Migrating A Lot of State with Python and the Terraform Cloud API

Neil Dahlke
Apr 10, 2020 · 10 min read

Overview

One of the first questions I hear in the field after explaining the merits of Terraform Cloud (TFC) is: how hard is it to migrate all my existing state? This is a fair question. When you have built up years worth of Terraform configuration and state over time, it can be a daunting proposition to consider migrating it all to one centralized platform. That is, if you haven't read this blog post.

For those who may not yet be familiar with it, TFC is an application that allows teams to use Terraform together in a consistent and reliable environment with access controls and reliable state management features. HashiCorp also provides a self-hosted distribution of TFC, called Terraform Enterprise (TFE).

In TFC and TFE, infrastructure is organized into **Workspaces**. Each TFC workspace is typically linked to a VCS repository (which contains the Terraform configuration) and has its own associated metadata (variable values, credentials, and secrets).

If you've used Terraform OSS in the past, there's no doubt you're familiar with the **State Files** that Terraform configurations have. Each TFC workspace also has its own state file. When you're planning for a migration to TFC or TFE, you can project that the number of Terraform OSS state files you currently have will map roughly one to one to the number of TFC workspaces you will need.

Note: TFC workspaces are different from **Terraform OSS workspaces**, which allow a single Terraform OSS configuration to have multiple state files. Each TFC workspace is associated with a single "default" Terraform OSS workspace and only has a single state file. So, if you have a Terraform configuration that uses 3 Terraform OSS workspaces with 3 corresponding state files, you will migrate that configuration to 3 TFC

workspaces. Finally, if your existing Terraform configurations use the current workspace interpolation, `${terraform.workspace}`, you will need to remove it because it will always evaluate to "default" in TFC.

The above discussion of workspaces and state files leads us to a big question: What do I do if I have *a lot* of state files?

While there are some good recommendations in the TFC docs for Migrating State from Local Terraform to Terraform Cloud and Migrating State from Multiple Local Workspaces, the methodology documented there could be burdensome for a large number of state files. However, with a little creativity and the help of the TFC API, we can automate this painful process away.

Challenge

Consider the following challenge: your team has been leveraging the GCS backend with Terraform OSS and has a bunch of state files in a GCS bucket. You've already either signed up for a Terraform Cloud account or stood up your own Terraform Enterprise deployment. You have also already connected your VCS provider. You want to migrate entirely to TFC, but you don't want to spend hours updating the backend configuration in each of the workspaces and migrating the state files one at a time.

Note: You'll be using Python in this example, and will lean on the Google Cloud Storage Python library and `terrasnek`, a Python library for the TFC API. While all of the code below is Python, the concepts can be translated to any language you want and there are other client libraries and tools available here.

You've created a GCS bucket called `hc-neil` (great choice!) with three state files in it. Each of them will eventually be mappped to a TFC VCS repository in your connected VCS provider. You might have many more than three state files in a bucket. While this example focuses on GCP, you can easily translate all of this work to another cloud provider like AWS or Azure.

If you use the `gsutil` CLI tool to traverse the bucket, you'll see something like the following output.

```
gs://hc-neil/demo-tfstates/:

gs://hc-neil/demo-tfstates/
```

```
gs://hc-neil/demo-tfstates/demo-project-aws/:

gs://hc-neil/demo-tfstates/demo-project-aws/

gs://hc-neil/demo-tfstates/demo-project-aws/two-tier-demo-app-
aws.tfstate

gs://hc-neil/demo-tfstates/demo-project-azure/:

gs://hc-neil/demo-tfstates/demo-project-azure/

gs://hc-neil/demo-tfstates/demo-project-azure/two-tier-demo-app-
azure.tfstate

gs://hc-neil/demo-tfstates/demo-project-gcp/:

gs://hc-neil/demo-tfstates/demo-project-gcp/

gs://hc-neil/demo-tfstates/demo-project-gcp/two-tier-demo-app-
gcp.tfstate
```

That output is helpful in telling you where all the state files live; determining that is the first step in mapping them over to TFC. You'll also want to take a look at your VCS repository's layout, so that you understand which Terraform configuration code you will eventually map to each state file.

```
./two-tier-tfc-demo-app:

aws azure gcp

./two-tier-tfc-demo-app/aws:

main.tf outputs.tf terraform.tfvars variables.tf versions.tf

./two-tier-tfc-demo-app/azure:

main.tf outputs.tf terraform.tfvars variables.tf

./two-tier-tfc-demo-app/gcp:

main.tf outputs.tf terraform.tfvars variables.tf versions.tf
```

The Approach

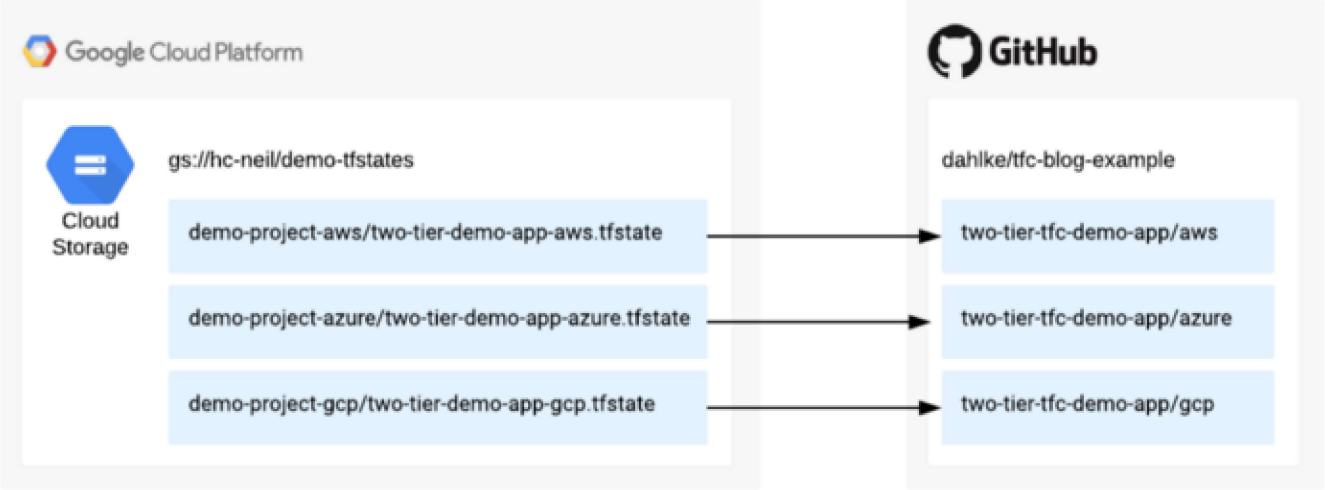
Knowing both the state file layout in GCS and the Terraform Configuration layout in your VCS repository, you can start to formulate a mental model of how they will be

mapped to each other.

```
./two-tier-tfc-demo-app/aws → gs://hc-neil/demo-tfstates/demo-project-aws/

./two-tier-tfc-demo-app/azure → gs://hc-neil/demo-tfstates/demo-project-azure/

./two-tier-tfc-demo-app/gcp → gs://hc-neil/demo-tfstates/demo-project-gcp/
```



Mapping GCS buckets to VCS repositories, illustrated

Once you understand how they map to each other, you can begin the automation process by defining a migration strategy, implemented in this case in a JSON file so that it is machine-readable. Let's call it the "Migration Map". You'll also add a couple other pieces of data to the migration map, including the branch of the VCS repository you want to use, the Terraform version to use, and the name you want to give to the workspace.

```
[
    {
        "gcs-blob-path": "demo-tfstates/demo-project-aws/two-tier-demo-app-aws.tfstate",
        "statefile-local-path": null,
        "repo": "dahlke/tfc-blog-example",
        "working-dir": "AWS-two-tier-tfc-demo-app/aws",
        "workspace-name": "AWS-gcs-to-tfc-demo",
        "branch": "master",
        "tf-version": "0.12.1"
    },
```

```
        ...
    ]
```

In the end, you should have one of these JSON objects for each state file you plan to migrate. With this migration strategy defined in a machine readable format, you can begin prepping your script for migration.

Migration Prep

Step 1: Configure Environment for Google Compute Engine

In order to make requests against the GCP API using the Google Cloud Storage Python library, you will need to authenticate. There are some instructions for setting up authentication here. Make sure that you follow these instructions and set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable properly for your OS.

```
export
GOOGLE_APPLICATION_CREDENTIALS="/Users/neil/.gcp/hashi/foo.json"
```

Warning: Do NOT store these credentials in VCS.

You'll also have to use an environment variable to tell the script which GCS bucket to read from.

```
export GCS_BUCKET_NAME="hc-neil"
```

Step 2: Configure Environment for TFC

Once you have programmatic access to the GCS bucket with your state files in it, you need to configure authentication for programmatic TFC API access.

Terraform Enterprise has multiple different types of tokens (User, Team and Organization) which come with different access levels. For simplicity, let's assume that you are a TFC admin and are using a User Token. Follow the instructions to generate a new user token for yourself and then set it as the `TFC_TOKEN` environment variable.

```
export TFC_TOKEN="YOUR_TFC_USER_TOKEN"
```

Warning: Do NOT store this token in VCS.

Since you're going to connect each TFC workspace you create to a VCS repository, you'll need to get the OAuth Token ID for your VCS from TFC. This is simple: Go to your organization settings, choose VCS Providers, and then copy the OAuth Token ID you want to use. Then export it.

```
export TFC_OAUTH_TOKEN_ID="YOUR_TFC_OAUTH_TOKEN"
```

You'll also need to tell the script where your TFC or TFE instance lives, and what organization to use. For example, if you were using HashiCorp's managed deployment of Terraform Cloud, you would use `https://app.terraform.io`.

```
export TFC_ORG="YOUR_TFC_ORG"
export TFC_URL="YOUR_TFC_URL"
```

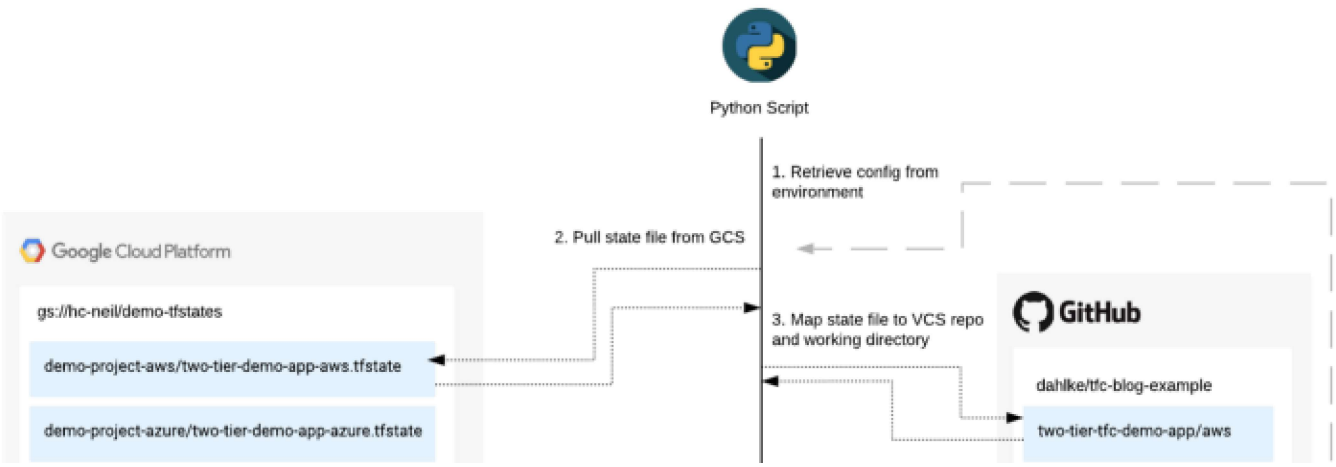
Step 3: Install the Python Dependencies

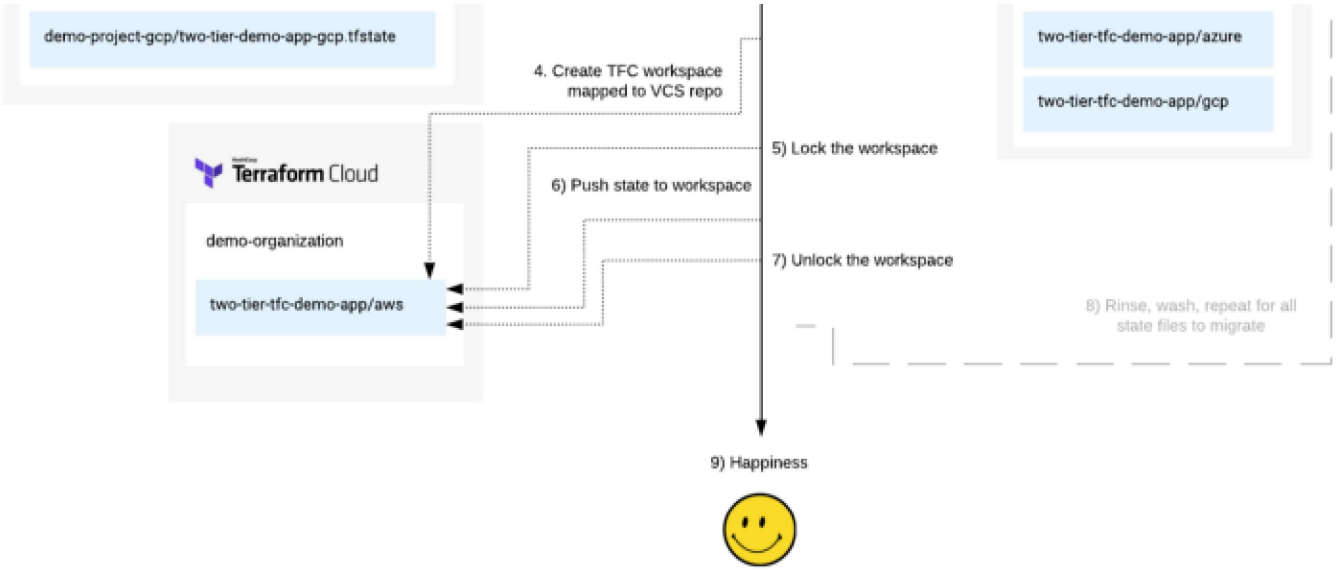
Since this example uses Python, you need to install some dependencies to make writing our script a little easier.

```
pip3 install google-cloud-storage==1.26.0
pip3 install terrasnek==0.0.2
```

With all the pieces in place, you can execute the migration.

Executing the Migration





Migration workflow, illustrated

You can see the workflow above. Below, you'll walk through each component. Since the focus in this blog is to help you understand the individual steps, we will focus on those. If you want to see the full script that you can use for automation, it is available here.

Step 1: Runtime Configuration

The first thing to do is read in all the variables your script will need from the environment. You'll notice that the GCP credentials are not read in explicitly since that is done by the GCS library itself. You also should read in the Migration map from your JSON file. Once all the configuration is loaded into the script, you can create the GCS and TFC clients.

```
TFC_TOKEN = os.getenv("TFC_TOKEN", None)
TFC_URL = os.getenv("TFC_URL", "https://app.terraform.io")
TFC_ORG = os.getenv("TFC_ORG", None)
GCS_BUCKET_NAME = os.getenv("GCS_BUCKET_NAME", None)
migration_targets = []

# Read the Migration Map from the file system into a Python dict
with open("migration.json", "r") as f:
    migration_targets = json.loads(f.read())

# Create the GCS client
storage_client = storage.Client()

# Create a Terraform Enterprise client with the TFC_TOKEN from the
# environment
api = TFC(TFC_TOKEN, url=TFC_URL)

# Set the organization to work in for our client
api.set_organization(TFC_ORG)
```

Now, the script is ready to start pulling down state files and pushing them up to TFC.

Step 2: Retrieve the Latest State Files and Save Locally

Once authenticated, you'll pull down the state file from each specified GCS blob path.

Warning: This demo does not account for users accessing GCS while the migration is being performed. If this is a concern, be sure to lock your state before proceeding.

```
# Connect to the bucket we want to download blobs from
bucket = storage_client.bucket(GCS_BUCKET_NAME)

# Retrieve the path from the migration target dict
blob_path = mt['gcs-blob-path']

# Create a blob object
blob = bucket.blob(blob_path)

# Extract the state file name from the blob and use
# it to define the path we want to save the state file
# locally
statefile_name = blob.name.split("/")[-1]
statefile_path = f"statefiles/{statefile_name}"

# Download the state file to the local path just defined
blob.download_to_filename(statefile_path)

# Add the local path we saved the state file to
# in the migration targets dict for usage later
mt["statefile-local-path"] = statefile_path
```

Step 3: Create a TFC Workspace

Up until this point, you have not used the TFC API at all. Let's change that. The first thing you'll need to do in TFC is create a workspace, so you'll want to reference the Workspace API docs to do that. Target the "With a VCS Repository" sample payload and then start building your own.

```
# Configure our create payload with the data
# from the migration targets JSON file
create_ws_payload = {
    "data": {
        "attributes": {
            "name": mt["workspace-name"],
            "terraform_version": mt["tf-version"],
```

```
                "working-directory": mt["working-dir"],
                "vcs-repo": {
                    "identifier": mt["repo"],
                    "oauth-token-id": oauth_client_id,
                    "branch": mt["branch"],
                    "default-branch": True
                }
            },
            "type": "workspaces"
        }
    }

    # Create a workspace with the VCS attached
    ws = api.workspaces.create(create_ws_payload)

    # Save the workspace ID for usage when adding a state version
    ws_id = ws["data"]["id"]
```

You can see directly how some of the keys you configured in your migration map are leveraged here. You'll use `workspace-name`, `repo`, `tf-version`, `branch`, and `working-dir` to properly set up your workspace. You'll also want to save the workspace ID after you create it for the next step.

Step 4: Upload the State File as a State Version

Your TFC workspace is now created and connected to your specified VCS repository, however, it still does not have the state you wanted to migrate over. This part can be a bit tricky. The state file you downloaded earlier is in plaintext. The State Versions API requires that the state file be encoded as a base64 string, and you need to provide an md5 hash of that so Terraform can verify the upload.

```
    # Read in the state file contents we just pulled from GCS
    raw_state_bytes = None
    with open(mt["statefile-local-path"], "rb") as infile:
        raw_state_bytes = infile.read()

    # Perform a couple operations on the data required for the
    # create payload. See more detail here:
    # https://www.terraform.io/docs/cloud/api/state-versions.html
    state_hash = hashlib.md5()
    state_hash.update(raw_state_bytes)
    state_md5 = state_hash.hexdigest()
    state_b64 = base64.b64encode(raw_state_bytes).decode("utf-8")
```

Now you have massaged your state file into base64 and have created an MD5 hash of it. You can build the payload for creating a State Version. Please reference the State

Versions API docs.

```
# Build the payload
create_state_version_payload = {
    "data": {
        "type": "state-versions",
        "attributes": {
            "serial": 1,
            "md5": state_md5,
            "state": state_b64
        }
    }
}
```

There is one last caveat: you cannot upload state to a workspace that is not locked. This is done to make sure you don't modify state underneath an existing run. You'll need to lock the workspace, create your state version, and then unlock the workspace.

```
# State versions cannot be modified if the workspace isn't locked
api.workspaces.lock(ws_id, {"reason": "migration script"})

# Create the state version
api.state_versions.create(ws_id, create_state_version_payload)

# Unlock the workspace so other people can use it
api.workspaces.unlock(ws_id)
```

Now you have a complete pipeline. You have pulled state files from GCS, created new TFC workspaces to hold those state files, and uploaded those state files to your new workspaces. Congrats on an easy migration!

Conclusion

The TFC API provides much more value than just migration — it is the key to unlocking the true automation potential of Terraform Cloud and Terraform Enterprise. While this simple example focuses primarily on state and workspaces, you can use that same API to manage variables, trigger runs, export plan and apply logs, get results from cost estimates and Sentinel policy checks, and more. The Terraform Cloud API is a very powerful tool in your tool belt. I hope you can use this blog post as inspiration for automating your migration — get building!

Full script available in the GitHub repository.

Note: All of the code is available in this repository. While this is a GCP-based example, you can apply the same logic of extracting state files and creating TFC workspaces to your favorite cloud provider.

Terraform State Files Migration Terraform Cloud Hashicorp