Terraform State. HashiCorp recommends that you store state in Terraform Cloud (free version available or contact HashiCorp account team for an entitlement for state storage just for Terraform Enterprise install). If access to Terraform cloud is not possible we recommend use one of the recommend state storage and locking mechanisms documented here. Note: We do not recommend that you store state in VCS or any unprotected location. Terraform Sate contains sensitive information and needs to be protected. The good thing is that this is the only state you will have to worry about if you do not use Terraform Cloud as all other state generated by your organization will be protected by Terraform Enterprise.
3. Select a machine where the terraform code will be executed. This machine will need to have the Terraform CLI available.
4. Ensure that the cloud credentials are available on the machine for terraform execution (more details provided in the appropriate cloud section).

The HashiCorp account team will provide you with the appropriate Terraform Code to install Terraform Enterprise.

5.3 Overview of the process

If you have chosen to install Terraform Enterprise on AWS, your HashiCorp account team would have provided you with Terraform related AWS code along with this document.

To deploy TFE on AWS using the provided modules, you will need to:

1. (optional) Create the SSL certificates for your TFE installation.
2. Deploy the prerequites with the `terraform-aws-tfe-prerequisites` module.
3. Deploy TFE with the `terraform-aws-tfe` module.

If you have the pre-requisites already installed, then that module is not necessary. You can directly proceed to `terraform-aws-tfe` that installs Terraform Enterprise. Terraform configuration (code) is meant to be self documenting. We recommend that you review the code thoroughly before executing it.

The provided Terraform modules have configurable options to create new AWS resources or reference existing AWS resources. For instance, the module contains conditional variables to determine whether to create, or to consume resources. For example, the TFE on AWS prereqs module contains a conditional variable to determine if the deployment creates an AWS VPC. The conditional variable `create_vpc is` true by default, but you can override it in the module configuration. When the conditional variable is true, the expected behavior is to create a new VPC for Terraform Enterprise.

When you override the default value to false, the deployment team must provide the ID of the target VPC for Terraform Enterprise. In this particular case, the variable `vpc_id` contains the value for the VPC ID. Note the sample excerpt from a `terraform.auto.tfvars` configuration file:

```
# Excerpt from terraform.auto.tfvars
create_vpc = false
vpc_id     = "vpc-1a23456b789c01def"
```

5.4 Review important security groups & ports

Security group	Description
TLS	This security group allows communication between the private subnets in the deployment. The intent is to safeguard private communication with TLS traffic within the VCP. In the scheme, this security group allows communication between the following endpoints in the VPC: - EC2 instances - ECM - SSM - KMS - S3 object storage 443 TLS traffic within the VPC

Security group	Description
Load Balancer	This security group is an intermediary between the public Internet Gateway and the Terraform Enterprise instances. It allows HTTPS (port 443) traffic inbound for Terraform Enterprise. The rules also allow for HTTPS (port 8800) traffic inbound for Terraform Enterprise. It also routes requests for the embedded Vault instances on TCP port 8201 internally, meaning this port is not exposed to the public Internet Gateway.
EC2	This security group controls HTTPS port 443 traffic between the EC2 instances and the Load Balancer security group above. Also, it controls traffic for the embedded Vault instances to communicate with each other in HA mode. When enabled, it allows SSH inbound from a trusted CIDR. Allow external monitoring tools to gather Terraform Enterprise metrics. Port 22 When configured, allows SSH access to the EC2 instances. Port 8201 for Replication traffic for embedded Vault Port 9091/9090 HTTPS/HTTP metrics requests
Redis	This security group controls Redis traffic between the Redis Replication Group and the Terraform Enterprise instances. By default it uses TCP port 6379.
PostgreSQL	This security group controls PostgreSQL traffic between the PostgreSQL cluster and the Terraform Enterprise instances. By default it uses TCP port 5432.

5.5 TFE license

Before starting the TFE installation, make sure you have a valid TFE license. If you don't, first reach out to your HashiCorp account team to request the license file.

5.6 TFE deployment steps using Terraform modules

5.6.1 Create the project directory

Create the deployment directory:

```
mkdir -p fdo-deployment/{1-certificate,2-prerequisites,3-tfe}
mkdir fdo-deployment/{2-prerequisites,3-tfe}/modules
```

Uncompress the archive and move the modules in the proper sub-directories.

Module	Target Location
`terraform-aws-tfe-prerequisites`	`fdo-deployment/2-prerequisites`
`terraform-aws-tfe`	`fdo-deployment/3-tfe`

```
unzip terraform-aws-tfe-bundle.zip
mv terraform-aws-tfe-prerequisites fdo-deployment/2-prerequisites/modules/
mv terraform-aws-tfe fdo-deployment/3-tfe/modules/
```

The directory structure of `fdo-deployement` will look like this:

```
fdo-deployment
`-- 1-certificate
|-- 2-prerequisites
|   `-- modules
|       `-- terraform-aws-tfe-prerequisites
|           |-- LICENSE
|           |-- README.md
|           |-- Taskfile.yaml
|           |-- examples
|           |-- images
|           |-- main.tf
|           |-- modules
|           |-- outputs.tf
|           |-- pkg_modules.json
|           |-- variables.tf
|           `-- versions.tf
`-- 3-tfe
    `-- modules
        `-- terraform-aws-tfe
            |-- LICENSE
            |-- README.md
            |-- examples
            |-- main.tf
            |-- modules
            |-- outputs.tf
            |-- pkg_modules.json
            |-- templates
            |-- variables.tf
            `-- versions.tf
```

Copy the certificate helper code:

```
cp fdo-deployment/2-prerequisites/modules/terraform-aws-tfe-prerequisites/examples/
    hvd-tfe/supplemental-modules/generate-cert/* fdo-deployment/1-certificate/
```

The directory structure of `fdo-deployment/1-certificate` will look like this:

```
.
`--README.md
|-- main.tf
|-- outputs.tf
|-- providers.tf
|--terraform.auto.tfvars.example
|--variables.tf
`--versions.tf
```

5.6.2 Set up Terraform Cloud (optional)

This step is **optional** and will set up workspaces in Terraform Cloud to store your state files.

1. Log in to Terraform Cloud.
2. Create a new **project** named `hvd-tfe-deployment`.
3. Create the **workspaces** according to the table below. Set the workspace for **local execution** mode.

Workspace Name	Purpose
tfe-instance-certificates	The workspace to manage the certificates
tfe-instance-prerequisites	The workspace to manage the TFE infrastructure prerequisites
tfe-instance	The workspace to manage the TFE instance

5.6.3 Create certificates (optional)

This step is **optional** and will create the necessary SSL certificate for TFE.

First move to the proper directory:

```
cd fdo-deployment/1-certificate
```

Configure `backend.tf`

This step is **optional** and required if you are using Terraform Cloud to store your state file.

Create the `backend.tf` file:

```
touch backend.tf
```

Replace the content of `backend.tf` with:

```
terraform {
  cloud {
    organization = "<YOUR TFC ORGANIZATION NAME>"
    workspaces {
      name = "tfe-instance-certificate"
    }
  }
}
```

Set the parameters

Copy the example `tfvars` file:

```
cp terraform.auto.tfvars.example terraform.auto.tfvars
```

Update the `terraform.auto.tfvars` file with your values:

```
route53_zone_name = "<your Route 53 zone name>"
region            = "<target AWS region>"
cert_fqdn         = "<FQDN for your TFE installation>"
email_address     = "<contact email address>"
```

Initialize Terraform

```
terraform init
```

Run a Terraform plan

```
terraform plan
```

Review the plan for any unexpected message.

Run a Terraform apply

```
terraform apply --auto-approve
```

The apply `operation` will create three files:

1. `tfe-no-root.pub`
2. `tfe.key`

3. `tfe.pub`

The directory should look like this:

```
.
`--README.md
|--backend.tf
|--main.tf
|--outputs.tf
|--providers.tf
|--terraform.auto.tfvars
|--terraform.auto.tfvars.example
|--tfe-no-root.pub
|--tfe.key
|--tfe.pub
|--variables.tf
`--versions.tf
```

5.6.4 TFE prerequisites

This step will create the infrastructure resources that are required to run TFE (network, database, etc.).

First move to the proper directory and copy the code example:

```
cd fdo-deployment/2-prerequisites
rsync -avzh --exclude=".*" --exclude="supplemental-modules" \
  modules/terraform-aws-tfe-prerequisites/examples/hvd-tfe/* .
```

The directory structure of `fdo-deployment/2-prerequisites` will look like this:

```
.
`--README.md
|--files
|--main.tf
|--modules
    |--  terraform-aws-tfe-prerequisites
|--output.txt
|--outputs.tf
|--providers.tf
|--terraform.auto.tfvars.example
|--variables.tf
`--versions.tf
```

Adjust the `main.tf` file as follows:

From	To
`source = "../../"`	`source = "./modules/terraform-aws-tfe-prerequisites`

Change the `source` property for the module `pre_req_primary`:

```
module "pre_req_primary" {

  source = "./modules/terraform-aws-tfe-prerequisites"

  ...
}
```

Copy your TFE licence file (`terraform.hclic`) to `fdo-deployment/2-prerequisites/files`.

Configure `backend.tf`

This step is **optional** and required if you are using Terraform Cloud to store your state file.

Create the `backend.tf` file:

```
touch backend.tf
```

Replace the content of `backend.tf` with the example code taken from TFC:

```
terraform {
  cloud {
    organization = "<YOUR TFC ORGANIZATION NAME>"
    workspaces {
      name = "fdo-instance-prerequisites"
    }
  }
}
```

Set the parameters

Copy the example `tfvars` file:

```
cp terraform.auto.tfvars.example terraform.auto.tfvars
```

Update the `terraform.auto.tfvars` file with your values:

```
route53_zone_name    = "<your Route 53 zone name>"
region               = "<target AWS region>"
cert_fqdn            = "<FQDN for your TFE installation>"
email_address        = "<contact email address>"

(...)

friendly_name_prefix = "tfe"

(...)

ssh_public_key       = <replaced with your public key>

(...)

license = {
  name = "tfe-license"
  path = "files/terraform.hclic"
}

(...)

cert_pem_secret = {
  name = "cert_pem_public"
  path = "../1-certificate/tfe-no-root.pub"
}

cert_pem_private_key_secret = {
  name = "cert_pem_private"
  path = "../1-certificate/tfe.key"
}

ca_certificate_bundle = {
  name = "cert_pem_bundle"
  path = "../1-certificate/tfe.pub"
}
```

Update `main.tf` to create the ASG service IAM role

This step is **optional** and only need if you have never created an ASG in the target AWS account.

If you run `terraform plan` and you get the following error message, then you are missing the IAM role and should follow these instructions:

```
|
 Error: reading IAM Role (AWSServiceRoleForAutoScaling): couldn't find resource||

  with module.pre_req_primary.module.iam[0].data.aws_iam_role.asg_role,|
  on .terraform/modules/pre_req_primary.iam/main.tf line 19, in data "aws_iam_role" "
      asg_role":|
  19: data "aws_iam_role" "asg_role" {
```

Set the `create_asg_service_iam_role` to **`true`** in the IAM section of `main.tf`:

```
module "pre_req_primary" {

  # Required if you have never created an autoscaling group before in your aws
      account
  create_asg_service_iam_role = true

}
```

Initialize Terraform

```
terraform init
```

Run a Terraform plan

```
terraform plan
```

Review the plan for any unexpected message.

Run a Terraform apply

```
terraform apply --auto-approve
```

5.6.5 Deploy the TFE instance

This step will deploy the TFE instance.

First move to the proper directory and copy the example:

```
cd fdo-deployment/3-tfe
rsync -avzh --exclude=".*" modules/terraform-aws-tfe/examples/hvd-tfe/* .
```

The directory structure of `fdo-deployment/3-tfe` will look like this:

```
.
|-- README.md
|-- main.tf
|-- modules
|   `--terraform-aws-tfe
|-- outputs.tf
|-- providers.tf
|-- terraform.auto.tfvars.example
|-- variables.tf
`-- versions.tf
```

Configure `backend.tf`

This step is **optional** and required if you are using Terraform Cloud to store your state file.

Create the `backend.tf` file:

```
touch backend.tf
```

Replace the content of `backend.tf` with the example code taken from TFC:

```
terraform {
  cloud {
    organization = "<YOUR TFC ORGANIZATION NAME>"
    workspaces {
      name = "tfe-instance"
    }
  }
}
```

Set the parameters

Copy the example `tfvars` file:

```
cp terraform.auto.tfvars.example terraform.auto.tfvars
```

Update the `terraform.auto.tfvars` file with your values. Use the following correspondance table to set the module parameters:

Deployment Module Input	Prerequisite Module Output
friendly_name_prefix	N/A
ssh_keypair_name	ssh_key_pair
vpc_id	vpc_id
tfe_hostname	route53_failover_fqdn
iam_instance_profile	iam_profile_name
kms_key_arn	kms_key_alias_arn
ec2_subnet_ids	private_subnet_ids
lb_tg_arns	lb_tg_arns
lb_type	lb_type
lb_security_group_id	lb_security_group_ids
s3_app_bucket_name	s3_tfe_app_bucket_name
s3_log_bucket_name	s3_log_bucket_name
cloudwatch_log_group_name	cloudwatch_log_group_name
license_secret_arn	license_secret_arn
enc_password_arn	tfe_enc_password_arn
console_password_arn	tfe_console_password_arn
tfe_cert_secret_arn	cert_pem_secret_arn
tfe_privkey_secret_arn	cert_pem_private_key_secret_arn
ca_bundle_secret_arn	ca_certificate_bundle_secret_arn
db_username	db_username
db_password	db_password
db_database_name	db_cluster_database_name
db_cluster_endpoint	db_cluster_endpoint
asg_hook_value	asg_hook_value
redis_host	redis_primary_endpoint

Deployment Module Input	Prerequisite Module Output
`redis_port`	`redis_port`
`redis_password`	`redis_password`
`redis_security_group_id`	`redis_security_group_ids`
`product`	N/A
`asg_max_size`	N/A
`asg_instance_count`	N/A
`common_tags`	N/A

Update `main.tf`

Adjust the `main.tf` file as follows:

From	To
`source = "../../"`	`source = "./modules/terraform-aws-tfe`

Change the `source` property for the module `tfe`:

```
module "tfe" {

  source = "./modules/terraform-aws-tfe"

  ...
}
```

Initialize Terraform

```
terraform init
```

Run a Terraform plan

```
terraform plan
```

Review the plan for any unexpected message.

Run a Terraform apply

```
terraform apply --auto-approve
```

6 Private cloud installation

Our Terraform-based Terraform Enterprise deployment modules focus on public cloud deployments because public CSPs offer a predictable set of services. Even though the overwhelming majority of our private cloud customers use VMware, there is still a bespoke approach to deployment, instead using a wide variety of additional configuration management technologies and deployment approaches to manage software installation, so an opinionated private cloud deployment Terraform module would be unlikely to be sufficiently useful for all private cloud customers.

This section thus provides recommendations and best practices for scaled, enterprise private cloud customers who need to apply their organisation-standard tooling to the deployment of Terraform Enterprise.

If you are deploying Terraform Enterprise to a public cloud, please refer to the Terraform-based Terraform Enterprise installation section for instructions.

6.1 Deployment topology

As listed in the architecture section, HashiCorp recommends the Active/Active deployment topology for customers seeking to guarantee high availability, resilience and scale.

As an automated deployment is a requirement of Active/Active, engineering to provide this should be a headline planning item for the project manager.

6.2 Component considerations

This section discusses a series of considerations specific to Active/Active private cloud installations irrespective of whether Docker or Kubernetes is used to manage Terraform Enterprise.

6.2.1 Redis

As a result of there being multiple concurrent instance nodes of Terraform Enterprise, your organization must consider the deployment of a Redis instance in your private datacenter as this is a hard requirement for Active/Active. Customers for whom this proves impossible are recommended to consider a public cloud deployment, or Terraform Cloud if SaaS is a viable option for your organization.

To avoid arranging product ownership and ongoing maintenance of Redis, the only self-managed alternative is to use the Single Instance + External Services deployment topology, but this means there being only one Terraform Enterprise container in the instance. In certain circumstances, if the container can be restarted by an orchestrator such as Kubernetes fast enough for your needs, and/or your RTO dictates this as a viable possibility, then Redis can be avoided, however, in our experience, your business will require higher availability and lower RTO over time, so at least, forward planning for Redis support on-premise is recommended.

6.2.2 PostgreSQL

Many customers use alternatives to PostgreSQL as their strategic database solutions. Terraform Enterprise depends specifically on PostgreSQL, which means a private cloud deployment requires that your organization has a pattern for the deployment and production management of PostgreSQL at the versions listed on our website.

Also be aware that there are specific schema requirements in the above link, so you are advised to liaise in the first instance with your DBA team so that all parties are aware of the requirements at the outset.

6.2.3 S3-compatibility

In addition to Redis and PostgreSQL, Terraform Enterprise requires S3-compatible storage to operate. In a private datacenter scenario, this means adopting a third-party technology to present an S3-compatible API to Terraform Enterprise.

Traditionally, HashiCorp has pointed customers who do not already have a solution in place to the most-used examples, however it is not for us to comment on the suitability of one solution for your business over another.

From working with our customers, we see significant success with private cloud deployments of Terraform Enterprise using either Dell ECS or MinIO as the object storage tier. Our MinIO setup guide for Terraform Enterprise is detailed here. If your organization has a pattern for on-premise S3-compatible object storage, our recommendation is to use that.

6.3 Compatibility and components sizing

6.3.1 Compute

We recommend deploying Terraform Enterprise on one the following operating systems:

- Red Hat Enterprise Linux 7.9 or 8.7 (also see RedHat Linux requirements). Our current deployment module currently deploys RHEL 7 but this can be overridden in case of organizational requirement.
- Ubuntu 22.04

Sizing

Each instance of the Terraform Enterprise application service runs as a container in Docker/Kubernetes. The following compute requirements are appropriate for most *initial* production deployments, or for development/testing environments.

- CPU - 4 vCPU
- Memory - 16 GB RAM
- Disk - 1TB

We recommend configuring the CPU and memory availability identically for each Terraform Enterprise application instance. Ultimately, actual computational workload requirements reflect your organization's needs, but these are safe limits HashiCorp uses in scaling tests.

Many of our scaled customers use 8 vCPU and 32GB RAM as an initial production specification. Either way, expect to scale past this as you begin to scale out.

When allocating CPU and memory resources, it is essential to consider the computational needs of the VM(s) and any adjacent containers running in the same orchestration instance. Thus, the CPU and memory allocation of your VM instances must be higher than the maximum configured for the Terraform Enterprise application. Consult the Terraform Operations Guides for more information on ongoing right-sizing and scale out of your Terraform Enterprise instances.

The disk space used by the object store will depend on a few things:

- The number of workspace runs - each run generates objects in the object store irrespective of whether the run succeeds or fails.

- The size of configuration directories submitted to the API during Terraform runs. Terraform runs aggregate the configuration directory and sends it to the API which, in turn, commit it to encrypted object storage using an internal process called archivist. We recommend limiting your configuration repositories to containing only necessary files needed for the Terraform workspace run to proceed. A 50MB `.tgz` file in a configuration directory will needlessly add 50MB to your object store and wastes hosting costs.

Disk

The performance of disk storage available to the Linux operating environment plays a role in the performance of Terraform Enterprise. We recommend using a disk with a minimum of 3000 IOPS to support data operations.

Terraform Enterprise requires a disk with at least 40GB of available space; for Docker deployments this space should be available to `/var/lib/docker`. This limit does not apply to Kubernetes deployments.

6.3.2 PostgreSQL

Sizing

We recommend the following minimum specification for the DB server.

- CPU - 4 core
- Memory - 32 GB RAM
- Disk - 2TB

6.3.3 Redis cache

The supported Redis cache versions are:

- Release 6.x
- Release 7.x

Redis Cluster and Redis Sentinel are not supported.

Sizing

We recommend the following characteristics for the cache.

- CPU - 4 core
- Memory - 16 GB RAM
- Disk - 500GB

6.4 Monitoring PostgreSQL and Redis servers

As a best practice we recommend that CPU, memory, available disk space and disk IO are closely monitored using your standard organizational machine telemetry solution. Create alerts for threshold breaches of 50% and 70% utilization. If utilization of any of the parameters exceeds 70% consistently, consider increasing the resource to ensure optimal performance. We recommend ongoing telemetry reporting, so the team owning the instance are aware of how quickly resources are being consumed.

6.5 Networking requirements

Terraform Enterprise has a number of ingress and egress requirements.

Please refer to the network requirements page for the latest information as these have recently changed. The requirements are specific to our Flexible Deployments Options (i.e. a single container which does not use the Replicated-based software distribution platform).

If a corporate proxy service is filtering outbound Internet traffic, then the destinations detailed in the above link need to be added to your allow-list.

> If your organization is not able to provide access to the required domains, Terraform Enterprise can work in an "Airgapped" mode. Please refer to the documentation here.

Terraform Enterprise works with both layers 4 and 7 load balancers, but consult the architecture section in this documentation for recommendations on TLS management which apply equally here.

6.6 Installation process - Docker

Write an automated Terraform Enterprise installation process which conforms to your organizational requirements and is based on the following. Bash has been used for illustration; the tasks required should be written into automation so that each machine starts up and installs the software the same way.

We recommend reading through the whole document prior to running the steps manually as part of your planning for automation.

6.6.1 Prepare requirements

The following list summarizes the infrastructure resources required prior to the deployment of Terraform Enterprise.

1. VMs for installation of Terraform Enterprise
2. PostgreSQL database
3. Redis cache
4. S3-compatible object storage
5. Terraform Enterprise FDO license file (if you already have a license file but are currently on a Replicated-backed instance, you will need a replacement file - the contract will be the same)
6. TLS certificate bundle
7. Load balancer

6.6.2 Tasks

As root, perform the equivalent of these commands:

```
sudo mkdir -p /opt/tfe/certs
cd !$
```

Put the PEM-encoded server certificate, private key and CA bundle files in this directory. For the sake of this installation, we assume the files are called `cert.pem`, `key.pem` and `bundle.pem`.

```
tfe
`-- certs
    |-- bundle.pem
    |-- cert.pem
    `-- key.pem
```

> Terraform Enterprise cannot use a private key that is protected by a passphrase.

6.6.3 Create a Docker Compose configuration

Write a Docker configuration with the consolidated information about your Active/Active environment. Base this `compose.yml`file on this one. Note that you will be using on-premise S3-compatible object storage, so use the `TFE_OBJECT_STORAGE_*` variables listed in the link as you would were you to deploy Terraform Enterprise on AWS.

> The `compose.yml` contains sensitive information that needs to be protected. Please treat this file as you would any other company secret.

In the configuration properties, we want to highlight the following:

- Treat the `TFE_ENCRYPTION_PASSWORD` value as a sensitive secret. While it can be defined arbitrarily, we recommend using a password generation utility or process to follow corporate policies. Terraform uses this password to decrypt the Vault token stored in the database; the Vault token is used to do all subsequent encryption/decryption processing - see below.
- Terraform Enterprise uses an embedded instance of HashiCorp Vault for data security operations. When identifying the `TFE_VAULT_CLUSTER_ADDRESS` for the Vault instance, point to the private IP of the VM. The internal instrumentation of Vault performs a discovery to find peer instances and add to a private cluster so the IP addresses of all nodes in the cluster need to be accessible to all the other nodes.
- The `TFE_DATABASE_HOST` points to your PostgreSQL database cluster endpoint name.
- The `TFE_REDIS_HOST` points to the endpoint of the primary node in the Redis Replication Group.
- The `TFE_LICENSE` entry requires a raw string from your Terraform Enterprise license file. The license is distributed as a JSON formatted file. The license string is the `data.attributes.signed_string` object.

This is not an exhaustive list of configuration options. Please refer to the configuration reference for documentation about all the configuration options.

Write the `commpose.yml` to a location where your nascent application machines can access it - each of them will need to use it for the installation.

6.6.4 Configure systemd

We recommend using the following systemd manifest to run `docker compose` on system start, so that if the machine reboots, Terraform Enterprise will automatically start. Create `/etc/systemd/system/terraform-enterprise.service` containing

```
[Unit]
Description=Terraform Enterprise Service
Requires=docker.service
After=docker.service network.target

[Service]
Type=oneshot
RemainAfterExit=yes
WorkingDirectory=/etc/terraform-enterprise
ExecStart=/usr/local/bin/docker compose up -d
ExecStop=/usr/local/bin/docker compose down
TimeoutStartSec=0

[Install]
WantedBy=multi-user.target
```

then run

```
systemctl enable --now terraform-enterprise
```

6.6.5 Download the Terraform Enterprise image

First, login to the Terraform Enterprise container image registry using

```
cat <PATH_TO_HASHICORP_LICENSE_FILE> \
| docker login --username terraform images.releases.hashicorp.com --password-stdin
```

If you do not receive the output `Login Succeeded` using the provided license, please contact your HashiCorp representative for support.

Next, pull the container image. Use the form `docker pull images.releases.hashicorp.com/hashicorp/terraform-enterprise:<vYYYYMM-#>` where `<vYYYYMM-#>` is the release required for example:

```
docker pull images.releases.hashicorp.com/hashicorp/terraform-enterprise:v202309-1
```

Check the releases page for the suitable version. We recommend using last month's latest release - if it is October, get `v202309-x` where `x` is the highest listed integer.

If necessary, you can pull the image and push it to your private Docker registry. When placing the image in a private Docker registry ensure to re-tag the image to support your hierarchical organization standards.

6.6.6 Run docker compose

Place the `compose.yml` in your working directory. When done, the directory structure must look like this:

```
tfe
|-- certs
|   |-- bundle.pem
|   |-- cert.pem
|   `-- key.pem
`-- compose.yml
```

Each node will then need to run

```
sudo docker compose up --detach
```

You should then see output messages indicating the progress of the running process. Logs can be followed using this in a separate shell:

```
docker compose logs --follow
```

6.6.7 Run a health check

Terraform Enterprise provides a `/_health_check` endpoint on the instance. If Terraform Enterprise is up, the health check returns a `200 OK`.

The `/_health_check` endpoint operates in 2 modes:

- Full check with `/_health_check?full=1`
- Minimal check with `/_health_check`

With a full check, the service will attempt to verify the status of internal components and PostgreSQL, in contrast to a minimal check which returns `200 OK` automatically after a successful full check.

To run a health check, use:

```
sudo docker exec terraform-enterprise \
  curl http://127.0.0.1:9292/_health_check?full=1
```

The command returns a JSON encoded message as follows:

```
{"postgres":"UP","redis":"UP","vault":"UP"}
```

Use the following or equivalent in your code in order to pause the deployment process until all Terraform Enterprise processes are healthy:

```
while ! docker compose exec terraform-enterprise curl http://127.0.0.1:9292/
    _health_check?full=1;
do
  echo "Waiting for Terraform Enterprise to be ready"
  sleep 30
done
```

The default behavior of this endpoint is to perform a full check during startup of the instance, and minimal checks after Terraform Enterprise is active and running. The Terraform Enterprise application supports a practical `tfe-health-check-status` command to provide full visibility. You can run the command in the VM as follows:

```
sudo docker exec terraform-enterprise tfe-health-check-status
```

When successfully started, Terraform Enterprise returns messages as follows:

```
checking: Archivist Health Check...
|  checks that Archivist is up and healthy
|- ✓ PASS

checking: Terraform Enterprise Health Check...
|  checks that Terraform Enterprise is up and can communicate with Redis and Postgres
|- ✓ PASS

checking: Terraform Enterprise Vault Health Check...
|  checks that Terraform Enterprise can connect to Vault and is able to encrypt and
    decrypt tokens
|- ✓ PASS

checking: Fluent Bit Health Check...
|  checks that the configure Fluent Bit server is healthy
|- SKIPPED

checking: RabbitMQ Health Check...
|  checks that RabbitMQ can be connected to and that we can send and consume messages
|- SKIPPED

checking: Vault Server Health Check...
|  checks that the configured Vault Server is healthy
|- ✓ PASS

  All checks passed.
```

When forcing a full check, take extra caution as every call makes requests to all external services, increasing system load and latency. This is not a concern during the initial installation stages as there are no in-flight operations expected.

6.6.8 Retrieve your initial admin creation token

When you have deployed all Terraform Enterprise application instances, you can retrieve the initial admin creation token. Choose one of your EC2 Instances and run the following command in a terminal session:

```
docker compose exec terraform-enterprise retrieve-iact
```

The command returns your initial admin creation token. The token is sensitive and must be protected as part of the handover process to the operating teams. The form is:

5db5c3ae5be1599804bc614ec976e1a64ad1cadf76fec518d61ed9d04ef565eb

To view the startup web page, open a web browser and navigate to the Terraform Enterprise URL.

The syntax for the URL endpoint is: `$URL/admin/account/new?token=$TOKEN`

For example:

```
https://alb_entry_point.your_domain.com/admin/account/new?
token=5db5c3ae5be1599804bc614ec976e1a64ad1cadf76fec518d61ed9d04ef565eb
```

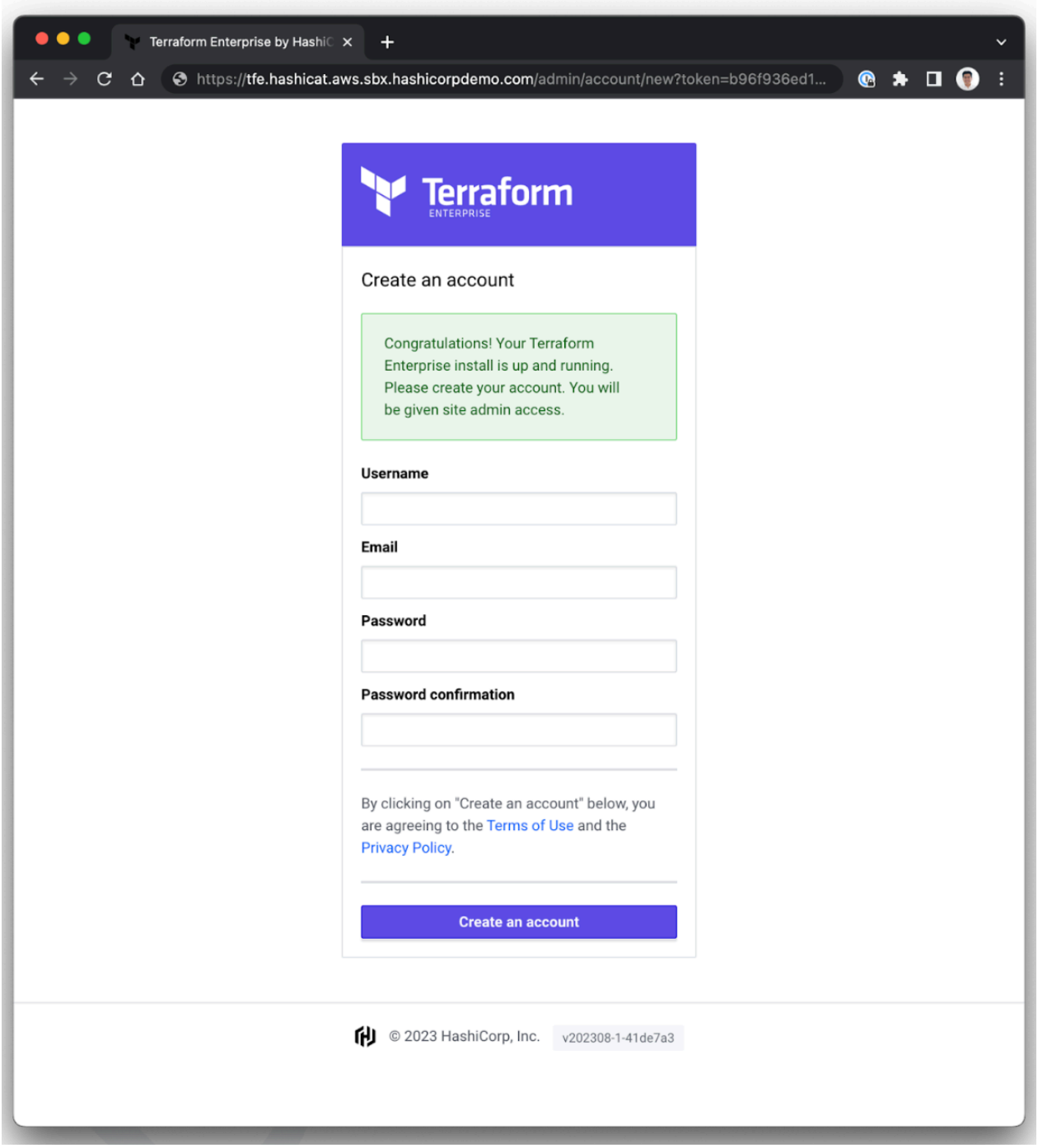


Figure 5: Login page

When the URL is loaded, a form is presented to the operator. The first user created in Terraform Enterprise becomes the site administrator. The only reason to complete this form is when the deployment team is responsible for post-installation tasks of the Terraform Enterprise deployment team.

7 Security hardening

7.1 Operating System

- Primarily, we strongly recommend using operating system configurations which are compliance to the CIS benchmark for the OS.
- Command line access to the machines should be minimized to a limited and well-known team, and access should cause SIEM/audit log to be augmented.

7.2 Application

- Use single-sign-on (SSO) with multi-factor authentication (MFA) for all users.
- The deployment configuration sets up TCP port restrictions for ingress to and egress from the Terraform Enterprise application and related services. These restrictions should not be altered except according to advice from HashiCorp support, a HashiCorp solutions or implementation engineer, or a certified HashiCorp partner.
- Direct SSH access to VMs should be restricted, and access should be available to designated administrators only.
- Enable the `Strict-Transport-Security` response header. Terraform Enterprise allows you to restrict access to the metadata endpoint from Terraform operations, preventing Terraform workspaces from reading any data from the native AWS metadata service.
- When performing a manual installation, set `restrict_worker_metadata_access` as a Docker environment variable to prevent Terraform operations from accessing the cloud instance metadata service. For additional information, please see this page.
- The automated deployment configuration used in this guide restricts the application instances from accessing the AWS metadata service; this should not be re-enabled.
- At the end of a deployment, there is an option to create an initial administrator for Terraform Enterprise. We recommend not creating an account and coordinating a hand-off to the operations team.

8 Next steps

8.1 Terraform Adopt Operating Guide

Please review the Terraform Adopt Operating guide.

8.2 Product support

Please ensure that all members of the Terraform Enterprise administration team:

- Have an account created through the HashiCorp Support Portal.
- Are configured to be an Authorized Technical Contact for your organization in the HashiCorp Support Portal. Authorized Technical Contact are the members of your team who are permitted to open support tickets on behalf of your company.
- Are versed in the guidance we prescribe through Support for Terraform Enterprise documentation for how to open a support ticket with HashiCorp Support, as well as generating and uploading a support bundle, when applicable. Please see the Support for Terraform Enterprise webpage at: https://developer.hashicorp.com/terraform/enterprise/support
- Are aware of what support plan level your company has purchased to manage expectations of response times, as well as the definitions for each severity level that can be selected for a ticket when opening one through the severity documentation. Please see the Customer Success and Enterprise Support web page for severity definitions: https://www.hashicorp.com/customer-success/enterprise-support

9 Appendix

9.1 AWS IAM policy

Sample AWS IAM Policy for S3 Bucket

Please combine the following 3 parts.

Part 1

```
{
  "Statement": [
    {
      "Action": [
        "s3:PutObject",
        "s3:ListBucketVersions",
        "s3:ListBucket",
        "s3:GetObject",
        "s3:GetBucketLocation",
        "s3:DeleteObject"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::hashicat-12a345-tfe-westeros-logging-us-east-2/*",
        "arn:aws:s3:::hashicat-12a345-tfe-westeros-logging-us-east-2",
        "arn:aws:s3:::hashicat-12a345-tfe-westeros-bootstrap-us-east-2/*",
        "arn:aws:s3:::hashicat-12a345-tfe-westeros-bootstrap-us-east-2",
        "arn:aws:s3:::hashicat-12a345-tfe-westeros-app-us-east-2/*",
        "arn:aws:s3:::hashicat-12a345-tfe-westeros-app-us-east-2"
      ],
      "Sid": "InteractWithS3"
    },
```

Part 2

```
    {
      "Action": [
        "kms:ReEncrypt*",
        "kms:GenerateRandom",
        "kms:GenerateDataKey*",
        "kms:Encrypt",
        "kms:DescribeKey",
        "kms:Decrypt"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:kms:us-east-2:441170333099:key/42845c0a-d750-4a1e-b505-26
          a76ebf0035",
      "Sid": "ManagedKmsKey"
    },
    {
      "Action": "secretsmanager:GetSecretValue",
      "Effect": "Allow",
      "Resource": [
        "arn:aws:secretsmanager:us-east-2:441170333099:secret:hashicat-12a345-tfe-
            license-l7ecol",
        "arn:aws:secretsmanager:us-east-2:441170333099:secret:hashicat-12a345-
            enc_password-test-h5Vl8l",
        "arn:aws:secretsmanager:us-east-2:441170333099:secret:hashicat-12a345-
            console_password-test-B9uhcm",
        "arn:aws:secretsmanager:us-east-2:441170333099:secret:hashicat-12a345-
            cert_pem_public-E3PNej",
        "arn:aws:secretsmanager:us-east-2:441170333099:secret:hashicat-12a345-
            cert_pem_private-h5Vl8l"
      ],
      "Sid": "RetrieveSecrets"
    },
```

Part 3

```
    {
      "Action": [
        "logs:PutRetentionPolicy",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "logs:DescribeLogGroups",
        "logs:CreateLogStream",
        "logs:CreateLogGroup"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:logs:us-east-2:441170333099:log-group:hashicat-12a345-tfe-log-group
           :*",
        "arn:aws:logs:us-east-2:441170333099:log-group:hashicat-12a345-tfe-log-group"
      ],
      "Sid": "WriteToCloudWatchLogs"
    },
    {
      "Action": [
        "ec2:DescribeVolumes",
        "ec2:DescribeTags",
        "cloudwatch:PutMetricData"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Action": "autoscaling:CompleteLifecycleAction",
      "Condition": {
        "StringEquals": {
          "autoscaling:ResourceTag/asg-hook": "hashicat-12a345-us-east-2-tfe-asg-hook
             "
        }
      },
      "Effect": "Allow",
      "Resource": "*",
      "Sid": "ASGHook"
    }
  ],
  "Version": "2012-10-17"
}
```

10 Change Log

- [Oct/26/2023] - v1.00.0 GA Release

11 Credits

This document owes its existence to the invaluable contributions of the individuals mentioned below. Hailing from various functional areas and representing diverse departments within HashiCorp, these experienced professionals have played pivotal roles in writing, editing, reviewing, and ensuring the completion of this material. Their collective efforts have been instrumental in shaping the final output.

- Adam Cavaliere
- Adeel Ahmad
- Alex Basista
- Amy Brown
- Anna Chen
- Brandon Ferguson
- Chris Steinmeyer
- Chris Stella
- CJ Obermaier
- Emmanuel Rousselle
- Jenna Degaust
- Judith Malnick
- Kalen Arndt
- Kranthi Katepalli
- Kyle Seneker
- Mark Lewis
- Michelle Roberts
- Prakash Manglanathan
- Purna Mehta
- Randy Keener
- Richard Russell
- Salil Subbakrishna
- Sean Doyle
- Sean Walker
- Simon Lynch
- Zeeshan Ratani