Unbreakable Azure Devops Pipeline using Dynatrace

Safia Habib

Pre Requisites

- Download the Source code from here.
- Create Azure Devops account from https://aka.ms/SignupAzureDevOps
- Create Azure Portal account from https://azure.microsoft.com/free/
- A Dynatrace account, get a SaaS free trial here: https://www.dynatrace.com/trial
- Create a cheat sheet in a text editor (name : Azure_Devops_values) with the following:
  - DT_TENANT:
  - DT_TENANT_TOKEN
  - DT_API_TOKEN
  - URL to Monspec File:
  - URL to PipelineInfo File
  - PAT:
  - URL to UnbreakablePipelineGate function:
  - Token for UnbreakablePipelineGate function:
  - URL to SelfHealing function:

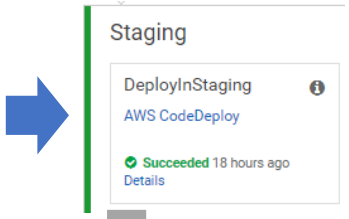
THE BIG PICTURE

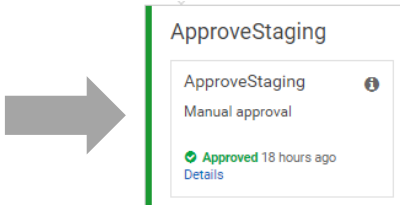
What we are going to do today!

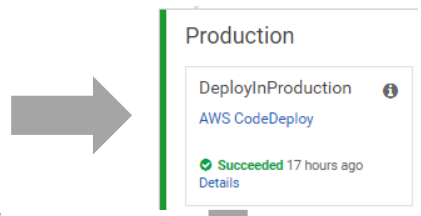
AWS CodePipeline

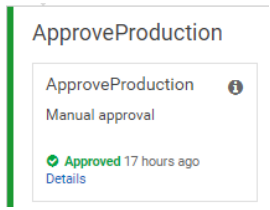
https://github.com/Dynatrace/AWSDevOpsTutorial

AppTeam





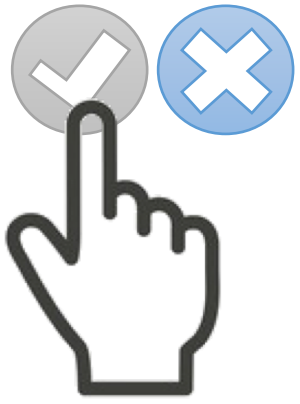




Users

AWS CodeDeploy

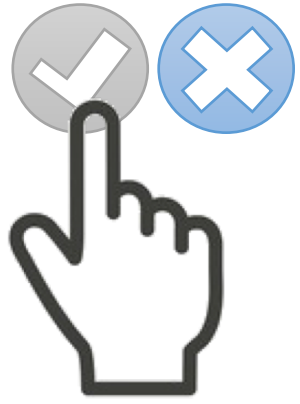
AWS EC2 (Tag=Staging)



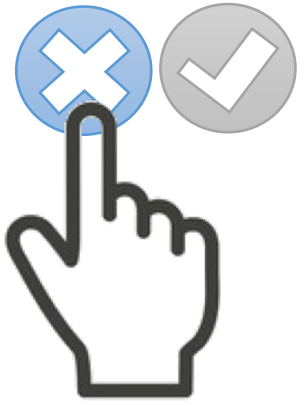
Manual Approval

AWS CodeDeploy

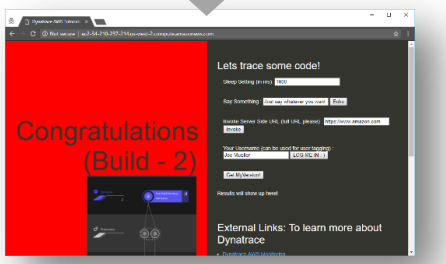
AWS EC2 (Tag=Production)

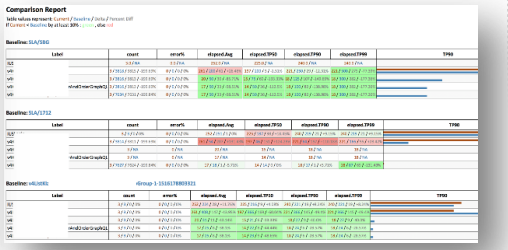


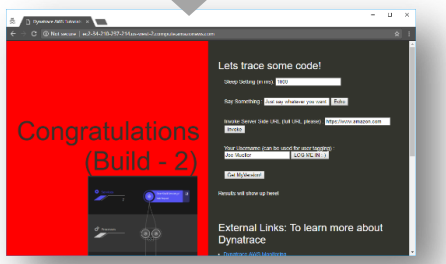
Manual Approval

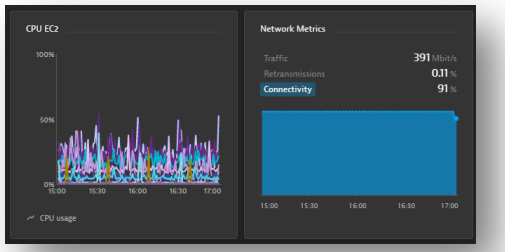


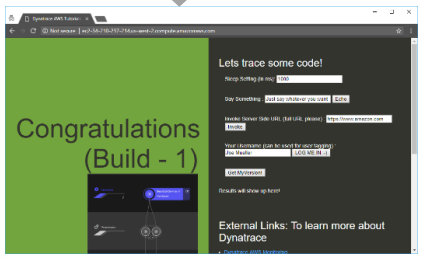
Manual Rollback



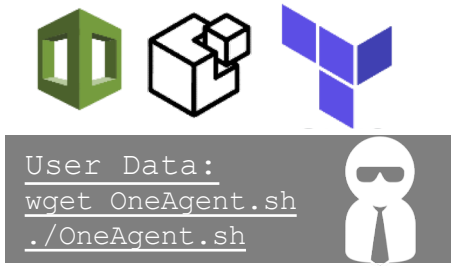




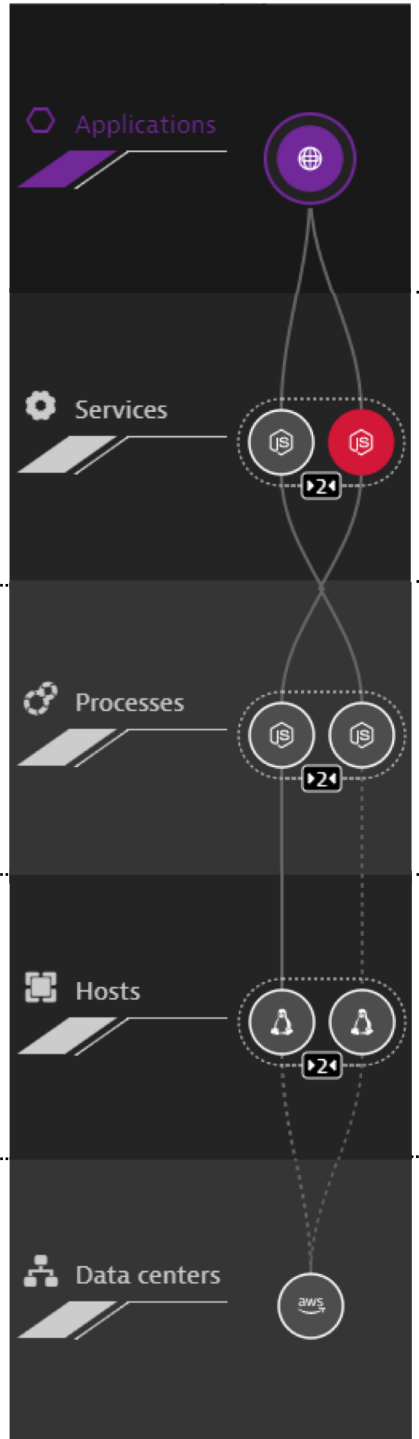


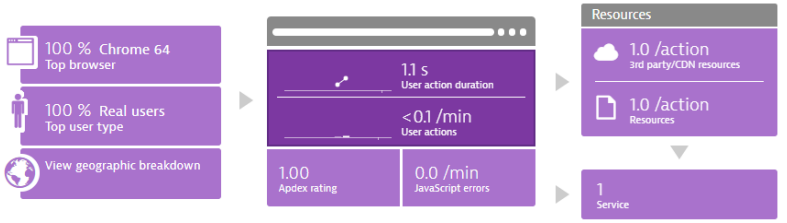


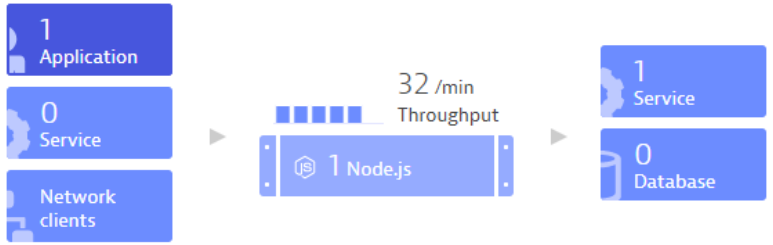
Add new rule

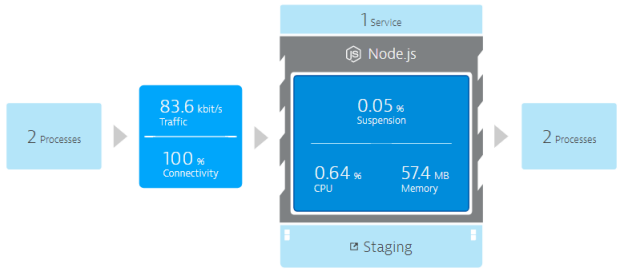


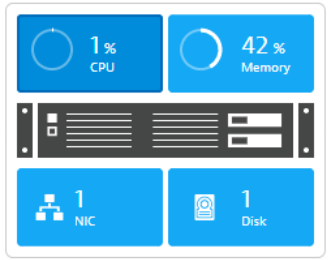
CloudWatch

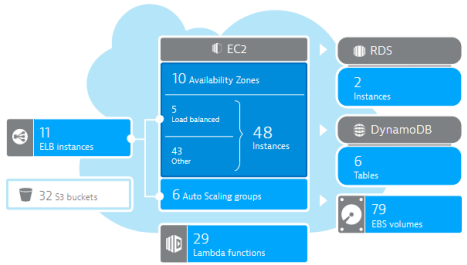


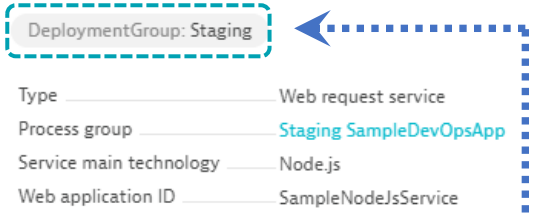


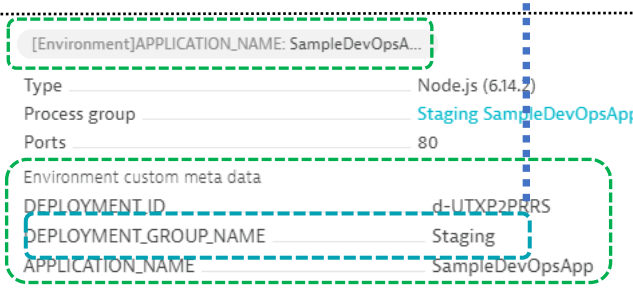


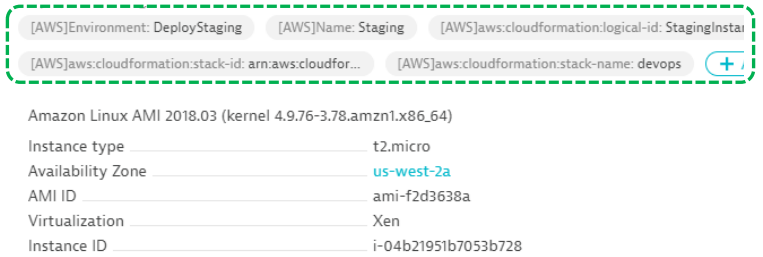






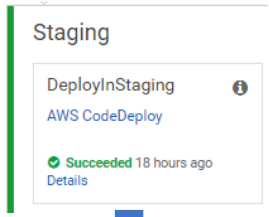


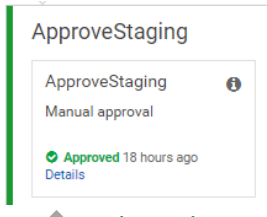




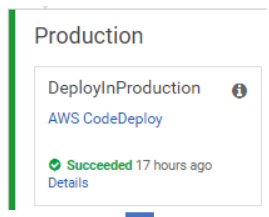
AWS CodePipeline + dynatrace

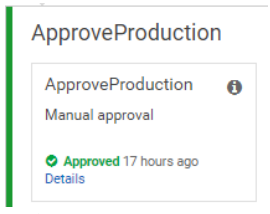
https://github.com/Dynatrace/AWSDevOpsTutorial



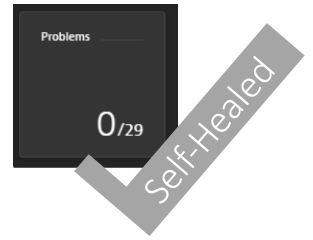


Auto-Approve!
Auto-Reject!



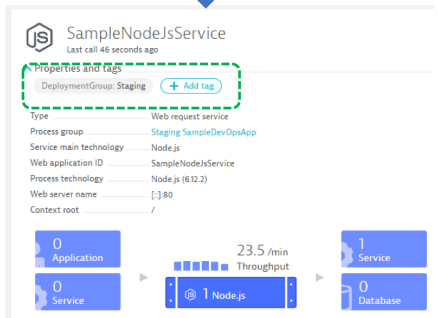


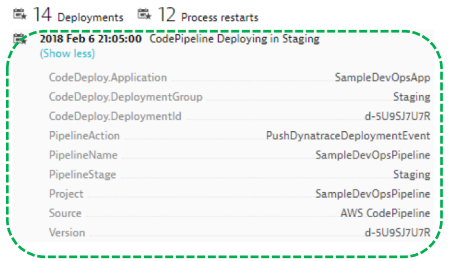
Auto-Approve!
Auto-Reject!



pushDynatraceDeploymentEvent

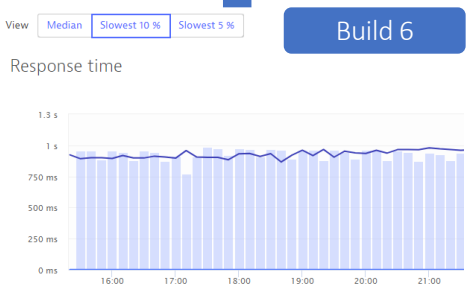
Pushes Deployment Info to Dynatrace Entities

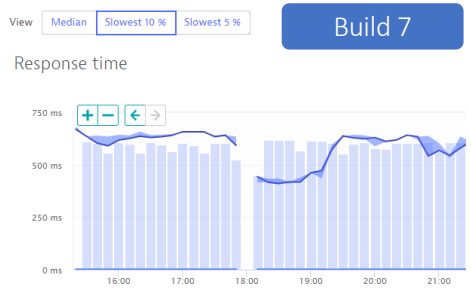




validateBuildDynatraceWorker

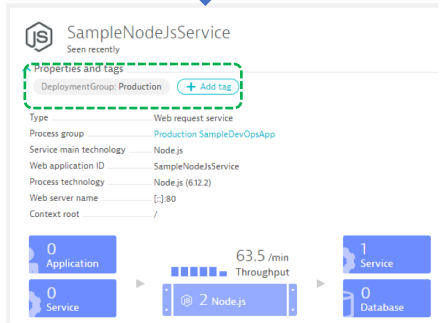
Compares Builds and Approves/Rejects Pipeline

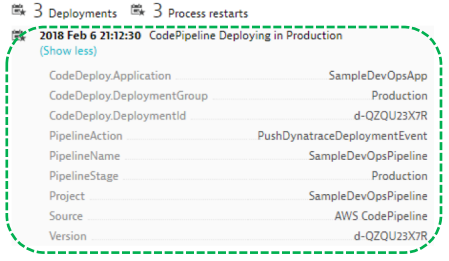




pushDynatraceDeploymentEvent

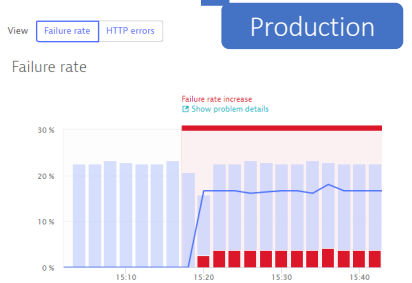
Pushes Deployment Info to Dynatrace Entities

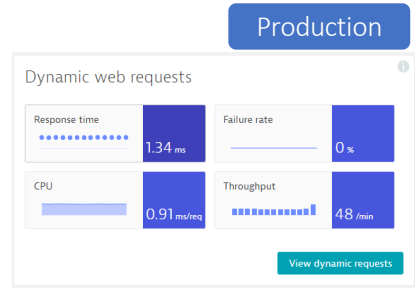


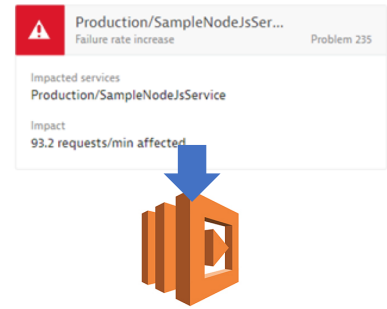


validateBuildDynatraceWorker

Validates Production and Approves/Rejects Pipeline

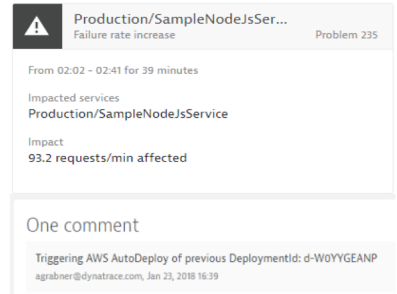






handleDynatraceProblemNotification

Executes **Auto-Remediating** Actions, e.g: Rollback



For Azure and Azure DevOps

- Azure Devops provides multiple development collaboration tools. We will be using:
  - Azure Repos: Source control repository for the test application
  - Azure Pipelines: Creating CI/CD pipelines for deploying the test application
  - Azure Devops Dynatrace Extension: Deployed in Azure Marketplace.
    - Push Deployment to Dynatrace Task
    - Release Gate
  - Azure Functions :
    - Self Healing Function: This function will redeploy a good build in the event that the bad build has made it to production upon receiving a Performance issue from Dynatrace.
    - Unbreakable Release gate Function – This is called post deployment from Staging to validate the build
  - Azure Storage account: Used for storing the monspec file and the pipelineinfo files.
  - Azure Proxy: Dynatrace Proxy: Utility with Dynatrace cli installed for comparing the Performance information with the monspec file.

More on the Azure DevOps Dynatrace extension

- The Dynatrace Extension consists of two components:

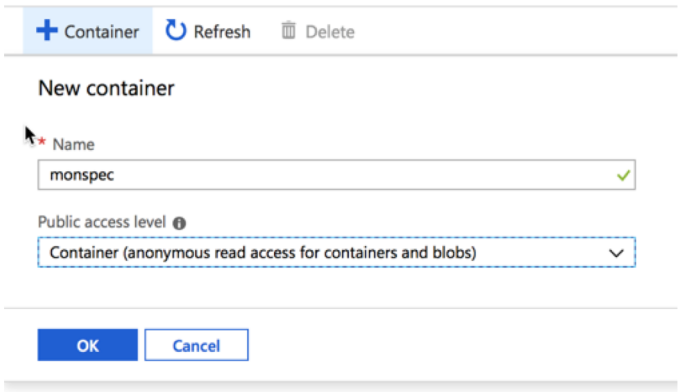
  - Push Deployment Event Task:
    - The task pushes a Deployment event to Dynatrace when a new version of the application is deployed.
    - It uses the Dynatrace events API to publish the details of the Deployment.
    - The event requires URL to the Dynatrace Tenant, API token, Tag Name and Context (Note the Tag needs to be available in Dynatrace for the event to show up).

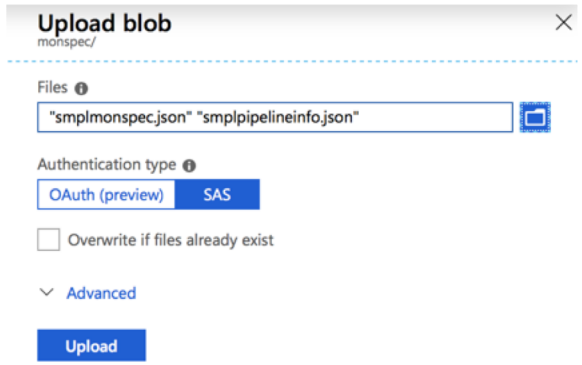
  - Unbreakable Release Gate:
    - The quality gate is executed to test the quality of the current build in Staging against a Monspec file.
      - In the case that there are no violations the gate succeeds and pushes the code to Production.
      - In case there are reported Violations then the Release Pipeline fails and the code is not deployed to Production.

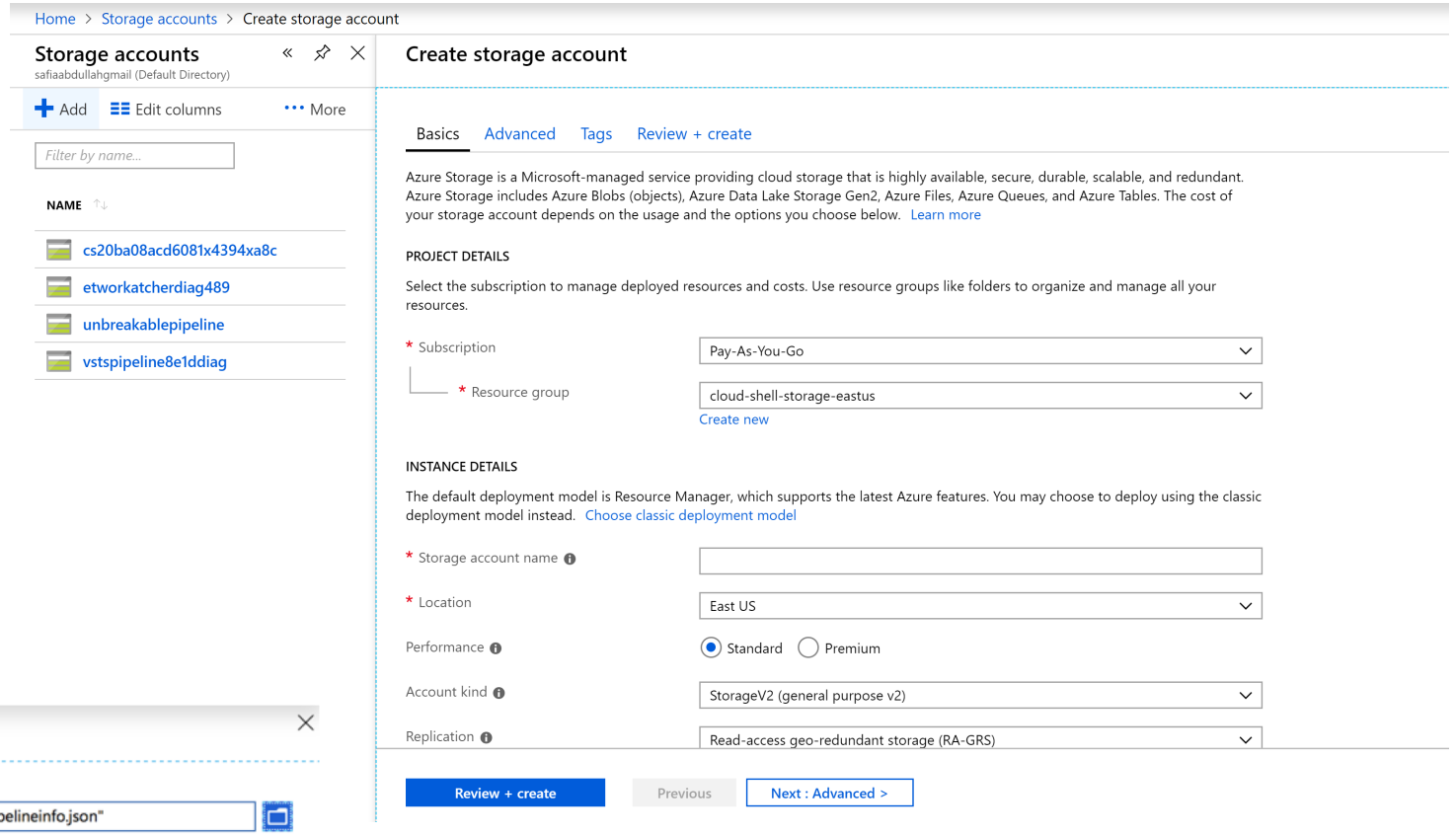
Step by Step

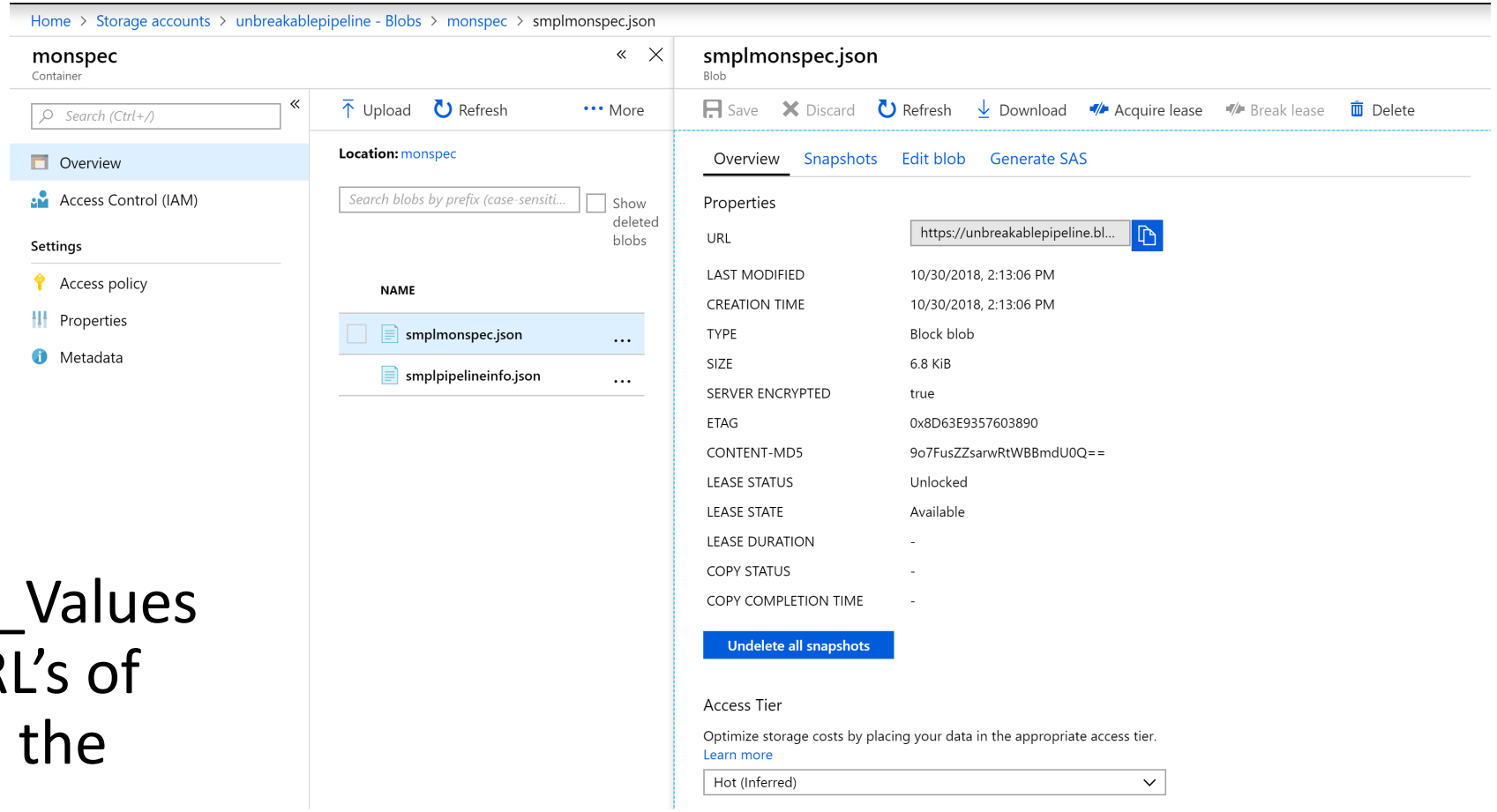
Step 1 : Storage Account

- Login to Azure Portal
- Create Storage Account
- Upload the Monspec folder from the source code to a new Blob container





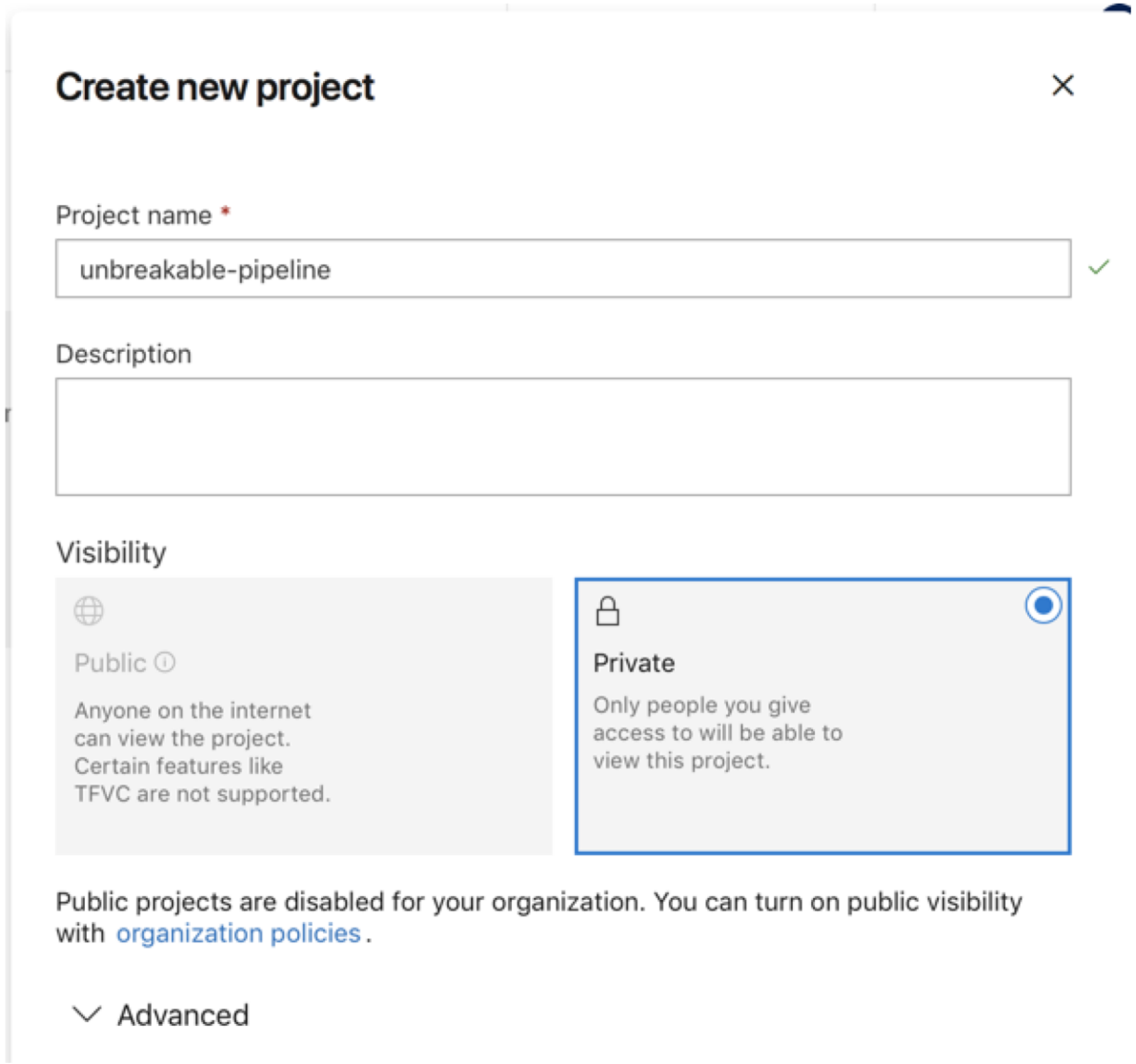




In the Azure_Devops_Values Notepad copy the URL's of the Monspec file and the Pipeline Info File

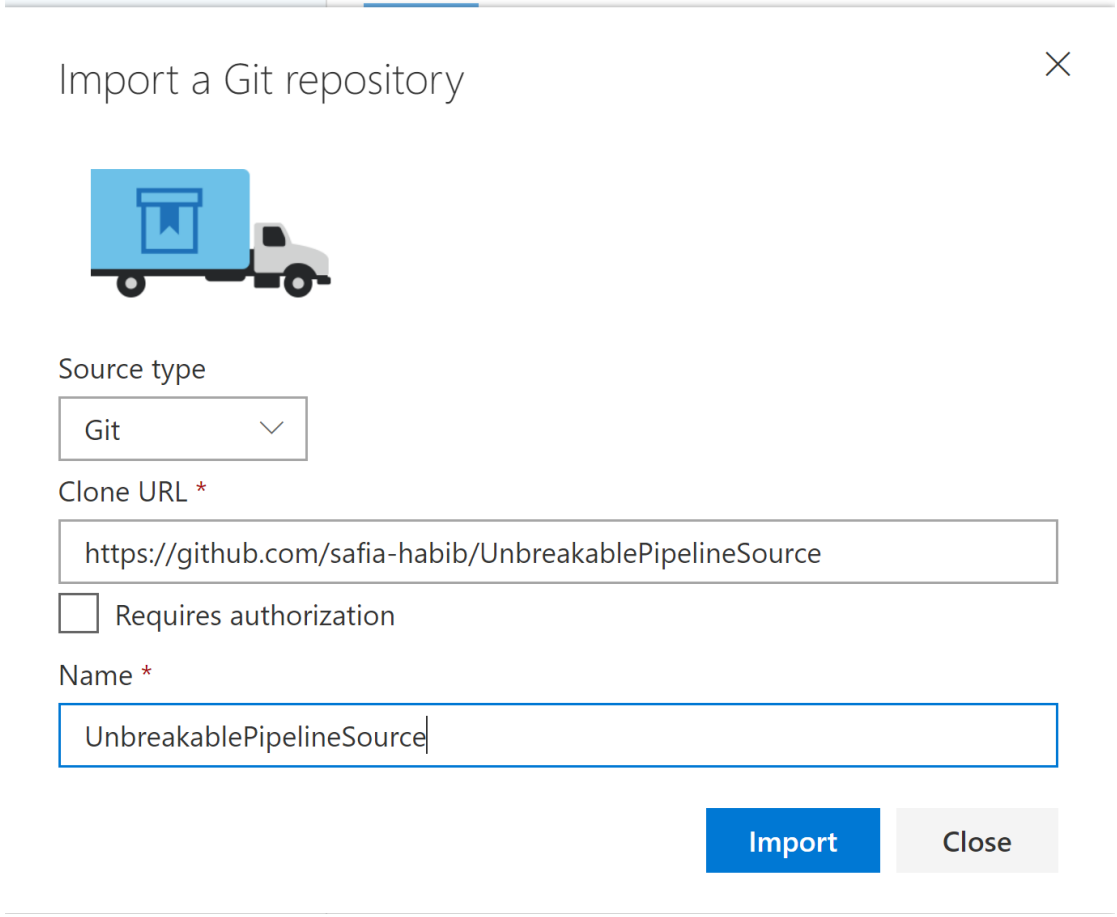
Step 2: Create Azure DevOps Organization and Project

- Login to https://dev.azure.com
- Create a New Organization
- Create a New Project

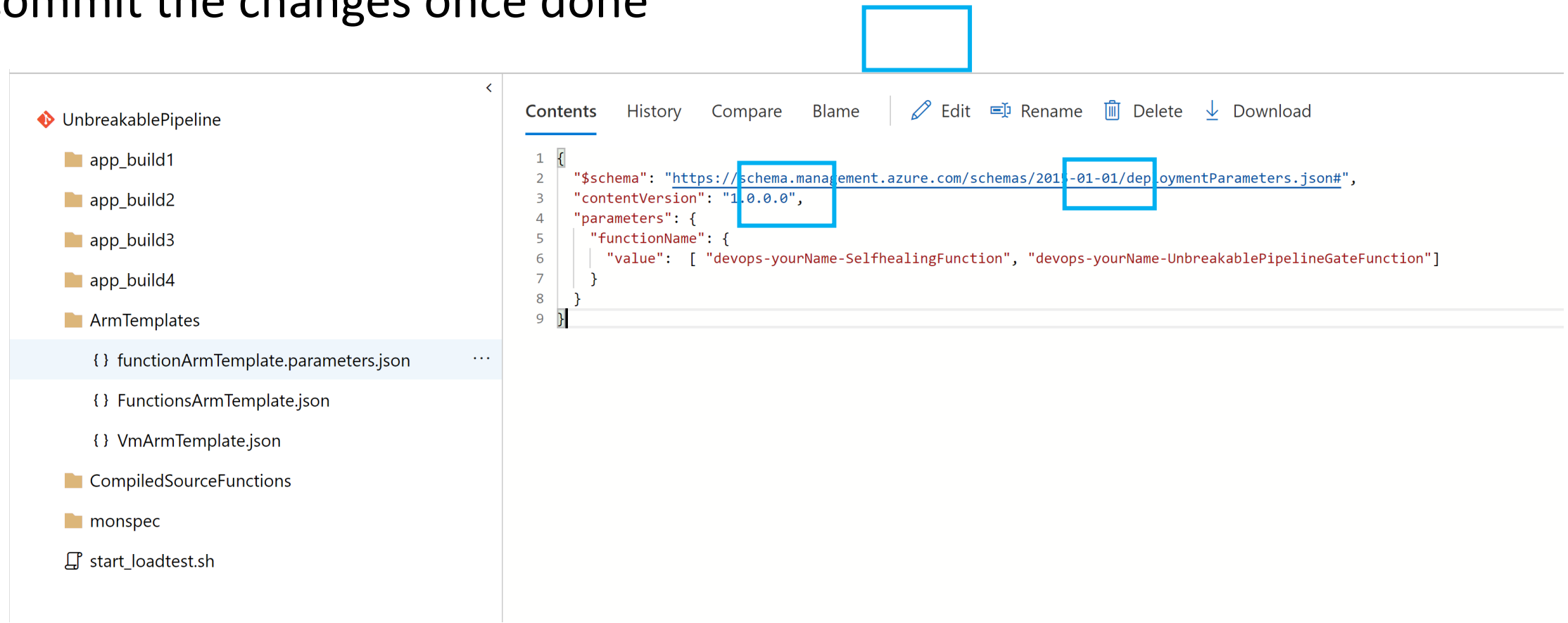


Step 3: Setup Azure Repos

- In your project, go to Repos
- Import a Git Repository
- URL to clone is: https://github.com/safia-habib/UnbreakablePipelineSource

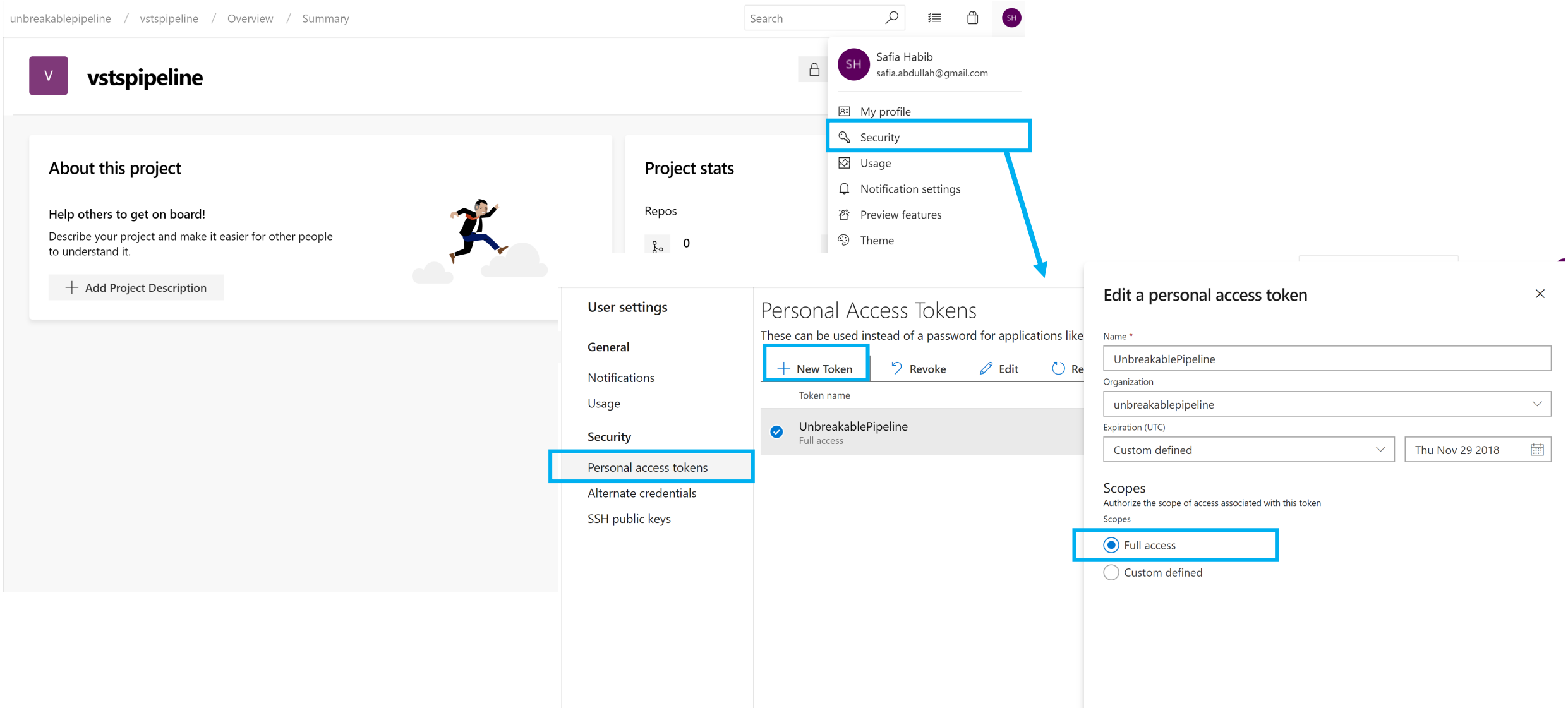


- In the imported code navigate to ArmTemplates/functionarmTemplate.parameters.json
- Replace <your Name> With your First + Last name, for example : stevecaron
- Commit the changes once done



Step 4: Create a Personal Access Token

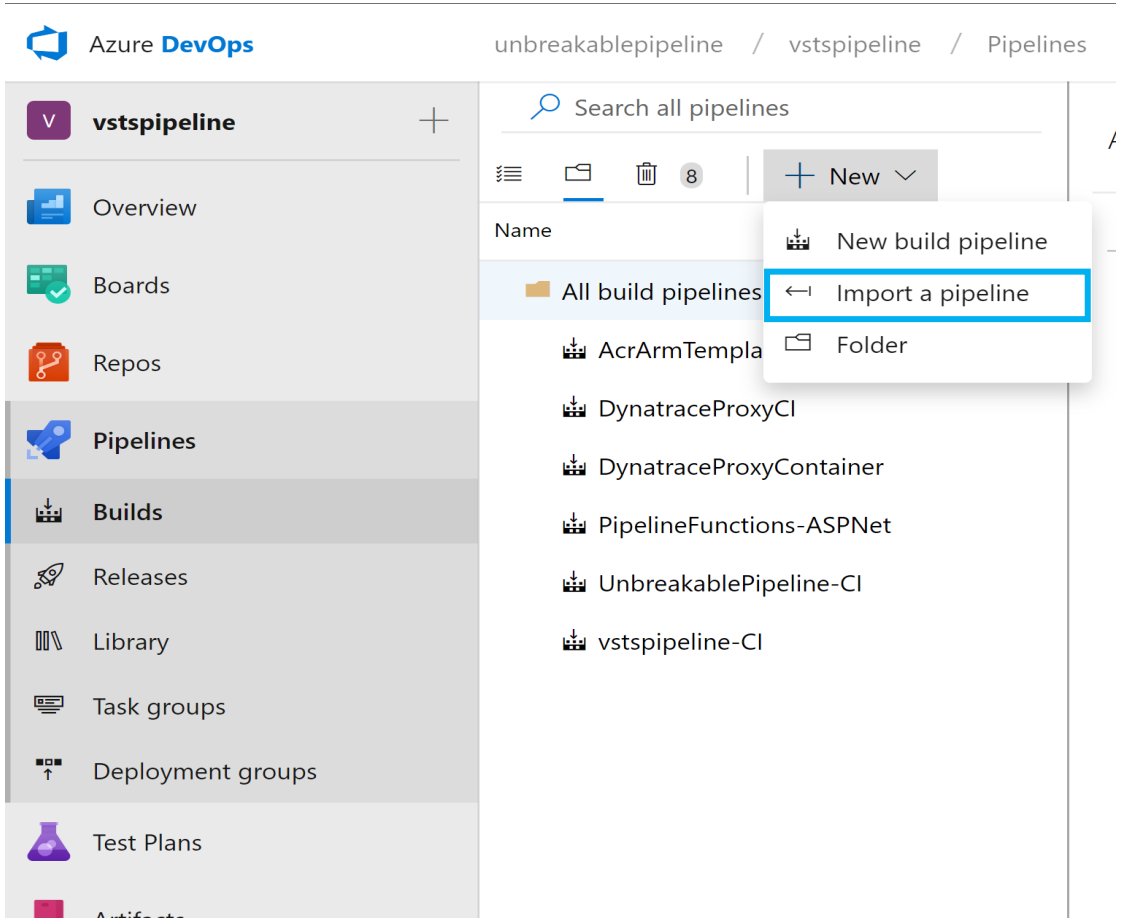
- Create a PAT and Copy Value to the Notepad



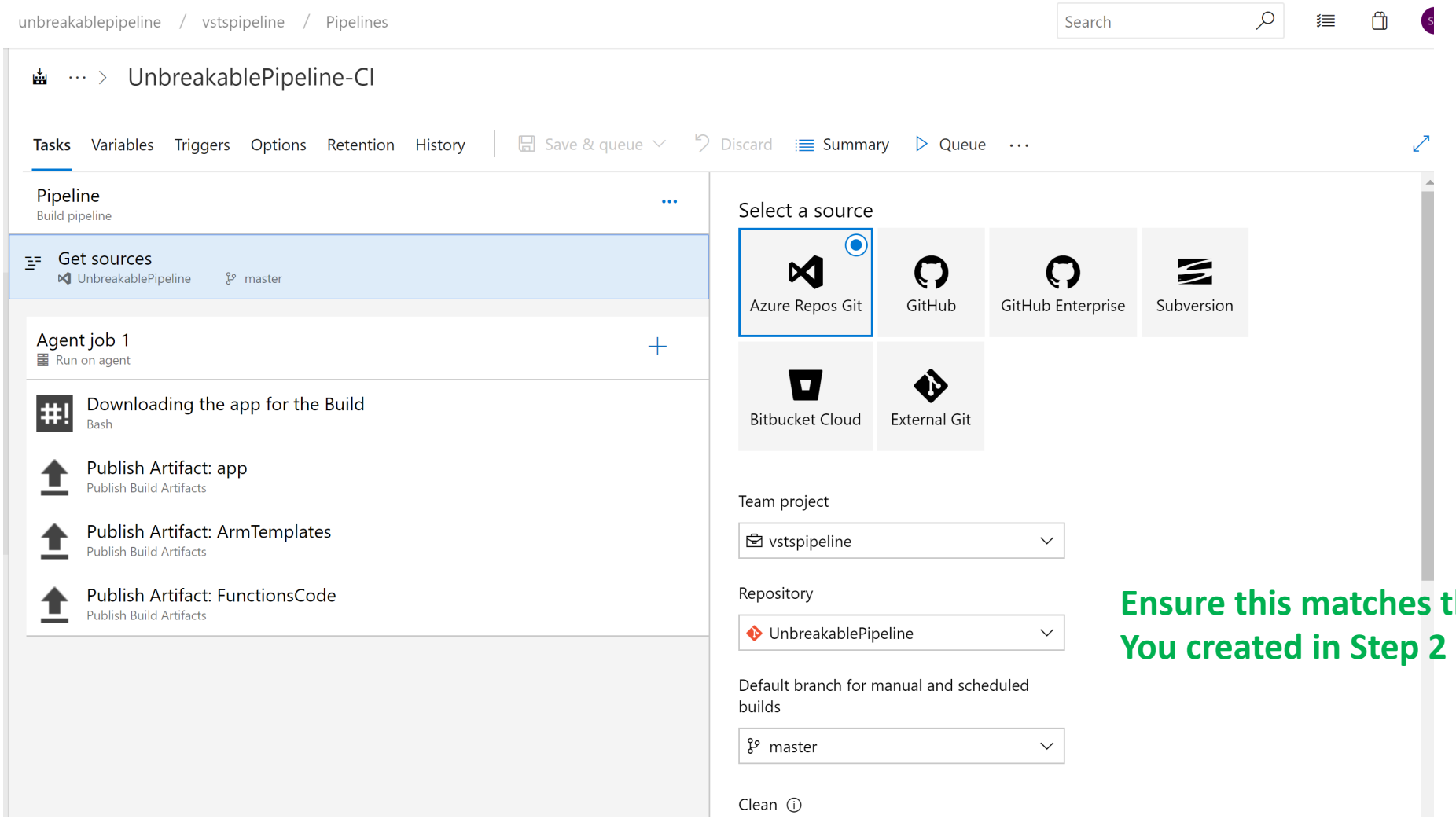
Step 5: Build Pipeline

We have noticed that for new users there is sometimes no option to Import pipeline
In this case create a Dummy Pipeline With any name
To get to the below view and then import the Build pipeline

- Go to Pipelines->Build
- Create a New Pipeline
- Import the Unbreakablepipeline-CI.json
- Current Limitation – Name of the Pipeline should match Unbreakablepipeline-CI so you might have to change the name assigned at the import (remove -import)



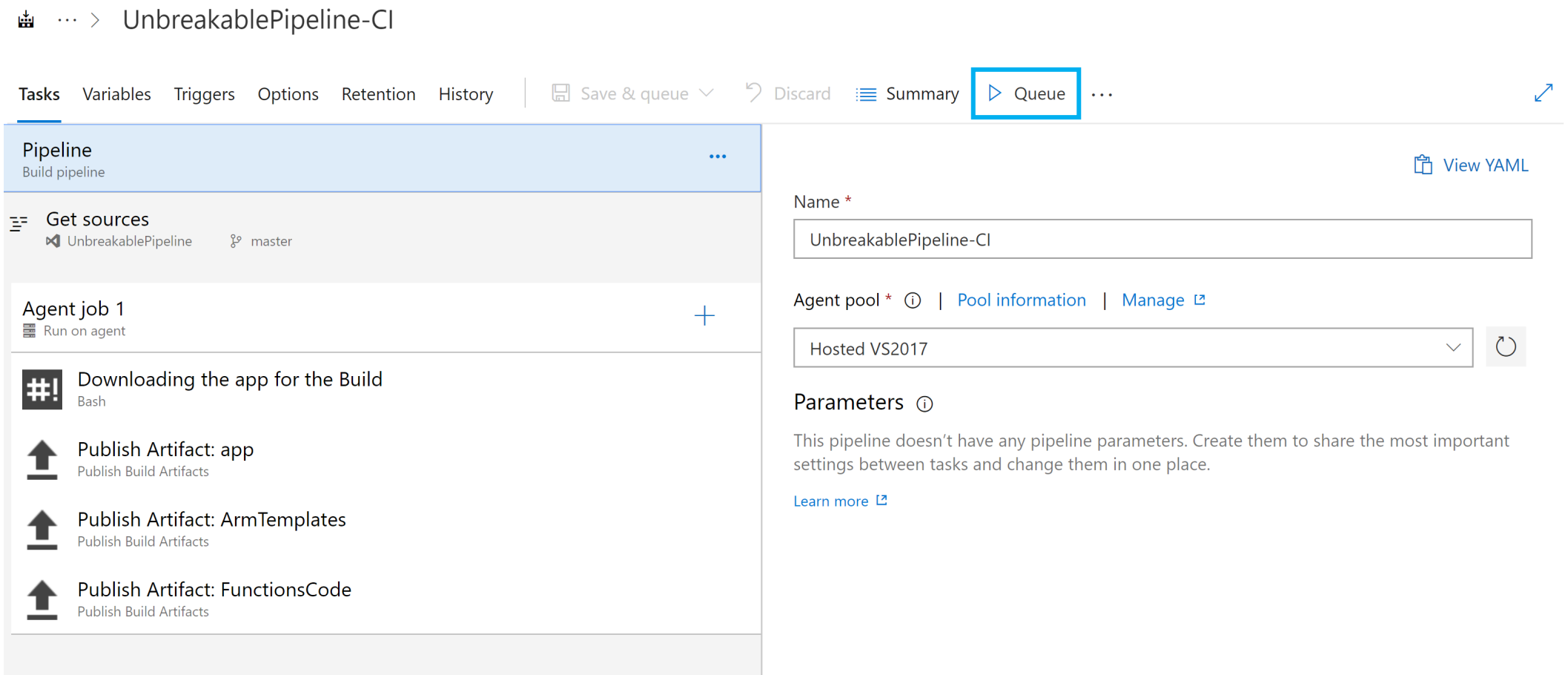
Result



Ensure this matches the Repo Name You created in Step 2

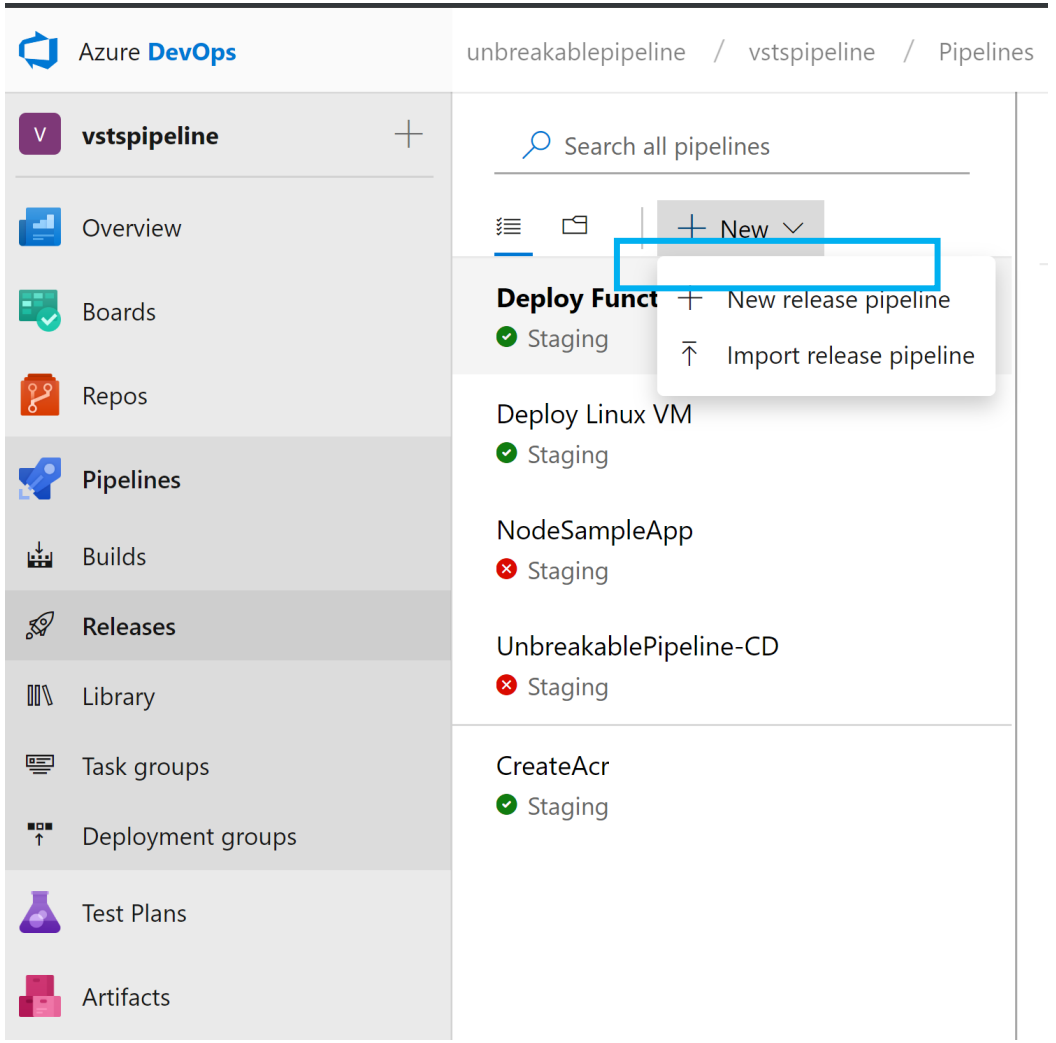
Step 6: Run the Build Pipeline

- Queue a Build Pipeline



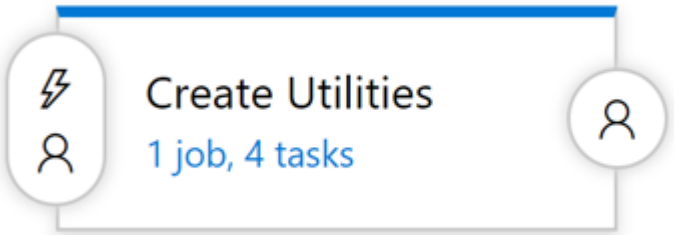
Step 7: Release Pipeline

- Go to Pipelines Release
- Import the CreateUtilities.json



We have noticed that for new users there is sometimes no option to Import pipeline
In this case create a Dummy Pipeline With any name
To get to the below view and then import the Build pipeline

Release Stage



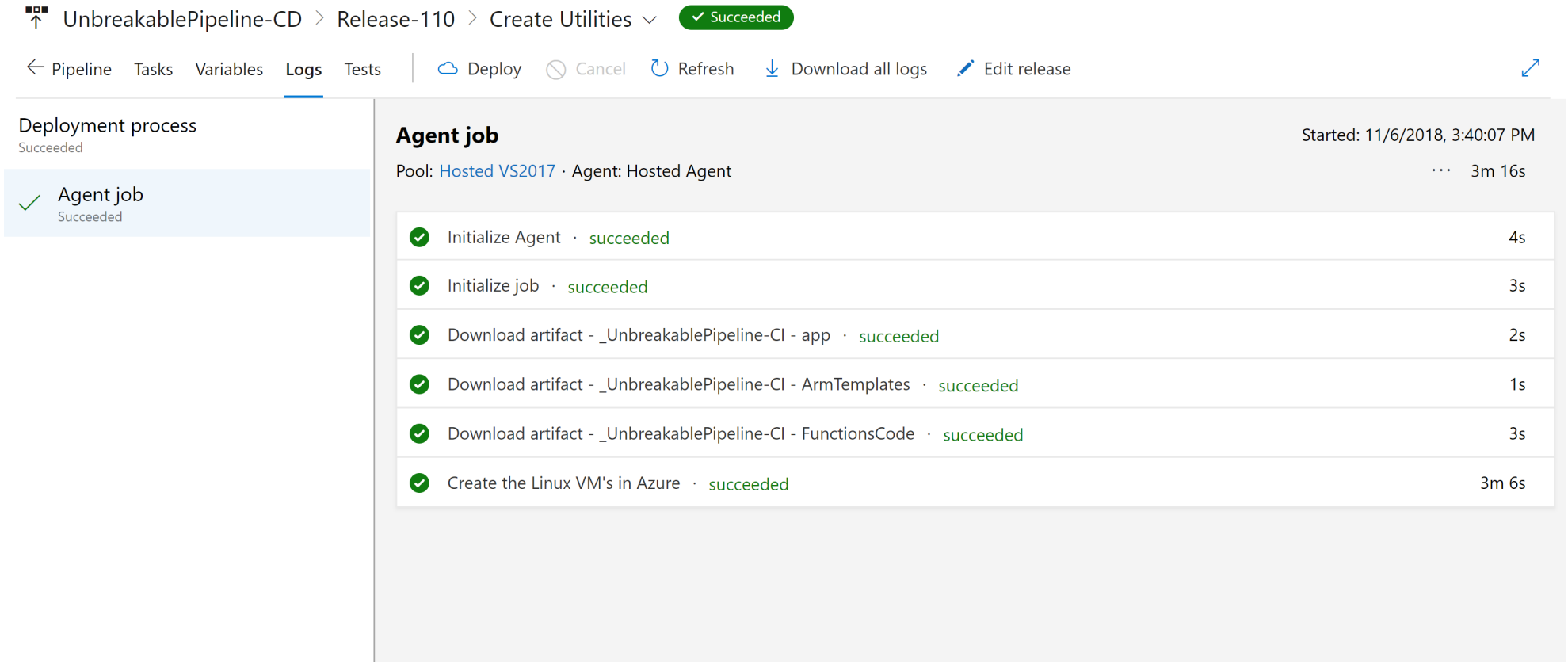
Create utilities

Creates the following in Azure:

- Self Healing Function
- Unbreakable Pipeline Gate Function
- For today we will just deploy the app to the Hosted agent / factory VM available provided by Azure Devops
- Staging Linux VM (Excluded for this Workshop)
- Production Linux VM (Excluded for this workshop)

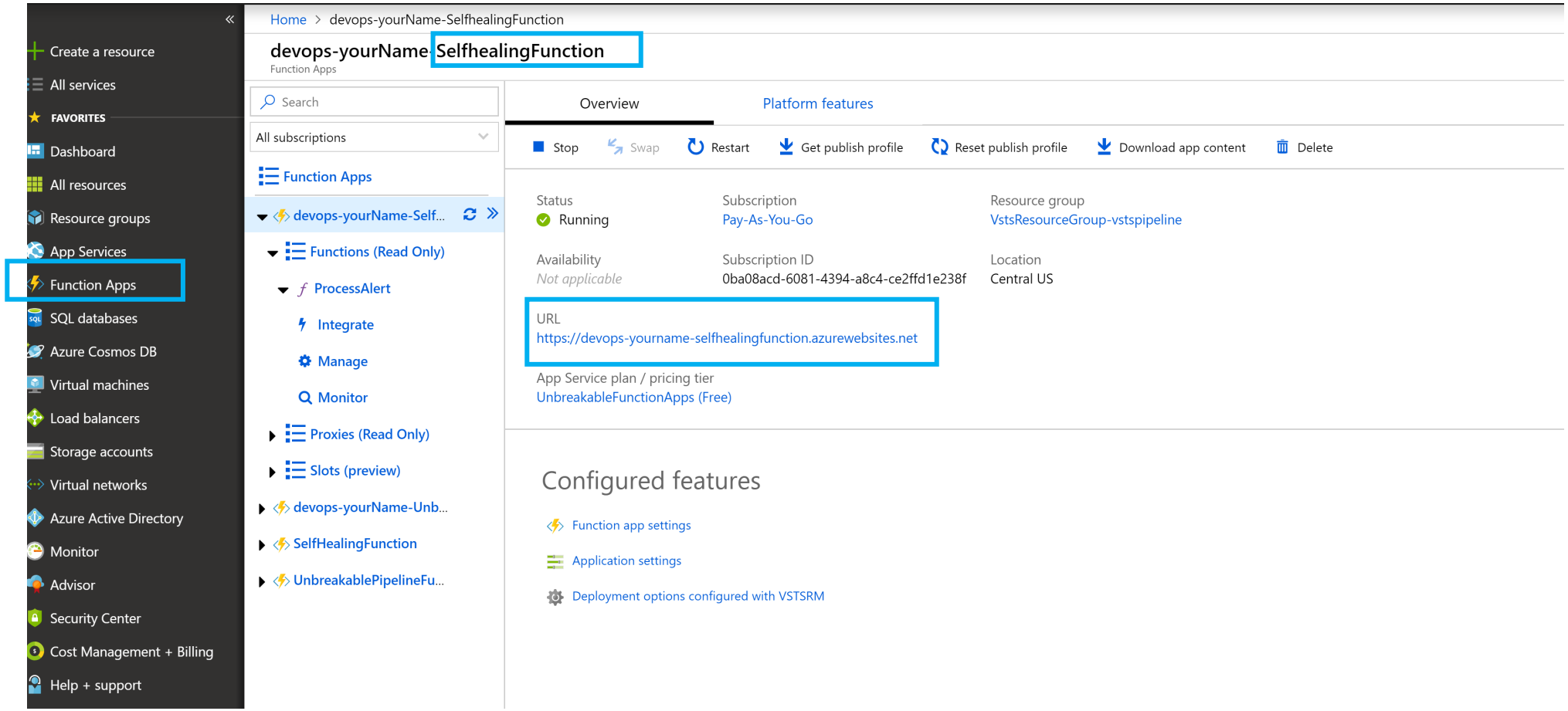
Step 8: Deploy time

- First run the Create Utilities Stage



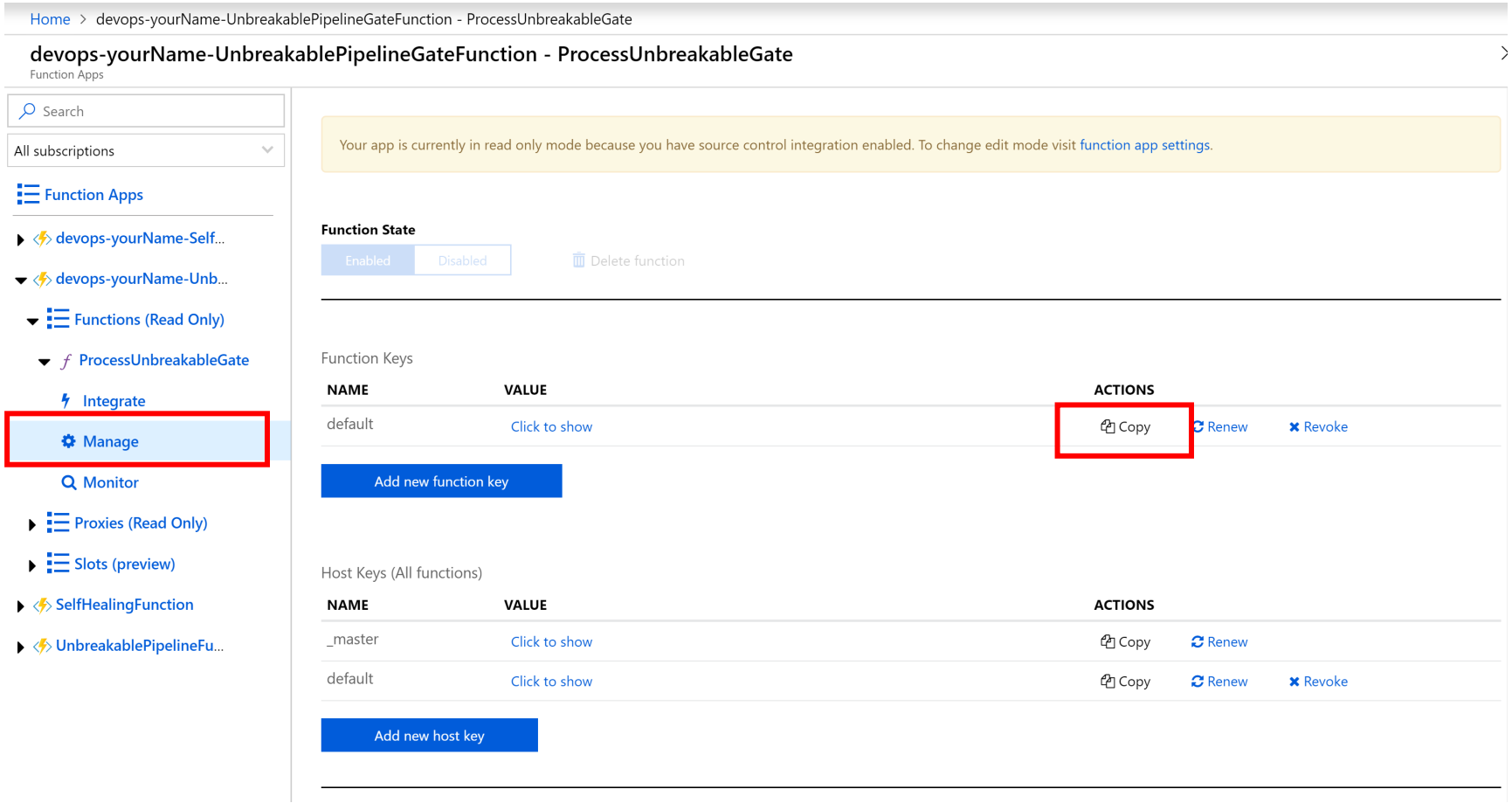
Step 9: Functions URL and Key

- Go to Azure Portal and copy the Self Healing Function URL Azure_Devops_values notepad.



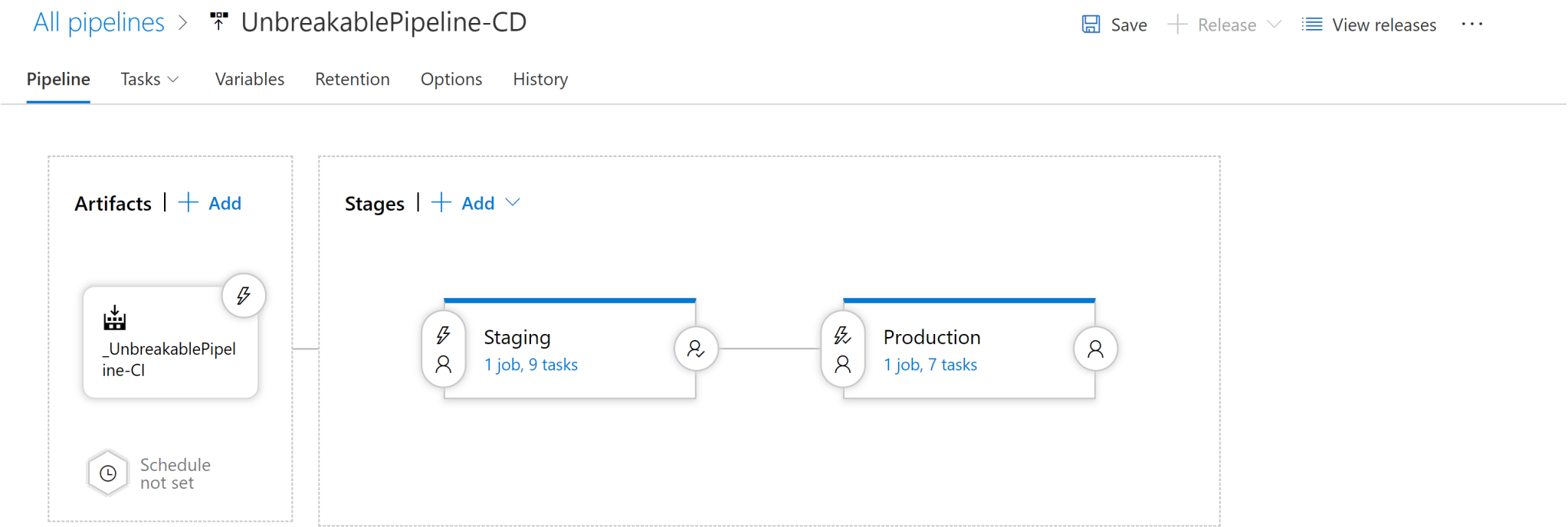
Step 9: Functions URL and Key

- Get the Unbreakablepipeline function URL and Key – Copy to Notepad



App Pipeline

- Import UnbreakablePipeline-CD.json into New Release Pipeline



App Release Pipeline

Staging

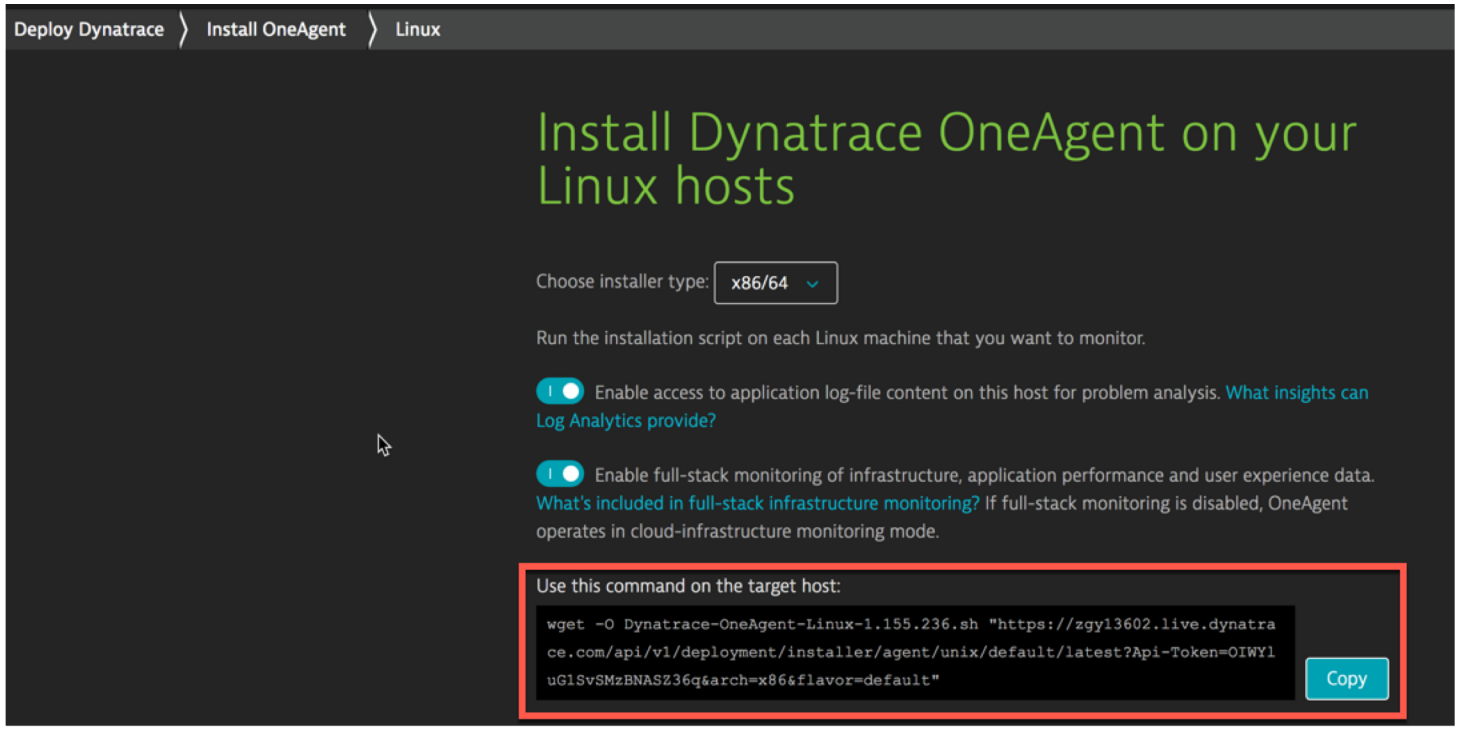
- Deploys the test app on Staging VM
- Starts Loadtest against the test application
- Pushes Deployment Event to Dynatrace

Production

- Deploys the test app on Staging VM
- Starts Loadtest against the test application
- Pushes Deployment Event to Dynatrace

Step 10 : Dynatrace Account Info

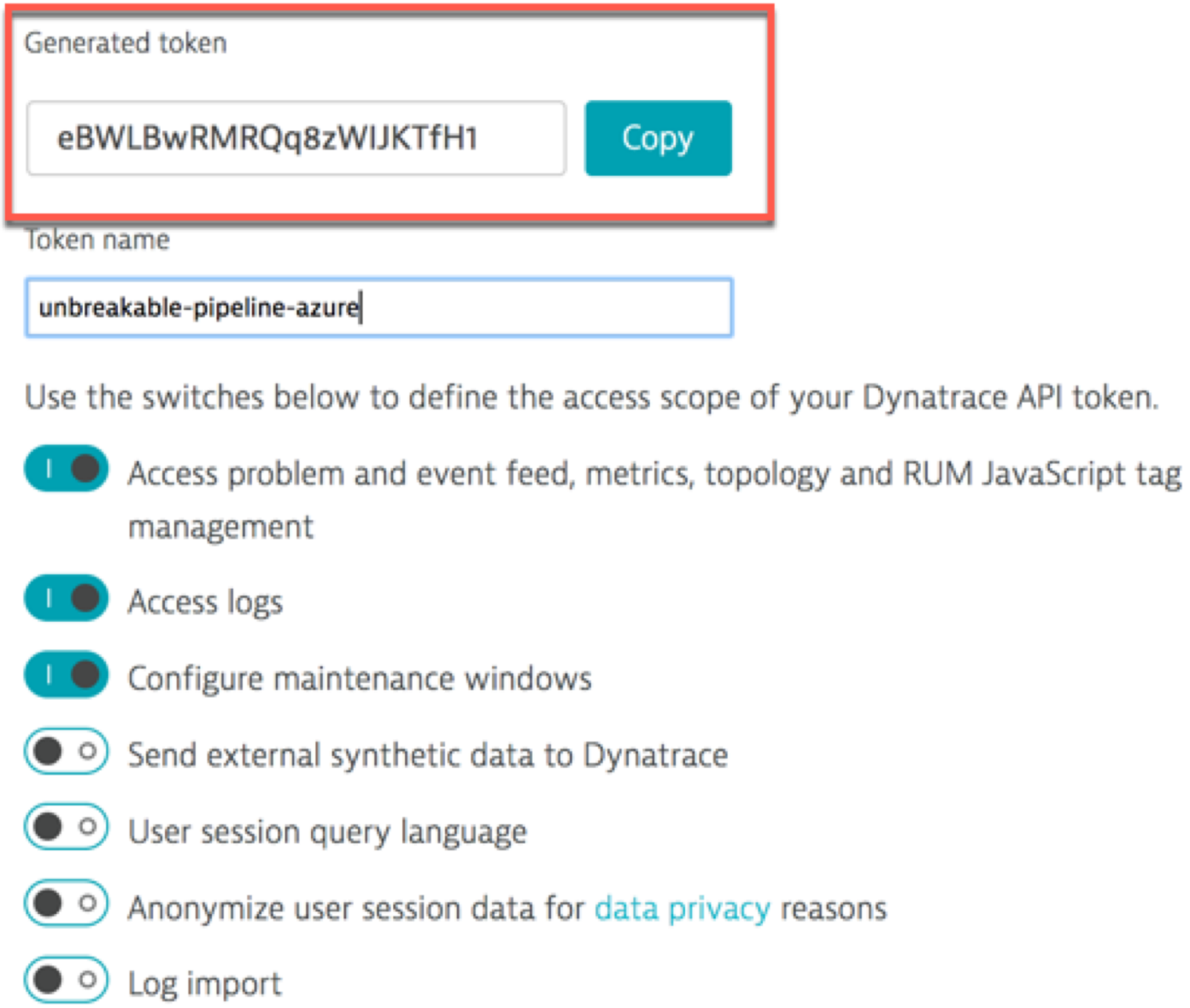
- Menu->Deploy Dynatrace->Install OneAgent->Linux
- Copy the wget command
- Copy the tenant URL and the installation token on your notepad
- DT_TENANT: https://<tenant>.live.Dynatrace.com
- DT_TENANT_TOKEN: Value of Tenant token



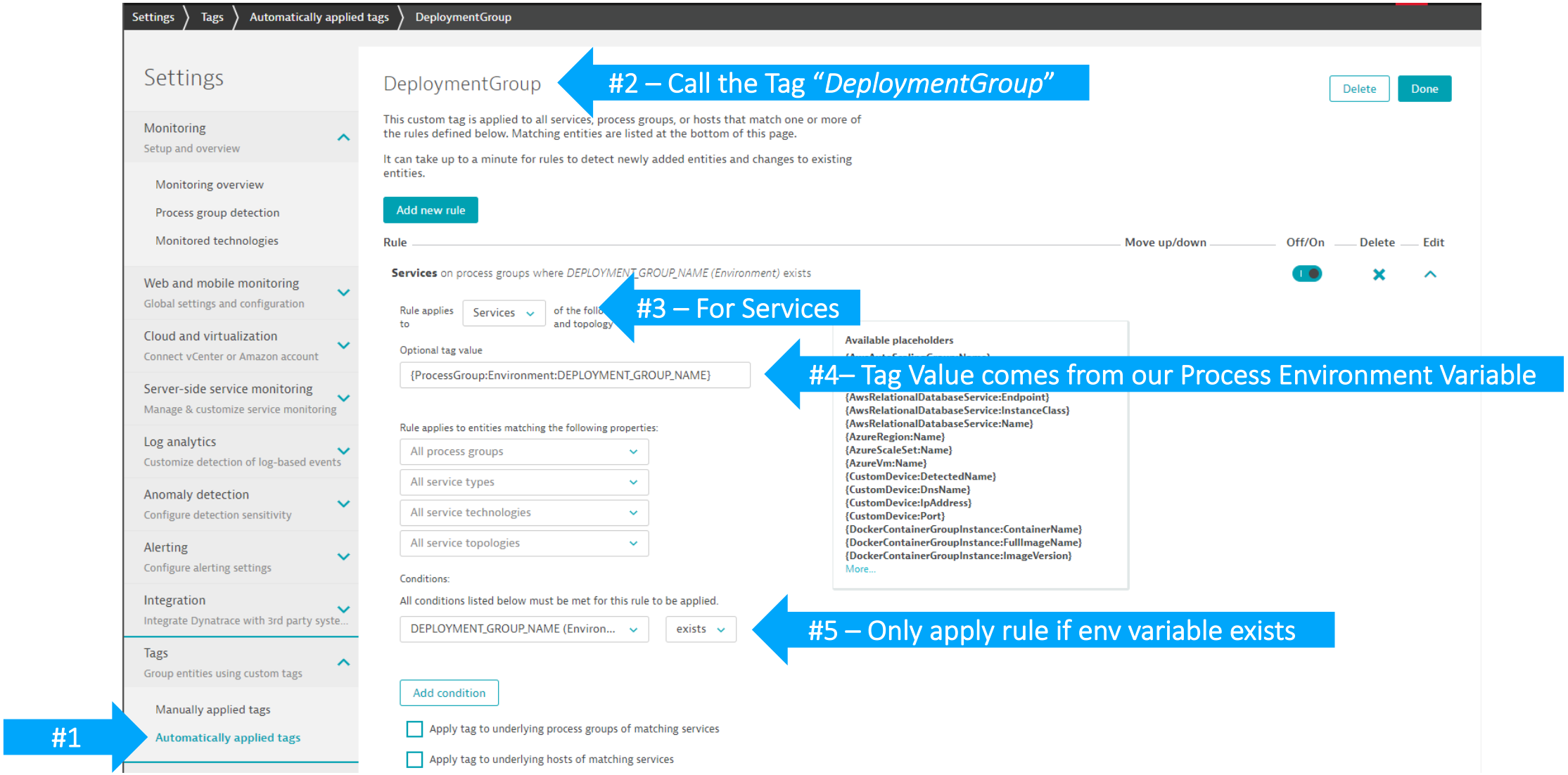

```
wget  -O Dynatrace-OneAgent-Linux-1.155.236.sh 'https://zgy13602.live.dynatrace.com/
api/v1/deployment/installer/agent/unix/default/latest?Api-
Token=OIWYluG1SvSMzBNASZ36q&arch=x86&flavor=default"
```

Step 11 : Dynatrace API Token

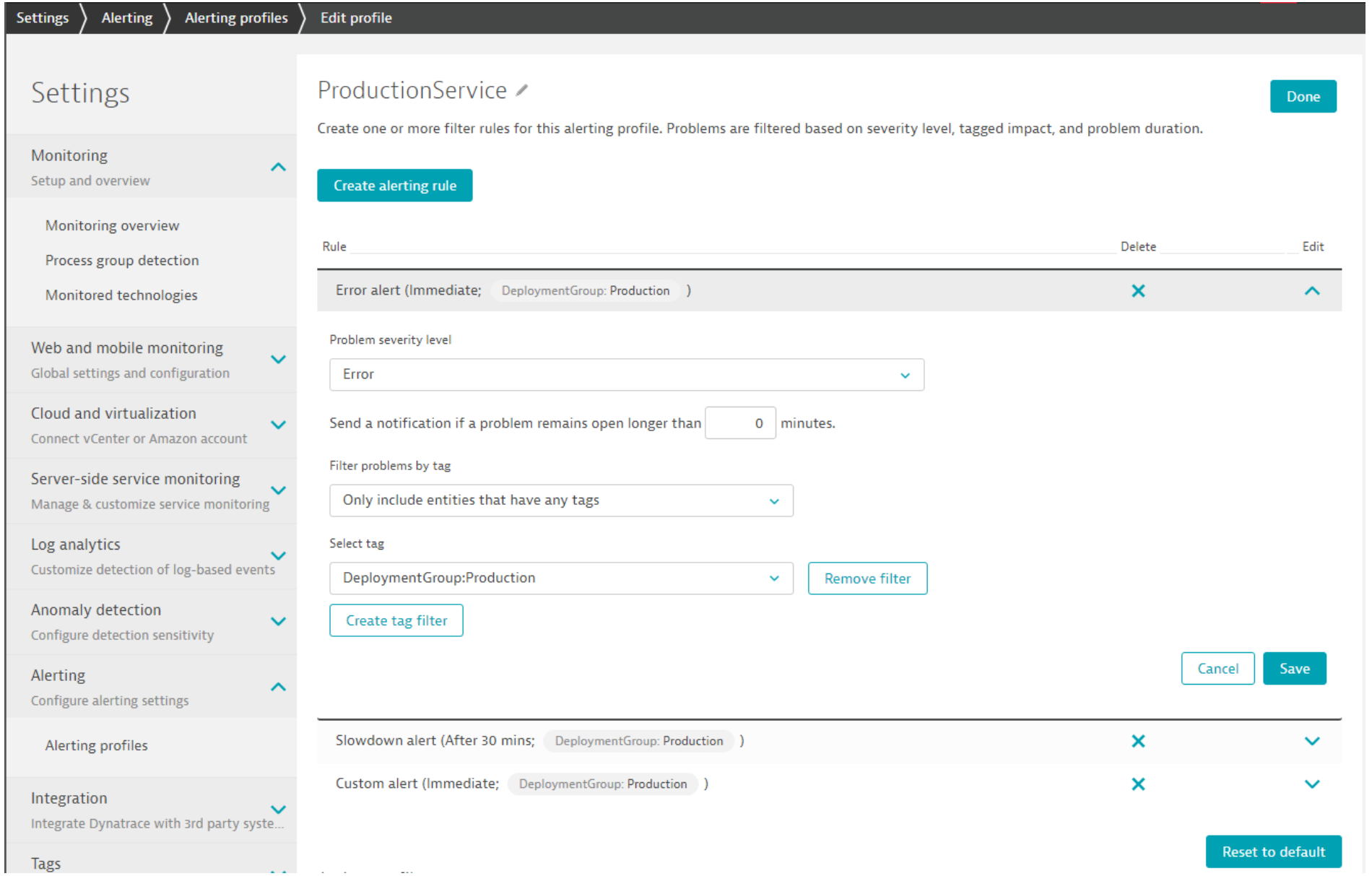
- Menu->Settings-> Integration-> Dynatrace API
- Generate Token
- Copy the token value



Step 12: Add Auto Tagging rule to Dynatrace

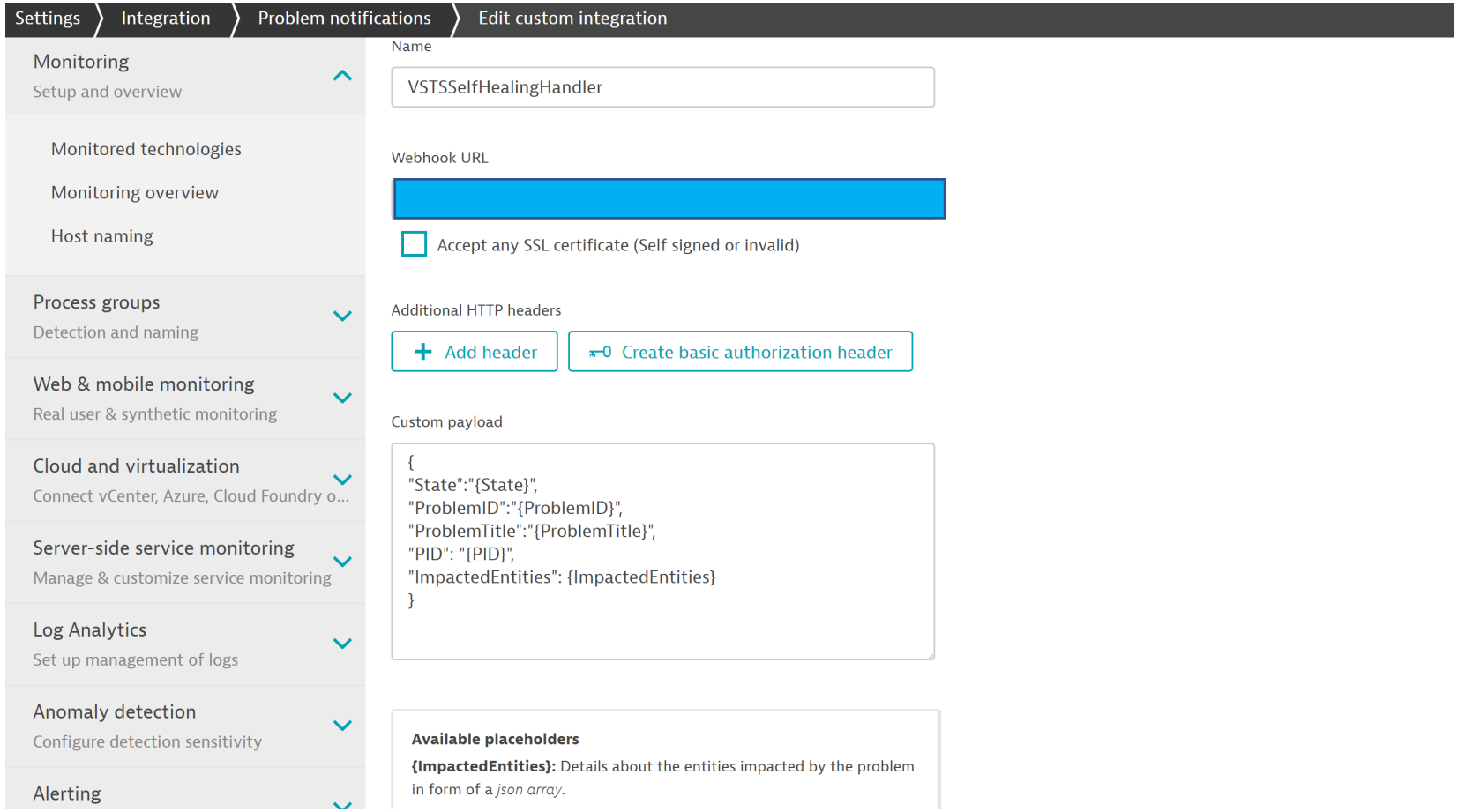


Step 13: Create Alerting Profile for Production



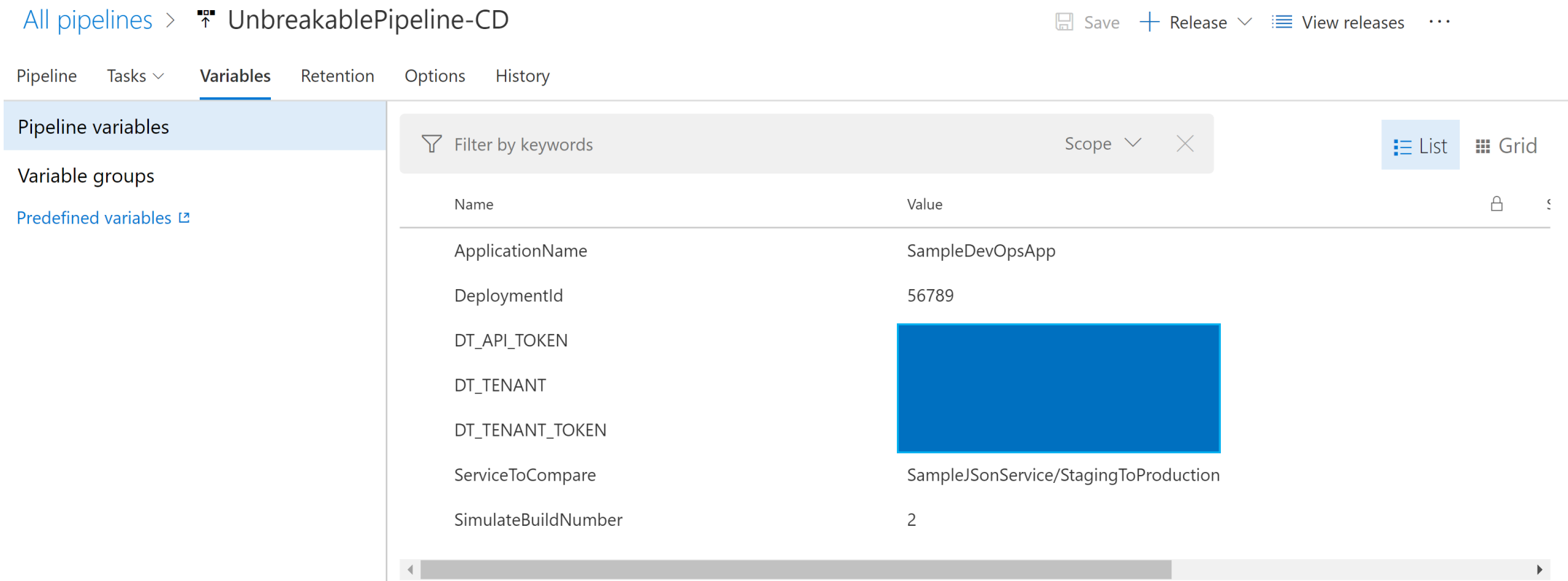
Step 14: Add SelfHealing Function URL Webhook Integration

- To enable auto remediation add the Self Healing Function URL to Dynatrace after creating the alerting profile
- WebhookURL: SelfHealingFunction URL + /api/ProcessAlert?name=<YourName>
- Don’t forget to select the Production alerting profile (at the bottom)



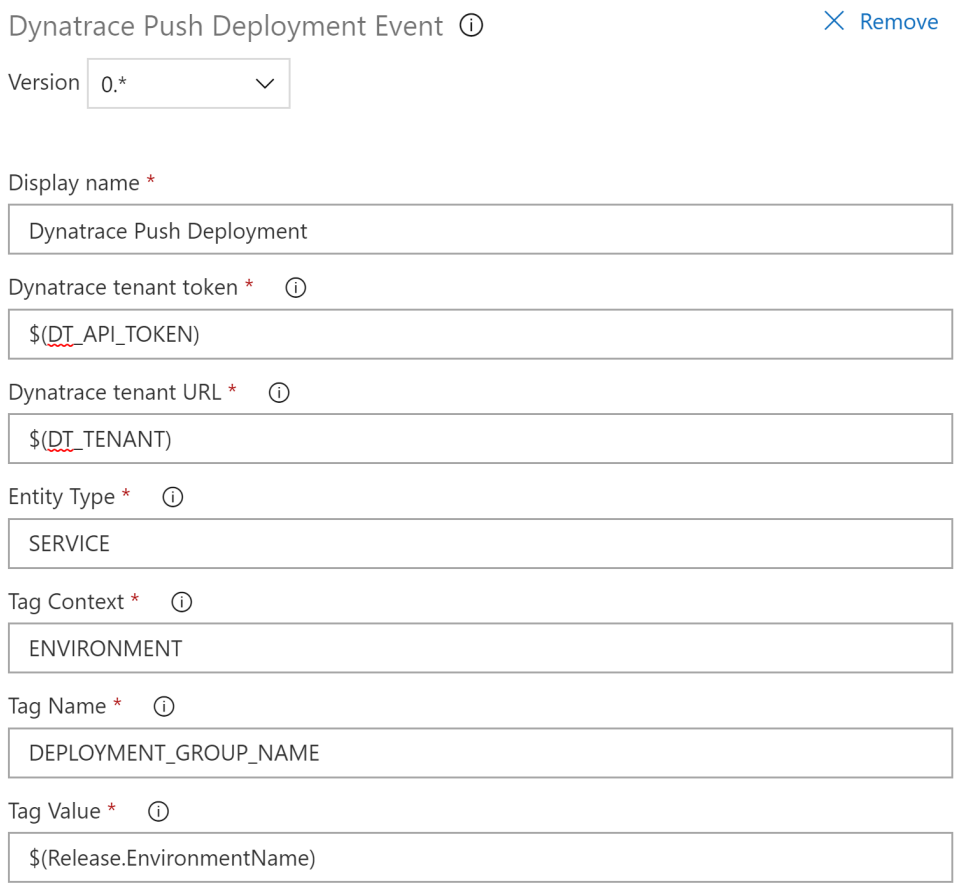
Step 14 – Release Pipeline Configuration

- After Importing the release pipeline go to each stage
- For Each Task in each stage you will have to select the Azure Subscription, Resource Group, Location
- Add the following values for the Variables in the Variables Section

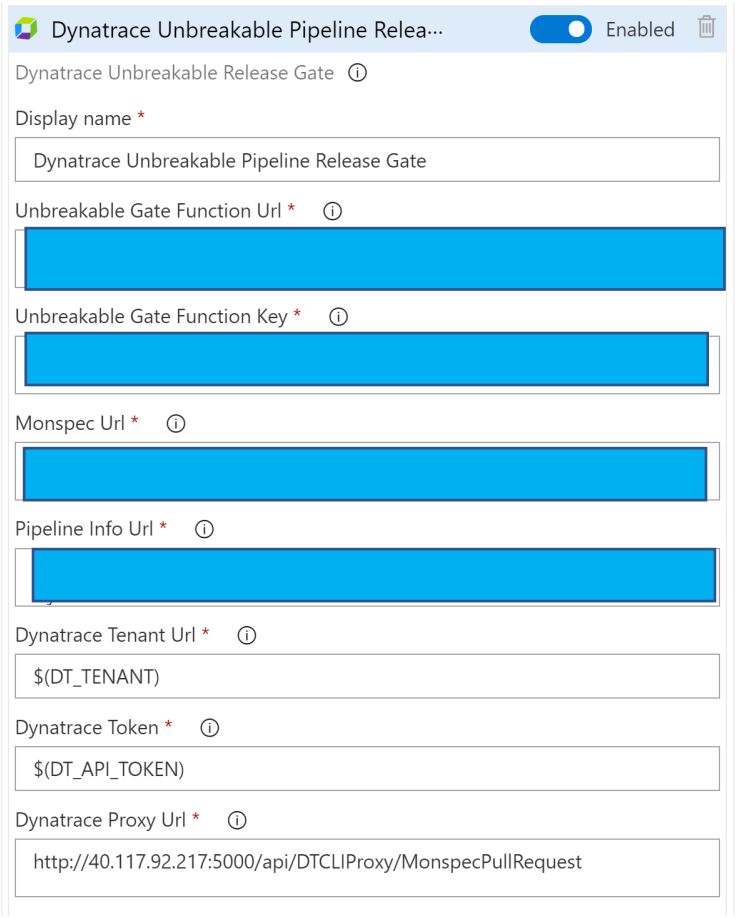


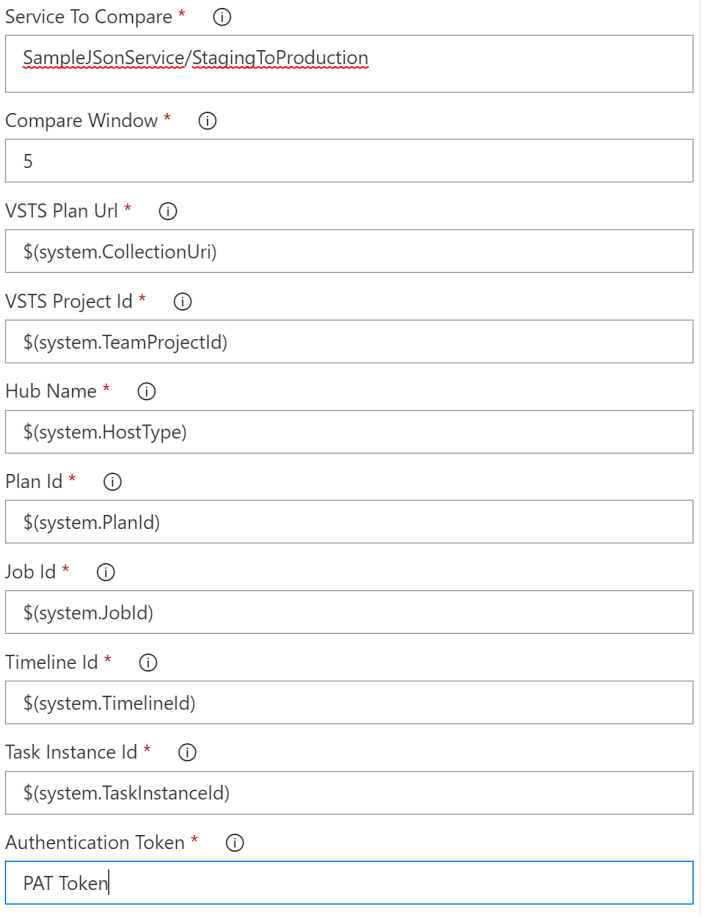
Push Deployment Event Task

- Update the values in Push Deployment Event Task for Staging and Production



Update Release gate





Values for Release Gate

- Note: Unbreakable Gate Function URL : URL + api/ProcessUnbreakableGate
- Dynatrace Proxy URL: http://40.117.92.217:5000/api/DTCLIProxy/MonspecPullRequest
- Service to Compare value: SampleJSonService/StagingToProduction
- Authentication Token : PAT Token

Create Release

- View Deployment in Dynatrace
- Validate Release Gates

Behind the Scenes: Deployment

Shift-Right: Version Information, Tags, Meta Data, …

What happens in Release Pipeline Stages?

```
#1 Executes script shell to start the app:
export DT_TAGS=APPLICATION_NAME=$APPLICATION_NAME
export DT_CUSTOM_PROP="DEPLOYMENT_ID=$DEPLOYMENT_ID DEPLOYMENT_GROUP_NAME=$DEPLOYMENT_GROUP_NAME
APPLICATION_NAME=$APPLICATION_NAME"
export DT_CLUSTER_ID="$DEPLOYMENT_GROUP_NAME $APPLICATION_NAME"
pm2 start app.js &> pm2start.log
```

Shift-Right: Tags, Deployments & Events

Release Automation

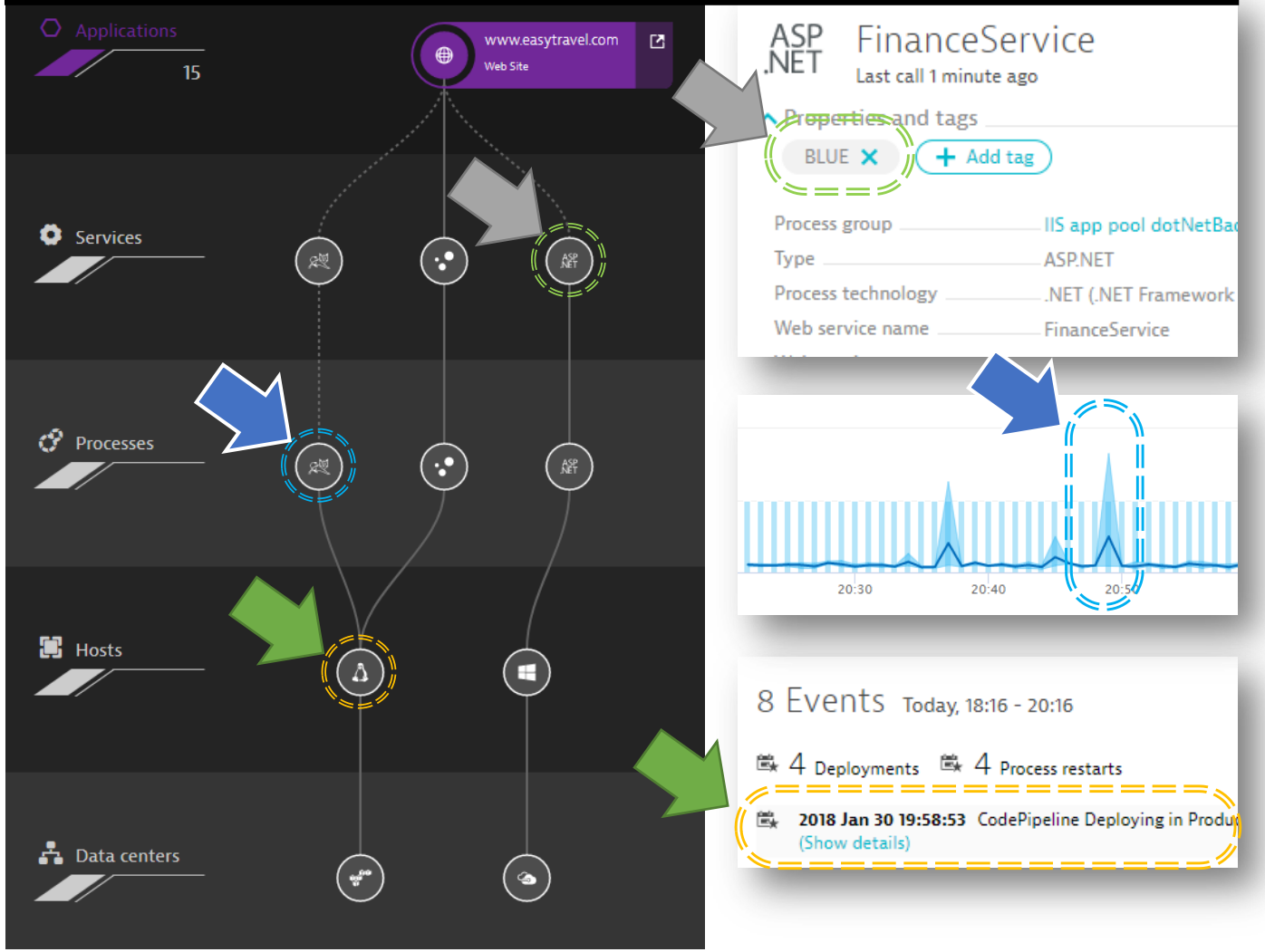
Dynatrace Automation
API, CLI, Auto-Detection

```
docker run -e DT_TAGS=BLUE
dtcli tag srv CartServicev2 GREEN
```

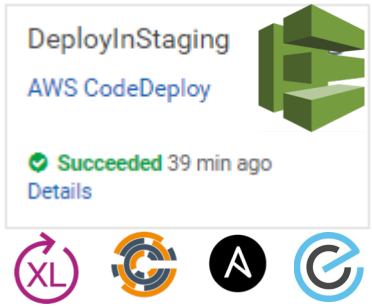
```
dtcli evt push pg tomcat1
desc=JVMMemIncr hint=+100MB
```

```
dtcli evt push host .*demo
version=123 source={code_deploy}
```

Dynatrace Smartscape

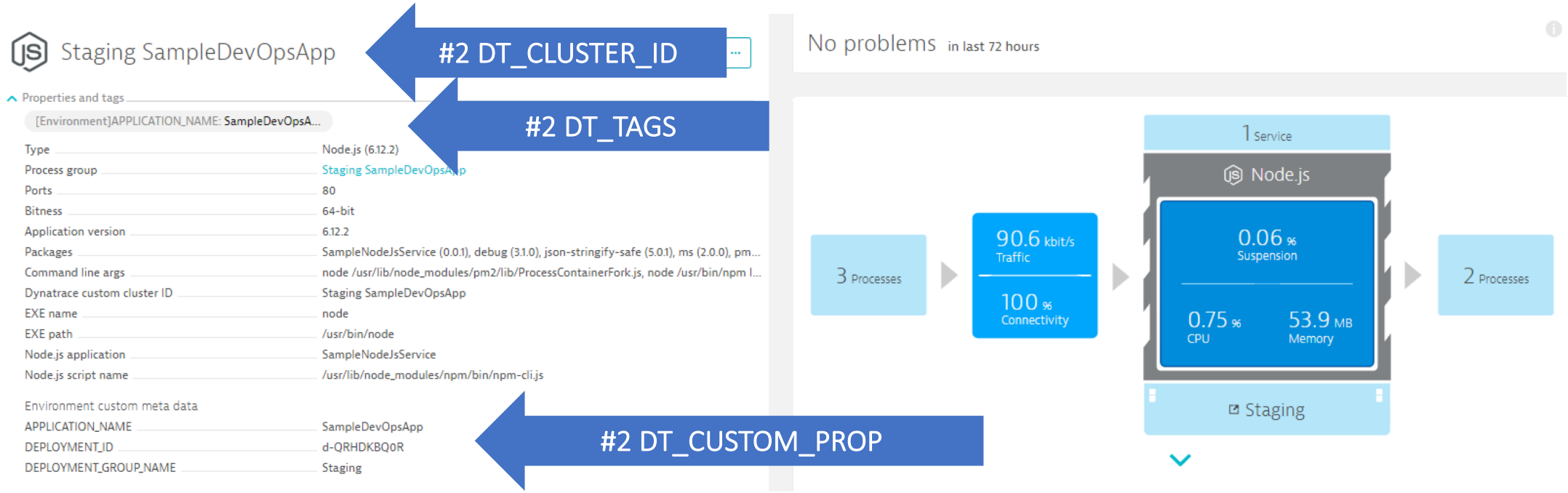


@ Deployment: Passing Meta Data to launched processes

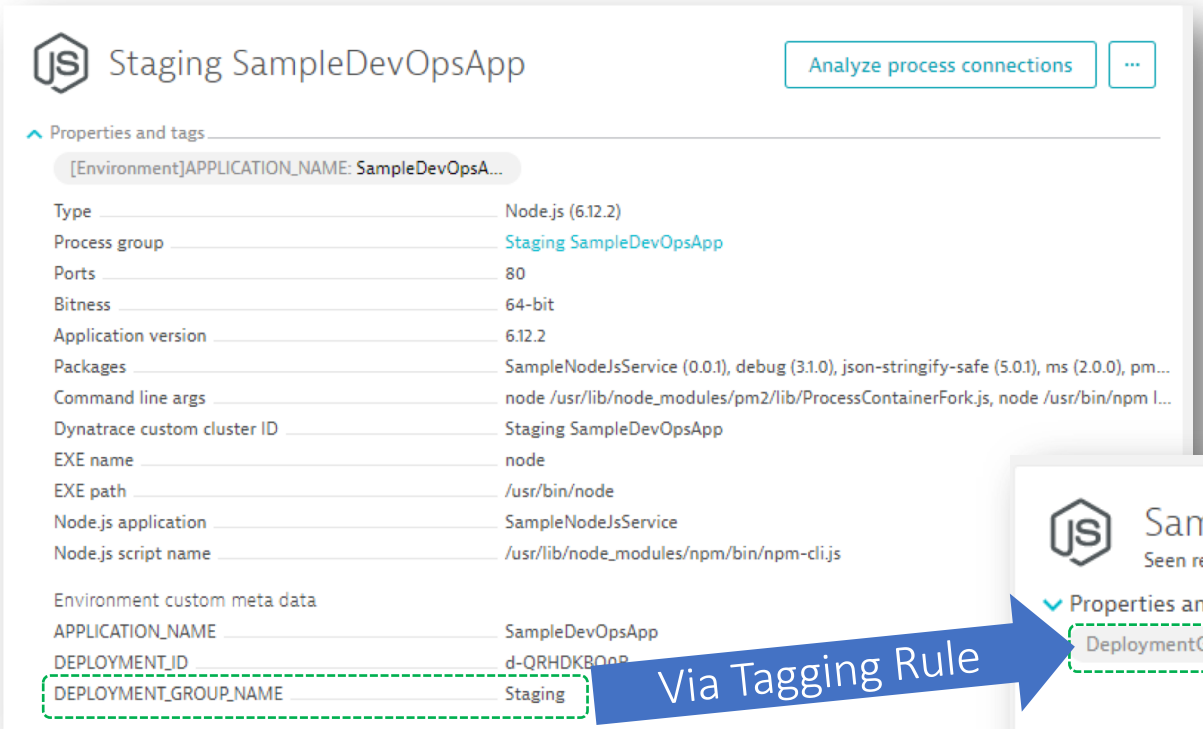


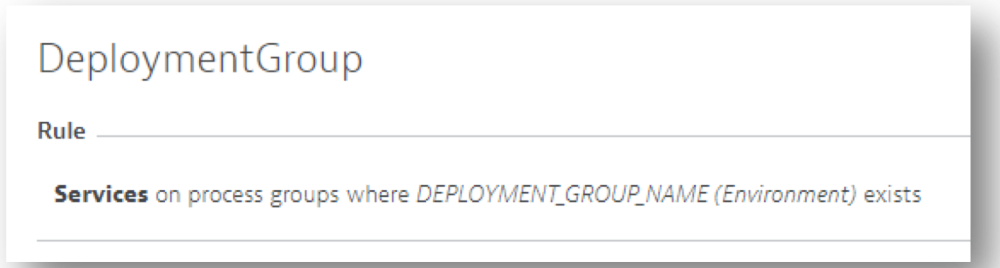
AWS CodeDeploy executes script during deployment:

```
export DT_TAGS=APPLICATION_NAME=$APPLICATION_NAME
export DT_CUSTOM_PROP="DEPLOYMENT_ID=$DEPLOYMENT_ID DEPLOYMENT_GROUP_NAME=$DEPLOYMENT_GROUP_NAME
APPLICATION_NAME=$APPLICATION_NAME"
export DT_CLUSTER_ID="$DEPLOYMENT_GROUP_NAME $APPLICATION_NAME"
pm2 start app.js &> pm2start.log
```

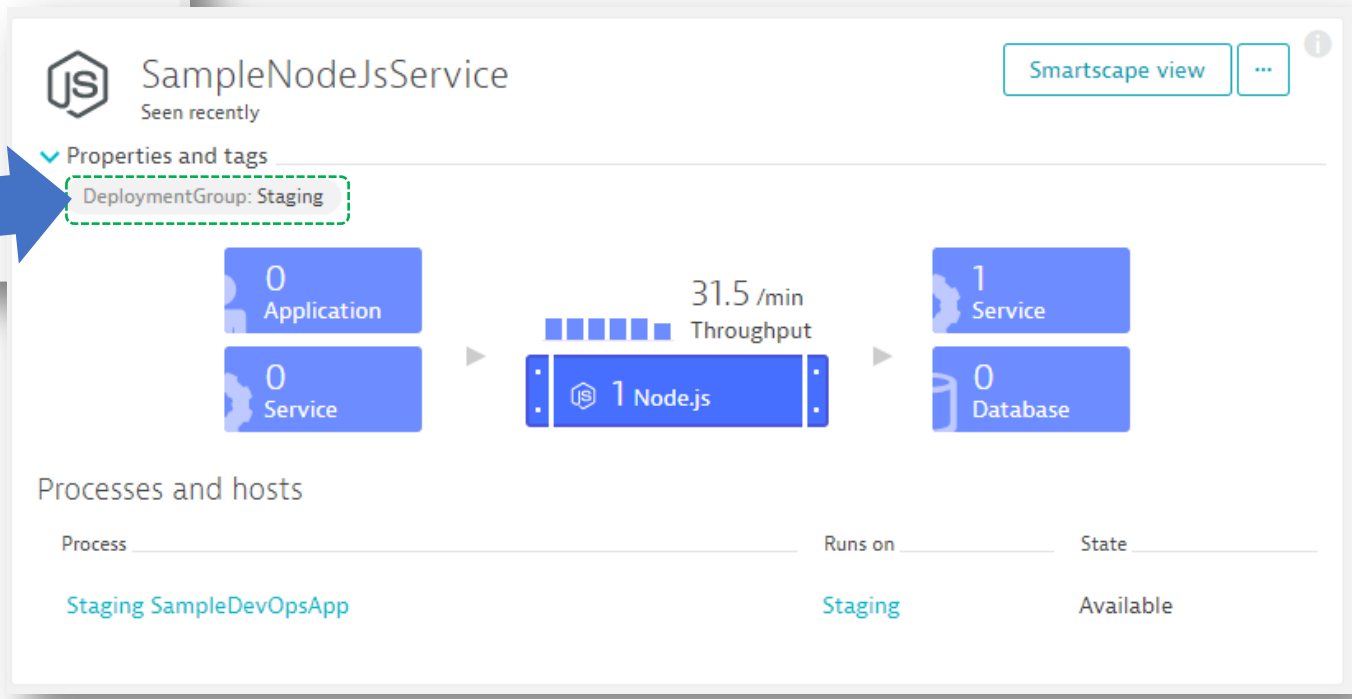


@ Deployment: Converting Process Meta Data to Tags on Services!

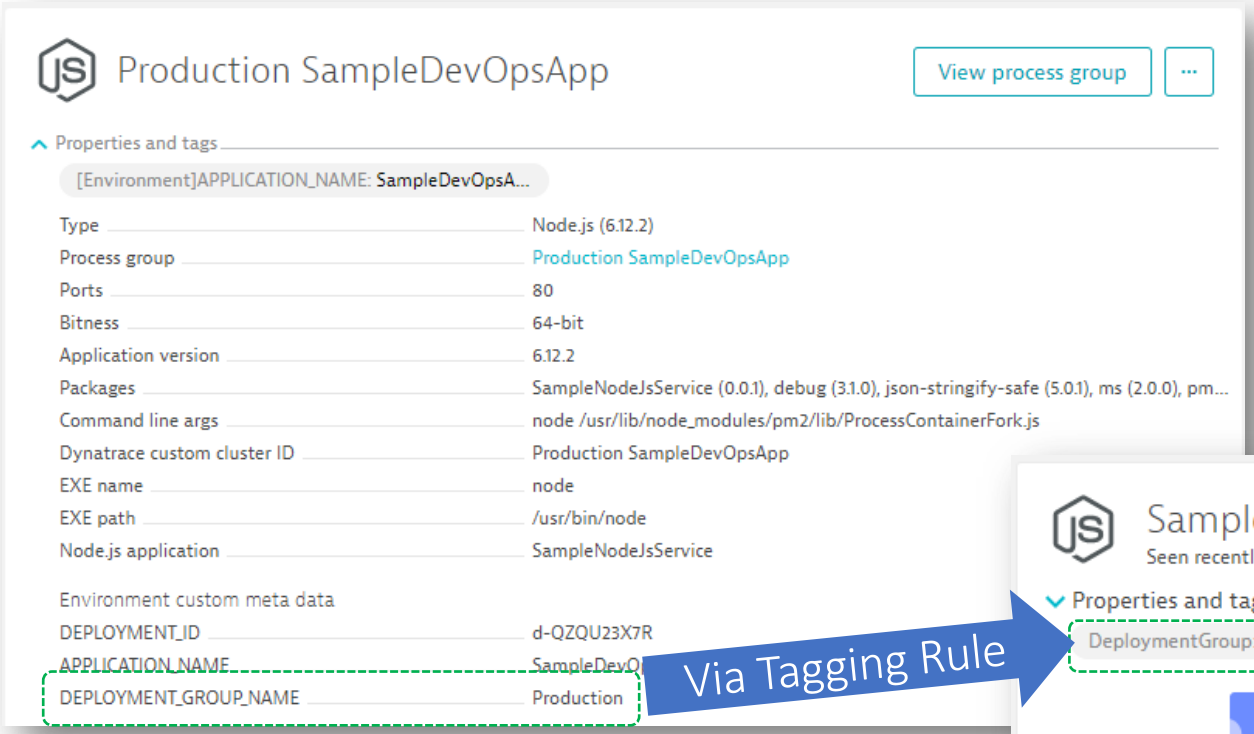


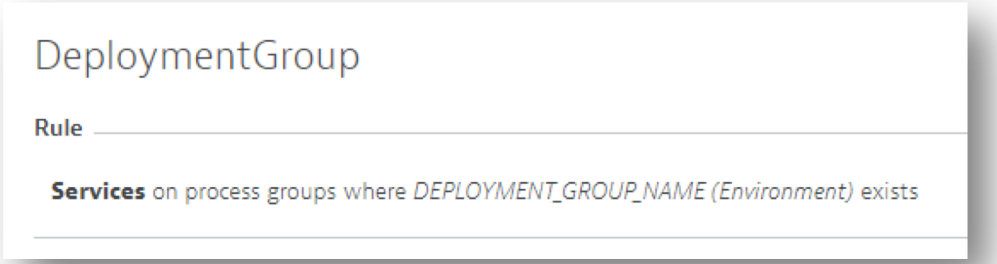


Via Tagging Rule

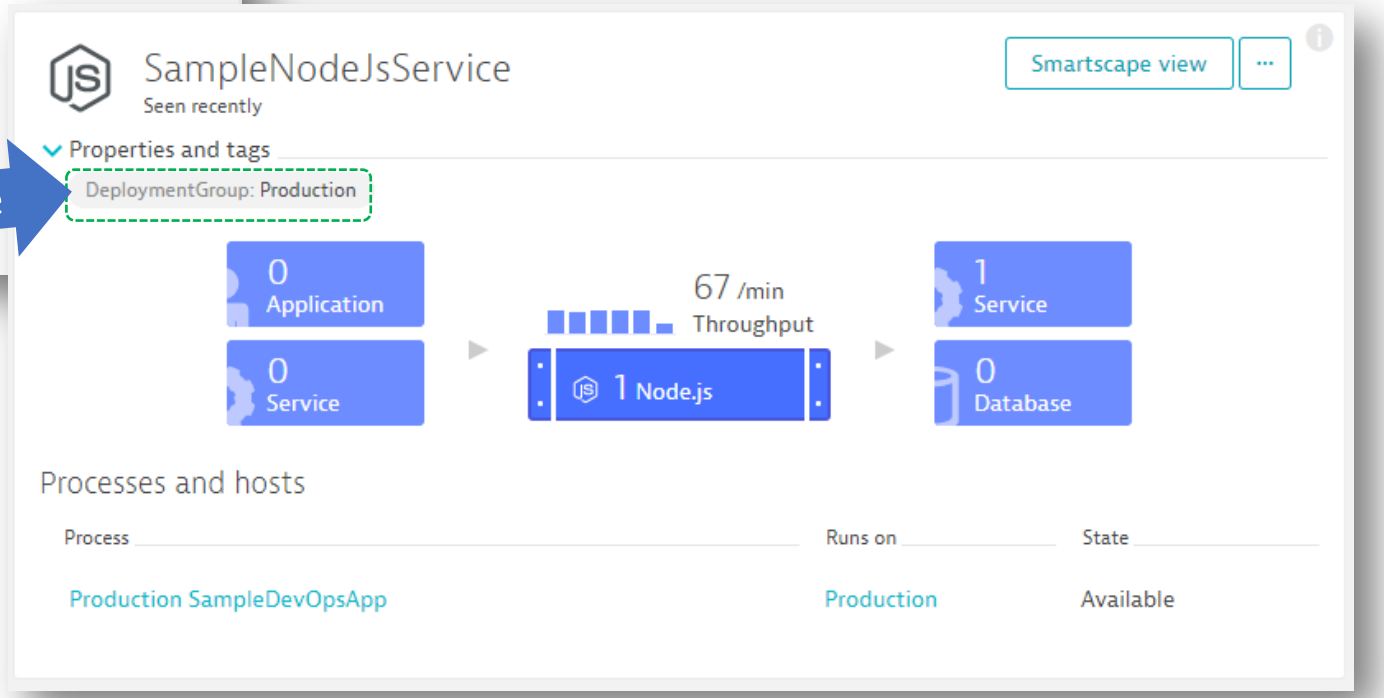


@ Deployment: Converting Process Meta Data to Tags on Services!

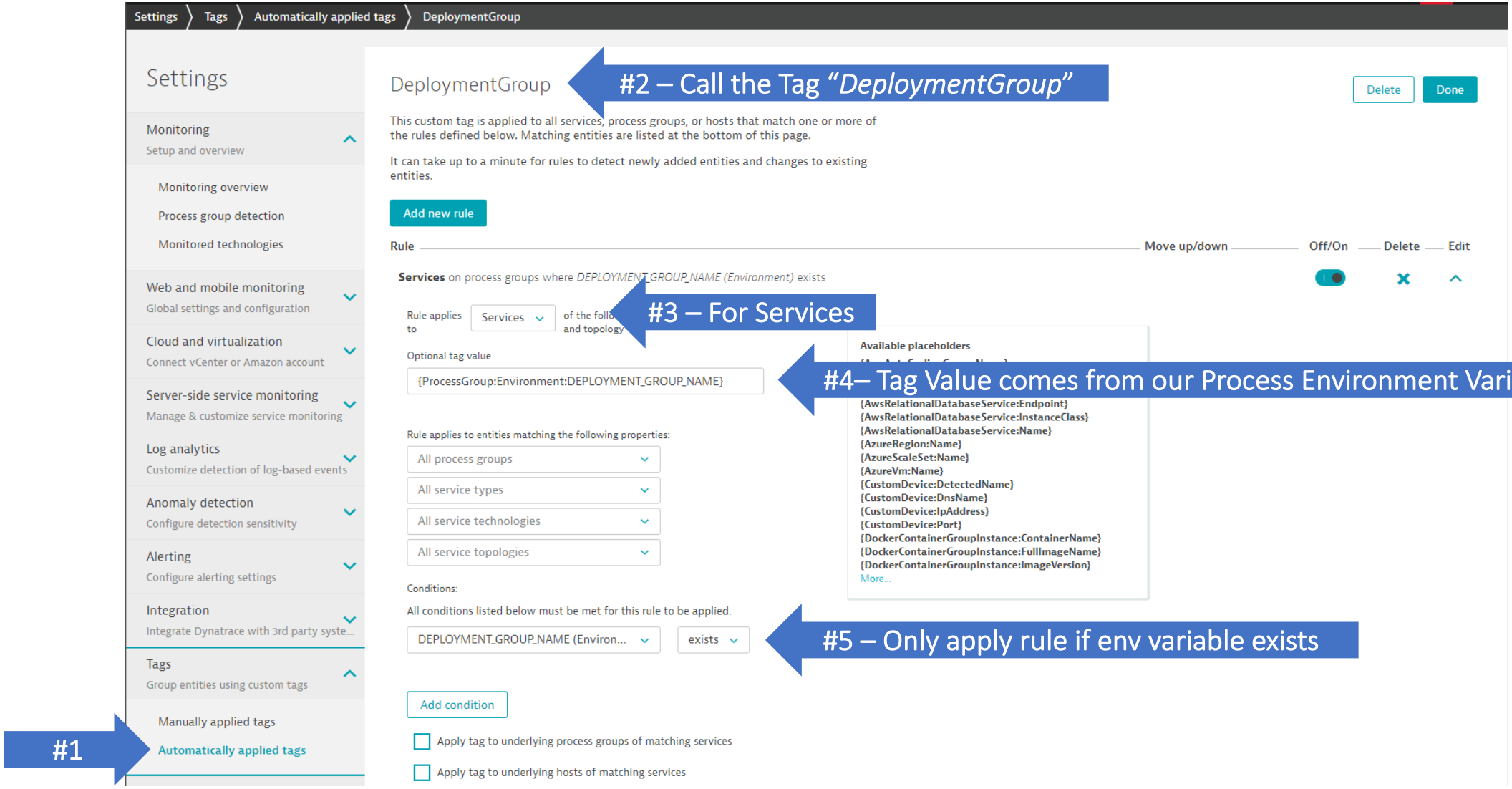




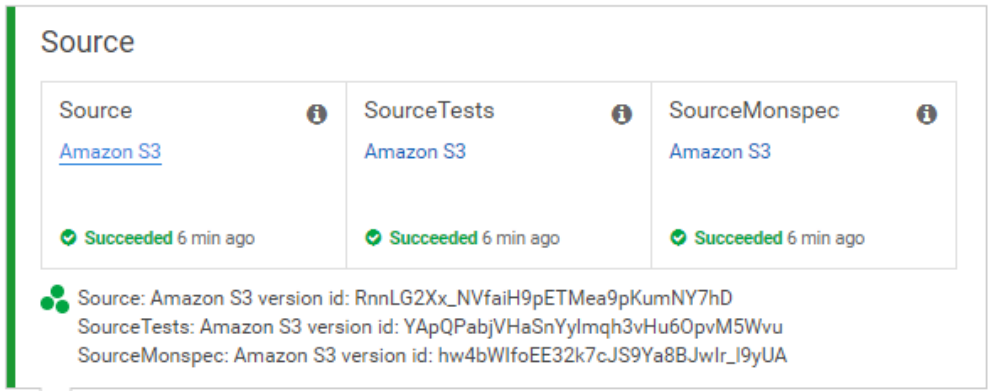
Via Tagging Rule

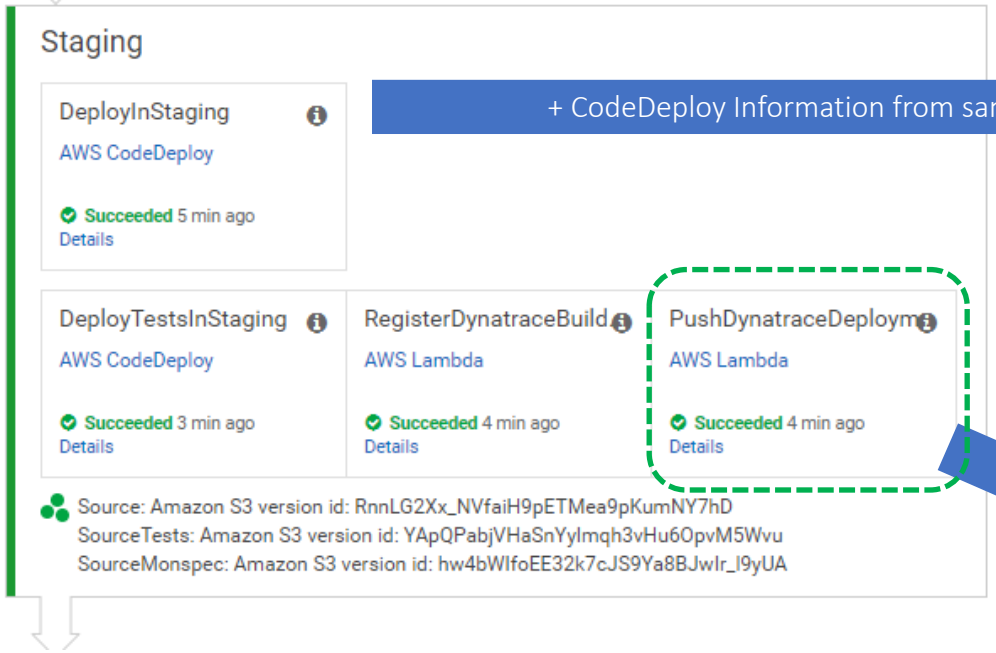


Automated Tag Rules: Define ONCE in Dynatrace!

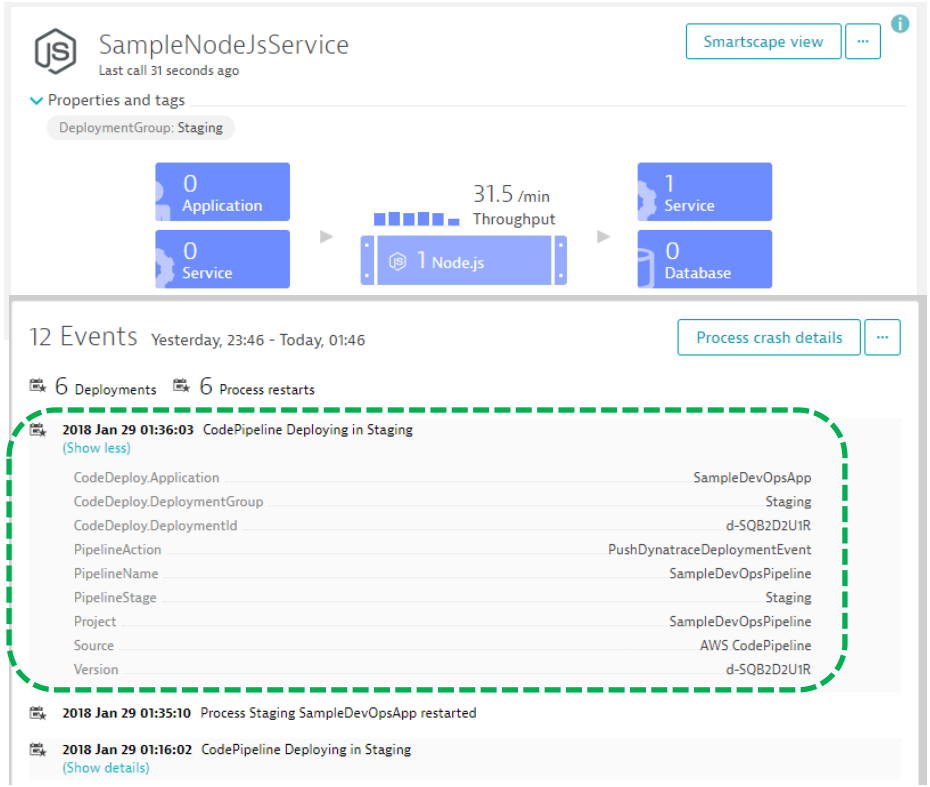


@Deployment: Pass Deployment Information to Deployed Entities





+ CodeDeploy Information from same Stage



Dynatrace Events REST API

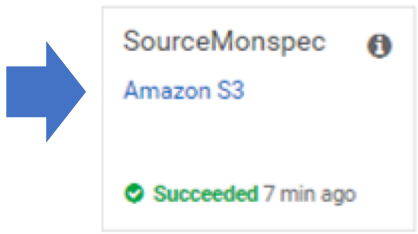
pushDynatraceDeploymentEvent

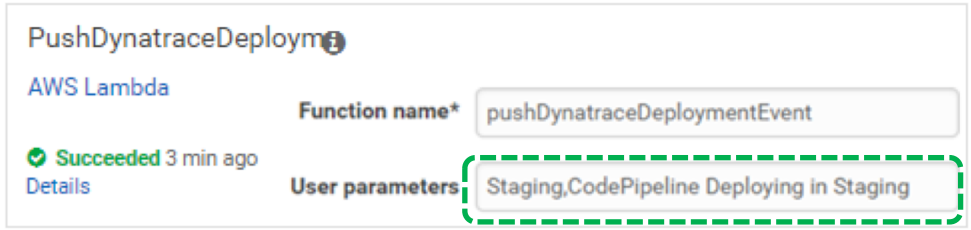
Pushes CodePipeline Info to Dynatrace

Monitoring as Code (Monspec): Define Services and their Environments as Code

Monspec (Monitoring as Code)

```
"SampleJSonService" : {
   "etype": "SERVICE",
   "environments" : {
     "Staging" : {
       "tags" : [
         {
           "context": "CONTEXTLESS",
           "key": "DeploymentGroup",
           "value": "Staging"
         }
       ]
     },
     "Production" : {
       "tags" : [
         {
           "context": "CONTEXTLESS",
           "key": "DeploymentGroup",
           "value": "Production"
         }
       ]
     }
   },
```



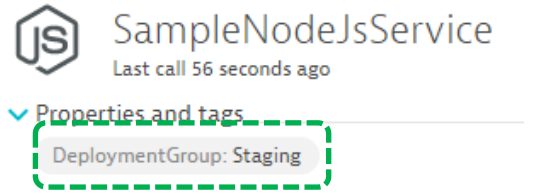


pushDynatraceDeploymentEvent

Pulls Tag Information from Monspec

Dynatrace Events REST API

Pushes Event for Entities with Tag DeploymentGroup:Staging



Behind the Scenes: Approval Stage

Shift-Left: Automate Metric Comparison to Validate Build

Monitoring as Code: Compare Metrics of Services of Different Environments/Builds

Entity: SampleJSonService

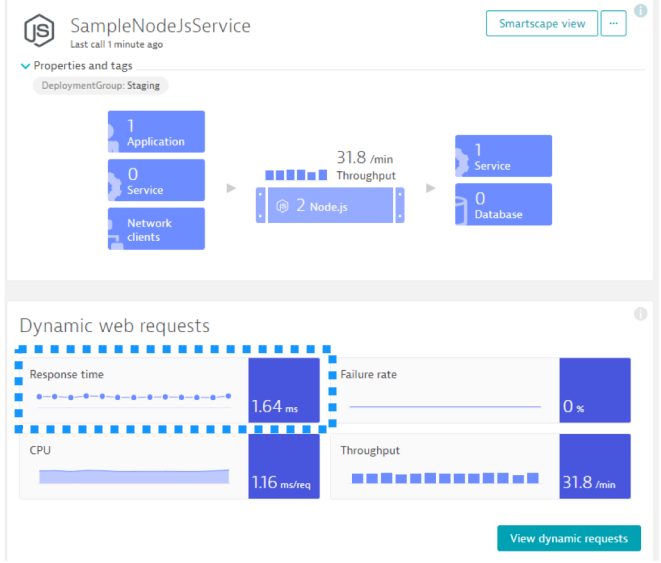
Environment: Staging *Tag: DeploymentGroup:Staging*	**Environment**: Production *Tag: DeploymentGroup:Production*		
Metrics: ResponseTime (Avg)	**Metrics**: ResponseTime(Max) < 10s	**Metrics**: FailureRate(Avg)	**M** nseCount *Less vs Production*
Compare: StagingToProduction *Source: Staging; Compare: Prod* *Factor: Staging = **10%** Slower*	**Compare**: StagingToProdYesterday *Source: Staging; Compare: Prod* *Factor: 10%, Timeshift: 86400s*	**Compare**: StagingToStagingLastHour *Source: Staging; Compare: Staging* *Factor: 0; Timeshift: 3600s*	**Compare**: StageToStageYesterday *Source: Staging; Compare: Staging* *Factor: 0; Timeshift: 86400s*

Different Metrics

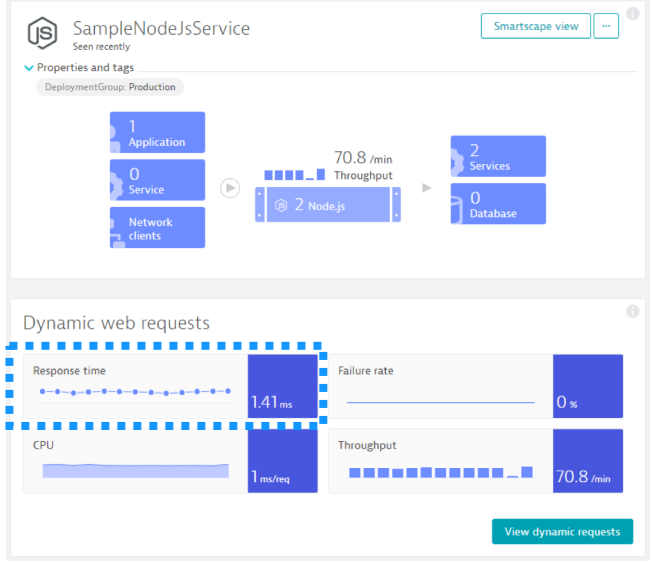
Different Comparisons

Pipeline Run: Validate Build with StagingToProduction

Environment: Staging



Environment: Production



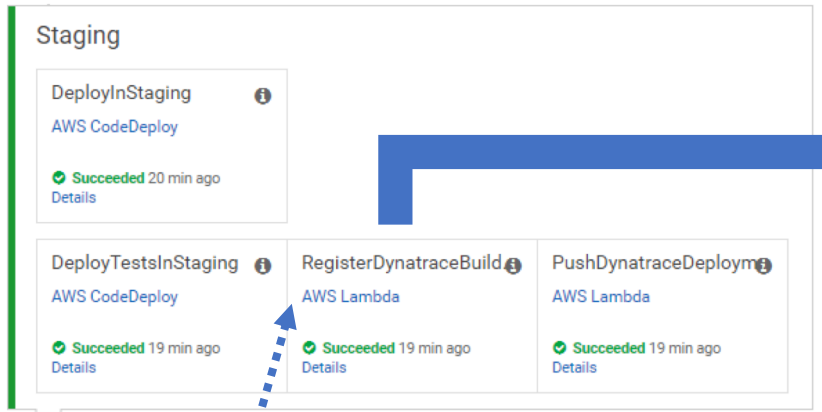

```
Source (Staging):        1.64ms

Compare (Production): 1.41ms

Threshold (+10%):        1.56ms

Status:                  VIOLATION
```

@Approval: Automating Build Validation into your CodePipeline



Monspec from S3

Adds Item

Build Validation Request Item

- Pipeline Information
- Monspec
- Timestamp + Timeframe
- Comparison Definition Name
- Action Name to Approve / Reject

registerDynatraceBuildValidation
Adds Build Validation Request

Updates Build Validation Request

- Updated Monspec
- Updated Status

Approves/Rejects IF “In Progress” & if RegisterBuildValidation was called with that Action Name

triggers

CloudWatch Events
(e.g: 1min)

validateBuildDynatraceWork

Dynatrace Entities & Timeseries REST API
Resolves Tags and gets list of Entities
Queries Metrics for these Entities

Monitoring as Code: The Full Picture

monspec.json

```
{
    "SampleJSonService" : {
        "etype": "SERVICE", // "Options are SERVICE, APPLICATIO
        "name": "SampleNodeJsService",
        "environments" : {
            "Staging" : {
                "tags" : [
                    {
                        "context": "CONTEXTLESS",
                        "key": "DeploymentGroup",
                        "value": "Staging"
                    }
                ]
            },
            "Production" : {
                "tags" : [
                    {
                        "context": "CONTEXTLESS",
                        "key": "DeploymentGroup",
                        "value": "Production"
                    }
                ]
            }
        },
```

Entity: What Entity do we want to validate?

Environments: Tell us how we can detect these entities in Dynatrace

Monitoring as Code: The Full Picture

monspec.json

```
{
  "SampleJSonService" : {
     ...
     "comparisons" : [
     {
        "name" : "StagingToProduction",
        "source" : "Staging",
        "compare" : "Production",
        "scalefactorperc" : {
          "default": 10,
          "com.dynatrace.builtin:service.requestspermin" : 90
        },
        "shiftcomparetimeframe" : 0,
        "shiftsourcetimeframe" : 0,
      },
      {
        "name" : "StagingToProductionYesterday",
        "source" : "Staging",
        "compare" : "Production",
        "scalefactorperc" : {
          "default": 10,
          "com.dynatrace.builtin:service.requestspermin" : 90
        },
        "shiftsourcetimeframe" : 0,
        "shiftcomparetimeframe" : 86400
      }
```

Comparision: Which environments do we compare and how!

monspec.json

```
{
    "SampleJSonService" : {
          ...
          "comparisons" : [
            {
                "name" : "StagingToStagingLastHour",
                "source" : "Staging",
                "compare" : "Staging",
                "scalefactorperc" : { "default": 0},
                "shiftsourcetimeframe" : 0,
                "shiftcomparetimeframe" : 3600
            },
            {
                "name" : "ProductionToProductionLastH",
                "source" : "Production",
                "compare" : "Production",
                "scalefactorperc" : { "default": 0},
                "shiftsourcetimeframe" : 0,
                "shiftcomparetimeframe" : 3600
            }
        ],
```

Monitoring as Code: The Full Picture

monspec.json

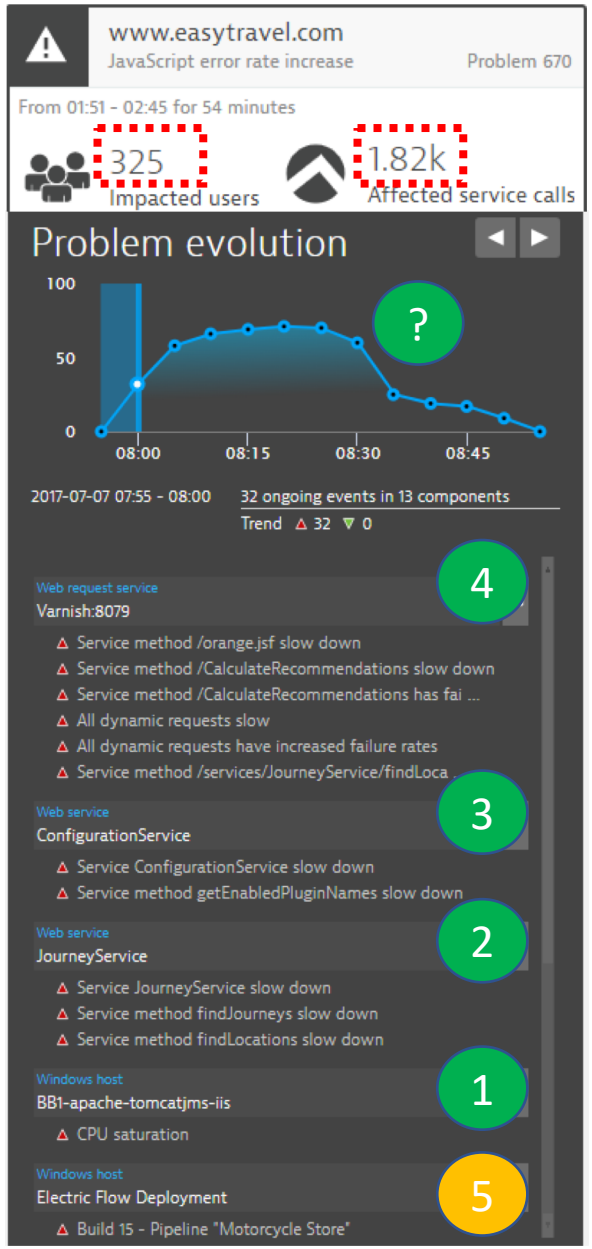
```
{
  "SampleJSonService" : {
     ...
     "perfsignature" : [
       {
         "timeseries" : "com.dynatrace.builtin:service.responsetime",
         "aggregate" : "avg", // min, max, avg, sum, median, count, percentile
         "validate" : "upper", // upper or lower
         // "upperlimit" : 100, // Optional: Can be used to define a FIXED THRESHOLD
         // "lowerlimit" : 50, // Optional: Can be used to define a FIXED THRESHOLD
       },
       {
         "timeseries" : "com.dynatrace.builtin:service.responsetime",
         "aggregate" : "p90"
       },
       {
         "timeseries" : "com.dynatrace.builtin:service.failurerate",
         "aggregate" : "avg"
       },
       {
         "timeseries" : "com.dynatrace.builtin:service.requestspermin",
         "aggregate" : "count",
         "validate" : "lower"
       }
     ],
```

Metrics: Which Metrics, Aggregation, Upper/Lower Boundaries

Behind the Scenes: Self-Healing

Smart Remediation Action based on AI-Detected Problems

Path to NoOps: Better and Smarter Auto-Remediation, Self-Healing, …



?Escalate at 2AM?

Auto Mitigate!

1 *CPU Exhausted?* Add a new service instance!

2 *High Garbage Collection?* Adjust/Revert Memory Settings!

3 *Issue with BLUE only?* Switch back to GREEN!

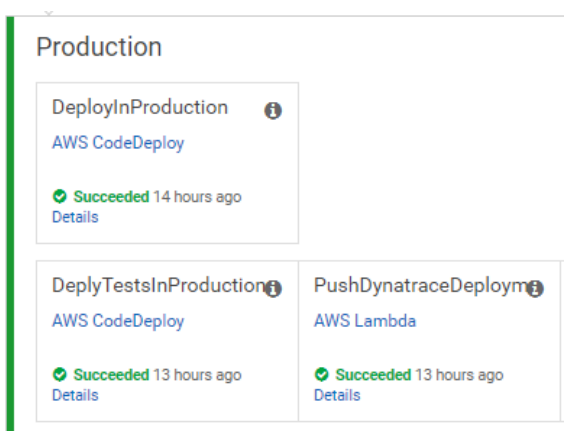
4 *Hung threads?* Restart Service!

? *Impact Mitigated?* → Update Dev Tickets

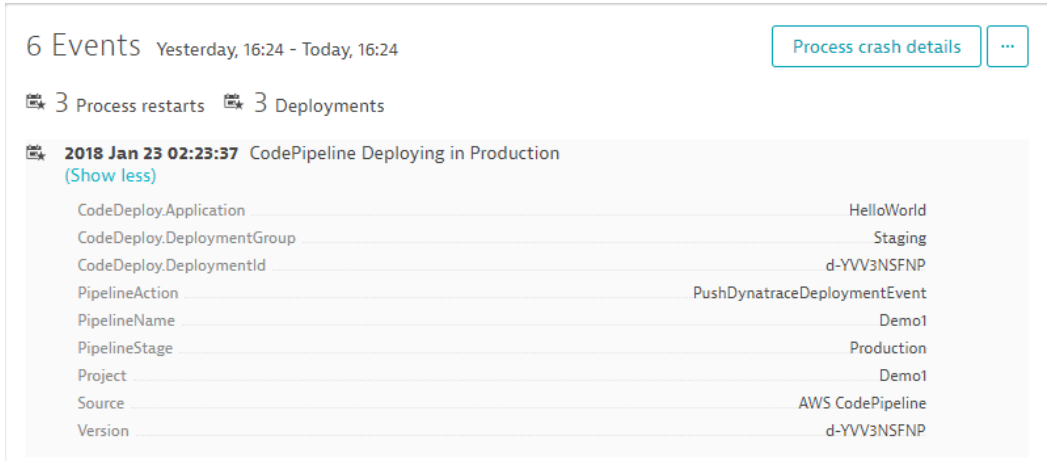
5 *Still ongoing?* Initiate Rollback! → Mark Bad Commits

? *Still ongoing?* → Escalate

Automated Rollbacks with AWS Lambda and AWS CodeDeploy



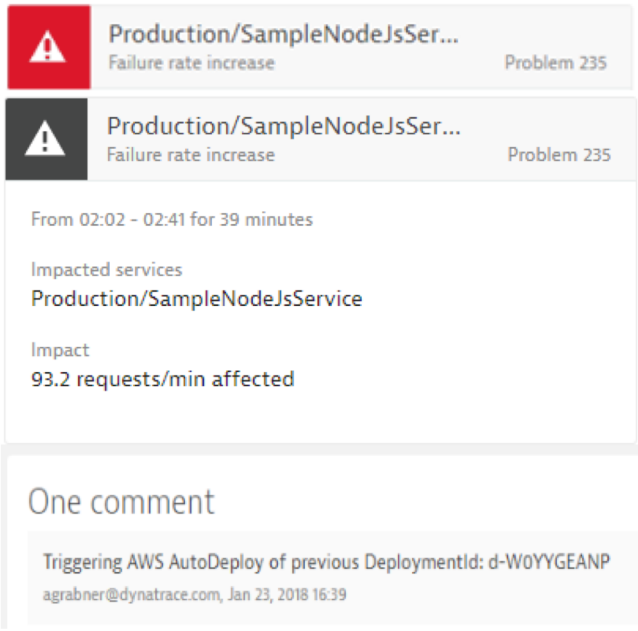
#1: Push Deployment Information, e.g: CodeDeploy DeploymentId



#3

Uses Dynatrace Events API

to pull CUSTOM_DEPLOYMENT events



#2: Calling Lambda via API Gateway

#4: Redeploy previous revision

handleDynatraceProblemNotification

#5: Push comment to Dynatrace

Simulating Different “Bad” Builds

- Application looks at startup for an environment variable BUILD_NUMBER. Allowed values, 1, 2, 3, 4

Build Number	Behavior
1	No Problem
2	50% of requests return HTTP 500
3	No Problem
4	Staging OK. 10% Requests fail in Production

- To Deploy a different build:
  - Change the build number value in the associated Build Pipeline variable
  - Save and Queue
  - Deploy a new build (Release Pipeline)