



MORPHEUS

Making Terraform Better with the Morpheus Hybrid Cloud Orchestration Platform

Integration Overview for
HashiCorp Terraform

Last Updated: September 2023

INTRODUCTION

Independently, Morpheus and Terraform can each provide benefits that help increase efficiency and simplify automation across cloud environments. However, leveraging both Terraform and Morpheus together will inherently provide more value by combining the infrastructure-as-code capabilities of Terraform with the multi-cloud governance, workflow automation, and policy engine within Morpheus. In this document, we will cover the additional benefits one would gain by leveraging Morpheus and Terraform in combination. We'll touch on the ability to trigger Terraform files from Morpheus and how to leverage the Terraform Provider for Morpheus. Before we jump in, let's spend a few minutes on the pros and cons of Terraform itself.

INFRASTRUCTURE-AS-CODE (IaC) USING TERRAFORM

Infrastructure-as-Code is the process of managing and provisioning infrastructure by leveraging simple-to-read and -write definition files, (rather than physical hardware configuration or interactive configuration tools). The value of using Infrastructure-as-Code includes speed of execution and automated delivery of infrastructure as well as a reduction of risk associated with human error that can in turn lead to downtime. IaC can also be key in enabling best practices in DevOps because developers become more involved in defining configuration and ops teams get involved earlier in the development process. Another enabler of DevOps is use of IaC to treat infrastructure resources of immutable so you can quickly create and destroy then re-create a fresh instance as needed.

In the case of Terraform, these definition files let you declaratively define the underlying infrastructure in template files (.tf) and manage the infrastructure's state. Other IaC languages include AWS CloudFormation as well as Azure Resource Manager (ARM) templates.

Terraform is cloud-agnostic as it relies on open-source provided cloud and infrastructure providers used in combination with the HashiCorp Configuration Language (HCL) IaC templating language. Terraform has a lot of appeal for software developers and IT infrastructure administrators because creating templates can be done quickly and with relative ease. There is broad open-source community for Terraform, plus engagement from vendors managing a network of Terraform providers for different infrastructure endpoints (VMware, AWS, Azure, etc.).

Terraform can be used to manage the state of infrastructure created with Terraform. However, it will not detect changes in a virtual machine that have occurred because of installing applications locally using a configuration management tool like Chef or Ansible, or manual changes that happen once infrastructure is in use. It is effectively automating 'infrastructure down', while configuration management tools traditionally work 'infrastructure up' to automate application configuration.

To sum it up, here are just some of the top reasons why developers and IT Infrastructure administrators like Terraform:

- Its perceived simplicity - it is like coding your infrastructure in English
- It leverages declarative Infrastructure-as-Code for speed
- It is cloud agnostic with a broad ecosystem
- It is an immutable approach to infrastructure

WHAT ARE SOME OF THE GAPS WITH OPENSOURCE TERRAFORM AND HOW CAN MORPHEUS HELP TO BRIDGE THEM?

BRITTLE IN PRIVATE CLOUD ENVIRONMENTS: Terraform is simple when you want to provision infrastructure using a single Terraform provider to abstract the interface into a public cloud - let's say to AWS, for example; you can write a Terraform file to deploy as many resources as you would like along with associated networks and other abstractions, and quickly deploy the declared infrastructure. But what happens when you need to interact with on-premises infrastructure? You'll need to mirror all the abstractions that public clouds make invisible, which is why private clouds are traditionally so hard to actually deliver.

For example, one provider to deploy a server, another to go grab an IP from an IP Address Management solution (like Infoblox), and another to provision a load balancer (like F5). This scenario requires you to call and maintain multiple Terraform providers, which increases complexity. What if, later, you choose to leverage a different IPAM solution, or a different load balancer technology? With this change you must leverage different Terraform Providers and your previous file(s) are no longer valid. This added complexity and re-work of multiple .tf files introduces fragility and risk.



With Morpheus, you can leverage the platform's dozens of built-in integrations with common on-premises technologies, along with the Morpheus Terraform Provider, to make private clouds as simple and straightforward as public clouds. In fact, with a single Morpheus Terraform Provider, you can consolidate access to both private and public cloud endpoints.


MISSING ROLE-BASED ACCESS CONTROL (RBAC): Control and Identity Access Management is vital to properly scoping and governing the activities of self-service consumers, especially in large Enterprise or Service Provider environments. RBAC is available in the paid version of Terraform (Terraform Enterprise and Sentinel), but this can get expensive and only addresses Terraform - meaning you may still need to invest in other tools such as Ansible Tower to manage RBAC for your other automation tools.



With Morpheus, you can leverage the platform's multi-tenancy and fine-grained role-based access control model to manage how users, groups, and roles can access


	<p>and execute terraform. Simply integrate your enterprise identity provider (Active Directory, Okta, SAML, etc.) and set policies accordingly.</p> <p>These governance policies not only control Terraform, but they also span your hypervisors, clouds, and automation tools - so you truly have a unified control plane.</p>
--	---

CHALLENGES BRIDGING INFRASTRUCTURE, APPS, AND ITSM: Terraform does a good job at tracking infrastructure state, but what about application and environment state? And how to you make sure your CMDB is updated and approvals are in place? Terraform lacks integrations into CM and ITSM solutions, and therefore doesn't address tracking application changes, nor does it integrate into an organization's ITIL processes the way many organizations need to operate at enterprise scale.

	<p>Morpheus integrates into all the popular configuration management tools (Ansible, Puppet, Chef, and SaltStack) and can even enable the well governed mixing and matching of automation task types. Morpheus also integrates deeply into ITSM tools (ServiceNow, Cherwell, and Remedy). This allows Morpheus to be part of the ITIL processes for approvals and CMDB updates, and hence involved in the overall GitOps workflow.</p>
---	--

FOCUSED ON INFRASTRUCTURE, NOT APPS: As noted earlier, Terraform is good at deploying many infrastructure resources to a single cloud and managing the infrastructure state, but what about applications? HashiCorp has embraced the notion of "Immutable Infrastructure," but at the end of the day it's all about the application. If something needs to be updated or fixed, new servers built from a common image (where the updated software is baked in) are provisioned to replace the old ones.

This process can be functional for smaller teams or projects; however, at the enterprise level, a pre-baked application image approach can fall apart. The high number of images and software versions that need to be managed and maintained can become an operations team's nightmare.

	<p>On the other hand, Morpheus defines the application(s) to be provisioned at the logical level of the service provisioning process, all the way down to policy-based automation workflows that need to occur in the provisioning of an environment. This enables Immutable Application Stacks - not just infrastructure. It enables Terraform to do what it does well, but provides a simple hook into configuration management, monitoring, logging, and other application centric tooling.</p>
---	--

NO GRAPHICAL USER INTERFACE (GUI): Terraform is, effectively, a command line tool. There are different tools out there that provide a bolt-on UI to Terraform; however, if you want

to use a HashiCorp UI you must move to Terraform Enterprise, which will provide a graphical user interface at a significant cost premium.



Morpheus provides a robust API for developers, so it certainly checks all the Dev boxes as far as full fidelity API/CLI goes. However, it also appeals to IT Operations and other users that may be more comfortable in a feature-rich GUI or interface from ITSM. Automation teams can focus on building the IaC patterns needed for the business but can then use Morpheus to turn those into easily consumable self-service catalog items to be executed by those without deep automation skills.

MORPHEUS AND TERRAFROM INTEGRATION TYPES

Morpheus has a few ways to interact with Terraform based on the desired outcome. Terraform blueprints, Terraform instance types, and the Morpheus Terraform provider.

1. Terraform Blueprints

The Terraform blueprint type allows you to bring your Terraform directly into Morpheus for immediate consumption. You can create blueprints in Morpheus that include Terraform code pasted directly into the onboard text editor, pull from Git repositories full of Terraform code, or even utilize modular Terraform files in a special Morpheus construct called a spec template, which allows for reuse of micro-templated Terraform code across the platform. Following are a few examples of how this can be utilized:

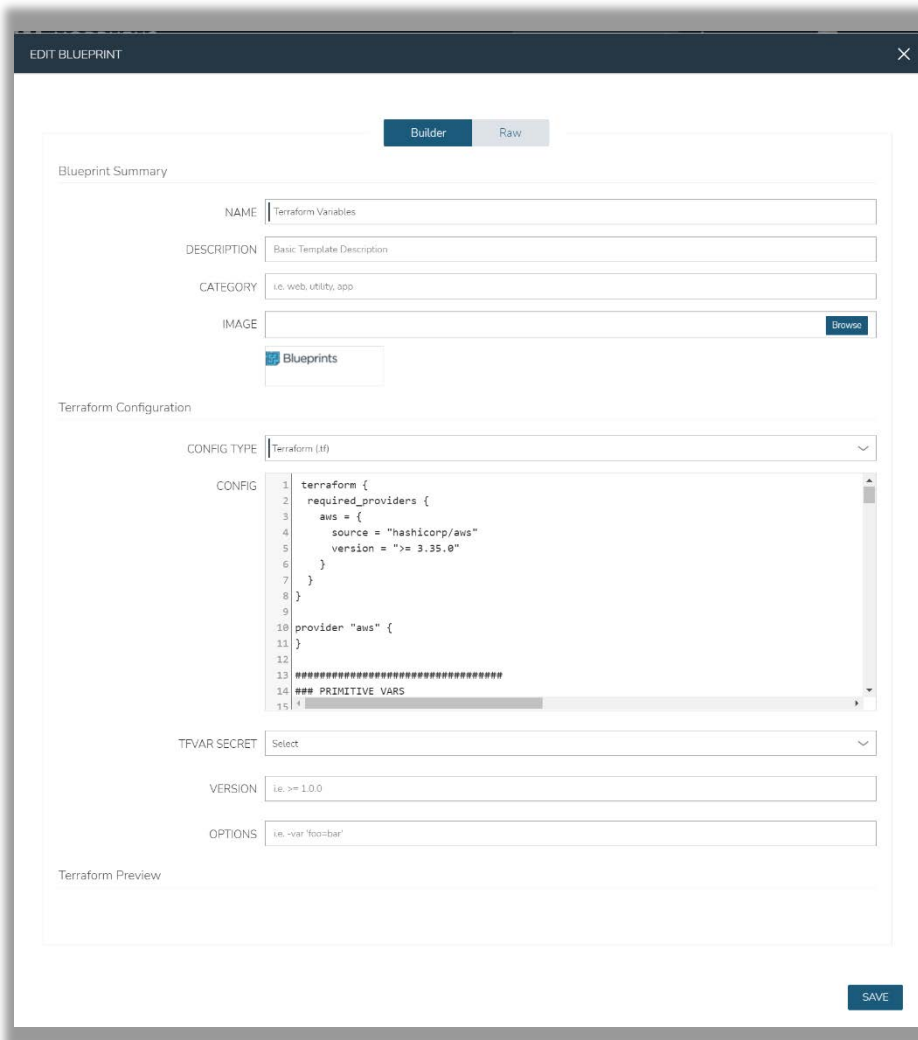


Figure 1: Terraform Code in Text Editor

In Figure 1, you can see the Terraform code is pasted directly into the config pane text editor. This is great for quick, down-and-dirty testing of a piece of Terraform code.

Limitations of this example are that you can only have a single Terraform file pasted into the config. This means large applications become extremely unwieldy and hard to maintain.

NAME > CONFIGURE

Builder Raw

Blueprint Summary

NAME: Terraform Variables Git

DESCRIPTION: Basic Template Description

CATEGORY: i.e. web, utility, app

IMAGE:

Terraform Configuration

CONFIG TYPE: Git Repository

SCM INTEGRATION: Terraform Testing

REPOSITORY: terraform_testing

BRANCH OR TAG: main

WORKING PATH: ./tests/variable_parsing/vars_by_module

TFVAR SECRET: Select

VERSION: i.e. >= 1.0.0

OPTIONS: i.e. -var 'foo=bar'

Terraform Preview

RESOURCES

- random_integer: number
- random_string: string
- random_shuffle: bool
- null_resource: string
- null_resource: number
- null_resource: bool

FILES

- output.tf
- provider.tf
- main.tf
- primitive_vars/vars.tf
- primitive_vars/output.tf

PREVIOUS COMPLETE

Figure 2: Terraform Blueprint from Git

In Figure 2, you can see that the Terraform code is being provided via a Git repo. In the preview pane, you can see the files and folders that are readable from the working path directory provided. What you cannot see in the screenshot is that Morpheus also shows the providers utilized, as well as the modules referenced in the Terraform code.

The screenshot shows the 'Builder' tab of the Terraform Blueprint interface. It contains the following fields and sections:

- Blueprint Summary:**
 - NAME: Terraform Variables by Spec
 - DESCRIPTION: Basic Template Description
 - CATEGORY: i.e. web, utility, app
 - IMAGE: [Browse]
 - Blueprints
- Terraform Configuration:**
 - CONFIG TYPE: Terraform Specs
 - SPEC TEMPLATES:
 - vars_by_spec: main
 - vars_by_spec: output
 - Terraform Provider: random
 - TFVAR SECRET: Select
 - VERSION: i.e. >= 1.0.0
 - OPTIONS: i.e. -var 'foo=bar'
- Terraform Preview:**
 - PROVIDERS:
 - random
 - MODULES:
 - primitive_vars
 - RESOURCES:
 - random_integer: number
 - random_string: string
 - random_shuffle: bool

A 'SAVE' button is located at the bottom right of the form.

Figure 3: Terraform Blueprint from Spec

In Figure 3: You can see that there are three (3) Morpheus spec templates utilized to create the blueprint. The main and output TF files are directly tied to the deployment that I would like to create. The provider TF is its own spec template so that I can reuse it for other Terraform deployments that would require said provider block.

Each of these methodologies provide different avenues to utilize already existing Terraform code to create the blueprint based on the configuration of your application deployments. The one drawback to any of these deployment types is that they lack a direct tie into the power of the Morpheus provisioning automation engine.

2. Terraform Instance Types

Terraform Instance Types take Terraform out of the monolithic blueprint construct and allow you to create a Morpheus instance of Terraform deployed code. This Instance Type can have multiple layouts, each representing a different configuration of a similar construct. Think RDS deployments on different sizes, for example. Terraform spec templates, either created locally or pulled in from Git, are used to stitch together the Terraform instance type. If you recall from the spec template powered blueprint, each Terraform file is its own spec template. This same thing applies to Instance Types.

EDIT LAYOUT

NAME Private Module

VERSION 1.0

DESCRIPTION

CREATABLE

TECHNOLOGY Terraform

MINIMUM MEMORY 0 MB
This will override any memory requirement set on the virtual image

WORKFLOW Select Workflow

SUPPORTS CONVERT TO MANAGED

TFVAR SECRET Select

ENVIRONMENT VARIABLES
Name Value

Inputs

Search inputs

Spec Templates

Search Templates

- Private Module: output
- Terraform Provider: Random
- Private Module: main

SAVE CHANGES

Figure 4: Instance Type Layout

In Figure 4 you can see that the same provider spec template is being utilized from the blueprint. This is one of the benefits of the spec template methodology.

You will also notice that you have the option to select a workflow for this layout. Instance Types grant the ability to utilize the lifecycle automation capabilities of Morpheus alongside your Terraform code. This is very useful for taking actions after resource deployment to perform application configuration. It is one of the stronger reasons for utilizing Instance Types over blueprints.

For example, this capability is currently being used to deploy classroom training environments for Morpheus' own training classes. Terraform is utilized to spin up a VPC in AWS with networks and security groups as required, EC2 instances for the class (one of which gets Morpheus installed on it), and an IAM user with secret and access keys and all the permissions Morpheus needs to manage the account. All of this is spun up per student to grant a whole environment to them. Once the Terraform deployment is complete, a Morpheus provisioning workflow takes over and performs configuration on the Morpheus appliance to do the initial setup and configuration utilizing Python to make API calls to the appliance. An email is then sent to the instructor who initiated the classroom build with all the information about every student's environment for them to disseminate to the class.

One of the downsides of utilizing the Instance Type is that you cannot rely on local modules or files, as each spec template is cloned into its own workspace directory to run the Terraform plan, apply, and destroy commands from.

How to Decide Between Instance Types and Blueprints

This is going to be *highly* dependent on your current methodology for utilizing Terraform, your tolerance for code changes, and your desire to utilize the built-in functionalities of Morpheus with your Terraform deployments.

If you have large Terraform deployments with local file references or local modules, then you will most likely want to stick with the Git-based blueprint option.

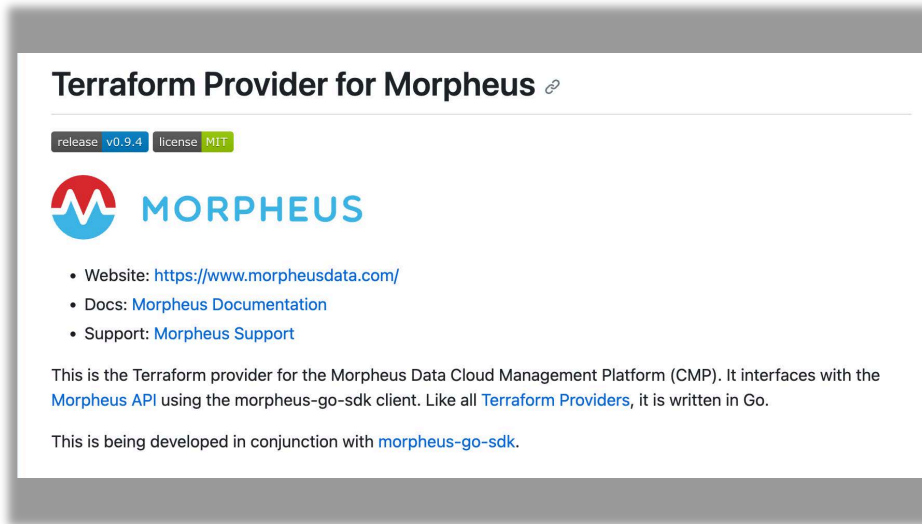
If you are calling a lot of remote modules and need a place to run a main.tf that utilizes those remote modules, or you want to utilize the provisioning lifecycle capabilities of Morpheus Provisioning Workflows, then Instance Types are the way to go.

3. Terraform Provider for Morpheus

The Terraform provider for the Morpheus Data appliance can be found on GitHub. <https://github.com/gomorpheus/terraform-provider-morpheus>

There's a distinct advantage to leveraging the Morpheus provider for on-premises infrastructure because it can become a single provider to offload the responsibility of provisioning and orchestrating heterogeneous infrastructure and managing application state. Morpheus integrates out of the box with 80+ tools and 20+ clouds. By writing Terraform files

to call Morpheus, you can let the CMP do all the heavy lifting and reduce the brittleness of IaC deployments.



There are several other benefits that Morpheus brings to the equation. Let's touch on some of them.

MORPHEUS AND TERRAFORM ARE BETTER TOGETHER

Enhance your overall security posture with Morpheus RBAC

As noted earlier, Terraform is limited when it comes to governance policy and RBAC on its own. If you have access to Terraform, you can plan/apply .tf files but there is no way for IT teams to control access to those .tf files or audit the result of execution. By integrating Terraform with Morpheus, you can take advantage of the fine-grained governance and RBAC available in Morpheus to extend those capabilities when providing access to Terraform files. Morpheus governance and control goes well beyond just Terraform. Because Morpheus is a Multi-Cloud Provisioning, Automation, and Orchestration platform, Morpheus takes that governance to Private and Public Clouds alike, as well as any additional integrated technologies, like configuration management, ITSM, logging, and backup systems. You can find all the integrations documented here:

https://docs.morpheusdata.com/en/latest/integration_guides/integration_guides.html#integration-guide

Morpheus Cypher Service for secure secret management

Morpheus has a built-in service called Cypher. It's a secure Key/Value store. Cypher allows the storage of secret data (like credentials or variable sets) in a highly encrypted way for future retrieval. When leveraging Morpheus and Terraform, these variables can be seamlessly passed to Terraform. Secrets and variables only need be maintained and managed in one place.

If you recall back to Figure 4, you may have noticed the names of the spec templates being utilized:

Private Module: main

Private Module: output

This particular instance type layout is making a call to a private repository in GitLab as the source for a module. The module is very basic and just takes some inputs, creates some null resources, and outputs the inputs back for you. The main for this spec template is simply calling that module and passing some variables to it. The keen-eyed will notice that there is a sneaky bit of code smooshed into the source url.

```
<%=cypher.read('secret/gitlab_tf_token') %>
```

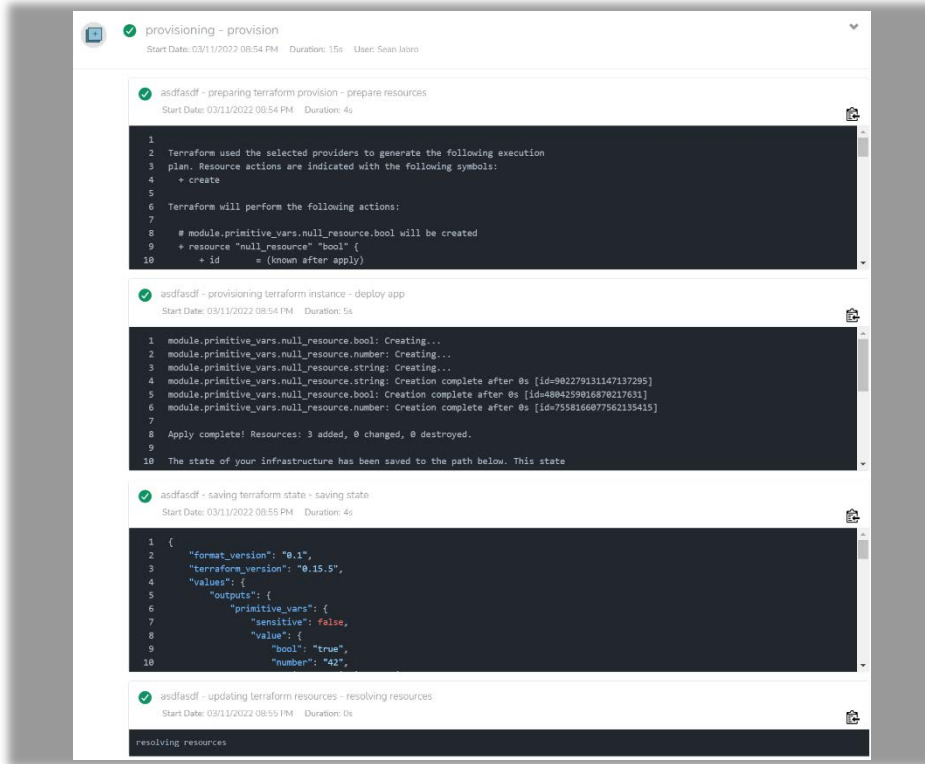
This little snippet takes an access token from GitLab stored securely in Morpheus Cypher as `gitlab_tf_token` and injects it into the url for the module

```
module "primitive_vars" {
  source =
  "git::https://oauth2:<%=cypher.read('secret/gitlab_tf_token') %>@gitlab.com/**redacted**/private_terraform_modules.git//random/primiti
  ve_vars"

  string = "This is a string"
  number = 42
  bool = true
}
```

Enable Execution History and Auditing of all Terraform actions

From Morpheus, whether you provision a Terraform blueprint or make any changes and plan/apply again, the output is available in the History tab of that instance.



HOW TO USE TERRAFORM WITH MORPHEUS

Let's go ahead and discuss the setup required for this integration, and then the interaction between Terraform and Morpheus. As mentioned earlier there are multiple ways to integrate and leverage Terraform with Morpheus. By creating Application blueprints that leverage Terraform files, or by creating instance types that utilize spec templates, the Terraform Provider for Morpheus allows you to run configurations and deployments against the Morpheus platform utilizing Terraform HCL.

Terraform Installation

The first step in taking advantage of this integration is to install Terraform on the Morpheus appliance. The great news is that Morpheus can automatically install Terraform locally based on a few different options. Firstly, you will need to set the Terraform Runtime to "auto" under Administration -> Settings -> Provisioning. You can also set your default Terraform version here. This is the version of Terraform that will be utilized by any deployment that does not specify another version to run.

Terraform Settings

Terraform Runtime

Default Terraform Version

Figure 5: Terraform Auto Runtime

In a spec template, you have the option to specify the version of Terraform. (See Figure 6) If you do not specify a version, then the default version from Administration -> Settings -> Provisioning will be used.

EDIT SPEC TEMPLATE

NAME

CATEGORY

TYPE

SOURCE

REPOSITORY

FILE PATH

VERSION REF

VERSION

Figure 6: Terraform Version in Spec Template

NOTE: Be sure not to mix and match Terraform versions in you spec templates on a single instance type or blueprint, as this can cause issues within the system.

You can also manually install and configure Terraform on the Morpheus appliance if you wish. More details on the manual installation are available here:

https://docs.morpheusdata.com/en/latest/integration_guides/Automation/terraform.html

GitHub/Git Repository Setup

To use .tf files from a Git repo, GitHub integration needs to be configured in Administration - Integrations. If one is not configured .tf or .tf.json files can be manually added to the Terraform application blueprints.

EDIT GIT INTEGRATION

NAME

ENABLED

GIT URL

USERNAME

PASSWORD

ACCESS TOKEN

KEY PAIR

SAVE CHANGES

Now that the setup is complete, we can leverage Terraform.

SUMMARY

We've discussed what Terraform is, and we've touched on the gaps that it has (especially in a heterogeneous private cloud), and when provisioning multi-cloud infrastructure and applications. We've noted the benefits of using Morpheus and Terraform together, and how Morpheus helps close the Terraform gaps. We've also highlighted how easy it is to setup this integration to take advantage of these benefits. We also discussed the challenges that Terraform has with private cloud, and by far the best way to leverage Terraform to provision to private cloud is by using the Terraform Provider for Morpheus. You only write to one provider, and Morpheus takes care of provisioning and state management of all the private cloud components, solutions, and constructs. Let Morpheus be the arbiter of all things private cloud, and multi-cloud! One final point I'd like to make is that any organization looking to procure Terraform Enterprise can save costs by leveraging Morpheus and open source Terraform.

To get started with Morpheus today, please visit www.morpheusdata.com Request a demo of Morpheus at www.morpheusdata.com/demo/