**Terraform IaC Runbook**

Last Modified Date: September 8, 2022

Author(s): Viktor Lee

Version   
State: Final

**Version Control**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Changes** |
|  | August 2022 | Viktor Lee | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Contents**

[1. Introduction 4](#_Toc113524150)

[**1.1** **Purpose** 4](#_Toc113524151)

[**1.2** **Audience** 4](#_Toc113524152)

[**1.3** **Document Syntax** 4](#_Toc113524153)

[2. Infrastructure as a code (IaC) 4](#_Toc113524154)

[**2.1** **Introduction of IaC** 4](#_Toc113524156)

[**2.2** **Terraform Overview** 5](#_Toc113524157)

[**2.3** **Terraform Module Assets** 6](#_Toc113524158)

[**2.3.1** **Objective** 6](#_Toc113524159)

[**2.3.2** **Maintenance** 7](#_Toc113524160)

[**2.3.3** **Client specific customization** 7](#_Toc113524161)

[**2.4** **Reference Architecture** 9](#_Toc113524162)

[3 Terraform User Guide 9](#_Toc113524163)

[**3.1** **Installing Terraform** 9](#_Toc113524164)

[**3.2** **Terraform Configuration** 12](#_Toc113524165)

[**3.3** **Running Terraform** 14](#_Toc113524166)

[**3.4** **Using the Terraform module** 15](#_Toc113524167)

[4 Azure DevOps for Terraform 16](#_Toc113524168)

[**4.1** **Creating a repository** 16](#_Toc113524169)

[**4.2** **Service Connection** 19](#_Toc113524170)

[**4.3** **Terraform State File** 21](#_Toc113524171)

[**4.4** **Azure DevOps extension** 23](#_Toc113524172)

[**4.5** **PAT token and central repo for modules** 24](#_Toc113524173)

[**4.6** **Creating a release pipeline for deployment** 26](#_Toc113524174)

[**4.7** **(Optional)Creating a release pipeline for clean up** 29](#_Toc113524175)

[5 YAML Pipeline in Azure DevOps 33](#_Toc113524176)

[**5.1 What is a YML pipeline?** 33](#_Toc113524177)

[**5.2 Creating a YML pipeline in Azure DevOps** 33](#_Toc113524178)

[**5.3 Managing YML pipelines in Azure DevOps** 33](#_Toc113524179)

[**5.4 Best practices for using YML pipelines in Azure DevOps** 34](#_Toc113524180)

# **Introduction**

## **Purpose**

The purpose of this document is to define the objective of the Infrastructure as a Code (IaC) Terraform assets and how to use them. It will provide the reader with the understanding of what these assets are envisioned for and wasn’t, providing insights of how to apply and accelerate their customer’s engagement.

## **Audience**

* Pre-Sales/Solution Architects
* DevOps Engineers
* Infrastructure Engineers

## **Document Syntax**

|  |  |
| --- | --- |
| ! | Important Note |

|  |  |
| --- | --- |
| ! | Critical Note |

|  |  |
| --- | --- |
| \* | Design Decision |

|  |  |
| --- | --- |
| ? | Business Decision Required |

# **Infrastructure as a code (IaC)**



## **Introduction of IaC**

Infrastructure as Code (IaC) is the management of infrastructure (networks, virtual machines, load balancers, and connection topology) in a descriptive model, using the same versioning as DevOps team uses for source code. Like the principle that the same source code generates the same binary, an IaC model generates the same environment every time it is applied. IaC is a key DevOps practice and is used in conjunction with continuous delivery.

Infrastructure as code defines the environment in a versioned
file

## **Terraform Overview**

Terraform is a provider agnostics IaC tool that allow users to define their cloud and on-prem infrastructure in a human-readable configuration file. Terraform create and manages infrastructure resources through providers that are published by their developers, allowing it to work with virtually any platform or service. In the case of Azure, the resource providers are published and managed by Microsoft.

Terraform is a declarative programming language and is immutable via the management of state files to track the infrastructure managed. Among Azure IaC tools, Terraform provides multi-cloud capabilities and opportunity to reskill cloud engineers that have prior experience in other cloud platforms.

Comparison of Bicep and Terraform.

|  |  |  |
| --- | --- | --- |
|  | **Terraform** | **Bicep** |
| **Multi cloud capability** | Yes | No, vendor locked to Microsoft Azure |
| **Microsoft Support** | Resource providers are supported as is, with slight delay when exposing new features | Microsoft Premier Support and commitment to immediate exposure of new features |
| **Immutable deployment** | Relies on State file | Incremental deployment |
| **Processing** | Executed on the Terraform client (e.g. AzureDevOps run agent) | Executed directly on core Azure infrastructure service |
| **Integration** | Terraform are integrated via APIs of the resource provider | Tight integration with Azure and its portal  Bicep preflight integration can determine blockers like Azure policy deny action  Ability to automate Portal actions like exporting templates |
| **Resourcing** | Large amount of Terraform capable professionals working in Azure and other vendors.  Low lead time required to upskill individuals that have leveraged Terraform in other cloud providers. | Smaller group of professionals that is across Bicep/ARM.  Longer lead time to train resources up to be proficient in it. |
| **Ideal for** | Client with multi-cloud strategy | Client deeply entrenched within Microsoft ecosystem with no plans on expanding to other platforms |

## **Terraform Module Assets**

Terraform modules are Terraform configuration files that can be “called” and reused by other Terraform configurations.

### **Objective**

The Terraform modules are intended to accelerate Avanade’s deployment of Azure resources and/or Azure IaC strategy:

* **Make simple things simple and complicated things possible**: Achieve by exposing as many settings as possible from Azure resource provider while making only a small subset of them as mandatory. This allows users starting out the IaC journey or requirement of simple designs to accelerate their deployments while still allowing advanced user requiring building complex enterprise grade infrastructure to reuse the same set of modules.
* **Value Add**: Each module created incurs technical debt for its maintenance. To justify its existence, each module must provide value above what is available from Microsoft published resource provider by either accelerating deployment of a particular environment by building correlated resources providers together (e.g. VNET, subnet, NSG for networking build activities) and/or build in integration with logging

What it is:

* Highly reusable module with no client specific configuration
* With reusability as a primary objective, only some highly unsecure configurations are intentionally cordoned off from usage (e.g. unsecured HTTP channels), with most decisions left to the user’s discretion.
* Suited to accelerate deployment of Azure resources via IaC as-is
* Useful to be picked up as a baseline for further development of client/industry specific tools

What it is not (without further development):

* Enforcement of client specific settings within the module (e.g. naming convention, tagging structure)
* Enforcement of industry specific compliance and standards within the module

### **Maintenance**

The Terraform modules need to be maintained and updated as Microsoft release new functionality on their Azure platform and therefore update their Resource providers.

The following scenario will trigger the necessity to update the modules:

* New variables exposed in resource providers
* New code blocks exposed in resource providers
* Change of behavior within the resource providers that is in conflict with our modules
* Resource providers being deprecated and superseded by new Resource providers

|  |  |
| --- | --- |
| ! | Important Note  *Users can continue to use the modules during the period where the above scenario have happened but the modules have not yet been updated. However, they will need to enforce the azurerm version in their Terraform configuration to forcefully use the previous version of the resource providers.*  *This will result in missing out of the new features until the modules are updated.* |

### **Client specific customization**

It is common for enterprise to request their IaC modules to enforce configurations that is set out by their architecture team. This can range from naming conventions to enforcing industry compliances.

While the modules are not built for this in mind, it can form the baseline and accelerate the development.

Azure resource type that has no value to be created as a reusable Terraform asset might make sense to have a module created once client specific customizations are considered.

The following table is an example of how the development can be implemented if its module have already been developed during the asset building exercise.

|  |  |
| --- | --- |
| **Request** | **Development required** |
| Naming convention | Update the name variables to programmatically enforce the naming standards |
| Tagging convention | Option 1:  Update the tag variables into an object and define the Keys and Type of each required tags  Option 2:  Remove the tag variables and create individual variable for each tagging key required by the client |
| Environment specific setting (e.g. prod environment must use Premium SKU) | Expose environment as a new variable  Programmatically insert Premium value into SKU when prod value is detected |
| A specific functionality is not sanctioned to be used in the client environment | Remove the corresponding variable representing the functionality  Set the parameter to null within the module, or remove it entirely if its is an optional parameter |
| A specific functionality is always enforced in the client environment | Remove the corresponding variable representing the functionality  Hard code the parameter within the module |
| A group of Azure resources are always built together in the client environment (e.g. Az Hub always have Az FW and Express Route) | Option 1: Create another module to encapsulate and “call” the separate modules created by this asset build exercise.  Option 2: Create a new module specific to the client where all the resources are provisioned together, while reusing the codes already developed in this asset build exercise when applicable. |

## **Reference Architecture**

ARM templates for the resources visible in the below reference Architecture have been created and the templates have been modularized for flexibility and re-useability.

Graphical user interface, diagram, application

Description automatically generated

# **Terraform User Guide**

## **Installing Terraform**

This section guides you on installing Terraform locally on your workstation. Installing Terraform locally allows your to quick validate and iterate your Terraform configurations before submitting them to a code repository and DevOps deployment process.

1. URL to download Terraform binary: <https://www.terraform.io/downloads>
2. Extract the executable to a directory of your choosing (e.g. c:\terraform)
3. Go to Control Panel>System>Advance System settings

Graphical user interface, text, application, chat or text message

Description automatically generated

1. Select Environment Variables

Graphical user interface, text, application, email

Description automatically generated

1. Select System variables>Path and click the edit button

Graphical user interface, text, application

Description automatically generated

1. Add the directory of where you have extracted Terraform in step 2 into the variable

Graphical user interface, text, application

Description automatically generated

1. Open a terminal and run the following code, if you have installed your Terraform correctly, the command will return the version of Terraform you have installed

|  |
| --- |
| terraform -version |

## **Terraform Configuration**

To provision Azure resources, you need to write Terraform configuration file. These file using the .tf extension. As Terraform is a declarative language, all .tf files are read and executed regardless of how you position your code or name your files.

Developers have taken advantage of this to break the Terraform configuration into the following files to encourage readability. This is option but a good practice that is adopted by the industry:

* Main.tf: File to define the Terraform and resource provider configuration. Sometimes, people will add the main code of resource providers or modules here for simple configurations
* Variable.tf: File to define any variables used
* Local.tf: File to define Local variables
* Output.tf: File to define output
* <custom name>.tf: Additional files to break the main Terraform configuration into readable chunks

Sample Terraform configuration to provision the following:

* Resource group named avatest in Australia East
* VNET named avanet
* Subnet named subnetI
* NSG named example-nsg which has no custom rules

|  |
| --- |
| terraform {    required\_providers {      azurerm = {        source  = "hashicorp/azurerm"        version = ">=3.13.0"      }    }  }  provider "azurerm" {      features {}  }  resource "azurerm\_resource\_group" "avatest" {    name     = "avatest"    location = "Australia East"  }  resource "azurerm\_virtual\_network" "example" {  name = " avavnet"  address\_space = ["10.0.0.0/16"]  location = "Australia East"  resource\_group\_name = azurerm\_resource\_group.avatest.name  }  resource "azurerm\_subnet" "example" {  name = " subnetI"  resource\_group\_name = azurerm\_resource\_group.avatest.name  virtual\_network\_name = azurerm\_virtual\_network.example.name  address\_prefixes = ["10.0.1.0/24"]  }  resource "azurerm\_network\_security\_group" "example" {  name = "example-nsg"  location = "Australia East"  resource\_group\_name = azurerm\_virtual\_network.example.name  }  resource "azurerm\_subnet\_network\_security\_group\_association" "example" {  subnet\_id = azurerm\_subnet.example.id  network\_security\_group\_id = azurerm\_network\_security\_group.example.id  } |

## **Running Terraform**

To connect to Azure via Terraform, use the az cli command to sign in via the terminal:

|  |
| --- |
| az login |

|  |  |
| --- | --- |
| ! | Important Note  *You will require az Powershell to be installed on your machine to run az cli: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli* |

To initiate and download the resource provider and modules required for Terraform to function, run the init command within the context of your Terraform configuration:

|  |
| --- |
| terraform init |

|  |  |
| --- | --- |
| ! | Important Note  *You will need to minimally run Terraform init for the first time and every time a new module is referenced.* |

To verify the code you have written will create the correct Azure resources, run the plan command.

|  |
| --- |
| terraform plan |

|  |  |
| --- | --- |
| ! | Critical Note  Verify the result to ensure the output is desired before proceeding. Watch out for unintended actions like destroy or modification that might not have been envisioned while writing your Terraform configuration. |

After verifying the plan, proceed to provision your Azure infrastructure using the apply command.

|  |
| --- |
| terraform apply |

If your configuration is correct, your intended resources will be provisioned in Azure and you can verify them on Azure Portal. Notice that Terraform state file is created in the same directory as your configuration by default. This file keeps track of what is provisioned in Azure and their values. The Terraform state file is to be treated as a sensitive file; we will revisit the location of the state file again in the DevOps chapter.

## **Using the Terraform module**

The following example show the Terraform configuration when modules are used. The modules can be used in the following method:

* Cloned to your repository or machine locally and reside within the same folder as your Terraform configuration (simplest and shown in this example)
* Hosted remotely in another repository from your Terraform configuration (recommended but will be demoed in the DevOps chapter later)

The module creates the following:

* Resource group named avatest in Australia East
* VNET named avanet
* Subnet named subnetI
* NSG named subnetI\_nsg with built in whitelisting rules

|  |
| --- |
| } terraform {    required\_providers {      azurerm = {        source  = "hashicorp/azurerm"        version = ">=3.13.0"      }    }  }  provider "azurerm" {      features {}  }  resource "azurerm\_resource\_group" "avatest" {    name     = "avatest"    location = "Australia East"  }  module "vnetmodule" {  source = "./VirtualNetwork/"  virtual\_network\_name = "avavnet"  location = "Australia East"  resource\_group\_name = azurerm\_resource\_group.avatest.name  address\_space = ["10.0.0.0/16"]  subnet\_definition = [{  name = "subnetI"  prefix = ["10.0.0.0/24"]  service\_endpoints = []  enforce\_private\_link\_endpoint\_network\_policies = false  enforce\_private\_link\_service\_network\_policies = false  delegations = []  }  } |

|  |  |
| --- | --- |
| ! | Important Note  *Notice that the module encapsulates the complexity of building related resources like NSG association from the user.* |

|  |  |
| --- | --- |
| ! | Critical Note  The sample uses “./” to reference the module, meaning the user have cloned a copy of the assets into their repo or local machine. While it is quick and easy for someone that first started out, this is not the best practice on leveraging reusable assets.  Refer to later chapter on “PAT token and central repo for modules” on how to use them in different repos. |

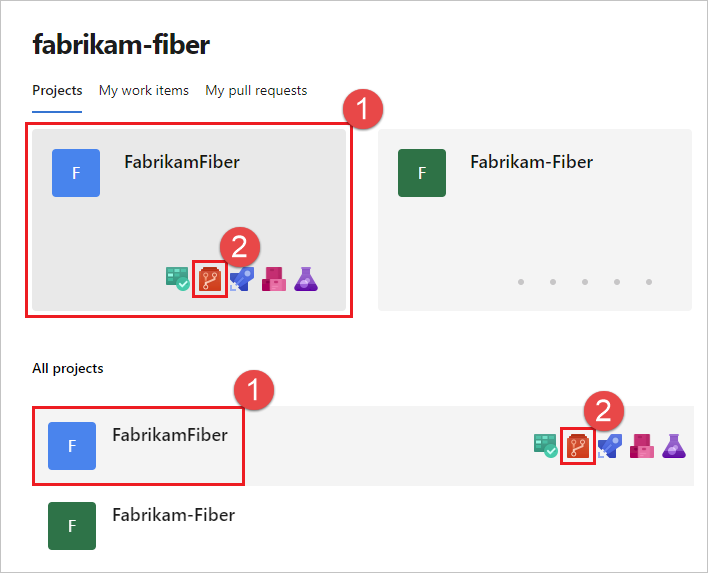
# **Azure DevOps for Terraform**

Terraform can be hosted and run using various DevOps tools. For the purpose of this document, we will use Azure DevOps.

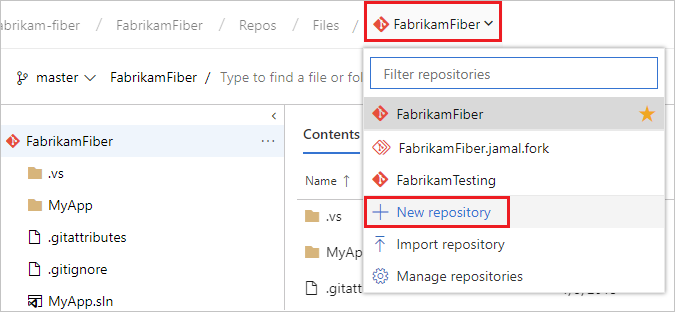
## **Creating a repository**

Repository provides version control for your artifacts, while providing capability for multiple users to collaborate in an independent and accountable manner.

1. Navigate to the **Repos** page in your project and hovering your mouse over the name of your project, then selecting the **Repos** icon. You can select it from the **All** projects list, or from the **Recent** projects list if you have accessed it recently.



1. From the repo drop-down, select **New repository**.



1. In the **Create a new repository** dialog, verify that Git is the repo type and enter a name for your new repo. You can also choose to add a README and create a .gitignore for the type of code you plan to manage in the repo. A README contains information about the code in your repo, and a .gitignore file tells Git which types of files to ignore, such as temporary build files from your development environment.Graphical user interface, text, application, email

   Description automatically generated
2. When you're happy with the repo name and choices, select **Create**.

A new empty Git repo is now created in your project.

You can now create and work with your Terraform configurations in that repo.

|  |  |
| --- | --- |
| ! | Important Note  *While Azure DevOps web interface supports creating and modification of text files like Terraform configurations, IDEs like Visual Studio Code is recommended.* |

## **Service Connection**

Your Azure DevOps require a credential to run that has permission to provision and modify Azure resources. In earlier steps, you have signed in using az cli using your own named account. For DevOps, a service account is required for this purpose and the credential must be safely stored in the DevOps project.

This is achieved via service connection in Azure DevOps.

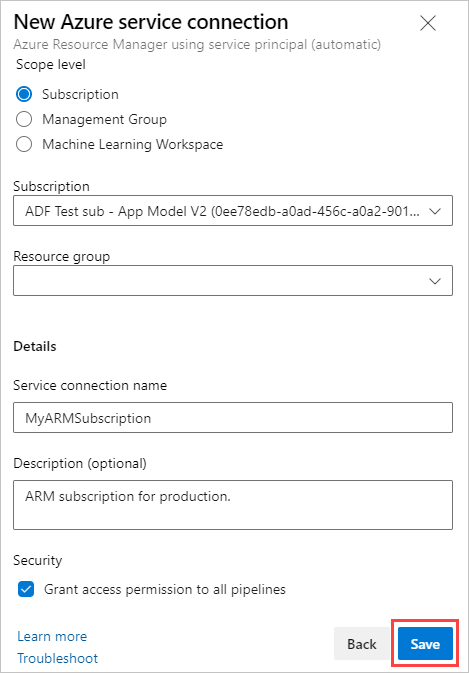
|  |  |
| --- | --- |
| ! | Important Note  *When using Service Principal (automatic) option described below to succeed, perform this action signed into the Azure DevOps with the named account that also has contributor access to the Azure subscription.* |

1. In Azure DevOps, select your project. Navigate to Project settings > Service connections
2. Select + New Service connection.

Graphical user interface, text, application, email

Description automatically generated

1. Select Azure Resource Manager
2. Select Service Principal (automatic)
3. Select Subscription as Scope level and wait for the Subscription name you have access to be retrieved
4. Select the desired subscription and populate the Service Connection Name



1. Click Save

## **Terraform State File**

State file tracks the Azure resources that are provisioned by Terraform and recommends any changes should configuration drift is detected. Since the state file also stores the exhaustive list of configurations describing the Azure resources, it should be treated as a sensitive file.

In earlier chapter, the state file is hosted on your local machine, and it is an acceptable arrangement when working alone on a test environment.

For enterprise environment where multiple users are working on the same environment (e.g. Azure DevOps) and requiring a secure place to store the file. The place would be an Azure storage blob container.

You can create the blob container manually or programmatically. In the following example, the Azure storage is programmatically created using an Azure DevOps pipeline using Az CLI.

|  |  |
| --- | --- |
| ! | Critical Note  You cannot use Terraform to programmatically create the Azure storage to store your state file since a state file location is required whenever Terraform deploys a resource. This results in a “chicken and egg” conundrum. |

Pipeline to create

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Azure CLI |
| Task Version | 2.\* |
| Script Type | PowerShell |
| Script Location | Inline script |
| Script Arguments | NA |
| ErrorActionPreference | Stop |
| Advanced | NA for all |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Environment Variables | NA |
| Output Variables | NA |

|  |  |
| --- | --- |
| ! | Important Note  *You can also add the above task as part of your Terraform apply pipeline since AZ CLI will not recreate the storage if it is already present.* |

Pipeline variables

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Value** | **Lock** | **Sample value** |
| container | Container name | No | livenonpord |
| location | Azure location of the storage account | No | australiaeast |
| rgname | Resource group name of the storage account | No | fy22assetrgname |
| storageaccount | Name of the storage account to be created | No | fy22assettfsa |

In line script

|  |
| --- |
| az group create --location $(location) --name $(rgname)  az storage account create --name $(storageaccount) --resource-group $(rgname) --location $(location) --sku Standard\_LRS  az storage container create --name $(container) --account-name $(storageaccount) |

Once the storage have been created, modify your Terraform configuration to use the Azure storage as your remote state file location. Key is the file name of your terraform state, ensure it is unique for every Terraform configuration.

|  |
| --- |
| backend "azurerm" {      storage\_account\_name = "fy22assettfsa"      container\_name       = "livenonprod"      key                  = "livenonprod.tfstate"    }  } |

|  |  |
| --- | --- |
| ! | Important Note  *Key is the file name of your terraform state, ensure it is unique for every Terraform configuration.* |

## **Azure DevOps extension**

Terraform are not installed by default on the run agents. To do so for release pipeline, extension have to be installed.

1. Navigate to Project settings > Extension
2. Browse marketplace and search for Terraform
3. Select and install the Terraform module by Microsoft DevLabs

Graphical user interface, application

Description automatically generated

1. Ensure the module is installed

Graphical user interface, text, application, email

Description automatically generated

|  |  |
| --- | --- |
| ! | Important Note  *You can also use other extensions of Terraform, but they will invalidate the guide to create the pipelines in the following sections.* |

## **PAT token and central repo for modules**

|  |  |
| --- | --- |
| ! | Important Note  *This is required if you are accessing the Terraform assets directly from its repository. You can skip this step if you choose to clone the assets directly into your repo.* |

The advantage of Terraform modules is to have reusable codes that accelerates deployment for the enterprise. Cloning the code to your own repo or local machine reduce the effectiveness as your cloned copy will get outdated as the modules get maintained by the author.

Therefore, the recommended solution is to store them in a central repository separate from your environment specific repo containing your Terraform configuration.

However, the service connection do not have access to access the modules that is residing in another repo. This can be solved by generating a Personal Access Token (PAT).

To generate a PAT token:

1. Click on your user at the top right corner and select Security
2. Select Personal access tokens
3. Click + New Token

Graphical user interface, text, email

Description automatically generated

1. Copy the PAT value into your notepad temporarily as you will not be able to view the value again once it closes.

PAT token can be use without any interaction via the following format:

|  |
| --- |
| https://$(pattoken)@dev.azure.com |

As PAT token is sensitive, it is also not recommended to store them within the Terraform configuration. Therefore, store the PAT as a secret variable in your pipeline and append them on the fly as the pipeline gets executed.

Pipeline to create

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Command Line Script |
| Task Version | 2.\* |
| ErrorActionPreference | Stop |
| Advanced | NA for all |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Environment Variables | NA |
| Output Variables | NA |

Script

|  |
| --- |
| cd $(System.DefaultWorkingDirectory)  git config --global url."https://$(pattoken)@dev.azure.com".insteadOf "https://dev.azure.com" |

Pipeline variable

|  |  |  |
| --- | --- | --- |
| **Variable** | **Value** | **Lock** |
| pattoken | Enter PAT token into this variable | Yes |

When calling the module in your Terraform configuration use the following value for source:

|  |
| --- |
| source = "git::https://dev.azure.com/FY22-Asset-build-funding/Sovereign%20MS%20with%20Adelaide%20Hub%20Delivery%20Model/\_git/Terraform-Modules//AzureMonitor" |

|  |  |
| --- | --- |
| ! | Critical Note  Update the address accordingly to the Azure DevOps project and repo hosting the modules.  The pipeline will append the PAT token into the URL programmatically during run time within the run agent and does not affect the code in the repo. |

## **Creating a release pipeline for deployment**

|  |  |
| --- | --- |
| ! | Important Note  *Some enterprise may want to be absolute on what was generated by Plan is deployed during Apply by using an output file for Plan and storing it for later use in Azure DevOps artefacts. This is beyond the scope of this document.* |

This pipeline creates a simple approval process for the Terraform Plan to be reviewed before the Apply step can be carried out. On the high level, the end to end pipeline will look like this:

* Grab the code from the repo
* Perform Terraform Plan
* Require approval for Plan
* Once approved, Apply

Diagram

Description automatically generated with low confidence

Within the Plan, are the following steps:

1. Azure CLI (optional for creating remote storage)
2. Install Terraform
3. Command Line Script (optional for PAT token)
4. Terraform init
5. Terraform plan

|  |  |
| --- | --- |
| ! | Important Note  *As the run agents gets wiped back to default state every run, Terraform install and init must be run everytime.* |

Install Terraform task:

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Terraform tool installer |
| Task Version | 0.\* |
| Version | latest |
| ErrorActionPreference | Stop |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Output Variables | NA |

Terraform init task:

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Terraform |
| Task Version | 3.\* |
| Provider | azurerm |
| Command | init |
| Configuration directory | Point to your Terraform configuration location (e.g. $(System.DefaultWorkingDirectory)/\_Terraform-Live/live-nonprod) |
| Additional command arguments | NA |
| Azure subscription | Select your Service Connection |
| Resource group | $(rgname) |
| Storage account | $(storageaccount) |
| Container | $(container) |
| Key | livenonprod.tfstate |
| ErrorActionPreference | Stop |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Output Variables | NA |

|  |  |
| --- | --- |
| ! | Important Note  *The Key variable must align with what is described in the Terraform configuration.*  *Example assume that the pipeline is creating the state file programmatically as described earlier chapters and the parameters are in place.* |

Terraform Plan task:

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Terraform |
| Task Version | 3.\* |
| Provider | azurerm |
| Command | plan |
| Configuration directory | Point to your Terraform configuration location (e.g. $(System.DefaultWorkingDirectory)/\_Terraform-Live/live-nonprod) |
| Additional command arguments | NA |
| Azure subscription | Select your Service Connection |
| ErrorActionPreference | Stop |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Output Variables | NA |

Within the Apply stage are the following steps:

1. Install Terraform
2. Command Line Script (optional for PAT token)
3. Terraform init
4. Terraform apply

For Terraform apply task:

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Terraform |
| Task Version | 3.\* |
| Provider | azurerm |
| Command | apply |
| Configuration directory | Point to your Terraform configuration location (e.g. $(System.DefaultWorkingDirectory)/\_Terraform-Live/live-nonprod) |
| Additional command arguments | -auto-approve |
| Azure subscription | Select your Service Connection |
| ErrorActionPreference | Stop |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Output Variables | NA |

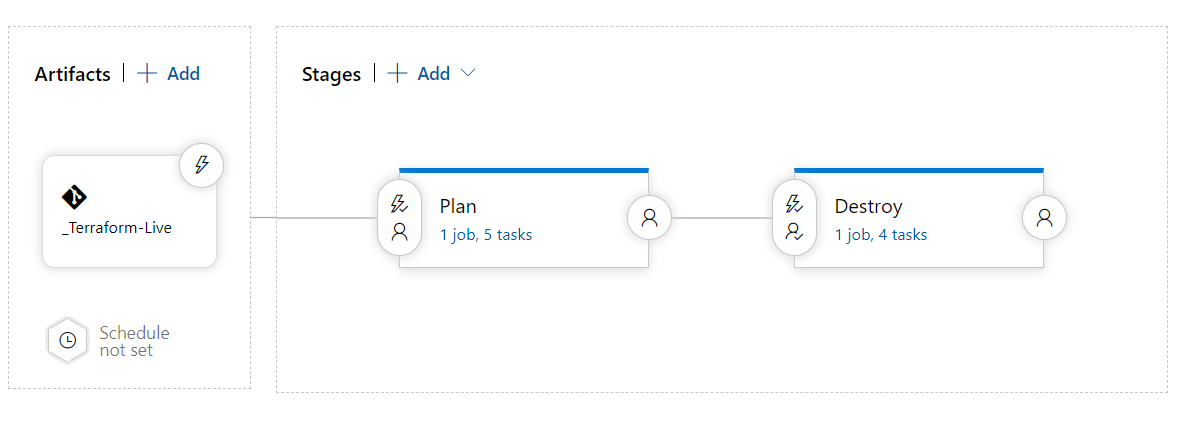
As best practices in a enterprise environment, you can add pre-deployment conditions on the Apply stage requiring approvers to review the plan output before apply is executed.

## **(Optional)Creating a release pipeline for clean up**

|  |  |
| --- | --- |
| ! | Important Note  *Clean up is usually relevant for test or demo environment only to preserve cost.* |

This pipeline creates a simple approval process for the Terraform Plan to be reviewed before the Destory step can be carried out. On the high level, the end to end pipeline will look like this:

* Grab the code from the repo
* Perform Terraform Plan
* Require approval for Plan
* Once approved, Destroy



Within the Plan, are the following steps:

1. Azure CLI (optional for creating remote storage)
2. Install Terraform
3. Command Line Script (optional for PAT token)
4. Terraform init
5. Terraform plan

|  |  |
| --- | --- |
| ! | Important Note  *The main difference in this plan step compared to the previous pipeline is that the plan has a -destroy argument to generate plan results to predict a destroy action.* |

Install Terraform task:

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Terraform tool installer |
| Task Version | 0.\* |
| Version | latest |
| ErrorActionPreference | Stop |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Output Variables | NA |

Terraform init task:

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Terraform |
| Task Version | 3.\* |
| Provider | azurerm |
| Command | init |
| Configuration directory | Point to your Terraform configuration location (e.g. $(System.DefaultWorkingDirectory)/\_Terraform-Live/live-nonprod) |
| Additional command arguments | NA |
| Azure subscription | Select your Service Connection |
| Resource group | $(rgname) |
| Storage account | $(storageaccount) |
| Container | $(container) |
| Key | livenonprod.tfstate |
| ErrorActionPreference | Stop |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Output Variables | NA |

Terraform Plan task:

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Terraform |
| Task Version | 3.\* |
| Provider | azurerm |
| Command | plan |
| Configuration directory | Point to your Terraform configuration location (e.g. $(System.DefaultWorkingDirectory)/\_Terraform-Live/live-nonprod) |
| Additional command arguments | -destroy |
| Azure subscription | Select your Service Connection |
| ErrorActionPreference | Stop |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Output Variables | NA |

Within the Destroy stage are the following steps:

1. Install Terraform
2. Command Line Script (optional for PAT token)
3. Terraform init
4. Terraform Destroy

For Terraform apply task:

|  |  |
| --- | --- |
| **Variable** | **Value** |
| Pipeline type | Release |
| Task type | Terraform |
| Task Version | 3.\* |
| Provider | azurerm |
| Command | destroy |
| Configuration directory | Point to your Terraform configuration location (e.g. $(System.DefaultWorkingDirectory)/\_Terraform-Live/live-nonprod) |
| Additional command arguments | -auto-approve |
| Azure subscription | Select your Service Connection |
| ErrorActionPreference | Stop |
| Control Options | Enabled: Ticked  Number of retries if task failed: 0  Timeout: 0  Run this task: Only when all previous tasks have succeeded |
| Output Variables | NA |

# **YAML Pipeline in Azure DevOps**

## **5.1 What is a YML pipeline?**

In Azure DevOps, a YAML pipeline is a set of instructions that will be executed in an automated fashion when certain conditions are met. YAML stands for "YAML Ain't Markup Language", and is often used for configuration files. In a YAML pipeline, each step is defined in a YAML file, and these files are then stored in a central location (usually in source control). When someone makes a change to one of the files, the pipeline will automatically re-run the affected steps.

YAML pipelines have several advantages over traditional "build" pipelines. First, they are much easier to read and understand. Second, they are more flexible and can be easily modified without having to go through a lengthy approval process.

## **5.2 Creating a YML pipeline in Azure DevOps**

Pipelines in Azure DevOps are defined in YAML files, allowing for greater control and customization than the visual editor.

To create a new pipeline, select "New Pipeline" from the Pipelines page in Azure DevOps. This will open the "Create Pipeline" wizard. Choose "Starter pipeline" to have a basic YAML file generated automatically.

The generated file will be placed in the root of your repository and named azure-pipelines.yml. You can edit this file directly to add more stages, jobs, or steps to your pipeline.

To learn more about what you can do with YAML pipelines in Azure DevOps, check out the documentation: https://docs.microsoft.com/en-us/azure/devops/pipelines/yaml-schema?

## **5.3 Managing YML pipelines in Azure DevOps**

Azure DevOps makes it easy to manage YML pipelines. With a few clicks, you can create, edit, and delete YML pipelines. You can also view the history of your YML pipelines and see when they were last run.

To create a new YML pipeline, click the "New Pipeline" button in the Azure DevOps interface. This will open the "Create Pipeline" page. On this page, you will need to select a repository type (GitHub, Bitbucket, or Azure Repos), and then select a template (Basic or Starter). Once you have selected a template, click the "Create Pipeline" button.

To edit an existing YML pipeline, click the "Edit" button in the Azure DevOps interface. This will open the "Edit Pipeline" page.

## **5.4 Best practices for using YML pipelines in Azure DevOps**

There are a few best practices that you should follow when using YAML pipelines in Azure DevOps:

1. Use comments liberally to document your pipeline.azure-pipelines.yml file. This will help other developers understand what your pipeline is doing and why.

2. Check your pipeline into source control so that it can be easily reviewed and updated by other members of your team.

3. Be thoughtful about which tasks should be run in parallel and which tasks should be run sequentially.