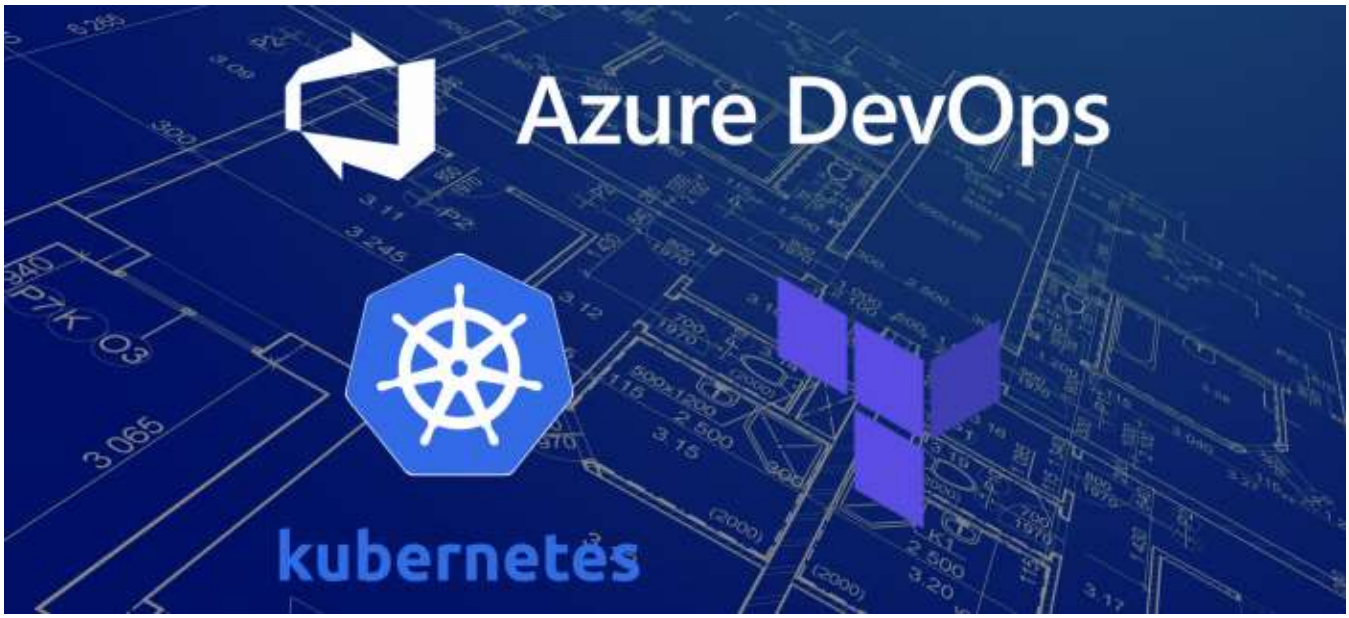
Nicolas Yuen

Follow 71 Followers About

Deploying AKS with Terraform and Azure DevOps

Nicolas Yuen Aug 20, 2019 · 11 min read



This is the first article of a multi-part series focused on AKS:

- Deploying AKS with Terraform and Azure DevOps
- Building and deploying a sample application with Azure DevOps and Azure Container Registry and AKS
- Performing a blue/green deployment to update the application
- Testing new features with Canary testing
- Exploiting the AKS and containers logs with Azure Monitor Logs

new AKS with Azure DevOps through different steps while trying to adhere to the **12 factor apps** guidelines:

- Version everything: Application Code, **Infra As Code, CI/CD pipeline**
- Disposability: Fast startups and graceful shutdowns (of the application and the infrastructure)
- Dev/Prod parity
- Logs: Treat logs as event streams

Source

The code is located on my github : **https://github.com/nyuen/AKS-Articles**

```
git clone https://github.com/nyuen/AKS-Articles.git
```

Requirements

- An Azure subscription
- An Azure DevOps organization + Git Repository: https://docs.microsoft.com/en-us/azure/devops/organizations/accounts/create-organization?view=azure-devops
- The Azure CLI (or the Azure Cloud shell), Terraform, and Kubectl

Preview features

I'm using a few preview features for AKS including the VM scaleSets and multi node pools. Follow the documentation below to enable these new features on your Azure Subscription

Azure/AKS

Please be aware, enabling preview features takes effect at the Azure subscription level. Do not install preview...

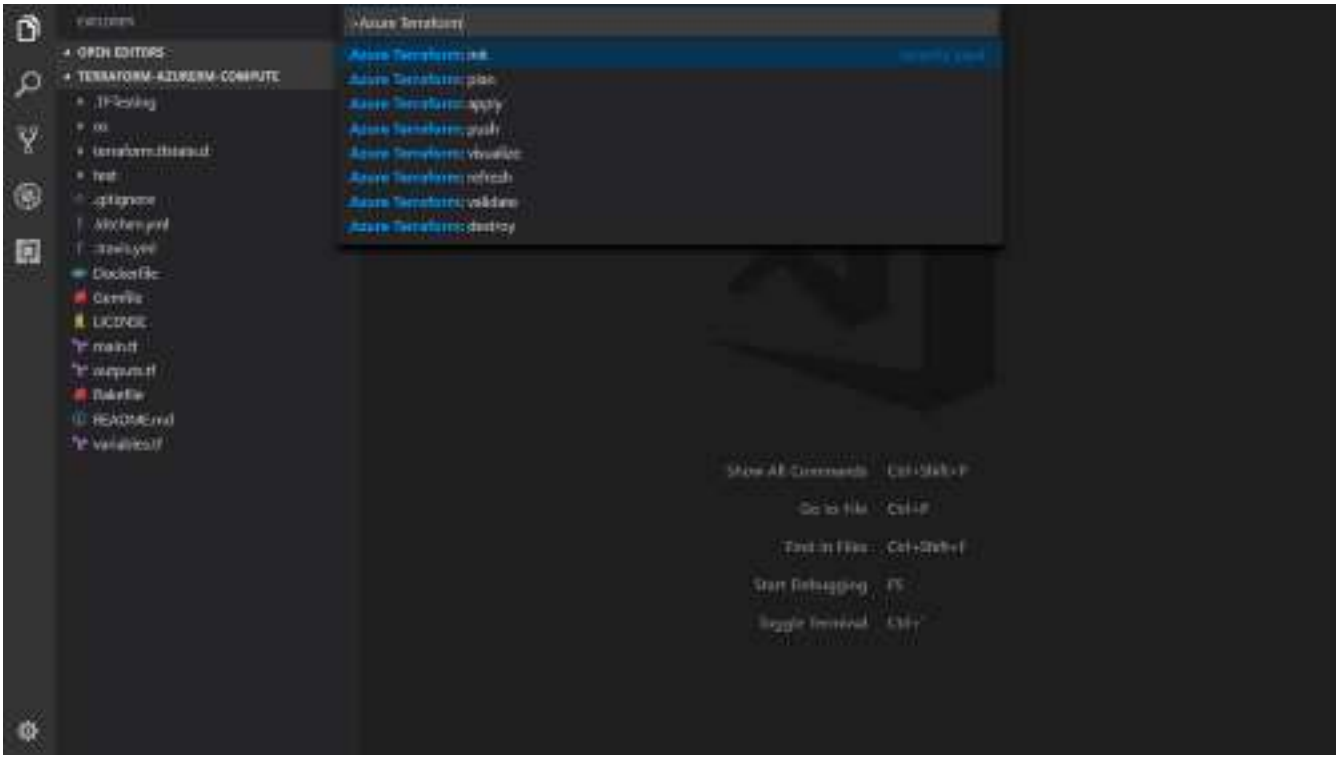
github.com

```
Microsoft.ContainerService

az feature register -n VMSSPreview --namespace
Microsoft.ContainerService

#Then refresh your registration of the AKS resource provider:

az provider register -n Microsoft.ContainerService
```

Terraform 0.12 and Visual Studio Code

I've decided to use Terraform 0.12 to create the infrastructure on Azure since it now supports first-class expression syntax and iteration constructs among a bunch of new features.

Before we dive in the Terraform code, here are a few tools I use to easily edit my Terraform code: Visual Studio Code for Mac + some handy extensions.



Azure Terraform Extension for VS Code: Allows you to run Terraform command from VSCode locally or directly on the Azure Cloud Shell

Terraform for VS Code: Supports code completion, formatting, linting, syntax highlighting, definition…

cluster including Advanced networking with Azure CNI which enables integration with existing Azure resources within a VNet.

To correctly maintain the state of our Terraform deployment we will require a few resources upfront:

- A storage account to act as a backend for Terraform: A "backend" in Terraform determines how state is loaded and how an operation such as `apply` is executed. Having a remote backend such as an Azure Storage Account has several benefits including working with multiple team members or performing long running remote operations (e.g Azure Pipeline for instance)
- An Azure KeyVault to store secrets and sensitive information, we will store a SAS token for the storage account in the Keyvault
- A Service Principal (required for AKS)
- An SSH Key (required for the Linux VM / AKS nodes)

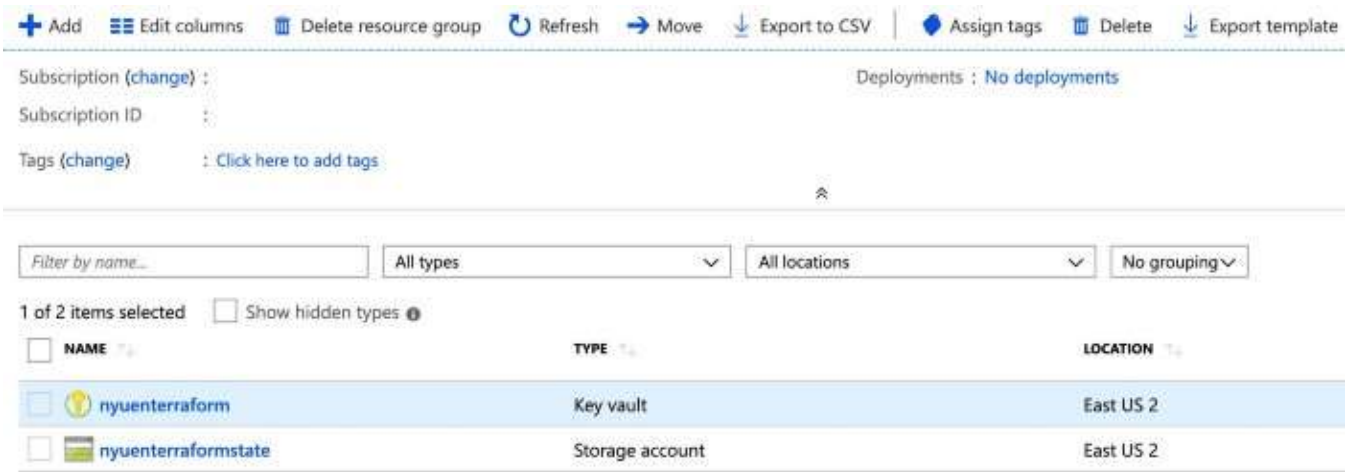
I've created an Azure Resource Group called AksTerraform-RG containing the two above mentioned Azure resources. We need to perform a few more steps to store sensitive data in KeyVault and minimize the risk of leaking a secret in a configuration file.

```
#creating the resource Group
az group create -n AksTerraform-RG -l eastus2
Location    Name
----------  ---------------
eastus2     AksTerraform-RG

#creating the storage account
az storage account create -n <YOUR_STORAGE_ACCOUNT_NAME> -g
AksTerraform-RG -l eastus2

#creating a tfstate container
az storage container create -n tfstate --account-name
<YOUR_STORAGE_ACCOUNT_NAME>

#creating the KeyVault
az keyvault create -n <YOUR_KV_NAME> -g AksTerraform-RG -l eastus2
Location    Name            ResourceGroup
```

```
#Creating a SAS Token for the storage account, storing in KeyVault
az storage container generate-sas --account-name
<YOUR_STORAGE_ACCOUNT_NAME> --expiry 2020-01-01 --name tfstate --
permissions dlrw -o json | xargs az keyvault secret set --vault-name
<YOUR_KV_NAME> --name TerraformSASToken --value

#creating a Service Principal for AKS and Azure DevOps
az ad sp create-for-rbac -n "AksTerraformSPN"

#creating an ssh key if you don't already have one
ssh-keygen  -f ~/.ssh/id_rsa_terraform

#store the public key in Azure KeyVault
az keyvault secret set --vault-name <YOUR_KV_NAME> --name
LinuxSSHPubKey -f ~/.ssh/id_rsa_terraform.pub > /dev/null

#store the service principal id in Azure KeyVault
az keyvault secret set --vault-name <YOUR_KV_NAME> --name spn-id --
value <SPN_ID> /dev/null

#store the service principal secret in Azure KeyVault
az keyvault secret set --vault-name <YOUR_KV_NAME> --name spn-secret
--value <SPN_SECRET> /dev/null
```

Make sure that you keep track of the generated **AppID**, **Password** and **Tenant**.

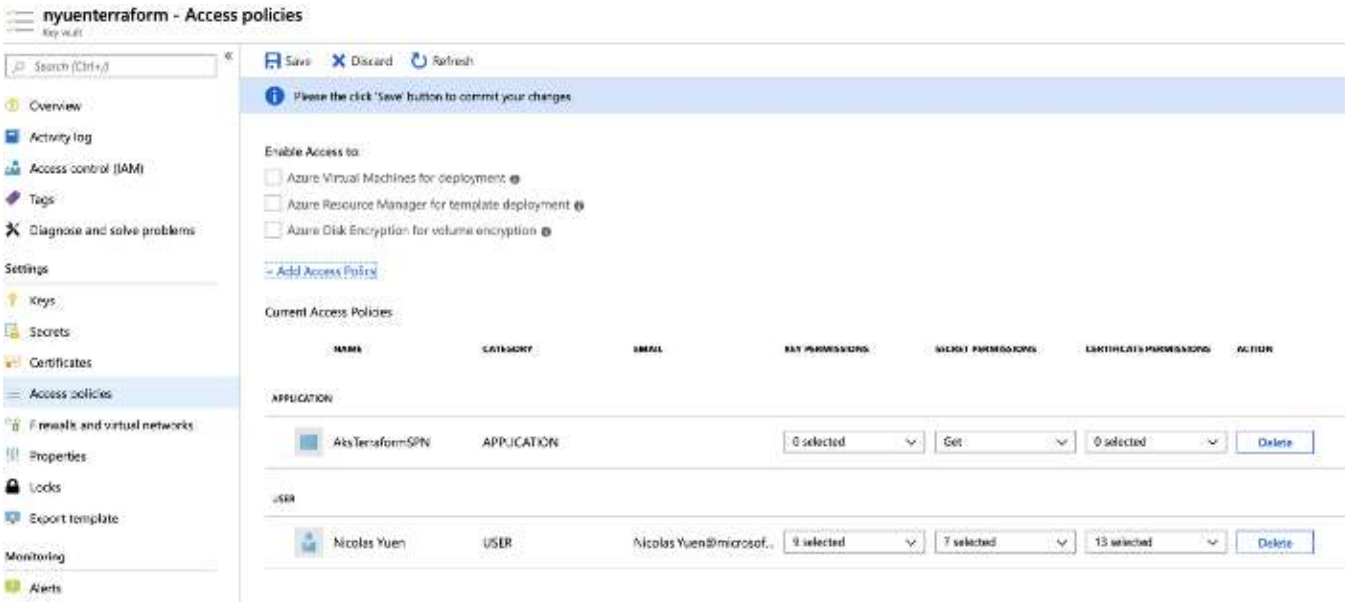
You should now have a **Resource Group** containing a **Storage Account + container** and a **KeyVault**



The KeyVault should contain four secrets:

spn-id	✓ Enabled
spn-secret	✓ Enabled
TerraformSASToken	✓ Enabled

You also need to grant Get access to the secret for the SPN we created



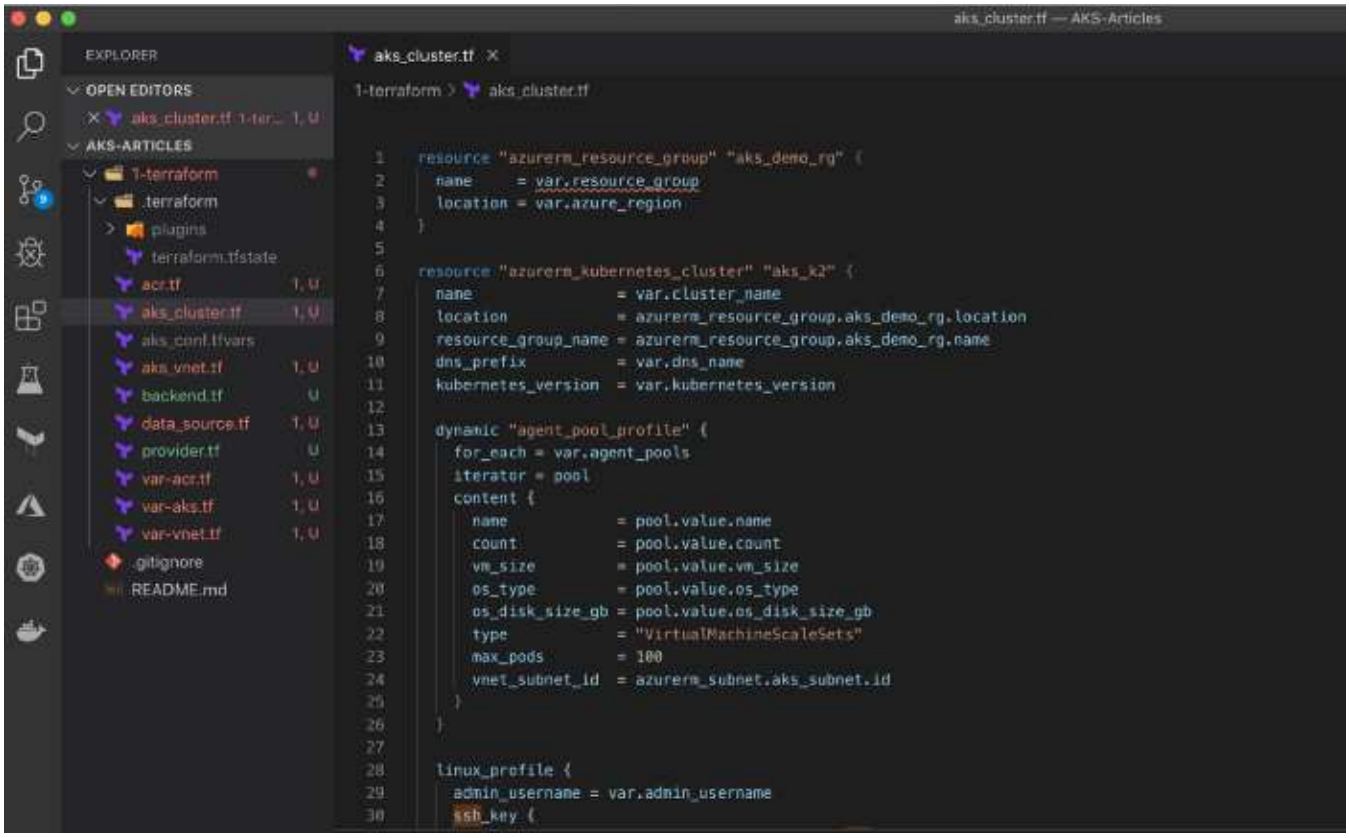
Terraform files

Let's now have a look at the Terraform files required to create the infrastructure (Vnet + Subnet, AKS Cluster, Azure Container Registry)

I've decided to split each resource in separate terraform files, the variable definitions are also isolated in dedicated files prefixed with *var-*. The code is located on my github at **https://github.com/nyuen/AKS-Articles**

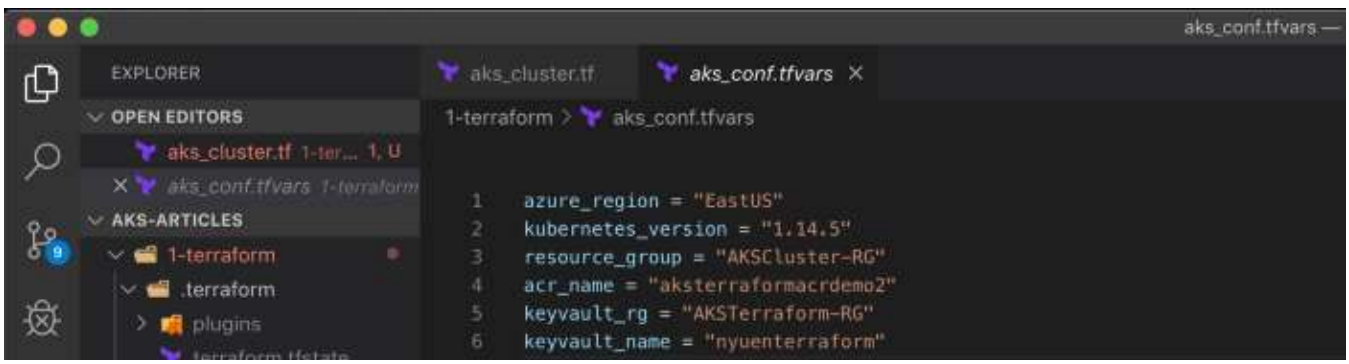
- acr.tf & var-acr.tf : creates the Azure Container registry
- aks_cluster.tf & var-aks.tf: creates the AKS cluster with Advanced Networking (CNI + Azure)
- aks_vnet.tf & var-vnet.tf: creates the Vnet and Subnet for the AKS cluster
- data_source.tf : Links the KeyVault data (the four secrets created)
- backend.tf : defines the Azure storage account as a backend for the Terraform State

be stored in a source repository as it often contains sensitive information.



Make sure to create a `aks_conf.tfvars` to input your configuration with your:

- resource group name (I chose a different resource group name to create the cluster)
- cluster name
- azure region
- kubernetes version
- KeyVault reference (KeyVault resource group and name)



To know which version of AKS is available on Azure you can use the following command:

```
az aks get-versions --location EastUS2

KubernetesVersion    Upgrades
-------------------  ------------------------
1.14.5               None available
1.13.9               1.14.5
1.12.8               1.13.9
1.12.7               1.12.8, 1.13.9
1.11.10              1.12.7, 1.12.8
1.11.9               1.11.10, 1.12.7, 1.12.8
1.10.13              1.11.9, 1.11.10
1.10.12              1.10.13, 1.11.9, 1.11.10
```

Unfortunately the Terraform Backend resource does not support variables, we will need to pass the parameter in the init method to correctly configure the backend in our CI/CD pipeline

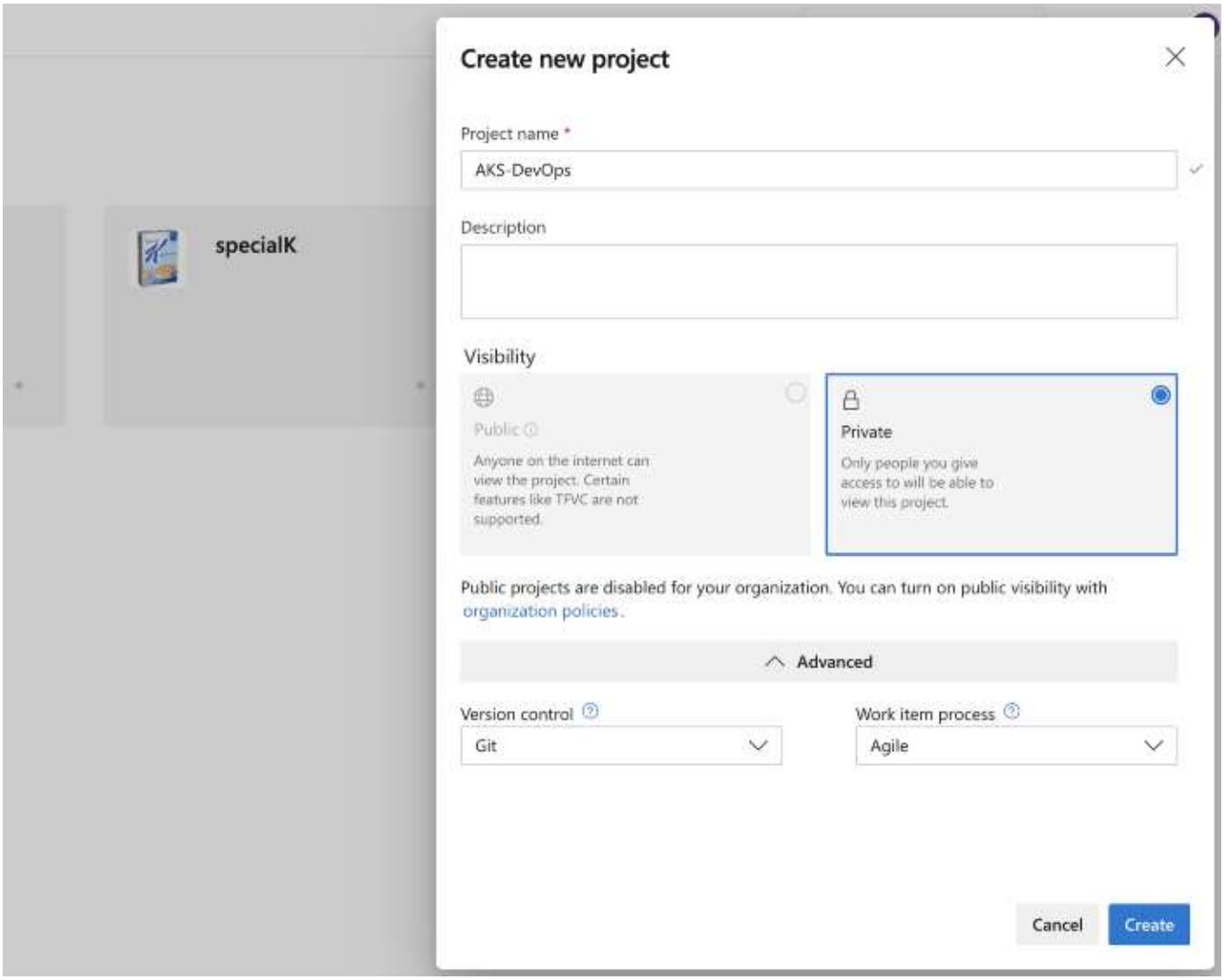
```
terraform init \
    -backend-config="resource_group_name=
<RG_containing_your_storage_account>" \
    -backend-config="storage_account_name=<YOUR_SA_NAME>" \
    -backend-config="container_name=tfstate"
```

We can now verify that the Terraform files are valid with the command line below

```
terraform validate -var-file=aks_conf.tfvars

Success! The configuration is valid.
```

it's time to automate the process. I'm using Azure Pipelines and the newly released **YAML multi-stage** definition which allows both CI and CD stages to be defined as a single YAML file. You can find more info on this new feature in the Azure DevOps documentation: https://www.azuredevopslabs.com/labs/azuredevops/yaml/

Let's first start by creating a new Project on Azure DevOps, I've called this project AKS-AzureDevOps, to do so login on dev.azure.com



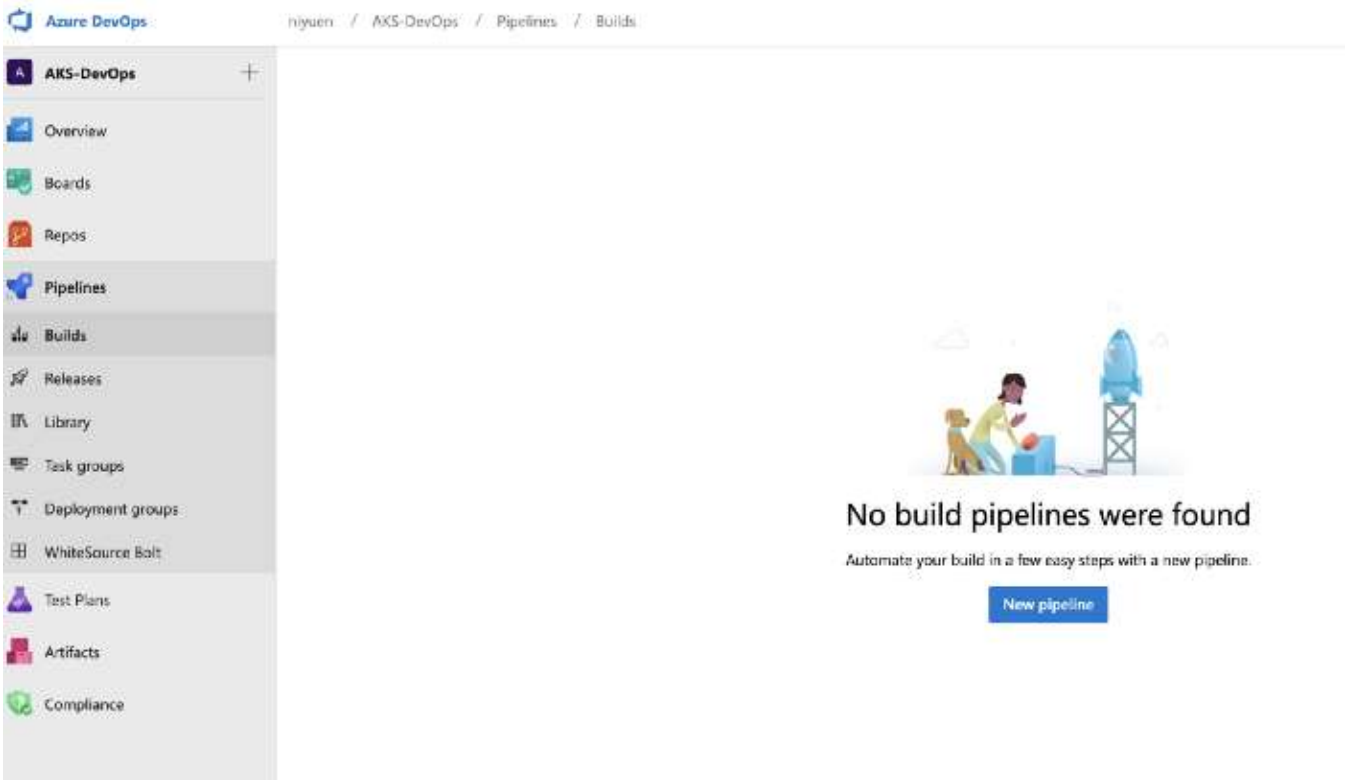
Installing a Terraform extension

I decided to use a MarketPlace extension for Azure DevOps : https://marketplace.visualstudio.com/items?itemName=charleszipp.azure-pipelines-tasks-terraform

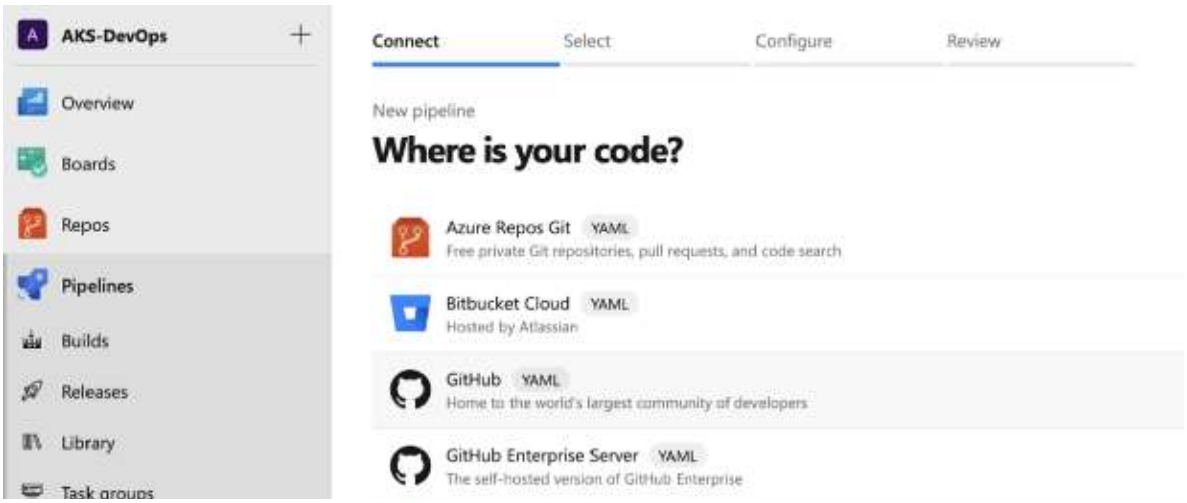
This extension will simplify the use of Terraform in the pipeline by wrapping commands and installing the Terraform CLI on the Agent.

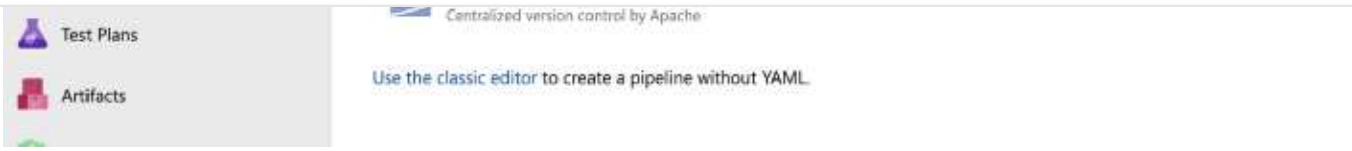
Setting up a basic YAML pipeline

Let's go straight to Azure Pipeline to create our deployment.

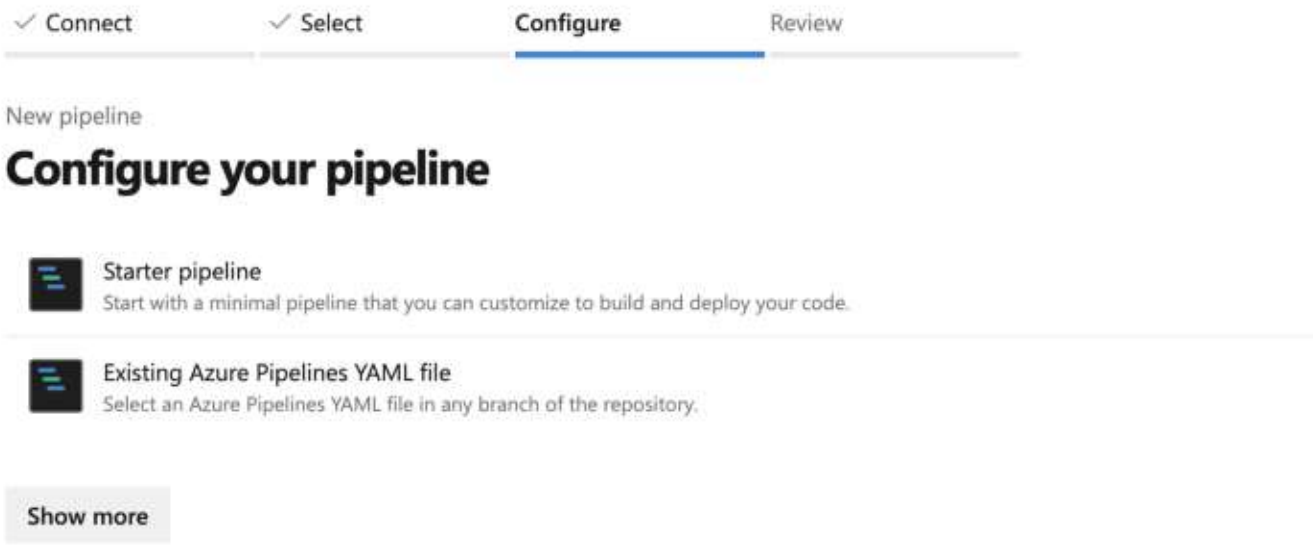


My repo is on a public GitHub, therefore I've selected GitHub `YAML`

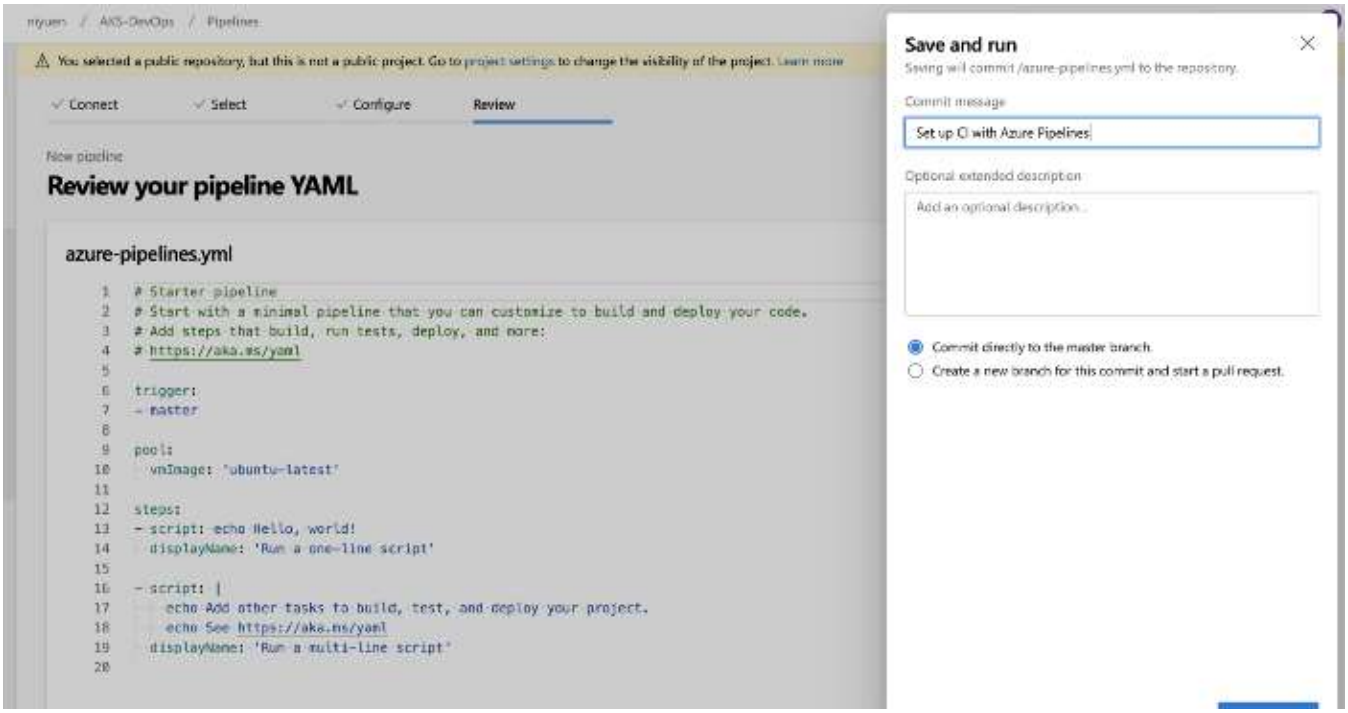




Azure Pipeline will automatically create a new YAML file based on the content of my repository, in this specific scenario I don't have much code so the pipeline will be fairly basic to start with



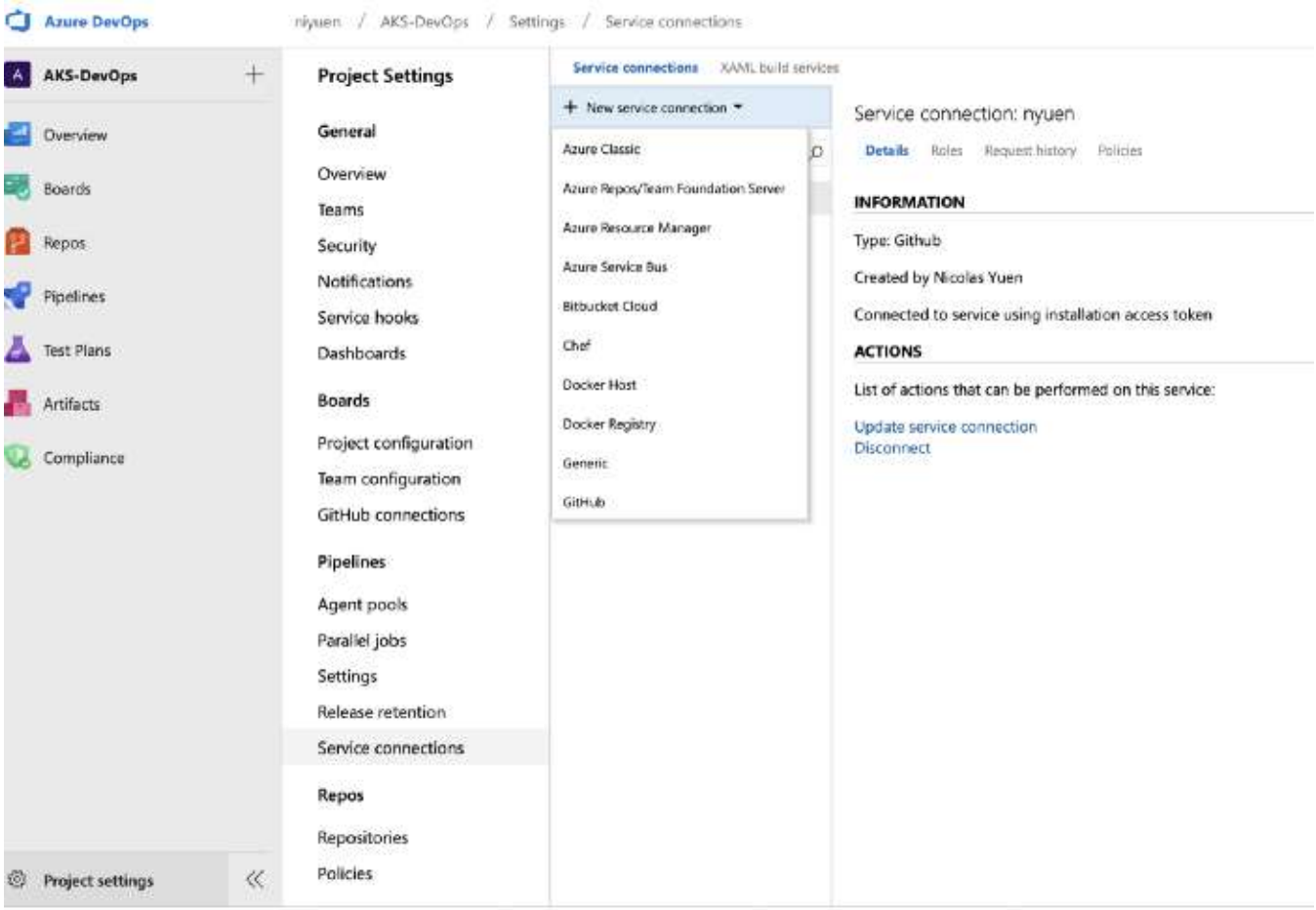
Clicking save and run will commit the basic pipeline to my repository and run a first pipeline



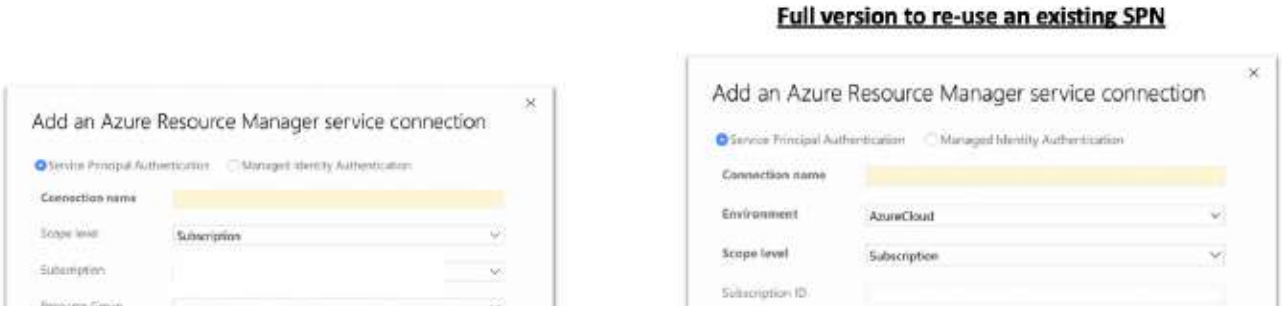
Obviously the first run is successful considering that so the pipeline is very basic right now.

Connecting Azure DevOps with Azure (with Service Principal)

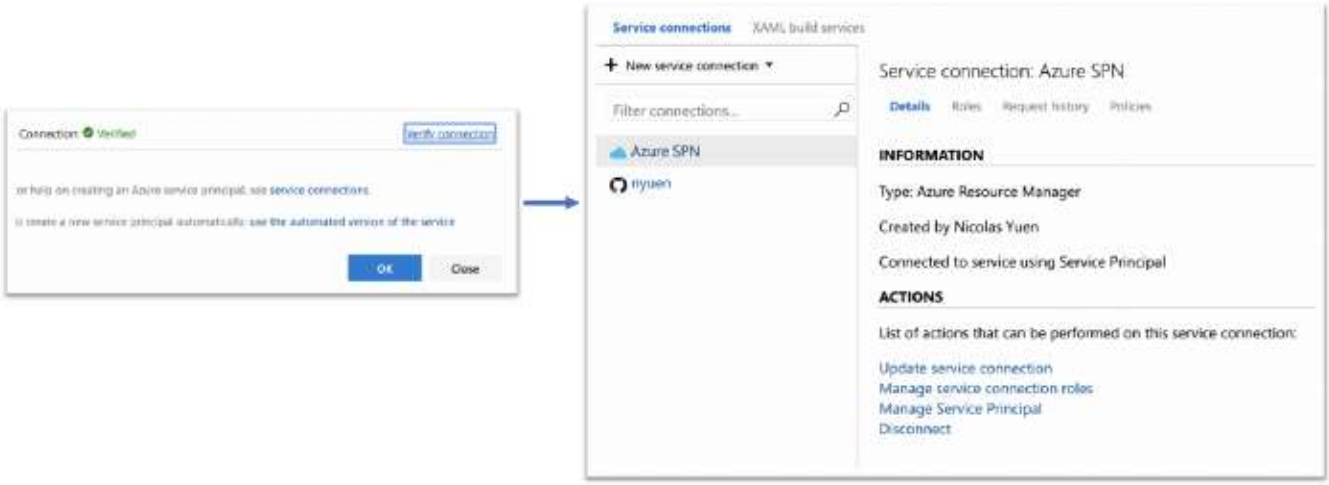
Before we start to modify the YAML file to deploy our infrastructure we need to configure Azure DevOps to be able to create resources on Azure using the Service Principal that we generated. Add a **service connection** on **Project Settings->Service Connections->New Service Connection -> Azure Resource Manager**



Since we already have a Service Principal created in *step 1* we don't need to let Azure DevOps create a new one, use the Advanced Service Connection Editor instead:

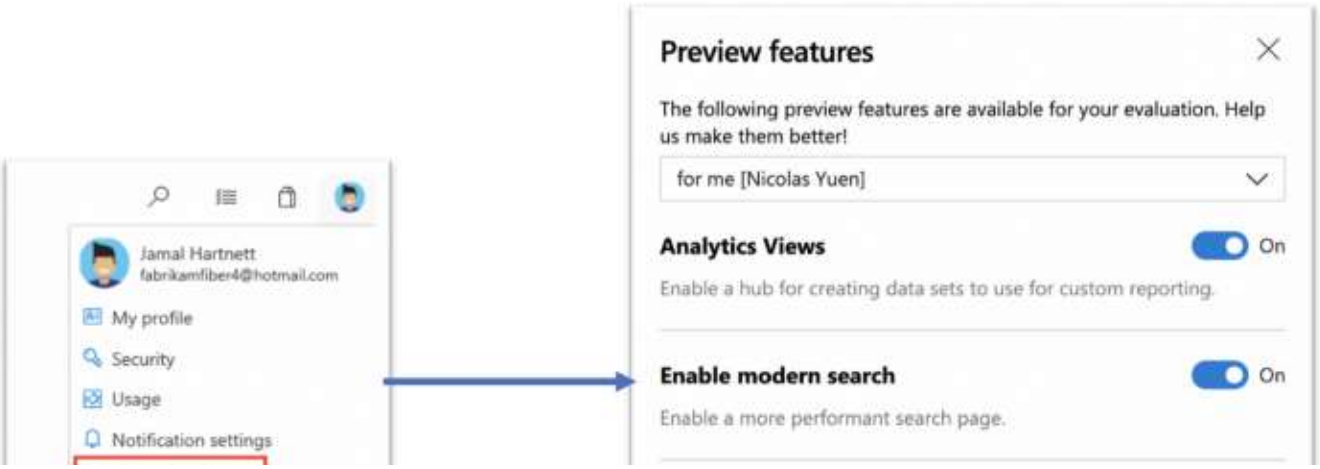


You will be able to manage Azure resources through Azure DevOps once the ARM connection is validated:



Enable the New Multi-stage pipeline

Stages are the major divisions in a pipeline: “build this app”, “run these tests”, and “deploy to pre-production” are good examples of stages. They are a logical boundary in your pipeline at which you can pause the pipeline and perform various checks. Since we are editing our pipeline using the new YAML experience we need to enable the Multi-Stage preview feature in the user settings:

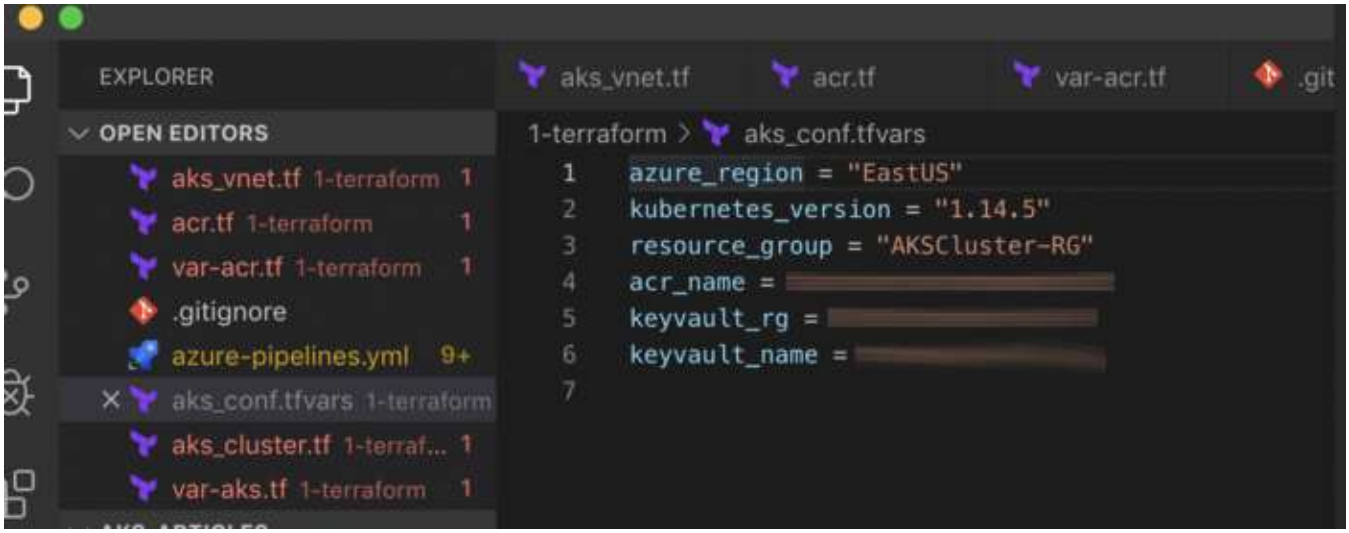


We had to ho through a few steps to get there but Azure Pipeline is now properly configured to deploy our resources on Azure.

3. Deploy the infrastructure with Terraform and Azure Pipelines

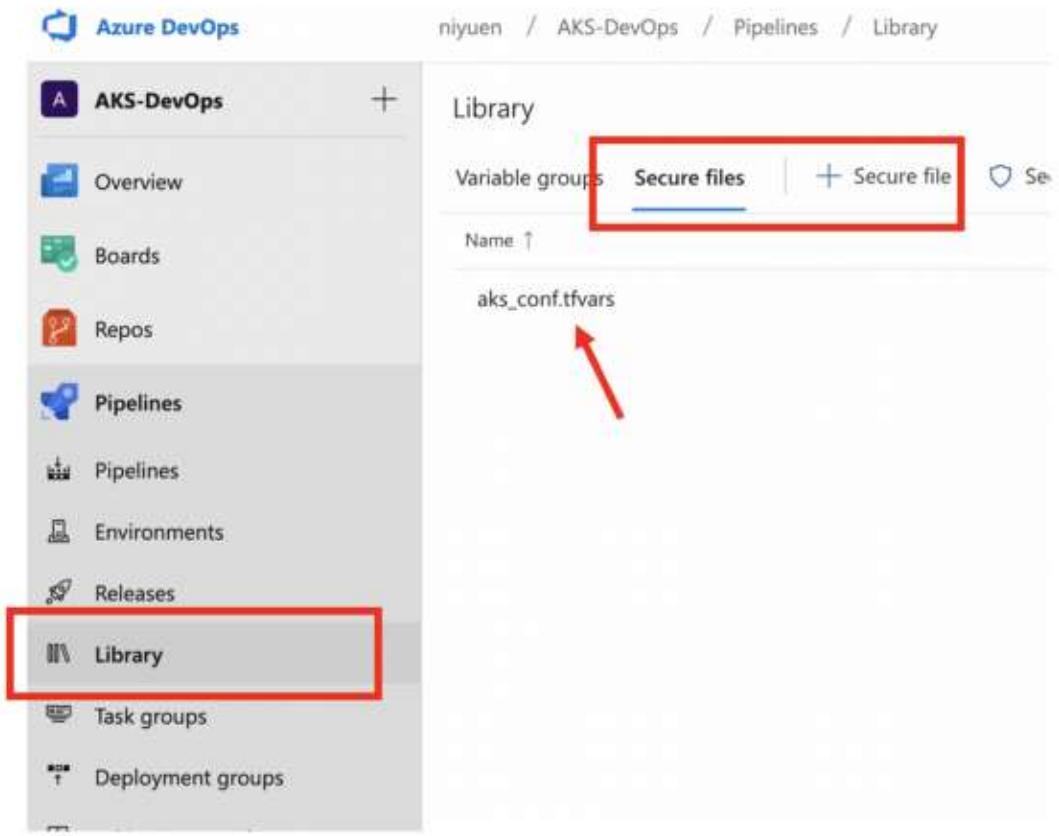
Adding the .tfvars file to the Azure Pipelines secure file library

I've added the *.tfvars pattern to my .gitignore to make sure secrets or sensitive data are not leaked. You can either store sensitive information in Azure KeyVault or leverage the Secure File Library from Azure DevOps. Below is a screenshot of my aks_conf.tfvars :



Here is the code of the aks_conf.tfvars

```
azure_region = "EastUS"
kubernetes_version = "1.14.5"
resource_group = "<YOUR_VALUE>"
acr_name = "<YOUR_VALUE>"
keyvault_rg = "<YOUR_VALUE>"
keyvault_name = "<YOUR_VALUE>"
```



Modifying the Pipeline YAML file

It's now time to update the pipeline to perform the following Terraform tasks using the ARM connection:

- Terraform init
- Terraform validate
- Terraform plan
- Terraform apply

We are going to split these commands in two stages : *Validation* and *Deployment*, here is the structure of our CI/CD pipeline

The **Validation** phase will perform Terraform `init`, `validate` while the **Deployment** phase will perform the `init`, `plan` and `apply` commands

Phase one is all about testing if the terraform files are valid through the `terraform validate` command

```
stages:
- stage: Validate
  jobs:
  - job: Validate
    continueOnError: false
    steps:
    - publish: 1-terraform
      artifact: terraform_out
    - task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-tasks-terraform-installer.TerraformInstaller@0
      displayName: 'Use Terraform v0.12.6'
      inputs:
        terraformVersion: 0.12.6
    - task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-tasks-terraform-cli.TerraformCLI@0
      displayName: 'terraform init'
      inputs:
        command: init
        workingDirectory: 1-terraform
        backendType: azurerm
        backendServiceArm: 'Azure SPN'
        backendAzureRmResourceGroupName: 'AKSTerraform-RG'
        backendAzureRmStorageAccountName: 'nyuenterraformstate'
        backendAzureRmContainerName: 'tfstate'
        backendAzureRmKey: 'demo.terraform.tfstate'
    - task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-tasks-terraform-cli.TerraformCLI@0
      displayName: 'terraform validate'
      inputs:
        command: validate
        workingDirectory: 1-terraform
```

let's break down the stage:

- We first copy the content of the terraform folder as a Pipeline Artifact using the `publish` feature of Azure Pipelines
- We `install` Terraform on the host using the Installation task from the Terraform Extension
- the `init` phase fetches the required Azure provider and links the deployment to the Backend (Azure Storage account)
- The `validate` phase performs a high level syntax check on the Terraform files

Next let's move to the **deployment** phase, I'm using the new deployment job provided by the Multi-stage pipeline experience, a deployment job is a special type of job t(a collection of steps to be run sequentially against the environment).Using deployment job provides some benefits:

- **Apply deployment strategy**: Define how your application is rolled-out

The deployment stage is also located in the YAML file:

```yaml
- stage: Deploy
  jobs:
    # track deployments on the environment
  - deployment: Deploy_Terraform
    pool:
      vmImage: 'ubuntu-latest'
    # creates an environment if it doesn't exist
    environment: 'DEV'
    strategy:
      # default deployment strategy
      runOnce:
        deploy:
          steps:
          # - download: current
          #   artifact: terraform_out
          - task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-tasks-terraform-cli.TerraformCLI@0
            displayName: 'terraform init'
            inputs:
              command: init
              workingDirectory: $(Pipeline.Workspace)/terraform_out
              backendType: azurerm
              backendServiceArm: 'Azure SPN'
              backendAzureRmResourceGroupName: 'AKSTerraform-RG'
              backendAzureRmStorageAccountName: 'nyuenterraformstate'
              backendAzureRmContainerName: 'tfstate'
              backendAzureRmKey: 'demo.terraform.tfstate'
          - task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-tasks-terraform-cli.TerraformCLI@0
            displayName: 'terraform plan'
            inputs:
              command: plan
              workingDirectory: $(Pipeline.Workspace)/terraform_out
              environmentServiceName: 'Azure SPN'
              secureVarsFile: 'aks_conf.tfvars'
              commandOptions: '-out $(Pipeline.Workspace)/terraform_out/terraform_aks_out'
              terraformVersion: 0.12.6
          - task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-tasks-terraform-cli.TerraformCLI@0
            displayName: 'terraform apply'
            inputs:
              command: apply
              workingDirectory: $(Pipeline.Workspace)/terraform_out
              environmentServiceName: 'Azure SPN'
              commandOptions: '$(Pipeline.Workspace)/terraform_out/terraform_aks_out'
              terraformVersion: 0.12.6
```

The deployment job is configured with a few parameters:

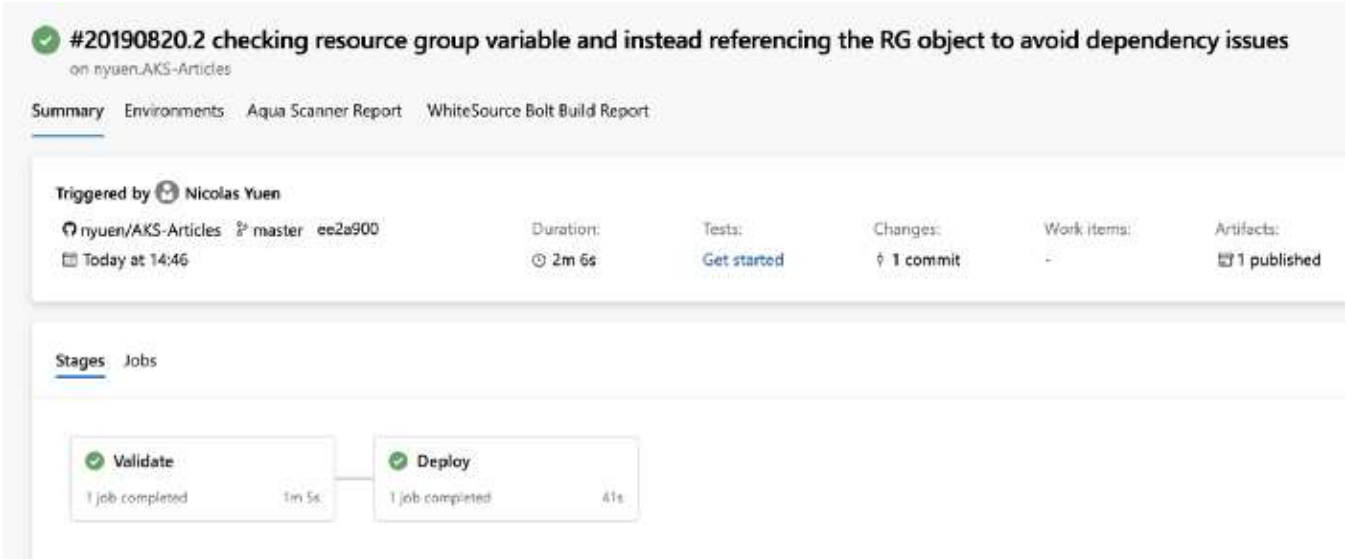
- deployment: the name of the job within the current stage
- strategy: as of August 2019, only RunOnce is supported
- environment : the name of the targeted environment, it is also possible to add a resource name (example dev.kubernetes)

the next tasks within the deploy steps are again leveraging the Terraform extension:

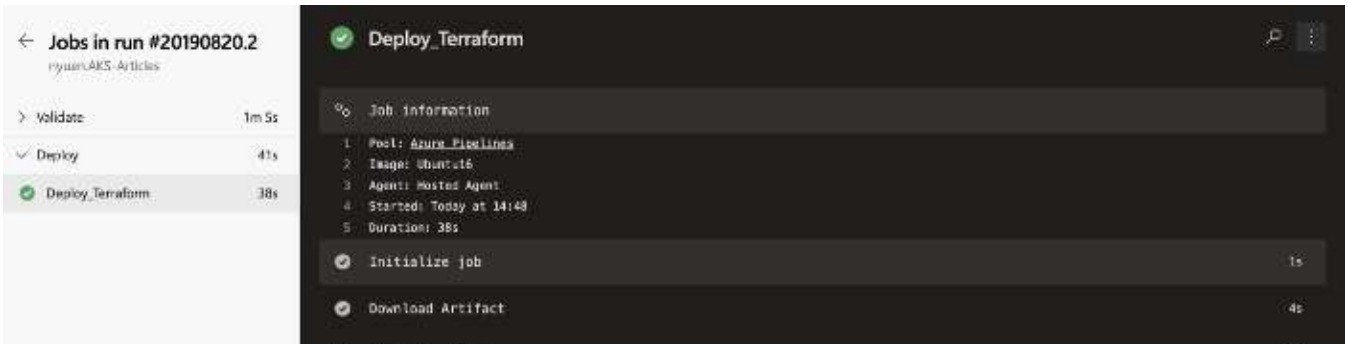
- init: assume that we have a new agent from the Microsoft hosted pool
- plan: outputs a terraform plan to terraform_aks_out
- apply: to deploy the resources to Azure

Note that the working directory is set to `$(Pipeline.Workspace)/terraform_out/` which maps to the artifact that we generated in the first step. By default, every artifacts generated in previous stages are automatically copied to the next stage, you can specify a specific artifact or none using the `-download` directive in yaml.

After running the pipeline you should have two successful stages : validation and Dev (as per the environment name set in the deployment job)

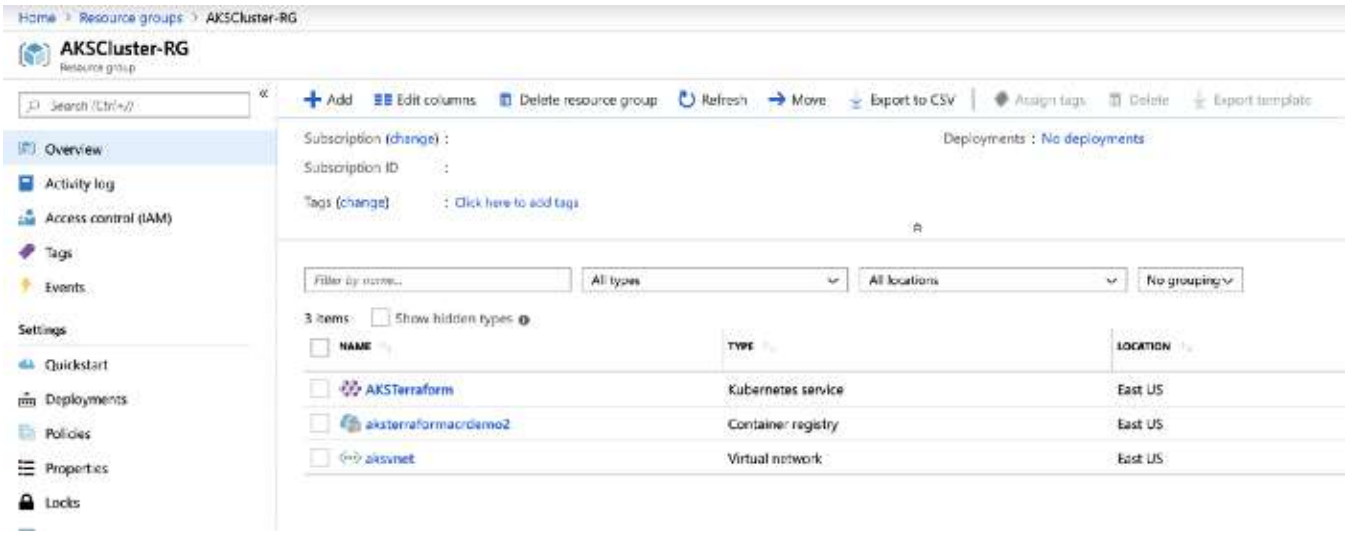


You can then navigate to the Environments tab and check the output of the deployment:

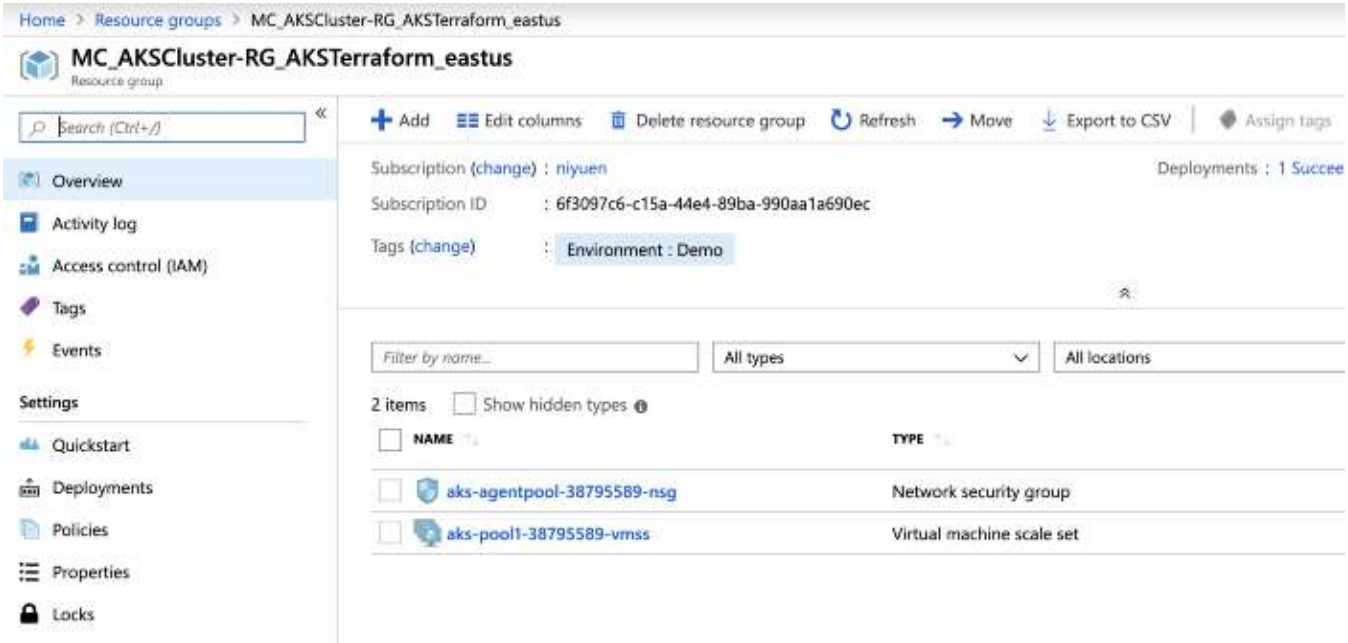


On the Azure side we now have our resources up and running including:

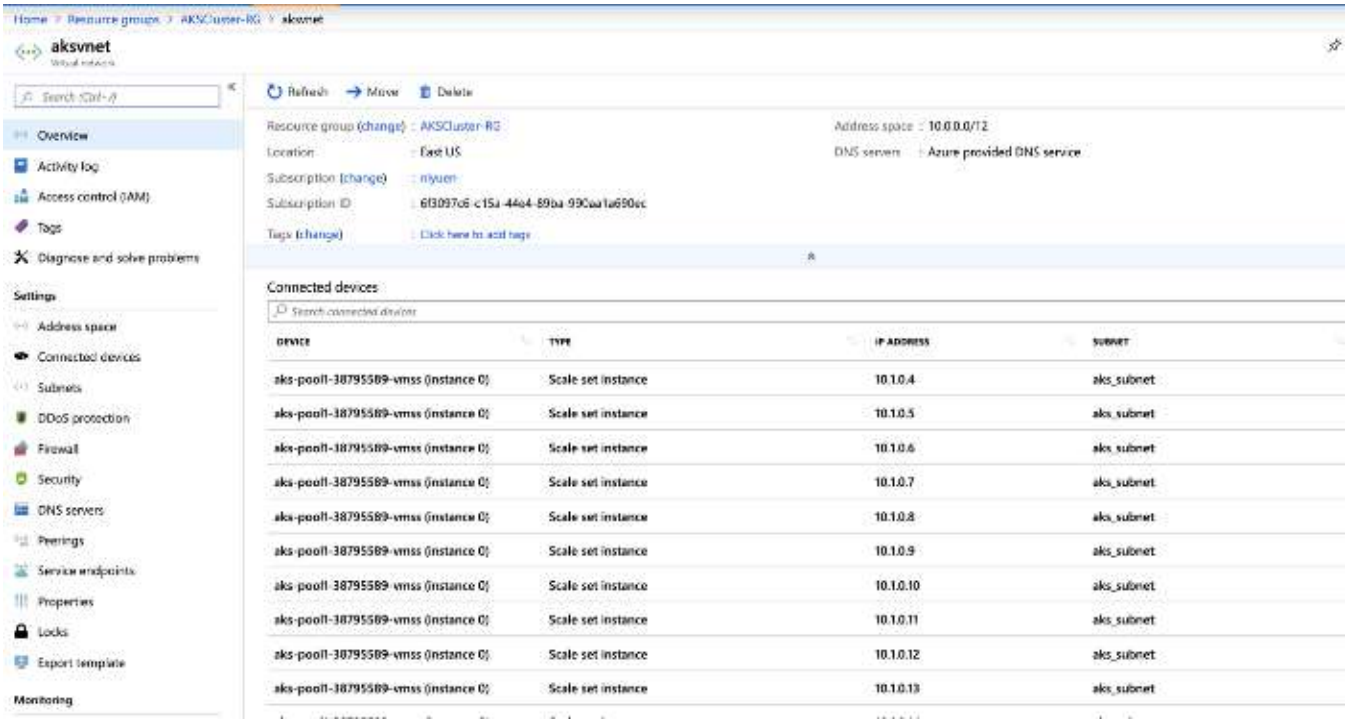
- an AKS cluster with 3 nodes using VMSS, Advanced networking with Azure CNI
- an Azure Container registry
- a VNet + Subnet



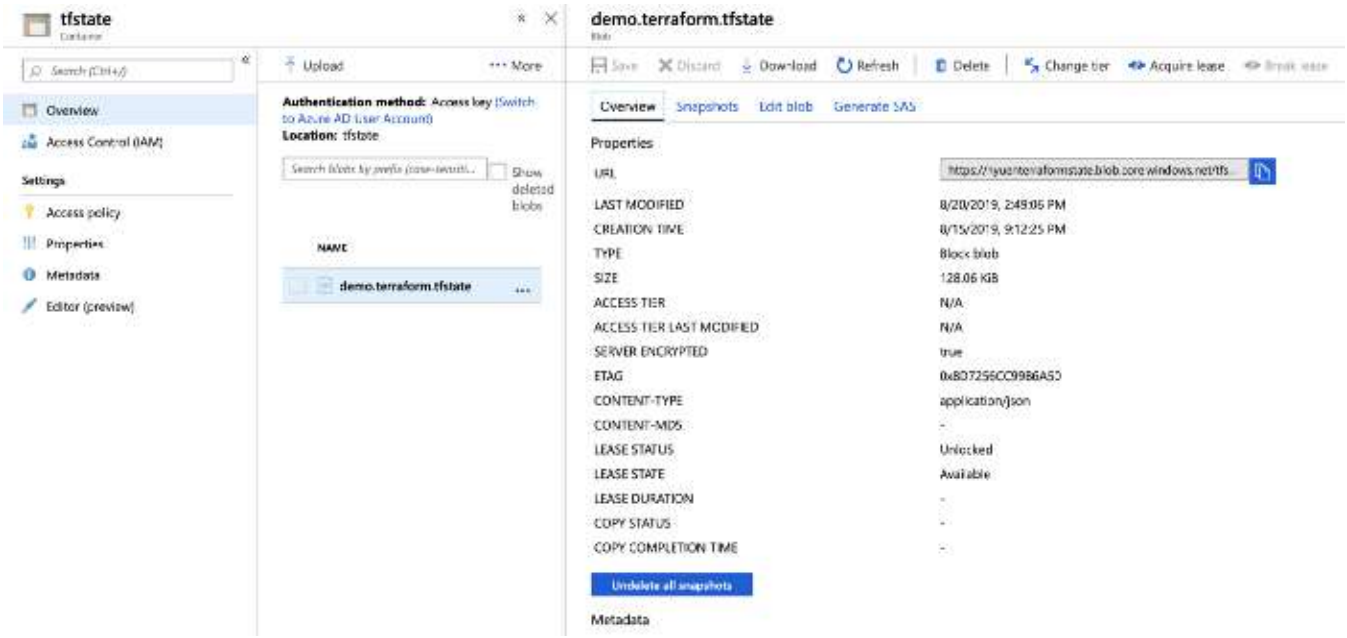
The nodes (Azure IaaS Vms) are located on a dedicated Resource Group



Azure CNI, we configured the cluster to support 100 pods per node and we have a total of 3 nodes : 3 x 100 = 300. The subnet is provisioned to secured the IPs upfront



The storage account for the backend now contains the deployment state of our resources



Conclusion

The deployment process is secured (KeyVault and Azure Pipelines secret files) and repeatable (CI/CD + Azure Backend for Terraform).

The next article will focus on deploying a sample Kubernetes application to this AKS instance + ACR using a dedicated CI/CD multi-stage pipeline.

Kubernetes Azure Azure Devops Cicd Terraform