

[Open in app](#)

## Nicolas Yuen

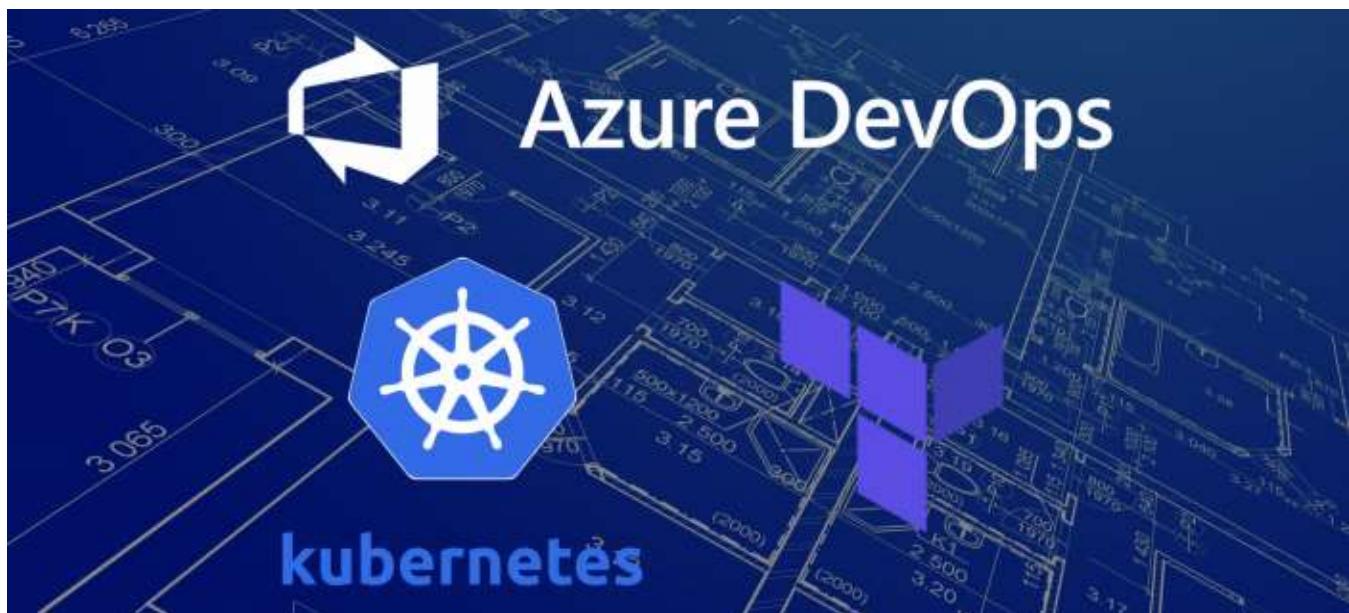
[Follow](#)

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

[Open in app](#)

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](https://github.com/nyuen/AKS-Articles)


[Open in app](#)

```
Microsoft.ContainerService

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

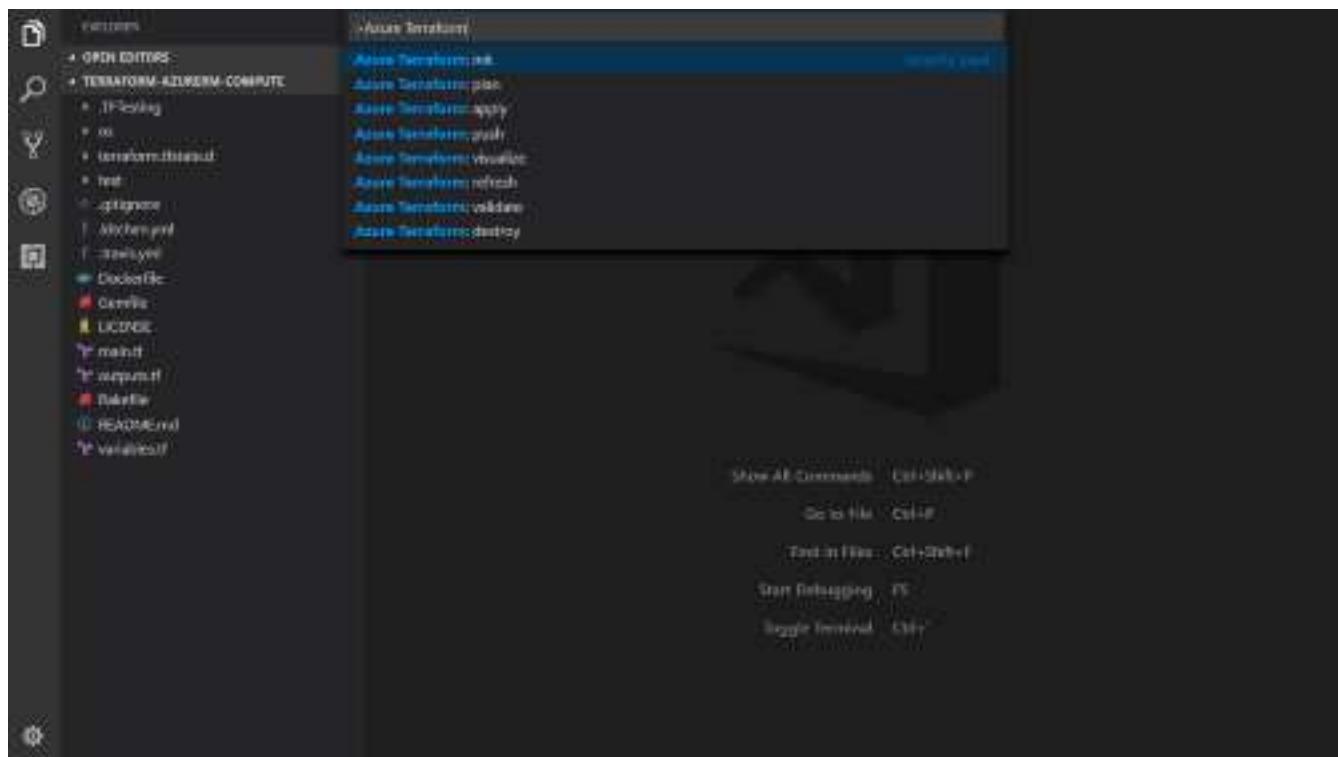
#Then refresh your registration of the AKS resource provider:

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...

[Open in app](#)

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
```


[Open in app](#)

```
#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**

NAME	TYPE	LOCATION
nyuenterraform	Key vault	East US 2
nyuenterraformstate	Storage account	East US 2

The KeyVault should contain four secrets:

[Generate/Import](#) [Refresh](#) [Restore Backup](#)


[Open in app](#)

spn-id	<input checked="" type="checkbox"/> Enabled
spn-secret	<input checked="" type="checkbox"/> Enabled
TerraformSASToken	<input checked="" type="checkbox"/> Enabled

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

NAME	CATEGORY	EMAIL	KEY PERMISSIONS	SECRET PERMISSIONS	LIMITED PERMISSIONS	ACTION
AksTerraformSPN	APPLICATION		0 selected	Get	0 selected	<button>Delete</button>
Nicolas Yuen	USER	Nicolas.Yuen@microsoft.com	0 selected	7 selected	15 selected	<button>Delete</button>

## 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



[Open in app](#)

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

```

1 resource "azurerm_resource_group" "aks_demo_rg" {
2   name         = var.resource_group
3   location     = var.azure_region
4 }
5
6 resource "azurerm_kubernetes_cluster" "aks_k2" {
7   name         = var.cluster_name
8   location     = azurerm_resource_group.aks_demo_rg.location
9   resource_group_name = azurerm_resource_group.aks_demo_rg.name
10  dns_prefix    = var.dns_name
11  kubernetes_version = var.kubernetes_version
12
13  dynamic "agent_pool_profile" {
14    for_each = var.agent_pools
15    iterator = pool
16    content {
17      name          = pool.value.name
18      count         = pool.value.count
19      vm_size       = pool.value.vm_size
20      os_type        = pool.value.os_type
21      os_disk_size_gb = pool.value.os_disk_size_gb
22      type          = "VirtualMachineScaleSets"
23      max_pods      = 100
24      vnet_subnet_id = azurerm_subnet.aks_subnet.id
25    }
26  }
27
28  linux_profile {
29    admin_username = var.admin_username
30    ssh_key {
31    }
32  }
33}
34
35
36
37
38

```

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)

```

1 azure_region = "EastUS"
2 kubernetes_version = "1.14.5"
3 resource_group = "AKSCluster-RG"
4 acr_name = "aksterraformcrdemon2"
5 keyvault_rg = "AKSTerraform-RG"
6 keyvault_name = "nyuenterraform"

```

[Open in app](#)

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=
<RGContainingYourStorageAccount>" \
  -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

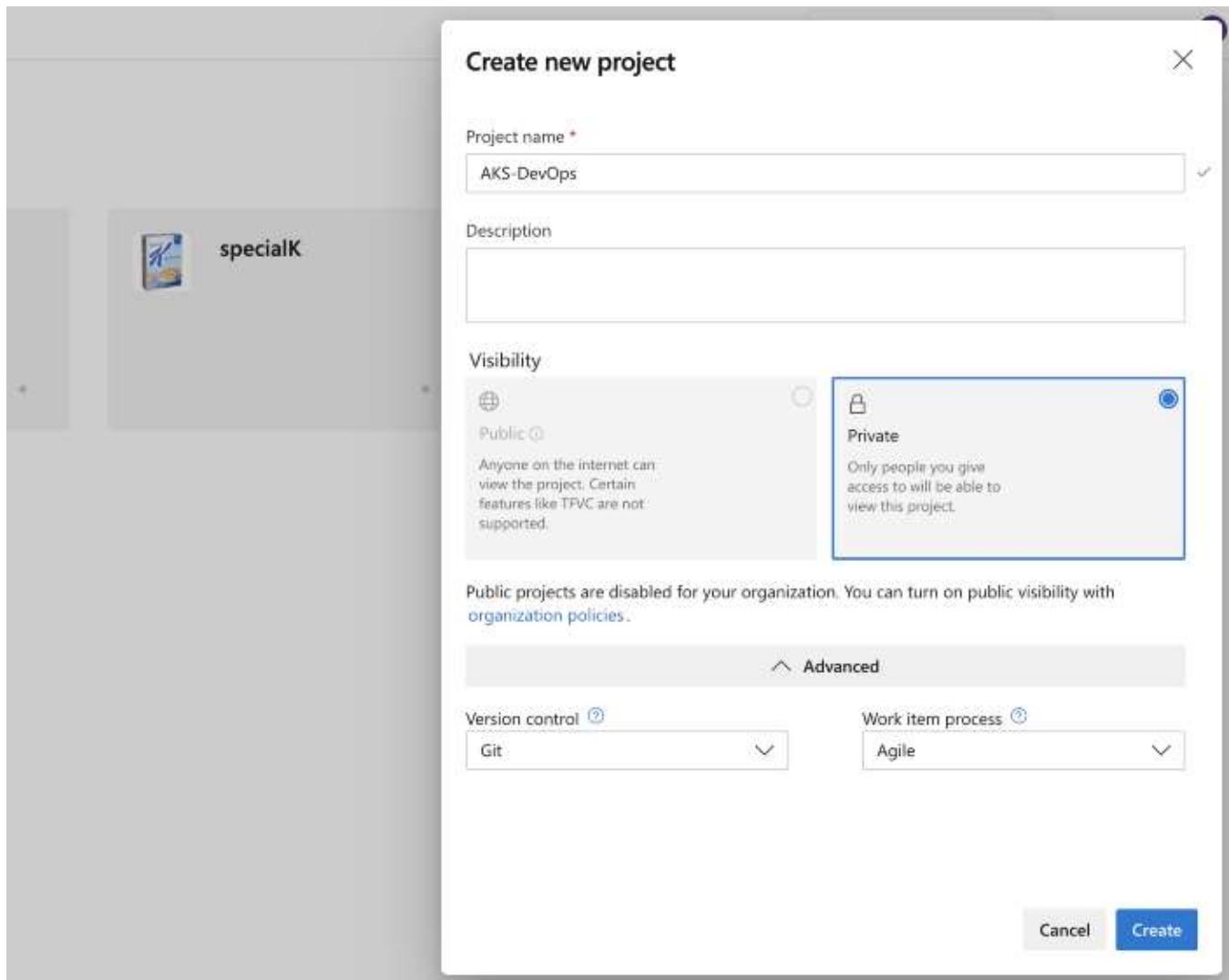
```
terraform validate -var-file=aks_conf.tfvars
```

Success! The configuration is valid.

[Open in app](#)

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](https://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.


[Open in app](#)



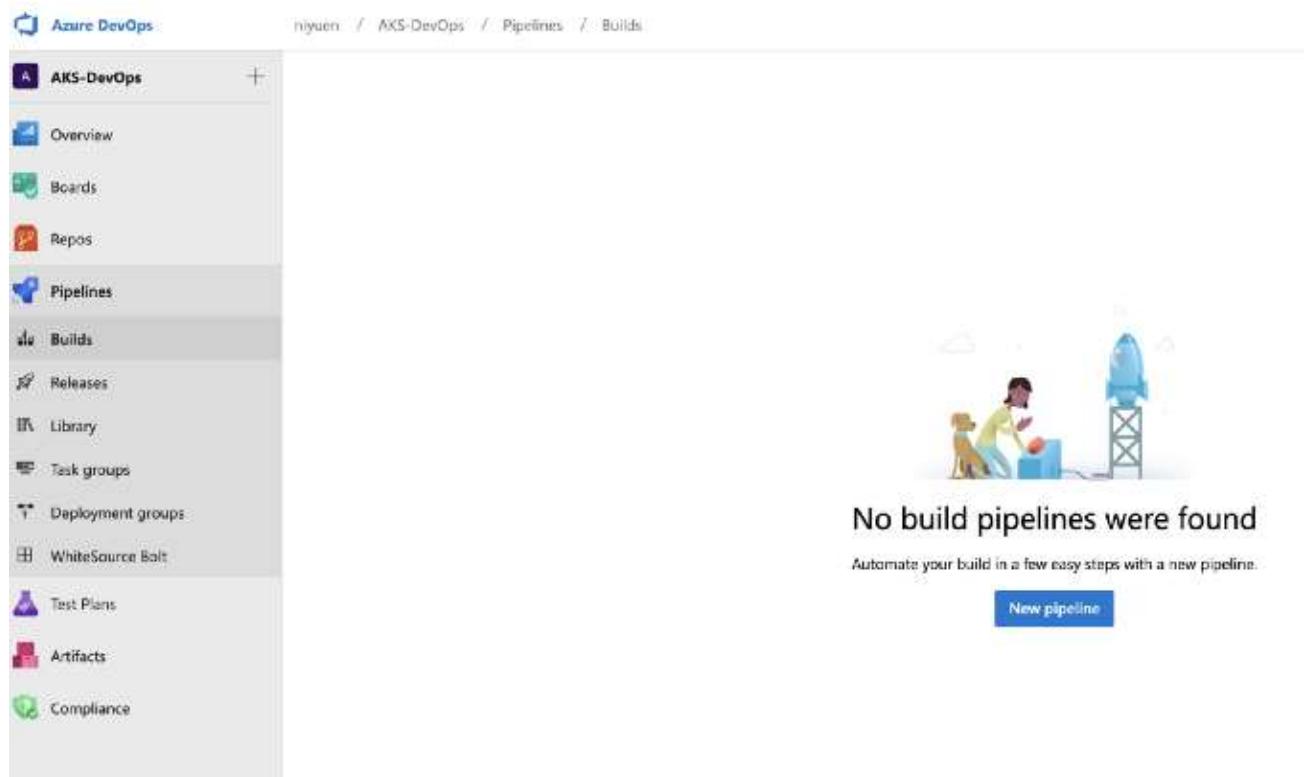
**Charles Zipp** | ± 1,885 installs | ★★★★★ (10) | Preview

Tasks to execute terraform commands during Azure DevOps Build & Release pipelines

[Get it free](#)

## Setting up a basic YAML pipeline

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

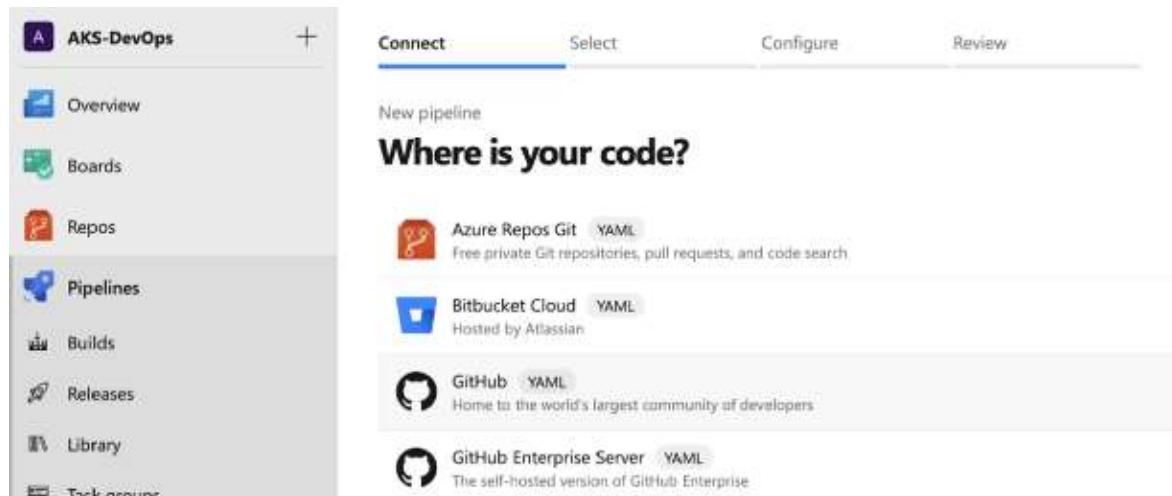


Azure DevOps / niyuen / AKS-DevOps / Pipelines / Builds

- AKS-DevOps
- + Overview
- Boards
- Repos
- Pipelines**
- Builds
- Releases
- Library
- Task groups
- Deployment groups
- WhiteSource Bolt
- Test Plans
- Artifacts
- Compliance

No build pipelines were found  
Automate your build in a few easy steps with a new pipeline.  
[New pipeline](#)

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



AKS-DevOps +

Connect Select Configure Review

New pipeline

**Where is your code?**

- Azure Repos Git `YAML`  
Free private Git repositories, pull requests, and code search
- Bitbucket Cloud `YAML`  
Hosted by Atlassian
- GitHub `YAML`**  
Home to the world's largest community of developers
- Github Enterprise Server `YAML`  
The self-hosted version of GitHub Enterprise


[Open in app](#)


Centralized version control by Apache

Use the classic editor to create a pipeline without 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

✓ Connect
✓ Select
Configure
Review

New pipeline

## Configure your pipeline



### Starter pipeline

Start with a minimal pipeline that you can customize to build and deploy your code.



### Existing Azure Pipelines YAML file

Select an Azure Pipelines YAML file in any branch of the repository.

[Show more](#)

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

The screenshot shows the Azure Pipelines interface for creating a new pipeline. On the left, the 'Review your pipeline YAML' section displays the generated YAML code:

```

azure-pipelines.yml
1 # Starter pipeline
2 # Start with a minimal pipeline that you can customize to build and deploy your code.
3 # Add steps that build, run tests, deploy, and more:
4 # https://aka.ms/yaml
5
6 trigger:
7 - master
8
9 pool:
10   vmImage: 'ubuntu-latest'
11
12 steps:
13 - script: echo Hello, world!
14   displayName: 'Run a one-line script'
15
16 - script: |
17   echo Add other tasks to build, test, and deploy your project.
18   echo See https://aka.ms/yaml
19   displayName: 'Run a multi-line script'
20

```

On the right, the 'Save and run' dialog box is open, showing the commit message 'Set up CI with Azure Pipelines' and options for committing directly to the master branch or creating a new branch for a pull request.


[Open in app](#)

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**

Azure DevOps  
nyuen / AKS-DevOps / Settings / Service connections

**AKS-DevOps** + Project Settings

General Overview Teams Security Notifications Service hooks Dashboards Boards Project configuration Team configuration GitHub connections Pipelines Agent pools Parallel jobs Settings Release retention Service connections Repos Repositories Policies

Service connections XAML build services + New service connection ▾

- Azure Classic
- Azure Repos/Team Foundation Server
- Azure Resource Manager**
- Azure Service Bus
- Bitbucket Cloud
- Chef
- Docker Host
- Docker Registry
- Generic
- Github

Service connection: nyuen

Details Roles Request history Policies

**INFORMATION**

Type: GitHub  
Created by Nicolas Yuen  
Connected to service using installation access token

**ACTIONS**

List of actions that can be performed on this service:  
Update service connection  
Disconnect

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:

[Full version to re-use an existing SPN](#)

Add an Azure Resource Manager service connection

Service Principal Authentication  Managed identity Authentication

Connection name

Environment: AzureCloud

Scope level: Subscription

Subscription ID:

Add an Azure Resource Manager service connection

Service Principal Authentication  Managed Identity Authentication

Connection name

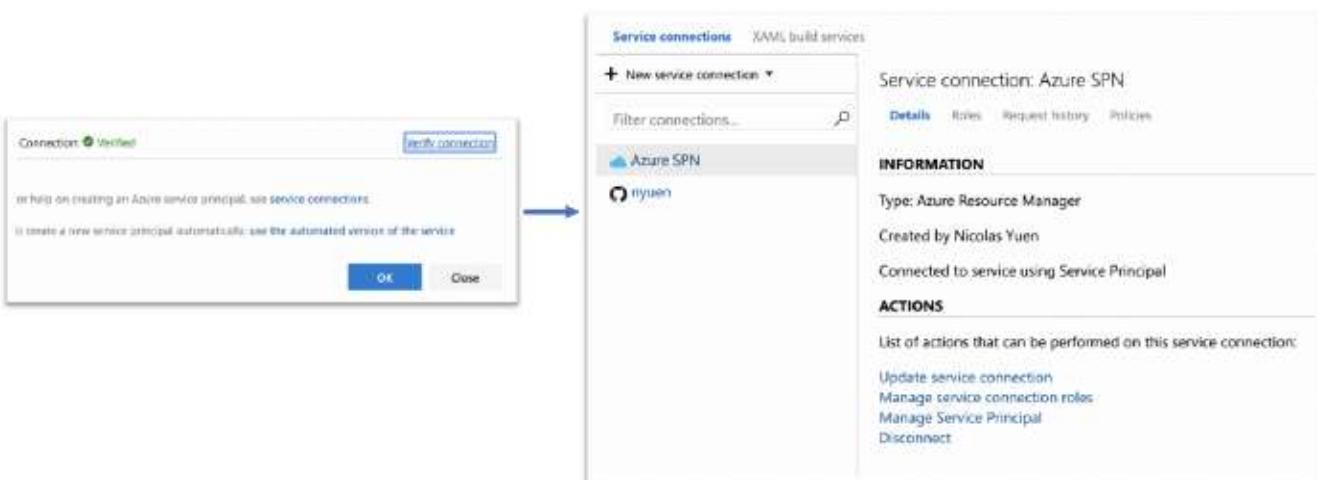
Environment: AzureCloud

Scope level: Subscription

Subscription ID:

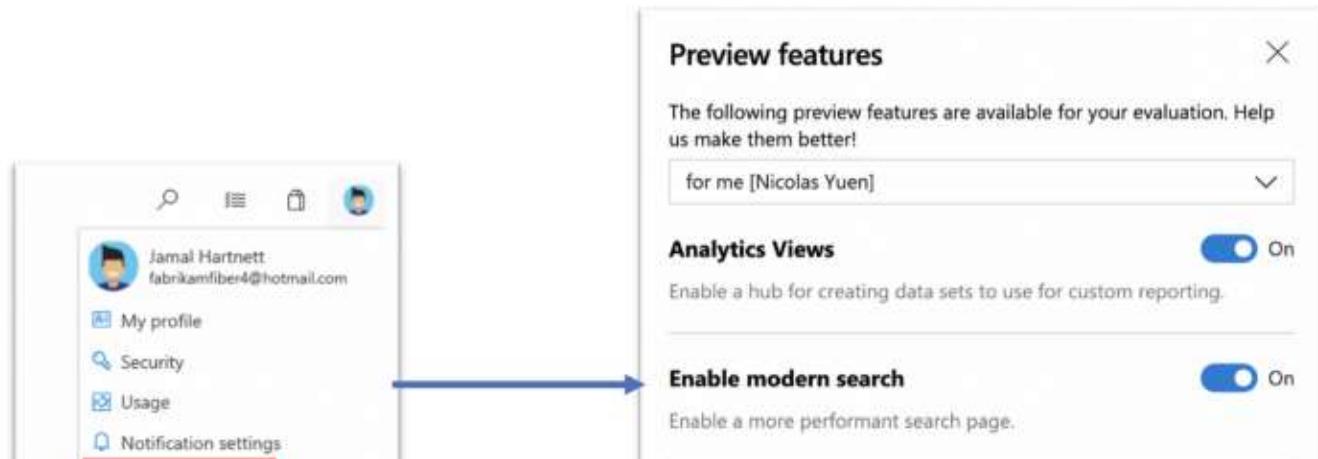

[Open in app](#)


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:




[Open in app](#)

### Multi-stage pipelines

On

Enables new multi-stage experiences for pipelines. [Learn more](#)

We had to go 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 :

```

1-terraform > aks_conf.tfvars
1  azure_region = "EastUS"
2  kubernetes_version = "1.14.5"
3  resource_group = "AKSCluster-RG"
4  acr_name = REDACTED
5  keyvault_rg = REDACTED
6  keyvault_name = REDACTED
7

```

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>"
```


[Open in app](#)

The screenshot shows the Azure DevOps interface for the 'AKS-DevOps' project. On the left, a sidebar menu is open, with 'Library' highlighted and surrounded by a red box. The main area is titled 'Library' and contains a 'Secure files' tab, also highlighted with a red box. A file named 'aks\_conf.tfvars' is listed under the 'Secure files' tab. A red arrow points from the 'aks\_conf.tfvars' file name towards the 'Secure files' tab.

## 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



[Open in app](#)

This post will be 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` that (a collection of steps to be run sequentially against the environment). Using deployment job provides some benefits:


[Open in app](#)

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

The deployment stage is also located in the YAML file:

```

- 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: 'nyuenteraformstate'
                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)


[Open in app](#)

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)

#20190820.2 checking resource group variable and instead referencing the RG object to avoid dependency issues  
on nyuen/AKS-Articles

[Summary](#) [Environments](#) [Aqua Scanner Report](#) [WhiteSource Bolt Build Report](#)

Triggered by Nicolas Yuen  
nyuen/AKS-Articles master ee2a900 Duration: 2m 6s Tests: 1 commit Work items: - Artifacts: 1 published  
Today at 14:46

[Stages](#) [Jobs](#)

Validate 1 job completed 1m 5s Deploy 1 job completed 41s

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

← Jobs in run #20190820.2  
nyuen/AKS-Articles

Deploy\_Terraform 38s

**Job Information**

- Post: Azure Pipelines
- Image: Ubuntu16
- Agent: Hosted Agent
- Started: Today at 14:49
- Duration: 38s

Initialize job

Download Artifact


[Open in app](#)

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

NAME	TYPE	LOCATION
AKSTerraform	Kubernetes service	East US
aksterrafraomcdemo2	Container registry	East US
aksnet	Virtual network	East US

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

NAME	TYPE
aks-agentpool-38795589-msg	Network security group
aks-pool1-38795589-vmss	Virtual machine scale set


[Open in app](#)

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

Device	Type	IP Address	Subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.4	aks_subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.5	aks_subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.6	aks_subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.7	aks_subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.8	aks_subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.9	aks_subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.10	aks_subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.11	aks_subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.12	aks_subnet
aks-pool1-38795589-vmss (instance 0)	Scale set instance	10.1.0.13	aks_subnet

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

NAME	LAST MODIFIED	CREATION TIME	TYPE	SIZE	ACCESS TIER	ACCESS TIER LAST MODIFIED	SERVER ENCRYPTED	ETAG	CONTENT-TYPE	CONTENT-MD5	LEASE STATUS	LEASE STATE	LEASE DURATION	COPY STATUS	COPY COMPLETION TIME
demo.terraform.tfstate	8/20/2019, 249:06 PM	8/15/2019, 9:12:25 PM	Block blob	128.06 KB	N/A	N/A	true	0x807256CC9986A50	application/json	-	Unlocked	Available	-	-	-

## Conclusion

[Open in app](#)

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

[About](#) [Help](#) [Legal](#)

Get the Medium app

