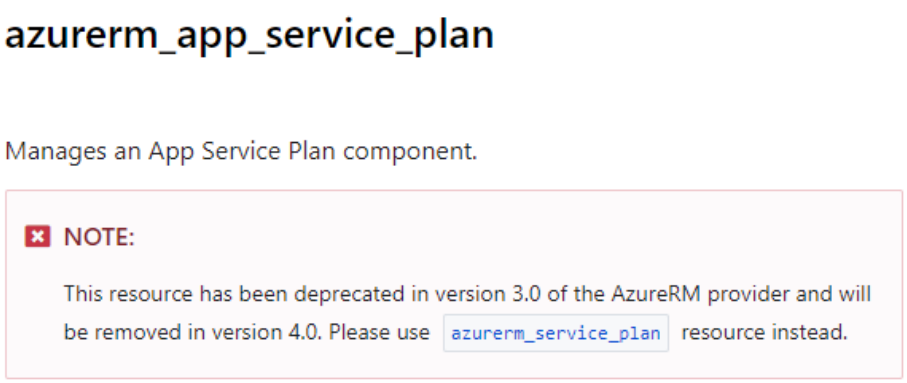
Terraform script monitoring through Azure Services

Problem Statement

Terraform is currently considered as one of the most popular cloud agnostic IaC tools. Terraform publishes its major/minor releases on regular intervals, and continuously evolving. To evolve with Terraform releases, organizations which opted Terraform as IaC tool, certainly need to monitor their own Terraform scripts and if required update the same. Along with monitoring, the other major problems are –

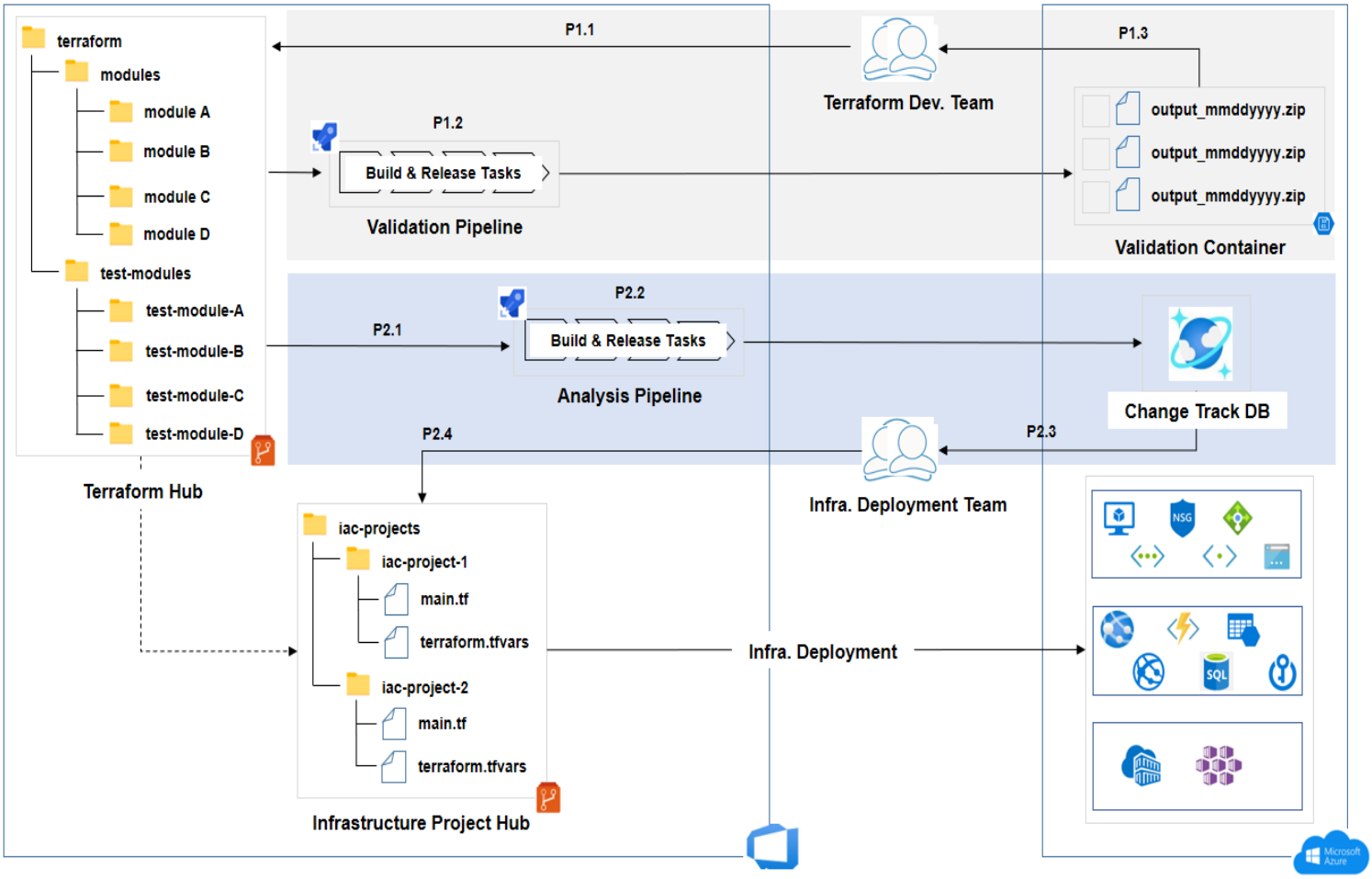
- To align with various Terraform providers, in case of deprecation or introduction of Terraform resource, such as removal of azurerm_app_service_plan by azurerm_service_plan.



- Considering updates required for Terraform resource variables (i.e. input/output).
- Terraform change communication with various teams within organization.

Solution/ Architecture

The solution depends on various out of the box features of Azure DevOps and custom code developed in .NET and PowerShell scripts.



The architecture is fairly simple to understand, and broadly divided in 2 processes which are loosely connected. The Process 1 and Process 2, are abbreviated as P1 and P2 respectively. The major solution components and its compositions and the processes and its stages are explained below.

- Terraform Hub – an Azure DevOps repo. The repo contains various Terraform modules (i.e. modules folder) like azurerm_app_service_plan etc. It also contains required Terraform scripts to validate and deploy (i.e. test-modules folder) each module.

- Validation Pipeline – an Azure DevOps pipeline. It contains tasks that behind the scene validate each Terraform modules of Terraform Hub on a scheduled interval.

- Analysis Pipeline – an Azure DevOps pipeline, responsible to track the changes on Terraform Hub.

The stages of each process are described below.

Process 1

P1.1

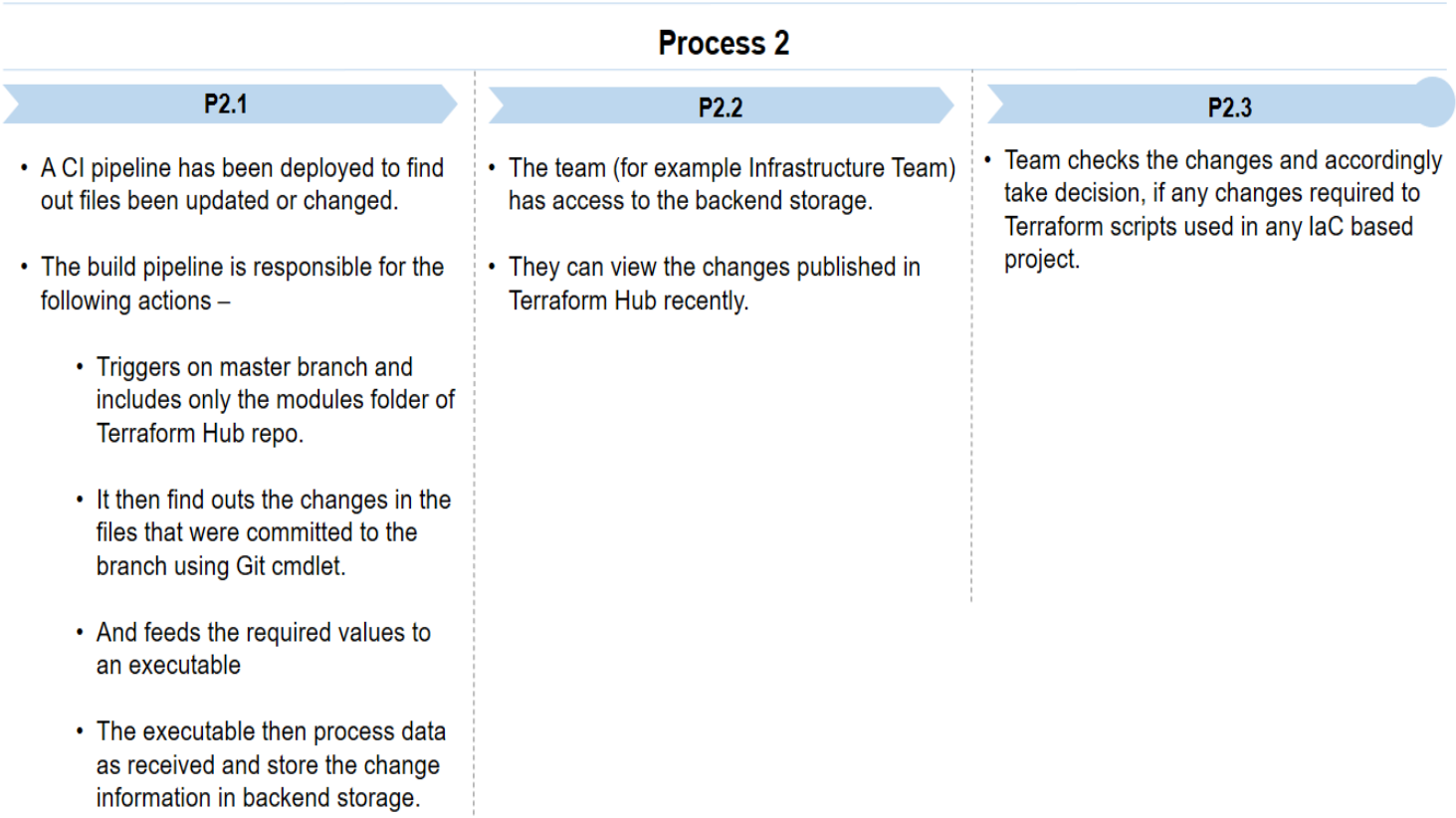
- The Terraform Dev. team is responsible for creating and updating the Terraform modules in the repository and testing the same before publishing it for other teams.
- The team implements new Terraform modules based on various business requirements.
- The team also gets notifications from the automated validation process on regular interval about the current status of various existing Terraform modules, and accordingly take decisions, if any changes required for the existing Terraform modules.

P1.2

- A CI-CD pipeline runs on a schedule basis. The pipeline is triggered using Azure DevOps CI triggers feature.
- The release pipeline consists of a single PowerShell task.
- The task invokes some PowerShell scripts which are responsible for the following actions –
    - Clone the Terraform Hub repo (modules and test-modules).
    - Iterating through each “test-module” folder and executing the Terraform Plan command and generate the report for each module.
    - Consolidating all reports and upload as zip to an Azure blob storage.

P1.3

- After pipeline completion, the Terraform Dev. team gets notification about the completion over mail.
- The team has access to the blob storage and can download the consolidated report.
- Team then checks the report, and mostly focusses on –
    - If any information regarding any changes that may get published very soon by the Terraform providers.
    - Any error occurred while executing the Terraform plan on any module.
    - Accordingly takes decision on new changes, if any.
        - The validation life cycle (i.e. P1.1) starts again.



Infrastructure Project Hub – an Azure DevOps repo. It contains various infrastructure deployable blueprints/projects written in Terraform. It depends on Terraform Hub modules for various types of Azure resource deployment. However this repo and its associated processes can be considered out of the core solution explained in this blog.

Technical Details and Implementation of solution

The technical details and solution implementation explained in below. There are 4 major components to be developed.

1. Change Track DB
2. Terraform Validator Executable
3. Validation Pipeline
4. Analysis Pipeline

Change Track DB

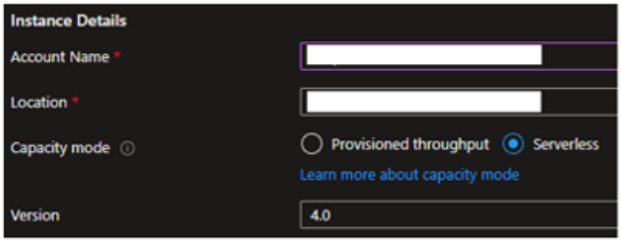
The section explain the Change Track DB creation. The solution uses Cosmos DB for MongoDB service offering of Azure.

Change Track DB Creation Steps

Step 1.

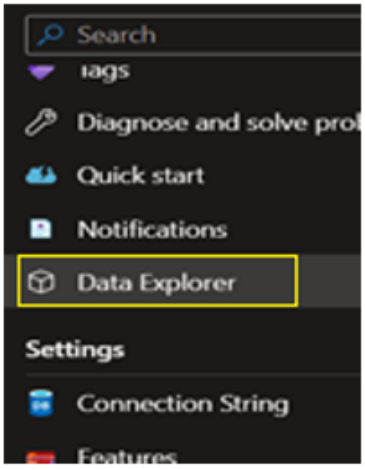
a. Login to Azure Portal
b. Search for Cosmos DB in Azure Portal top search box
c. Select the “Azure Cosmos DB for MongoDB” option
d. Provide the required information for Subscription, Resource Group

Step 2.



- Provide an account name
- Provide Location details
- Rest of the options values as shown in the picture.
- Please provide required information in rest of the tabs, like “Global Distribution” etc. as per the organization security policies if applicable or go ahead with the default settings.
- In “Review + create” tab, click on “Create” button to create the Cosmos DB for MongoDB

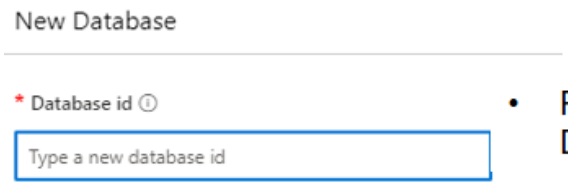
Step 3.



- Once the Azure Cosmos DB for MongoDB created, go to the newly database and from the left hand side panel, and click on the “Data Explorer” option.

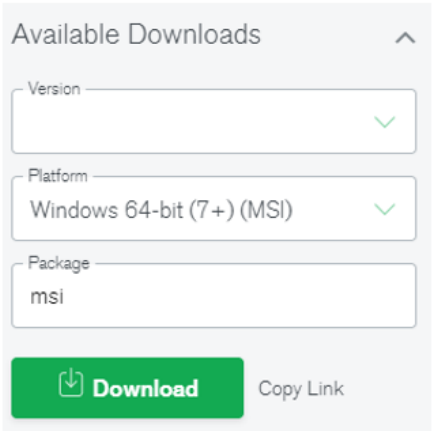
Step 4.

- Click “New Database” option from the top menu bar.



- Please provide the value of "Cosmos Database ID" as “**change-track-db**”.

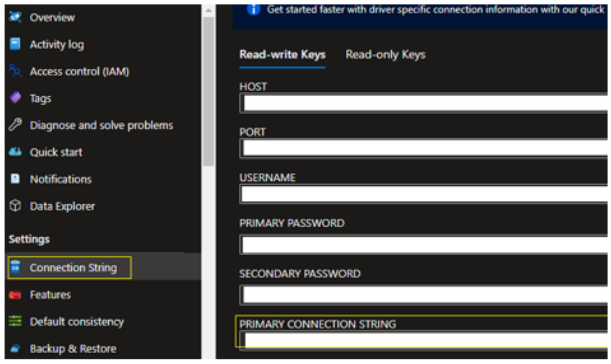
Step 5.



- Install MongoDB Compass in the system. This is a freeware. This will help interact with MongoDB in easy way.
- Visit the MongoDB Compass official web site at https://www.mongodb.com/try/download/compass
- From “Available Downloads” section, please select option as shown in the picture. Please select the stable version from “Version” option.
- Please run the downloaded executable locally in the system and follow the instructions as appeared in the installation wizard.
- Complete the installation.

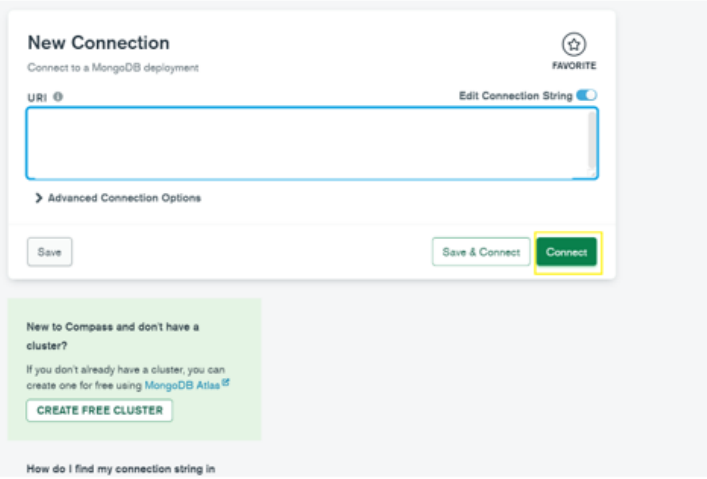
Create the change-track-db collections

Step 1.



- Log in to Azure DevOps Portal.
- Navigate to the Azure Cosmos DB for MongoDB instance created previously.
- Click on the “Connection String” link from the left hand side panel.
- Copy the Primary Connection String value

Step 2.



- Start MongoDB Compass
- Paste the Primary Connection String in the URI box and click Connect.
- Once the database server instance is connected. Select the change-track-db from the left hand side panel.

Step 3.

- Any Terraform Resource and its input and output variables can be easily structured in JSON based No-SQL database collections, for example “module A” and its variables could be represented as collection below.
- Create a collection (let’s say Resources) with a meaningful name to store Terraform resource information.

```
{
  "_id": {
    "$oid": "[redacted]"
  },
  "modulename": "module-a",
  "resourcename": "azurerm_module_a",
  "terraformregistrypath": "terraform/modules/module A"
}
```

- Create another collection (let’s say Resourceproperties) to store the Terraform Resource input and output variable information. The relationship could be established between the collection using MongoDB Document Reference approach.

```
{
  "_id": { "$oid": "[redacted]"},
  "resourceid": { "$oid": "[redacted]"},
  "ipvariables": [
    {
      "_id": { "$oid": "[redacted]"},
      "name": "input variable name",
      "description": "the description of input variable",
      "type": "the type of the variable",
      "required": "is it a required variable? (Yes/No)",
      "argumenttype": "primitive type variable (STRING, etc.) or complex type variable like LIST",
      "parentargumentid": "parent input variable id, if any",
      "userreadablename": "user readable name of the variable",
      "defaultdeploymentvalue": " "
    },
    {....},
    {........},
    {............}
  ],
  "opvariables": [
    {
      "name": "output variable name",
      "attributetype": "primitive type variable (STRING, etc.) or complex type variable like LIST",
      "description": "the description of output variable",
      "parentattributeid": "parent output variable id, if any",
      "userreadablename": "ID",
      "defaultvalue": [
        ""
      ],
      "_id": {  "$oid": "[redacted]"}
    }
  ]
}
```


- Create the documents in each collection based on the requirement for Terraform Resource and its variables.

Terraform Validator Executable

The Terraform Validator is a .NET Core executable which is triggered using Analysis Pipeline in later stage to track the changes of Terraform Resources and its variables from Terraform Hub and store the change information in the change-track-db.

Step 1.

- Create a .NET Core Console Application using Visual Studio IDE (2019/2022).
- Add MongoDB.Bson, MongoDB.Driver using Visual Studio Package Manager.
- Create the required Data Models for the application, as per the MongoDB collections created in the previous section.

```
public class Resource
{
        [BsonId]
        [BsonRepresentation(BsonType.ObjectId)]
        public string? Id { get; set; }

        public string? modulename { get; set; }
        public string? terraformregistrypath { get; set; }

        public string? resourcename { get; set; }


}
```

```
public class IPVariable
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string? Id { get; set; } = ObjectId.GenerateNewId().ToString();

    public  string? name { get; set; }
    public string? description { get; set; }
    public string? type { get; set; }
    public string required { get; set; }
    public string argumenttype { get; set; }

    [BsonRepresentation(BsonType.ObjectId)]
    public string parentargumentid { get; set; }
    public string userreadablename { get; set; }
    public string rule { get; set; }
    public string defaultdeploymentvalue { get; set; }

}
```

```
public class OPVariable
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; } = ObjectId.GenerateNewId().ToString();

    public string? name { get; set; }
    public string? description { get; set; }
    public string? attributetype { get; set; }

    [BsonRepresentation(BsonType.ObjectId)]
    public string parentattributeid { get; set; }

    public string? userreadablename { get; set; }
    public List<string>? defaultvalue { get; set; }

}
```

```
public class ResourceProperties
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }


    [BsonRepresentation(BsonType.ObjectId)]
    public string resourceid { get; set; }

    public List<IPVariable> ipvariables { get;set;}
    public List<OPVariable> opvariables { get;set;}
}
```

Step 2.

```
using MongoDB.Bson;
using MongoDB.Driver;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.Json;
using System.Threading.Tasks;
```

- The required namespace for the program shown
- MongoDB.Bson, MongoDB.Driver namespaces been added too.

```
class Program
{
    static async Task Main(string[] args)
    {
        List<Variables> variables;
        List<Output> outs;
        var connectionString = args[0];

        var modulename = args[1] != "_" ? args[1] : null;
        var resourcename = args[2];
        var folderpath = args[3];
        var varstring = args[4];
        var outstring = args[5];
        var dbname = args[6];
```

- The Main function depends on the input arguments.
- The arguments will be fed by the caller (i.e. Analysis Pipeline, described later)
- varstring – represents the Terraform resource input variables
- outstring – represents the Terraform resource output variables
- resourcename – represents the Terraform resource
- modulename – represents the module folder name
- dbname and connectionstring – the change-track-db name and its connection string.

Step 3.

```
var mdb = client.GetDatabase(dbname);
var resourcecoll = mdb.GetCollection<Resource>("Resources");
var filter = Builders<Resource>.Filter.Eq("modulename", modulename);

var resourcepropcoll = mdb.GetCollection<ResourceProperties>("Resourceproperties");
var resoucedocs = colle.Find(filter).Project(x => new Resource { Id = x.Id, modulename = x.modulename }).ToList();
var resoucedoc = resoucedocs?.FirstOrDefault();
```

- The database collections (i.e. Resources and Resourceproperties) are filtered.
- The MongoDB Document for a specific Terraform module is also filtered using modulename variable.

```
if (resoucedoc == null)
{
    resoucedoc = new Resource { modulename = modulename, terraformregistrypath = folderpath, resoucename = resourcename };
    await colle.InsertOneAsync(resoucedoc);
    var propertyItem = new ResourceProperties {
                           resourceid = resoucedoc?.Id,
                           ipvariables = new List<IPVariable> { },
                           opvariables = new List<OPVariable> { }};

    if (variables!= null && variables?.Count() != 0)
    {
        propertyItem.ipvariables.AddRange(variables);
    }

    if (outs != null && outs?.Count() != 0)
    {
        propertyItem.opvariables.AddRange(outs);
    }

    await resourcepropcoll.InsertOneAsync(propertyItem);

    return;
}
```

- This block performs the business logic to track the data for the various Terraform resources and its properties and fed the same to backend database.
- In case the Terraform resource included recently by the Terraform Dev. Team, it will capture that information save the new resource information to backend database collections

Step 4.

```
var pfilter = Builders<ResourceProperties>.Filter.Eq("resourceid", new BsonObjectId(new ObjectId(resoucedoc.Id)));
var propItem = resourcepropcoll.Find(pfilter).ToList();
var prop = propItem.FirstOrDefault();
```

- The MongoDB Document for a specific Terraform resource properties is filtered using resourceid variable.

```
if (variables != null && prop?.ipvariables != null)
{
    prop.ipvariables.RemoveAll(x => !variables.Any(y => y.name == x.name));

    prop.ipvariables.ForEach(x =>
    {
        var item = variables.Find(y => y.name == x.name);
        if (item != null)
        {
            x.description = item.description;
            x.type = item.type;
        }
    });

    var newvars = variables.FindAll(x =>
            !prop.ipvariables.Any(y => y.name == x.name)
    );

    prop.ipvariables.AddRange(newvars);
}
```

- This block performs the business logic to track the data for the various input properties of a Terraform resources and fed the same to backend database.
- In case the Terraform resource input properties included recently by the Terraform Dev. Team, it will capture that information save the new property information to backend database collections

```
            if (outs != null && prop?.opvariables != null)
            {
                prop.opvariables.RemoveAll(x => !outs.Any(y => y.name == x.name));

                prop.opvariables.ForEach(x =>
                {
                    var item = outs.Find(y => y.name == x.name);
                    if (item != null)
                    {
                        x.description = item.description;

                    }
                });

                var newouts = outs.FindAll(x =>
                    !prop.opvariables.Any(y => y.name == x.name)
                );

                prop.opvariables.AddRange(newouts);
            }

            await resourcepropcoll.ReplaceOneAsync<ResourceProperties>(doc => doc.Id == prop.Id, prop);

        }
    }
}
```

- This block performs the business logic to track the data for the various output properties of a Terraform resources and fed the same to backend database.
- In case the Terraform resource output properties included recently by the Terraform Dev. Team, it will capture that information save the new property information to backend database collections

- After being triggered by the Analysis Pipeline, Terraform Validator executable inserts required data to backend collections.
- Other teams can view data from backend database in various ways, they can use MongoDB Compass, or through web APIs they can get this as feed or may view the data in web application too.
- Accordingly the team can update their respective IaC project Terraform scripts, if required.

Validation Pipeline

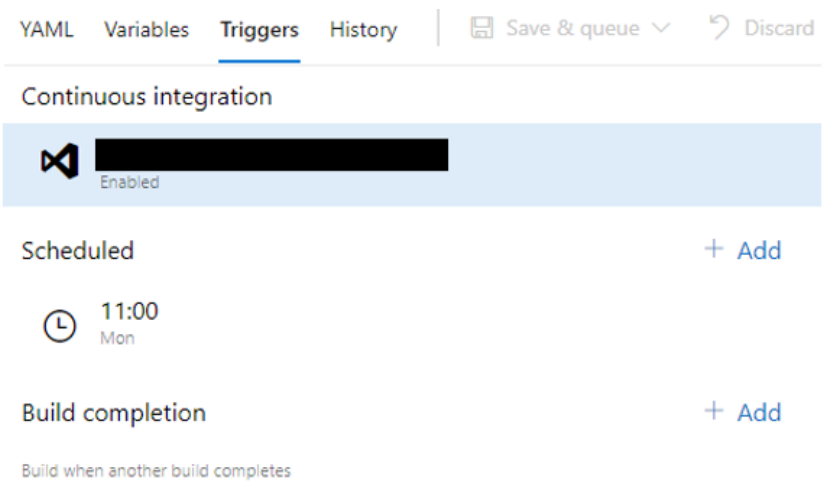
The Validation Pipeline is a CI-CD pipeline. The CI pipeline publishes the build artifact for the CD pipeline. The CD pipeline executes a PowerShell task. The PowerShell scripts then iterate through each Terraform Hub modules and generate a consolidated plan report and upload to the blob storage.

Step 1.

```
pool:
  vmImage: ubuntu-latest

stages:
- stage: Build

  displayName: "Development"

  jobs:

    - job: BuildAutomationIaC

      steps:

      - task: CopyFiles@2

        inputs:

          SourceFolder: '$(System.DefaultWorkingDirectory)/validation-job/[redacted]'

          Contents: |

            **

          TargetFolder: '$(Build.ArtifactStagingDirectory)'

      - task: PublishBuildArtifacts@1

        inputs:

          PathtoPublish: '$(Build.ArtifactStagingDirectory)'

          ArtifactName: 'terraform-plan'

          publishLocation: 'Container'
'
```

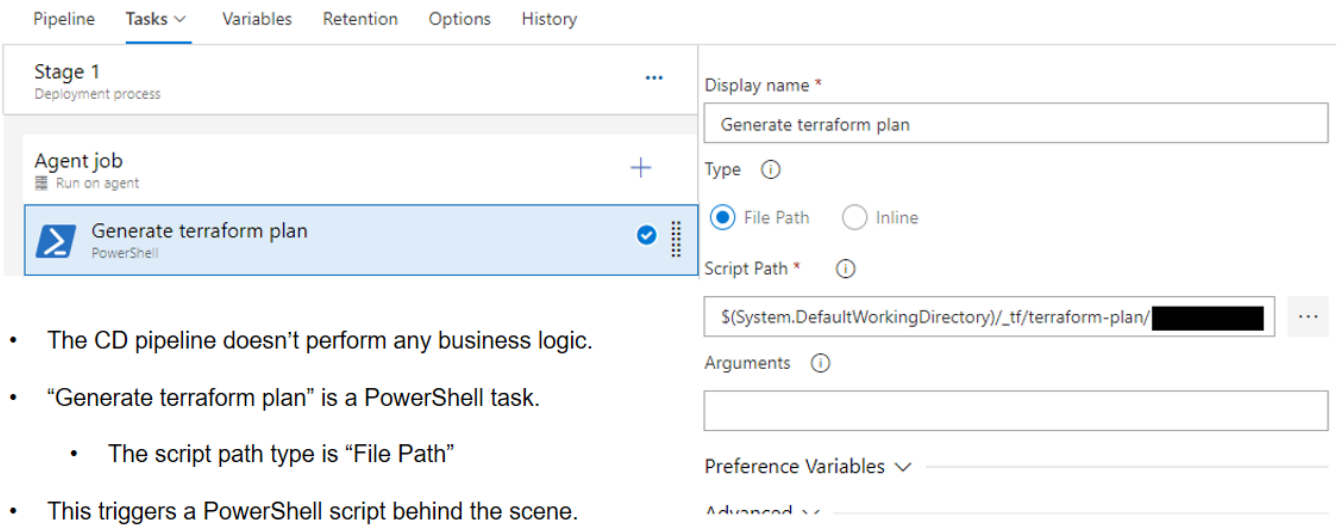
- The CI pipeline doesn't perform any business logic.
- CopyFiles@2 task prepared a publish folder will all required PowerShell scripts.
- PublishBuildArtifact@1 task creates the publish artefact for CD pipeline.

Step 2.



- The CI pipeline is scheduled to trigger on a specific time on weekly basis.

Step 3.



- The CD pipeline doesn't perform any business logic.
- "Generate terraform plan" is a PowerShell task.
  - The script path type is "File Path"
- This triggers a PowerShell script behind the scene.

Step 4.

- The PowerShell script performs the following tasks.
- Task 1: Clone the Terraform Hub
- Task 2: Execute the Terraform commands and generate output files
- Task 3: ZIP the output folder
- Task 4: Upload the output folder to storage container

```
$usernamepat = $username + ":" + $patkey
$b64pat = [Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($usernamepat))

$tmprepofoldername = "tmp_" + (Get-Date).ToString("MMddyyyy_hhmmss")
$folder = New-Item -Path $CurrentExecutionPath -Name $tmprepofoldername -ItemType "directory"
$clonefolderpath = $CurrentExecutionPath + "\" + $tmprepofoldername

$currentlocation = Get-Location
Set-Location $clonefolderpath

# Clone the module
git -c http.extraHeader="Authorization: Basic $($b64pat)" clone -b $branch [redacted] .

Set-Location $currentlocation
Write-Host $currentlocation
```

- Task 1: Clone the Terraform Hub
- To clone the Terraform modules from the Terraform Hub repo, PAT has been used.
- Using PAT and Git commands the repo cloned to an empty folder inside the DevOps agent server.

Step 5.

- Task 2: Execute the Terraform commands and generate output files
- Once the Terraform Hub cloned, iterate through each module folder.
- For each module folder run the Terraform Plan command to validate the each module. The plan will be executed using the latestversion of Terraform Providers (like azurerm, etc.)
- The plan will written directly to a file (for example .txt file) and saved inside a folder

```
# Create unique file name for tfplan.txt
$modulefoldername = Split-Path $RootModuleFolderPath -Leaf
$tfplanfilepath = $OutputFolderPath + "\" + $modulefoldername + "-tfplan-" + $UniqueIdentifier + ".txt"

$varsfile = tfvarsfile -RootModuleFolderPath $rootmodulefolderpath

terraform init -backend-config="terraform.tfvars"

$outputcontent = terraform plan -no-color > $tfplanfilepath 2>&1

Write-Host "terraform plan completed for $($modulefoldername)"
```

- Task 3: ZIP the output folder
- Once the Terraform Plan executed for every module and the output file has been generated inside the output folder.
- The folder will be compressed.

Step 6.

- Task 3: ZIP the output folder
- Once the Terraform Plan executed for every module and the output file has been generated inside the output folder.
- The folder will be compressed.

```
$zipfolderpath = $currentexecutionpath + "/" +"output_" + (Get-Date).ToString("MMddyyyy_hhmmss") + ".zip"
Write-Host ("Zip folder path : " + $zipfolderpath)
Compress-Archive -Path $outputfolderPath -Update -DestinationPath $zipfolderpath
Write-Host ("current execution path after zip folder" + (Get-ChildItem -Path $currentexecutionpath))
```

- Task 4: Upload the output folder to storage container
- The consolidated Terraform Plan report, the zip file then uploaded to the blob container (i.e. Validation Container)
- Azure PowerShell or Azure CLI commands can be used to connect to Azure and upload to the storage container (below is the Azure CLI option)

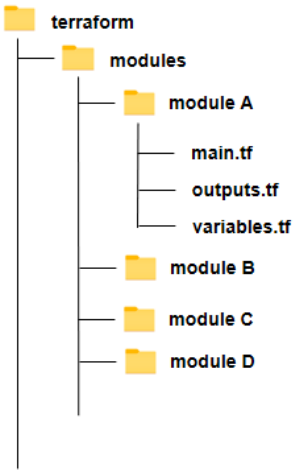
```
az login --service-principal -u $clientid -p $clientsecret --tenant $tenantid

az storage blob upload --account-name=$storageaccount --account-key=$storageaccountaccesskey --container-name $storagecontainername --file $ZipFolderPath

az logout
```

- After successful completion of the pipeline, the Terraform Dev. team can access the consolidated report from the container.
- Accordingly take decision, if there is any change required for the existing modules or new modules to be added to Terraform Hub repo.

Terraform Hub Files and Format

Before explaining the Analysis Pipeline, let's have a look of Terraform Hub and its files structure.

Step 1.



- Each Terraform module is created with proper hierarchy and naming convention .
- Each module folder contains main.tf, outputs.tf and varibales.tf
- Each file having same format and name naming convention across the modules.
- The main objective for maintaining same format and proper naming convention to maintain a big repository of resources in hassle-free way and programmatically can be read/parsed.

Step 2.

```
resource "azurerm_module_a" "msarm_modulea" {
  # Required
  name                = var.ip_armmodulea_name
  resource_group_name = var.ip_armmodulea_resourcegroupname
  location            = var.ip_armmodulea_location

  # Optional
  settings        = var.ip_armmodulea_settings
}
```

main.tf

Step 3.

```
variable "ip_armmodulea_name" {
  description = "(Required) The name of the service."
  type        = string
}

variable "ip_armmodulea_resourcegroupname" {
  description = "(Required) The name of the resource group in which to create the service."
  type        = string
}

variable "ip_armmodulea_location" {
  description = "(Required) Specifies the supported Azure location where the resource exists."
   type        = string
}

# Example - settings {"SOME_KEY" = "some-value"}
variable "ip_armmodulea_settings" {
  description = "(Optional) A key-value pair of Settings."
  type        = map
}
```

variables.tf

```
output "op_armmodulea_id" {
  value       = armmodulea.msarm_modulea.id
  description = "The ID of the service."
}
```

outputs.tf

Analysis Pipeline

The Analysis Pipeline is responsible for finding the changes done on any files in Terraform Hub after making commit to the repo master branch. The logic highly depends on the file structure of Terraform Hub modules and format. The changes are identified by RegEx expressions based on the pre-defined file format and naming convention of Terraform resources and its input and output variables.

Step 1.

- The pipeline includes 2 distinct steps and both are PowerShell tasks.

```
trigger:
 branches:
    include:
    - master
 paths:
    include:
    - terraform/modules


pool:
  vmImage: windows-latest

steps:
```

- The pipeline triggers on the master branch includes the “modules” folder of Terraform Hub repo.

Step 2.

```
- task: PowerShell@2
  name: getfile
  inputs:
    targetType: 'inline'
    script: |
      ${a} = git diff-tree --no-commit-id --name-only -r $(Build.SourceVersion)
      Write-Host $a
      Write-Host "##vso[task.setvariable variable=changedFile]$a"
```

- The first step utilizes the git diff-tree command to get the files been committed recently.
- Then it stores in a DevOps task variable so that in the next step value can be used for further information processing

Step 3.

```
- task: PowerShell@2
  inputs:
    targetType: 'inline'
    script: |
        $var1 = '$(changedFile)'.split(" ")
        echo $(changedFile)
        $var1.count
        $folderHash= @{}

        foreach ( $filepath in $var1 )
        {
                $folder =  $filepath -replace '/\w*.tf',""

                if( !$folderHash.Contains( $folder ))
                {
                        $folderHash[$folder] = New-Object string[] 3
                        $folderHash[$folder][0]= $filepath
                }
                else
                {
                        if ( $folderHash[$folder][1] -eq $null)
                        {
                                $folderHash[$folder][1]= $filepath
                        }
                        else
                        {
                                $folderHash[$folder][2]= $filepath
                        }
                }
        }
```

- The second step utilizes PowerShell commands and RegEx to find out the changes in the files (main.tf, variables.tf and outputs.tf).
- It first creates a dictionary kind of object which has key as the module folder path and value is string array.
- The array contains the information of main.tf, variables.tf and outputs.tf file paths respectively.

Step 4.

- Once the dictionary object created successfully.
- The object will iterated based on the unique key (i.e. folder path of module)

```
foreach($folderPath in $folderHash.Keys)
{
        $resourcename = "_"
        $modulename = "_"
        $filepathcontent = $folderPath -split('/|\\')
        $modulename = $filepathcontent[2]

        if ($folderHash[$folderPath].Contains($folderPath+'/main.tf'))
        {
                $mainfilepath=$folderPath+'/main.tf'

                $maincontent = [IO.File]::ReadAllText('./'+$mainfilepath)

                $found = $maincontent -match '^resource\s*"(.*)"\s*".*"\s{'
                if ($found)
                {
                    $resourcename=$matches[1]
                }
        }
```

- The iteration first process the main.tf file content.
- Using RegEx expression it finds out the Terraform resource name.
- From the RegEx expression it is clear that it highly depends on file format of Terraform Hub module main.tf file.

Step 5.

```
if ($folderHash[$folderPath].Contains($folderPath+'/outputs.tf'))
{

        $ofilepath=$folderPath+'/outputs.tf'
        $outputarr=""
        $content = [IO.File]::ReadAllText('./'+$ofilepath)
        $content =$content -replace '\/\*((^)?.*\r?\n?)*\*/|#.*',""
        $outputarr = $content -split '^output ',0,'Multiline'
        $outputarr.count
        $jstring = ""

        foreach ($outitem in $outputarr )
        {
                $found = $outitem-match '(".*")\s{\r?\n?(\s*)value(\s*)=\s*.*(\r?\n?\s*)description(\s*)=(\s*)(".*")\r?\n?}'

                if ($found)
                {
                        $jstring=$jstring+" { ""name"" : " + $matches[1]   + ", " +"""description"" : "+ $matches[7]  + " },"
                }
        }
}
```

- The iteration then takes the outputs.tf file.
- Using RegEx expression it removes any comment that is part of the file.
- Initialize a variable (i.e. jsstring) that holds the matched data.
- Keeping in mind the backend database collection the value of jsstring been prepared.

Step 6.

```
if ($folderHash[$folderPath].Contains($folderPath+'/variables.tf'))
{
        $outputarr=""
        $vfilepath=$folderPath+'/variables.tf'
        $content = [IO.File]::ReadAllText('./'+$vfilepath)
        $content =$content -replace '\/\*((^)?.*\r?\n?)*\*\/|#.*',""
        $outputarr = $content -split '^variable ',0, 'Multiline'
        $outputarr.count
        $jvstring = ""

        foreach ($outitem in $outputarr )
        {
                $found = $outitem -match '(".*")\s\{\r?\n?(\s*)description(\s*)=(\s*)(".*")\r?\n?(\s*)type(\s*)=(\s*)(.*)\r?\n?'
                if ($found)
                {
                        if ($matches[9].trim() -like 'list*')
                        {
                                $matches[9]= 'list(object)'
                        }
                        $jvstring=$jvstring+" { ""name"" : " + $matches[1]   + ", " +"""description"" : "+ $matches[5]  + ", "+"""Type"" : """+
$matches[9].trim()+""" },"
                }
        }
}
```

- The iteration then takes the variables.tf file.
- Using RegEx expression it removes any comment that is part of the file.
- Initialize a variable (i.e. jvstring) that holds the matched data.
- Keeping in mind the backend database collection the value of jvstring been prepared.

Step 7.

- Once all the required files are processed.
- The Terraform Validator Executable been triggered with all required arguments

```
    if ($modulename)
    {
            ./terraformvalidator/TerraformValidator.exe "${env:CON}", "$modulename", "$resourcename", "$folderPath", "$jvstring", "$jstring","${env:DB}"
    }
}
```

Challenges in implementing the solution

- The Microsoft Hosted vs Self-Hosted DevOps Agent
  - In case if end user has big Terraform module hubs that includes hundreds of modules, many times it has been observed that the validation pipeline gets aborted without prior notification, if the tasks takes more than 1 Hour to complete. In such cases it is better to use self-hosted agent instead of Microsoft hosted agent.
  - With Microsoft hosted agent, sometime “System.Management.Automation.RemoteException” may occur, using self-hosted agent also resolves the issue.

- Azure DevOps PowerShell tasks gives 2 options to execute the scripts. The “Inline” script option is good to quickly run the script logic and test, however it is recommended to run the PowerShell scripts using “File Path” based option where the logic is big and complex. It helps users to write logics better way, maintain and read.

- If any Terraform commands executed without proper configuration it outputs enormous non-printable / special characters as output. To avoid those unwanted characters and generate plain and easy to read output, it is better to use Terraform commands with configuration like “-no-color”. Also it is better to use UNIX output redirection notation along with the Terraform command to capture correct information in the command output file.

- It is better to use the Terraform provider backend configuration as a part of Terraform scripts rather than using Terraform Azure DevOps Task extension. The Terraform task extension is a free and easy to configure extension, however it depends on Azure DevOps Service Connection and user can select one subscription at design/configuration time only. To set different subscription based on the requirement it is better to write Terraform provider backend configuration as a part of Terraform script itself.

The benefits of the solution

The solution provides the benefits over the problem statements as mentioned previously. It outlines an automated way to keep updated Terraform modules and get in sync with Terraform release roadmap.

- With this solution user can detect any future or upcoming release that may impact their existing Terraform registry.
- User can experience a hassle free maintenance of Terraform module, even at the attribute, argument level.
- Solution used Azure DevOps as the collaboration tool to make the team aligned with Terraform module creation and modification. In turn helps end users to use the updated modules every time.
- Release of new modules, attributes, arguments or modification of the same, can be published various way to the intended users to keep them updated with new releases.