Beda Tse

86 Followers About Follow

Azure DevOps CI/CD with Azure Databricks and Data Factory— Part 1

Beda Tse Feb 28, 2019 · 13 min read

Let's cut long story short, we don't want to add any unnecessary introduction that you will skip anyway.

For whatever reason, you are using Databricks on Azure, or considering using it. Google, or your favourite search engine, has brought you here because you want to explore Continuous Integration and Continuous Delivery using Azure DevOps.

May be it is just me, I always find outdated tutorials when I wanted to learn something new, and by the time you are reading this, this has already outdated as well. But hopefully this series will give you some insight on setting up CI/CD with Azure Databricks.

Without further due, let us begin.

Prerequisites

You need to have an ***Azure account***, an ***Azure DevOps organisation***, you can leverage either ***GitHub*** as repository or ***Azure Repos*** as repository. In this series, we will assume you are using Azure Repos.

You will need to ***create a project*** on Azure DevOps, together with a ***repository***. A sample repository can be found here.

You will need a ***git client***, or command line git. We will use command line git throughout the series, thus assuming that you also have a terminal, such as Terminal on Mac, or Git-Bash on Windows.

You will need a ***text editor*** other than the normal Databricks notebook editor. Visual Studio Code is a good candidate for that. If the text editor have built-in git support, that will be even better.

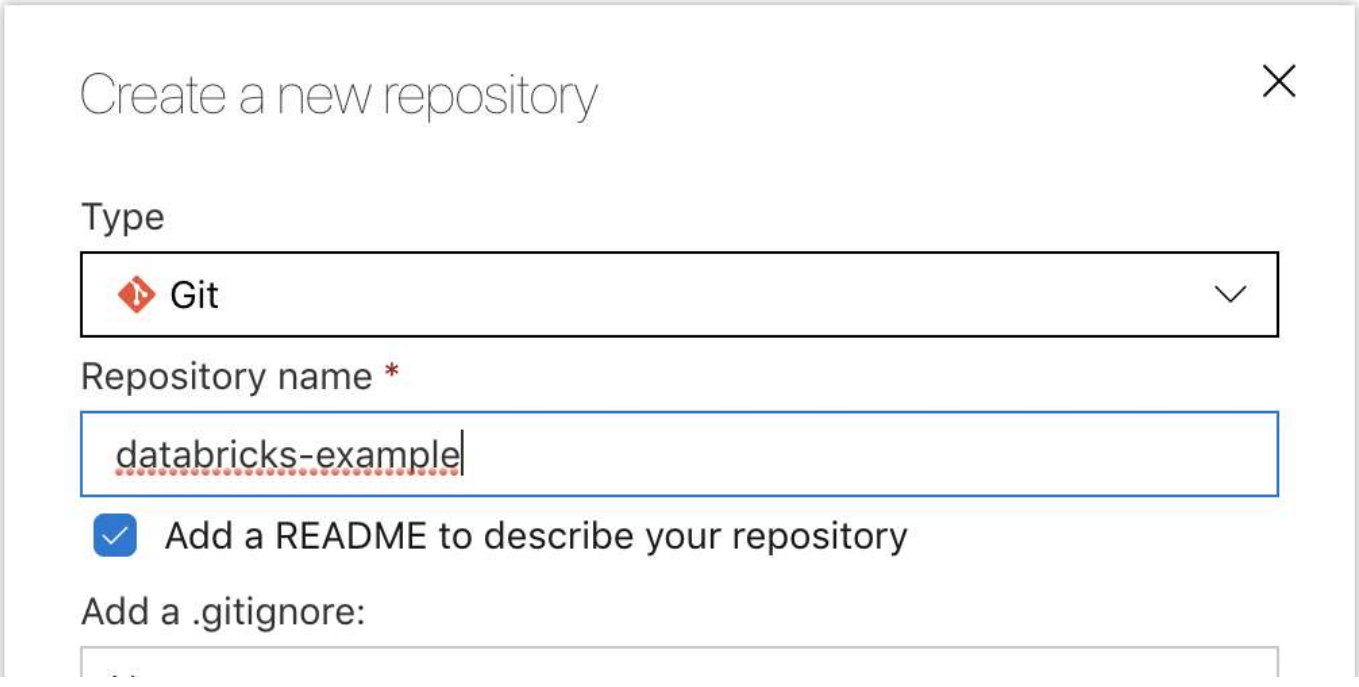
Checklist

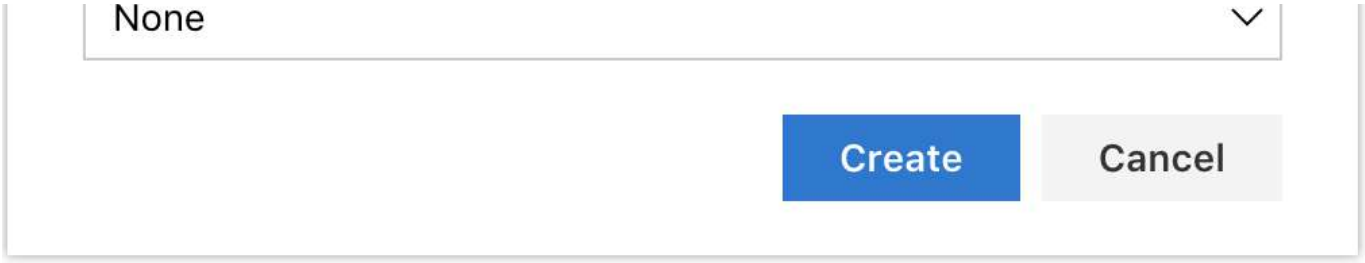
1. Azure Account
2. Azure DevOps Organisation and DevOps Project
3. Azure Service Connections set up inside Azure DevOps Project
4. Git Repos (Assuming you are using Azure Repository)
5. Git Client (Assuming you are using command line Git)
6. Text Editor (e.g. Visual Studio Code, Sublime Text, Atom)

Step 0: Set up your repository and clone it onto your local workstation

0–1. In your Azure DevOps, set up your SSH key or Personal Access Token.

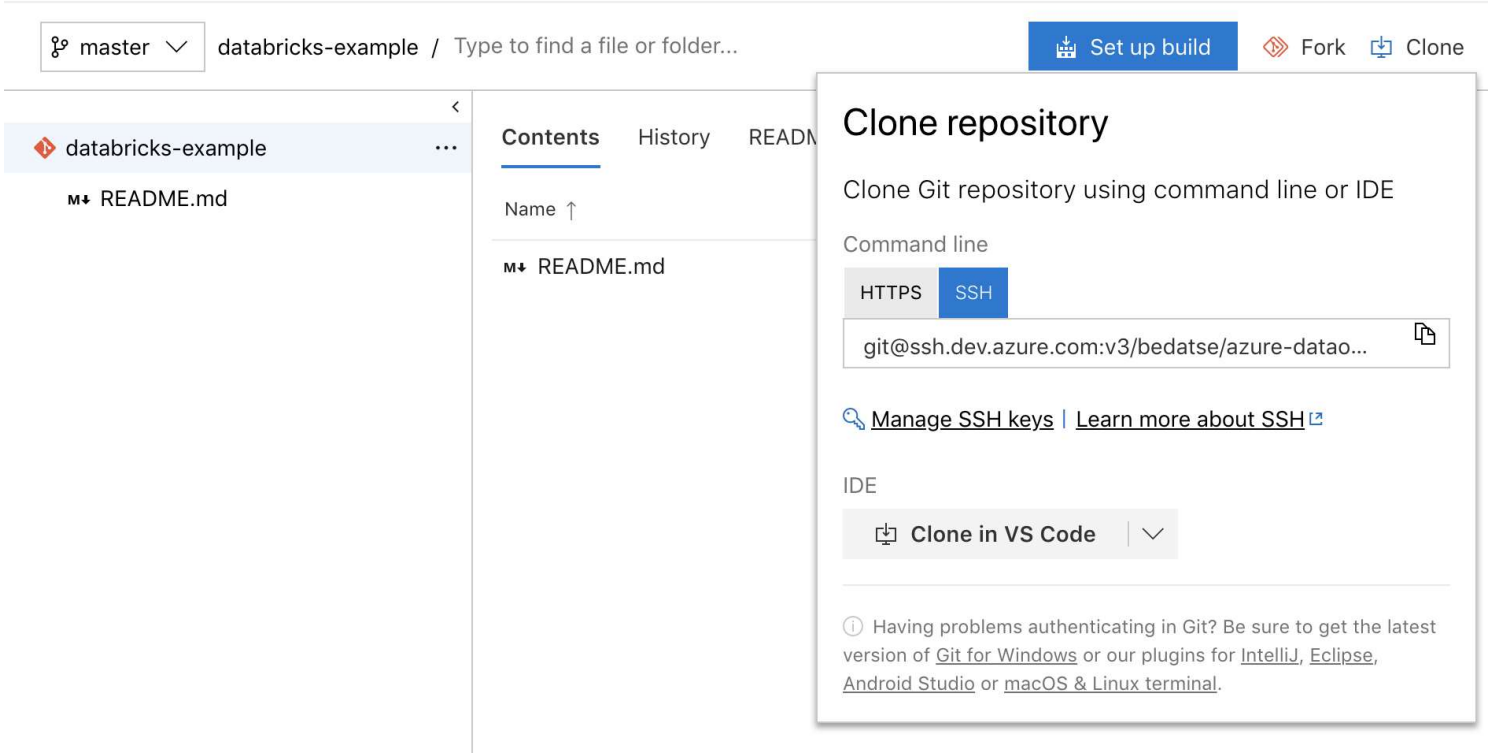
0–2. Create git repo on Azure DevOps with initial README





0–2. Create your git repo on Azure DevOps inside a project with initial README.

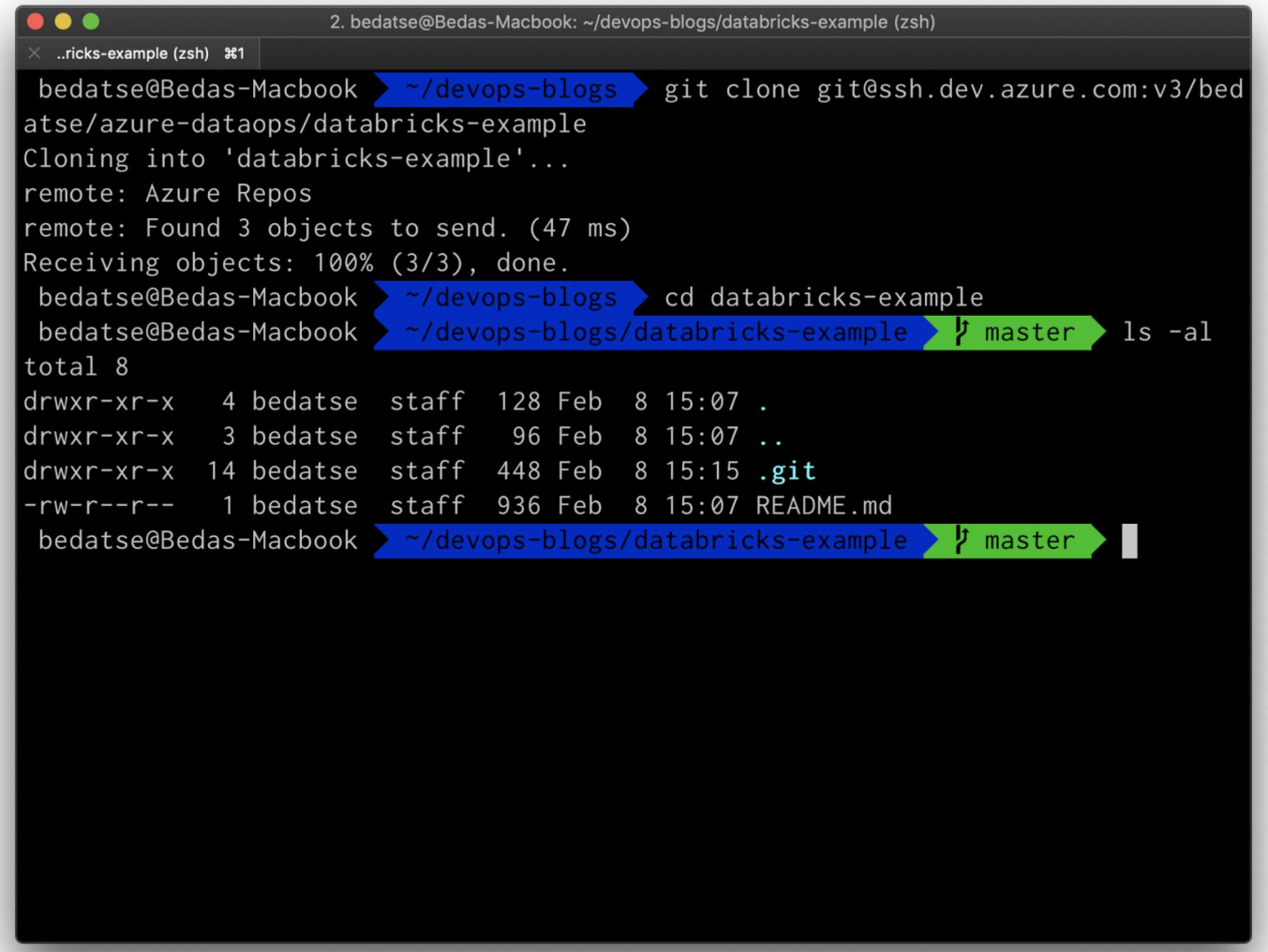
0–3. Locate your git repository URL for git clone



0–3. Locate your git URL for cloning

0–4. Clone the repository via git using the following command

```
$ git clone <repository-url>
```



0–4. Now you have cloned your repository locally.

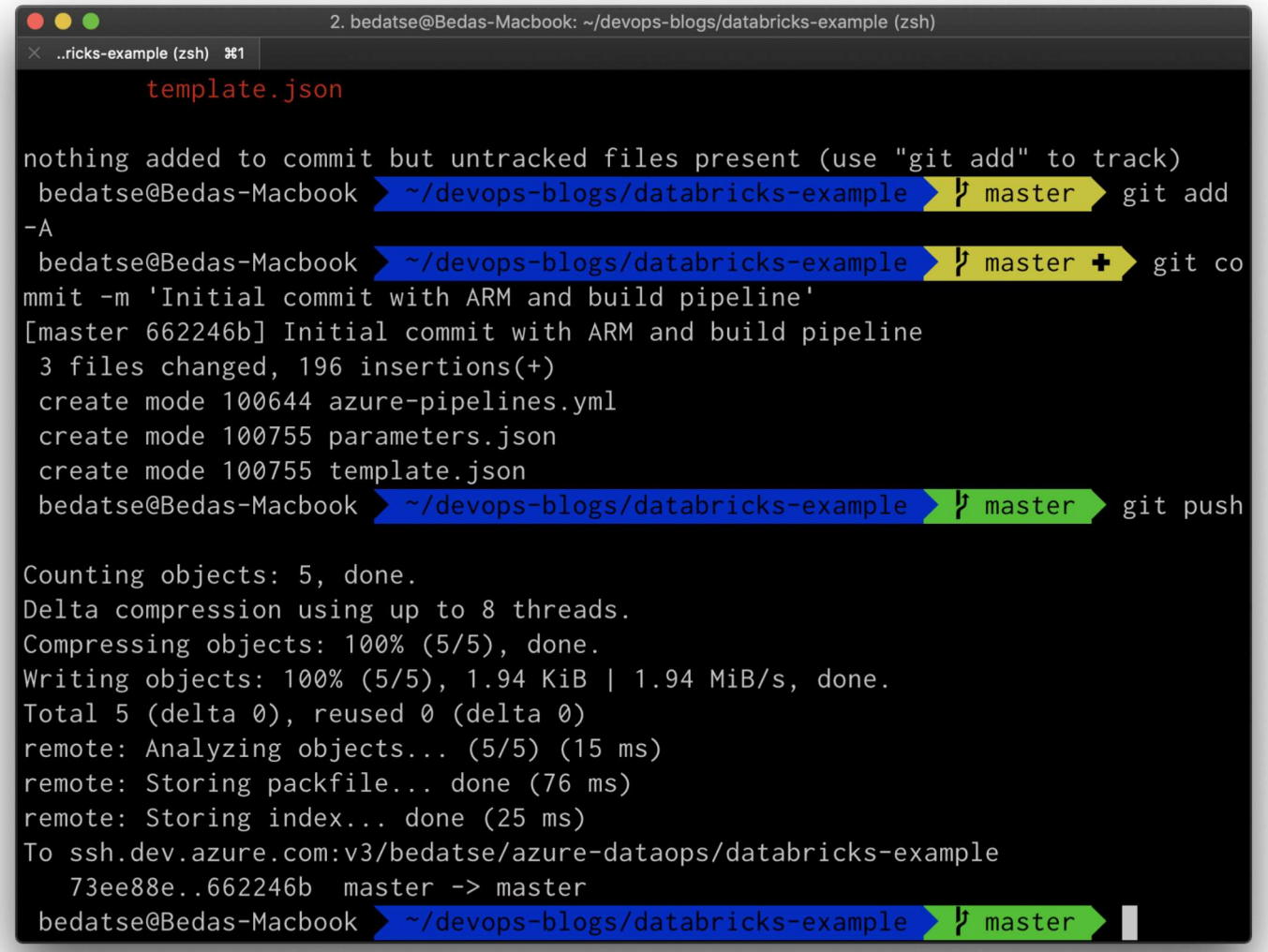
Step 1: Provisioning Azure Databricks and Azure Key Vault with Azure Resource Manager Template

We want to automated the service provisioning or service updates. When you need to set up another set of Databricks, or update anything, you can just update configuration json in the repository, or variables stored on Azure DevOps Pipeline, which we will cover in the next steps. Azure DevOps Pipeline will take care of everything else for you.

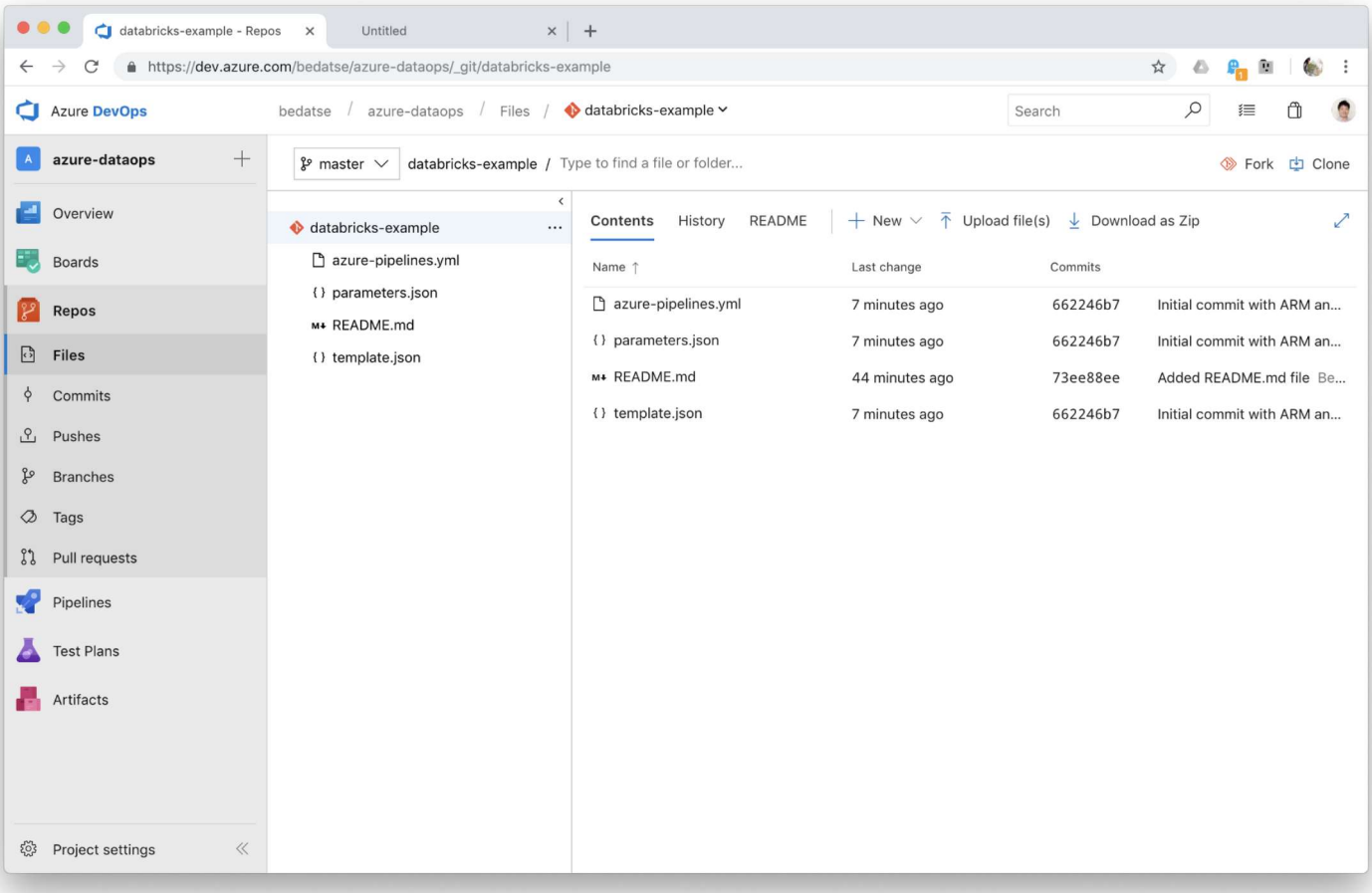
1–1. Copy `template.json parameters.json azure-pipeline.yml notebook-run.json.tmpl` from this commit of the example repository, put them into your repository local folder.

1–2. Stage the changed file in git, commit and push it onto the Azure Repo.

```
$ git add -A

$ git commit -m '<your-commit-message>'

$ git push
```

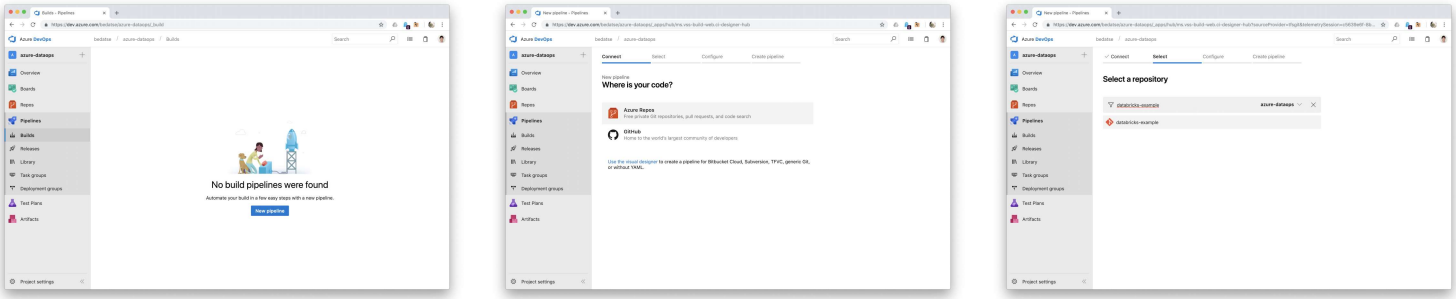


1–2. Commit and Push infrastructure code and build pipeline code onto repository

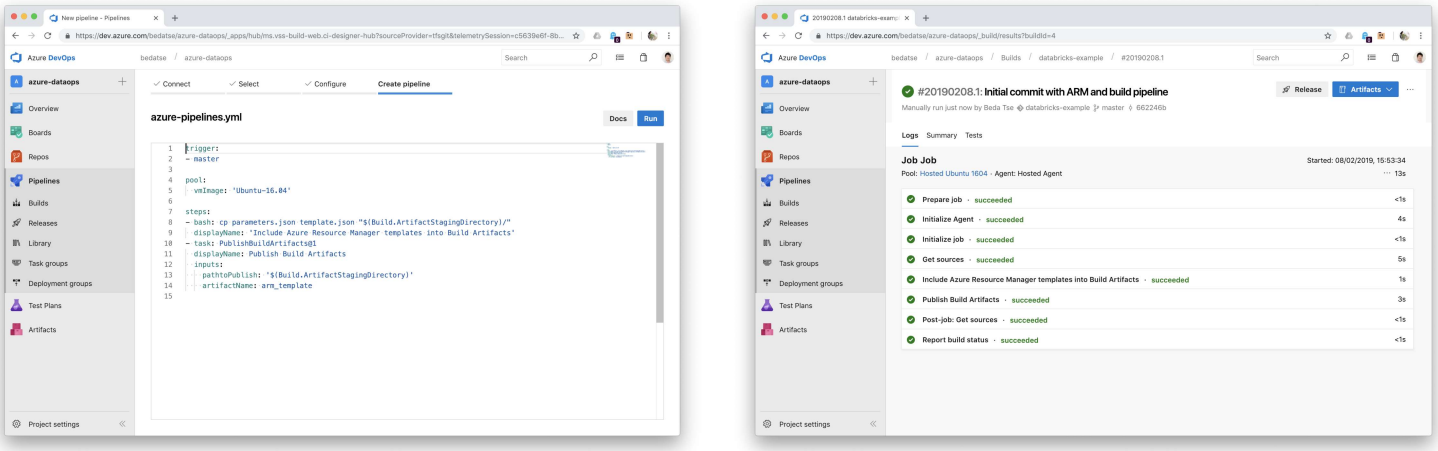


1–2. After pushing code back into repository, it should look like this.

1–3. Create your build pipeline, go to **Pipelines > Builds** on the sidebar, click **New Pipeline** and select **Azure DevOps Repo**. Select your repository and review the pipeline `azure-pipeline.yml` which has already been uploaded in step 1–2. Click **Run** to run the build pipeline for the first time.



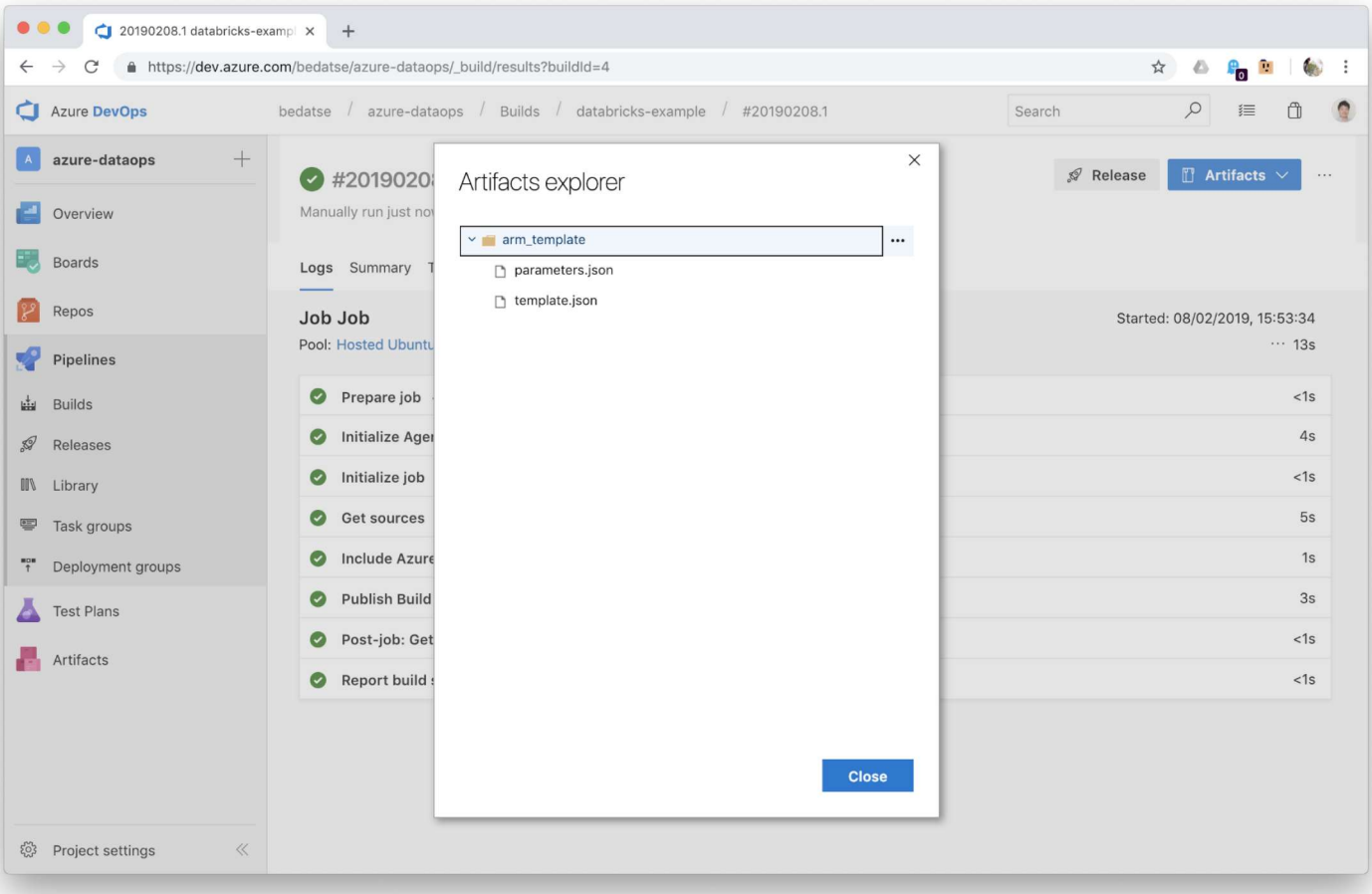
1–3–1 Create new Build Pipeline



1–3–2 Review the content of the pipelines and execution result

The build pipeline currently only do one thing, which is to pack the Azure Resource Manager JSONs into a build artifact, which can be consumed on later steps for deployment. Let take a look what is inside of the artefact now.

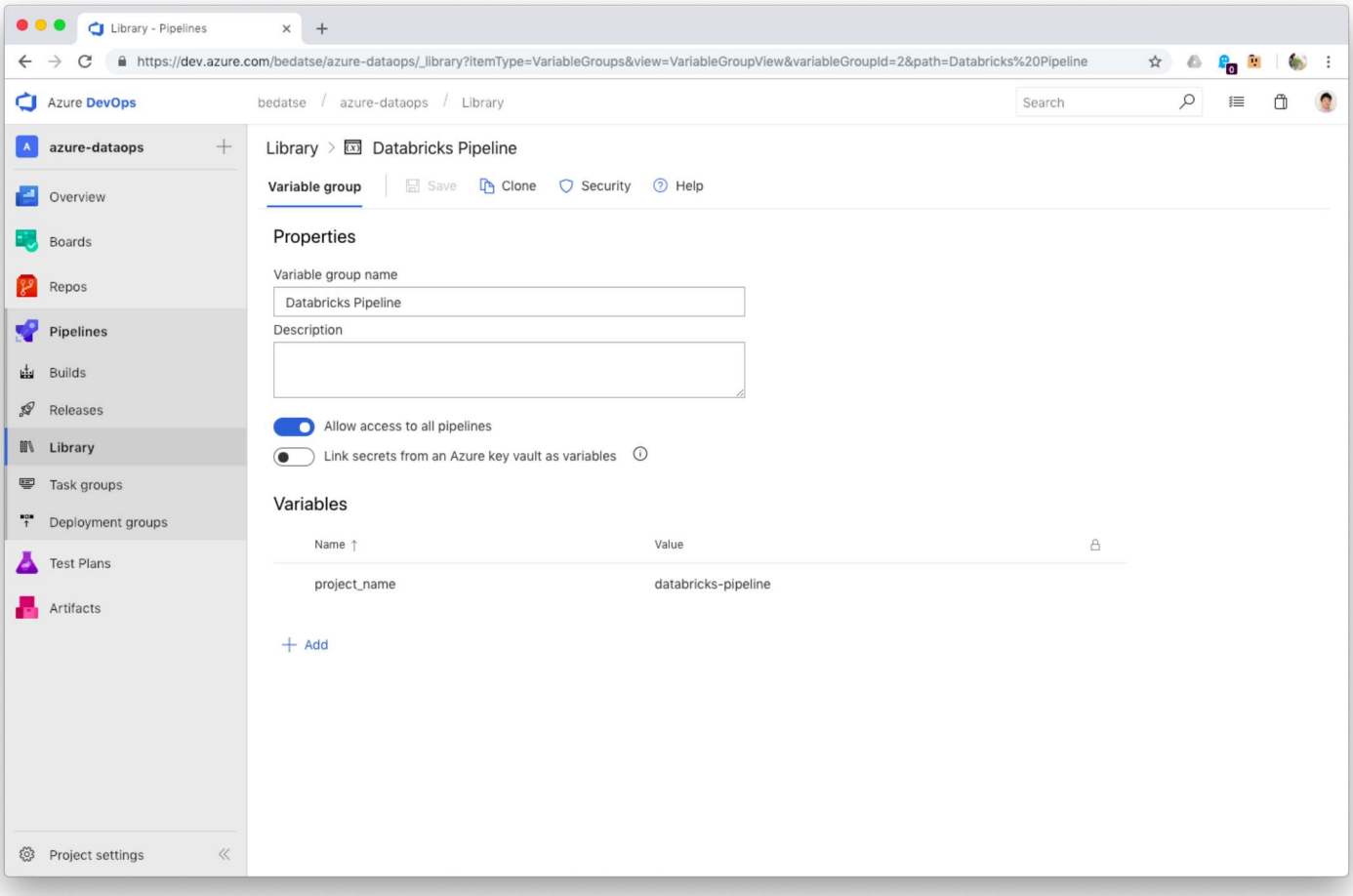
In your build, click **Artifacts** > **arm_templates**. The details of the artifact will be displayed.



The artifact **arm_template** contains ARM JSON files.

1–4. Create variable group for your deployment. You don't want to hardcode your variables inside the pipeline, such that you can make it reusable in another project or environment with least effort. First, let's create a group for your Project, storing all variables that would be the same across all environments.

Go to **Pipelines** > **Library**, Click on **+Variable group**. Type in your variable group name, as an example, we are using *Databricks Pipeline* as the variable group name. Add a variable with *Name* `project_name` with *Value* `databricks-pipeline` . Save the changes after you are done.

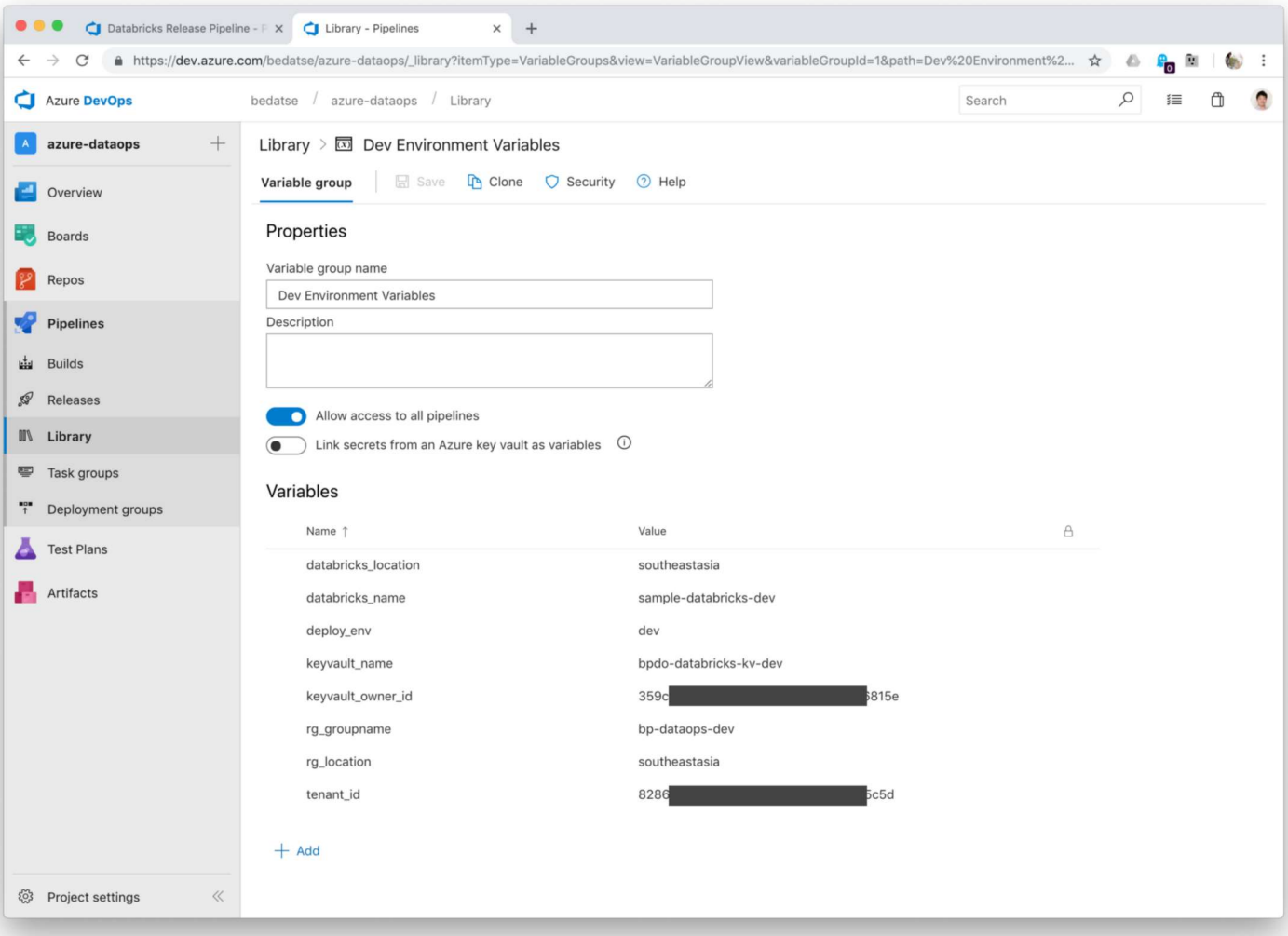


1–4–1 Project-wide Variable Group

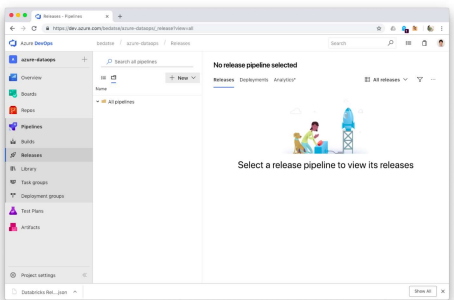
Create another group named *Dev Environment Variables*, this one will have more variables in it, as listed below.

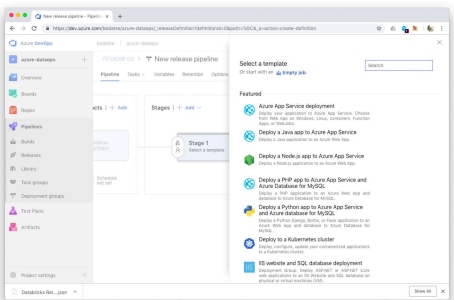
- `databricks_location` : <databricks location>
- `databricks_name` : <databricks name>
- `deploy_env` : <deployment environment name>
- `keyvault_owner_id` : <your user object ID in Azure AD>
- `keyvault_name` : <key vault name for storing databricks token>
- `rg_groupname` : <resource group name>
- `rg_location` : <resource group location>

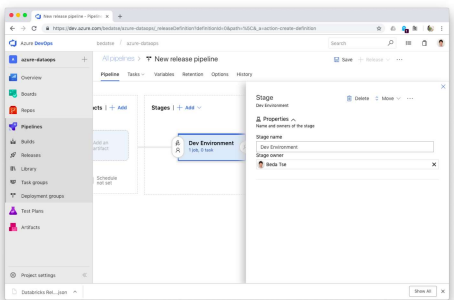
- `tenant_id` :<your Azure AD tenant ID>



1–5. Create a new release pipeline, Go to **Pipelines** > **Releases**, click on **+New**. Select start with an ***Empty job*** when you are asked to select a template. Name your stage ***Dev Environment***, in future tutorials, we will be cloning this stage for ***Staging Environment*** and ***Production Environment***.



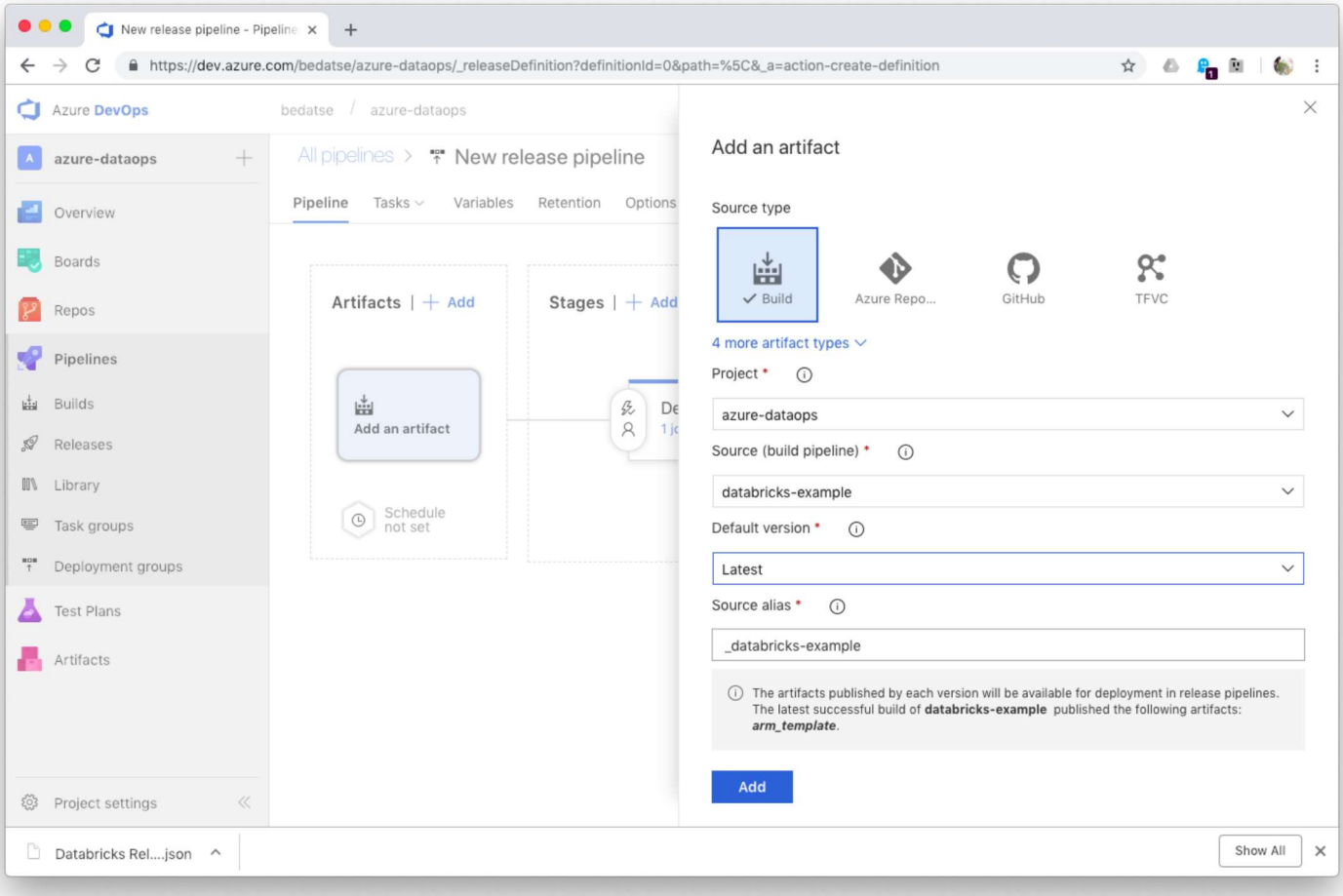




1–5–1 Create New Release Pipeline

Add your build artifact from your repository as the source artifact. In this example, we will add from databricks example. Click **+Add** next to ***Artifacts***.

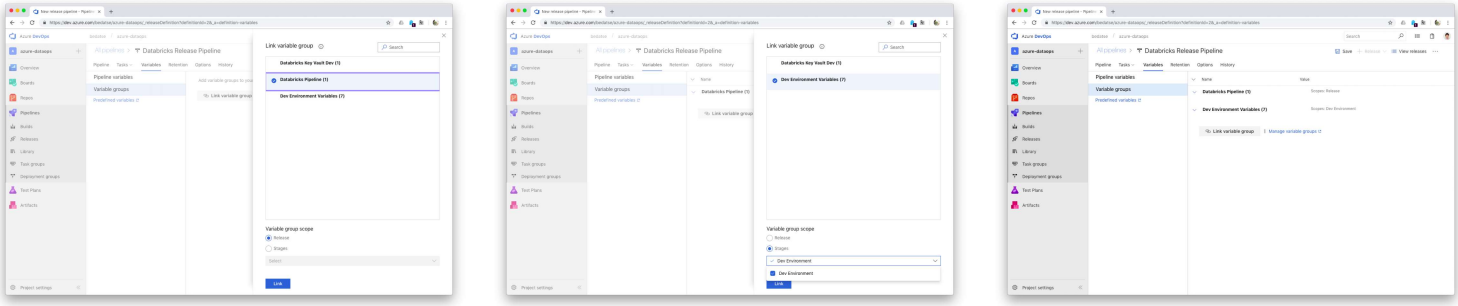
Select **Build** as ***Source type***, select your ***Project*** and ***Source***. Select **Latest** as ***Default version***, keep ***Source alias*** unchanged. Click **Add** when you are done.



1–5–4 Add an artifact to the release pipeline as trigger.

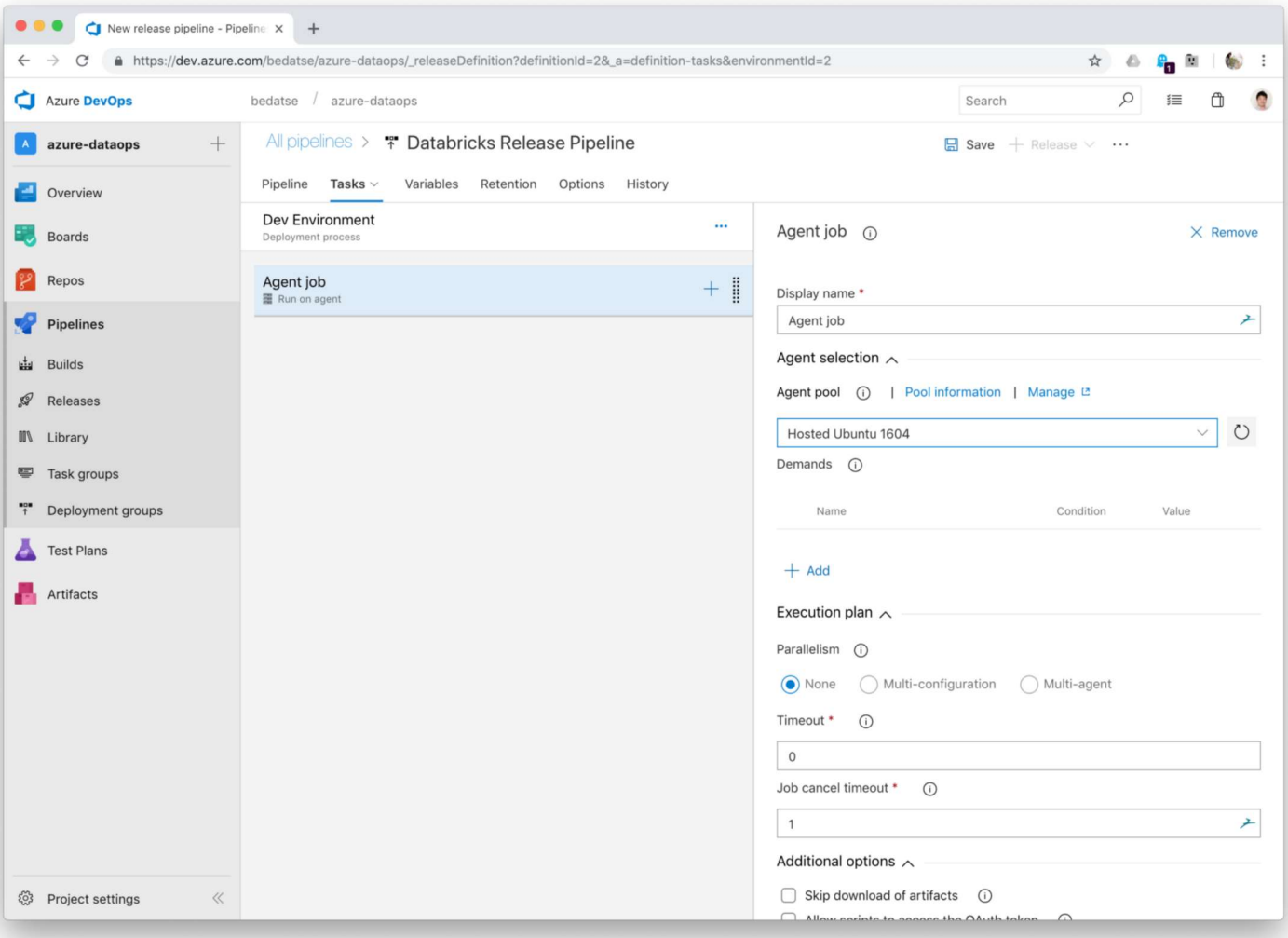
Before moving onto specifying *Tasks* in the ***Dev Environment Stage***, let's link the variable group with the release pipeline and the Dev Environment Stage.

Go to **Variables** > **Variable groups**, Click **Link variable group**. Link your ***Databricks Pipeline*** created in step 1–4 with scope set to ***Release***. Link your ***Dev Environment Variables*** created in step 1–4 with scope set to ***Stages***, and apply to ***Dev Environment*** stage.



1–5–5 Link Databricks Pipeline Project Variable Group with Release scope

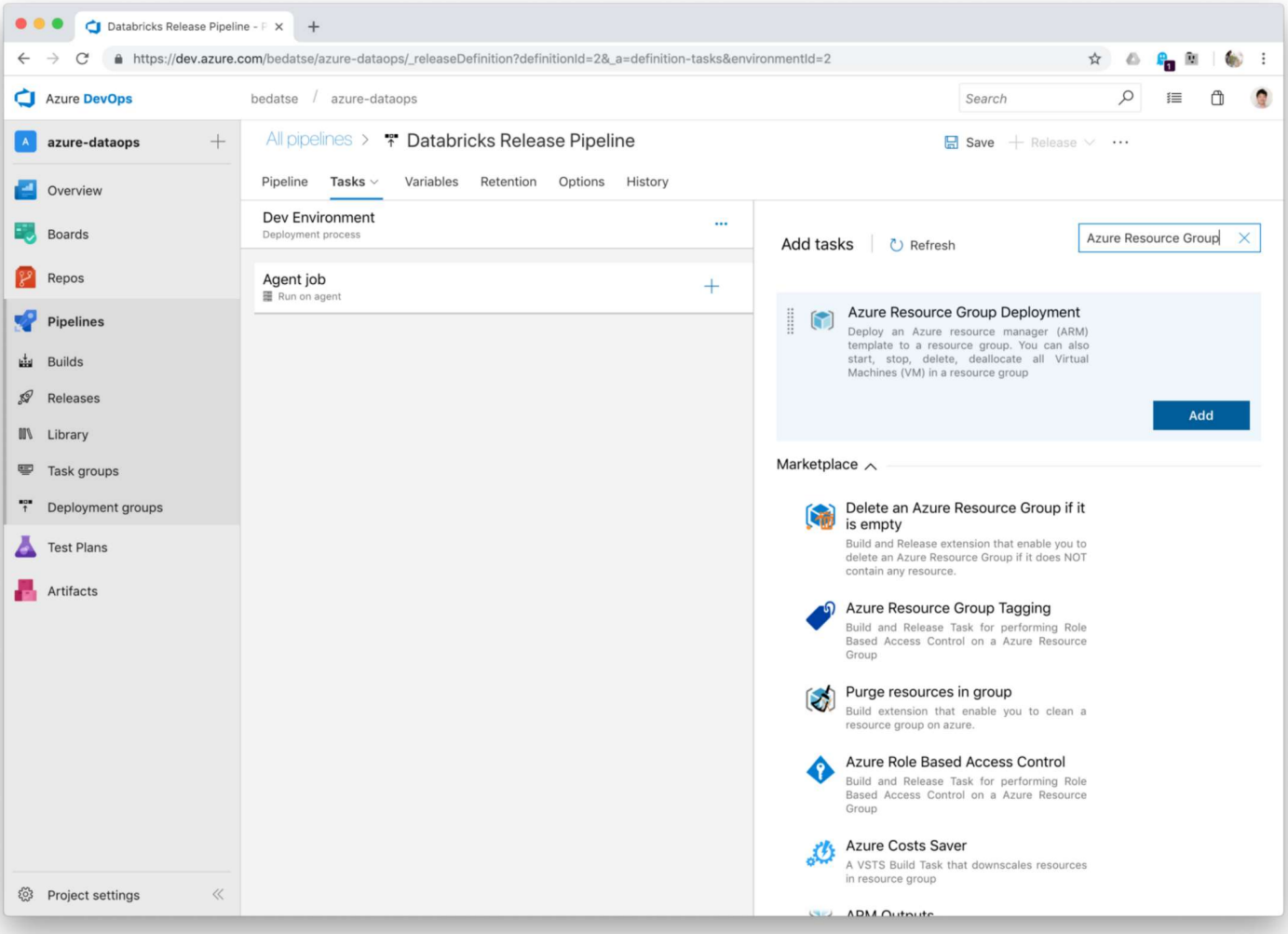
With variable groups linked, we are ready for setting up tasks in the ***Dev Environment*** stage. Click **Tasks** > **Agent job**, review the settings in there.



1–5–6 Empty Agent Job

Click the + sign next to the Agent job, add an **Azure Resource Group Deployment**

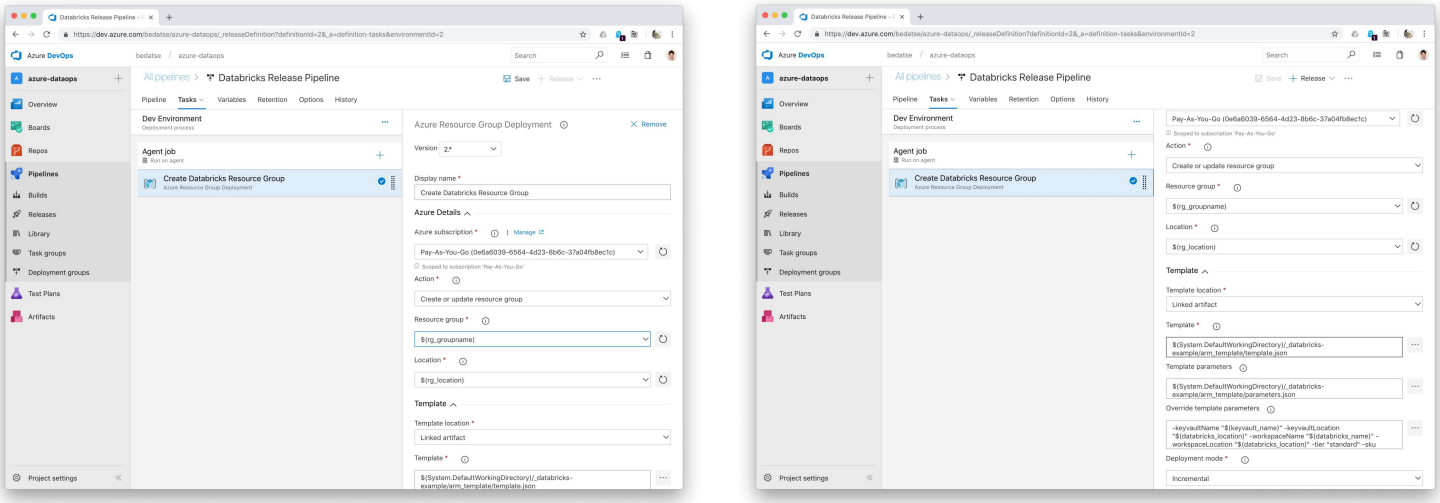
task.



1–5–7 Add task to Deployment stage

Now, we can configure **Azure Resource Group Deployment** task. Select the service connections from previous step, keep ***Action*** as **Create or update resource group**. For ***Resource group name*** and ***location***, type in `$(rg_groupname)` and `$(rg_location)` respectively.

For ***Template*** and ***Template Parameters***, click on the ***More*** action button next to the text field, select **template.json** and **parameters.json** inside *_databricks-example/arm_template*. Before we set our override template parameters, let us set the ***Deployment mode*** to **Incremental**.



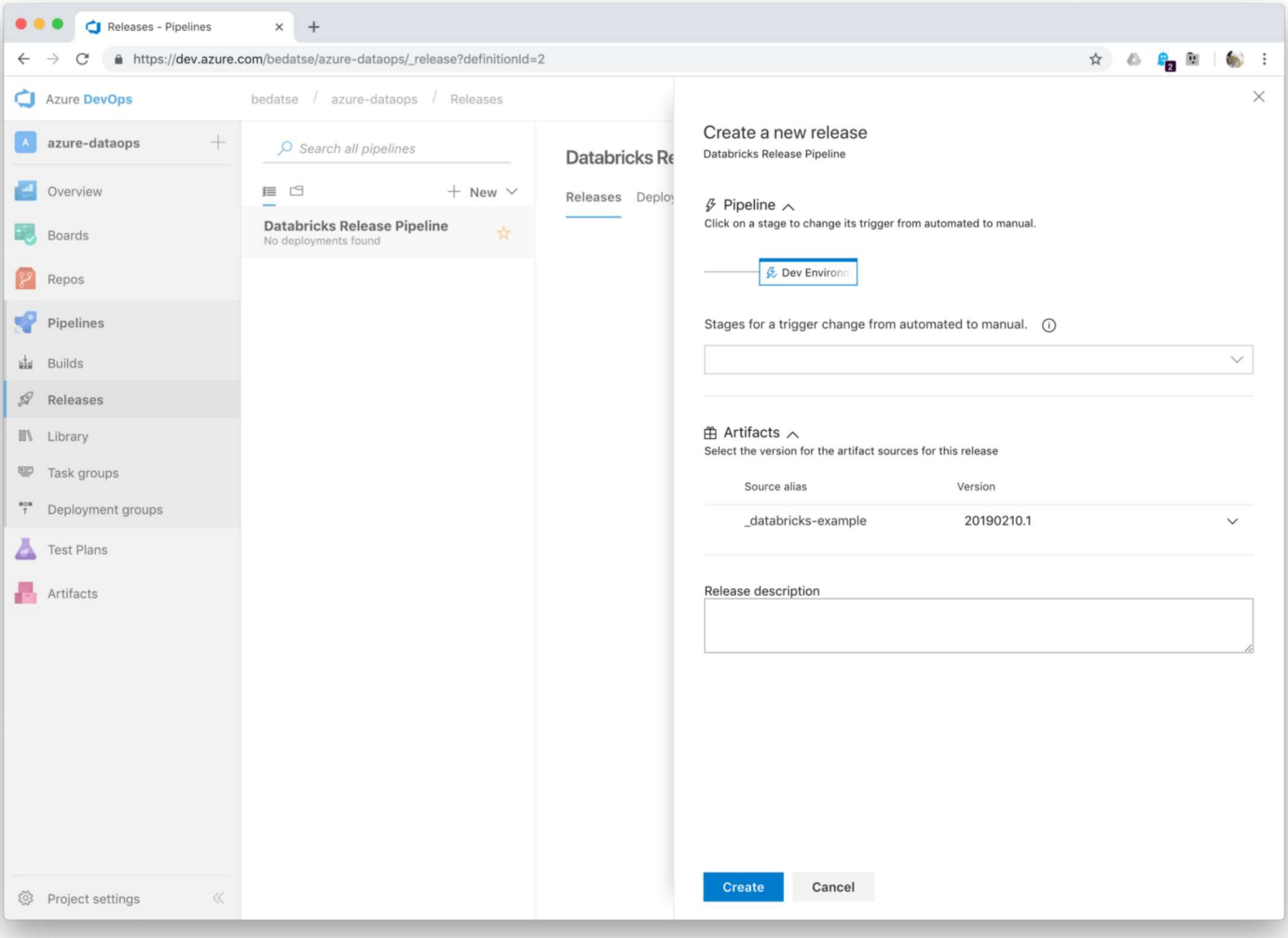
1–5–8 Configuring Resource Group Task

The most challenging part in this section is ***Override template parameters***, we have made this simple for you. Just copy the following snippet into the text field for now. This will allow you to override the default value specified in the *parameters file* by the value specified in *Variable Group*.

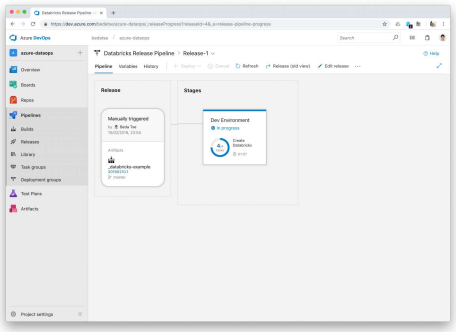
```
-keyvaultName "$(keyvault_name)" -keyvaultLocation
"$(databricks_location)" -workspaceName "$(databricks_name)" -
workspaceLocation "$(databricks_location)" -tier "standard" -sku
"Standard" -tenant "$(tenant_id)" -enabledForDeployment false -
enabledForTemplateDeployment true -enabledForDiskEncryption false -
networkAcls
{"defaultAction":"Allow","bypass":"AzureServices","virtualNetworkRule
s":[],"ipRules":[]}
```

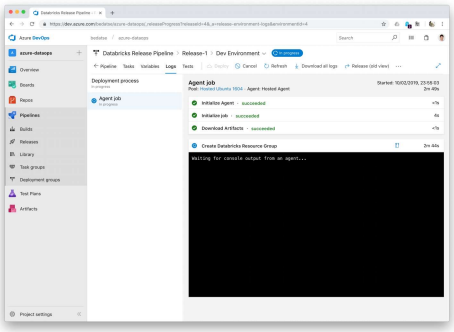
After all these, save your release pipeline and we are ready to create a release.

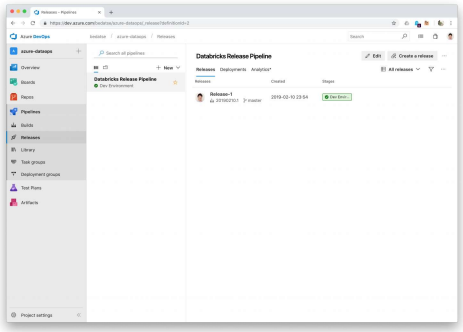
1–6. Create a release by going back to **Pipelines** > **Releases** screen. Click on **Create a release** button, then click Create. Your release will then be queued.



1–6–1 Create a release



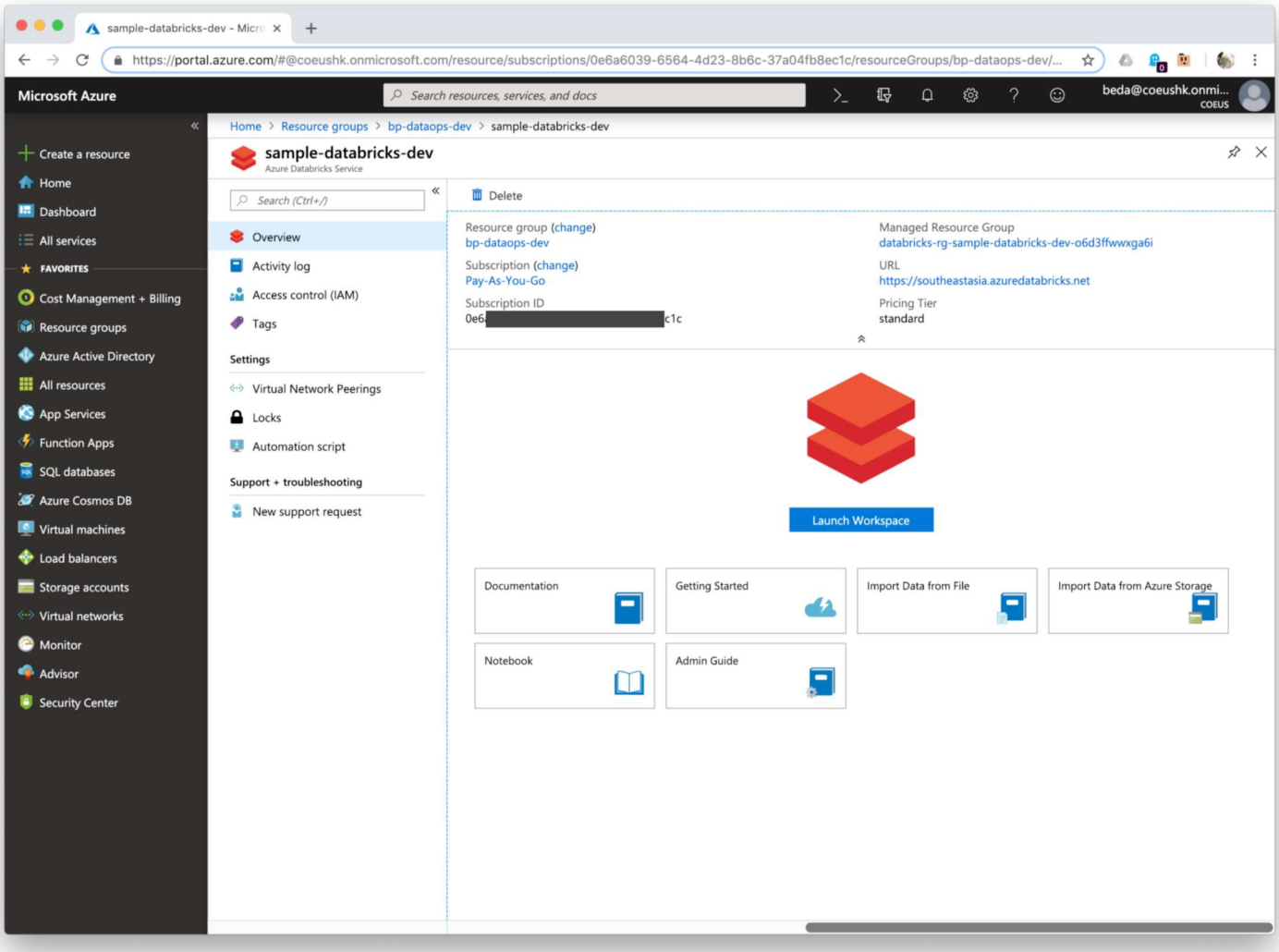




1–6–2 Wait for provisioning result

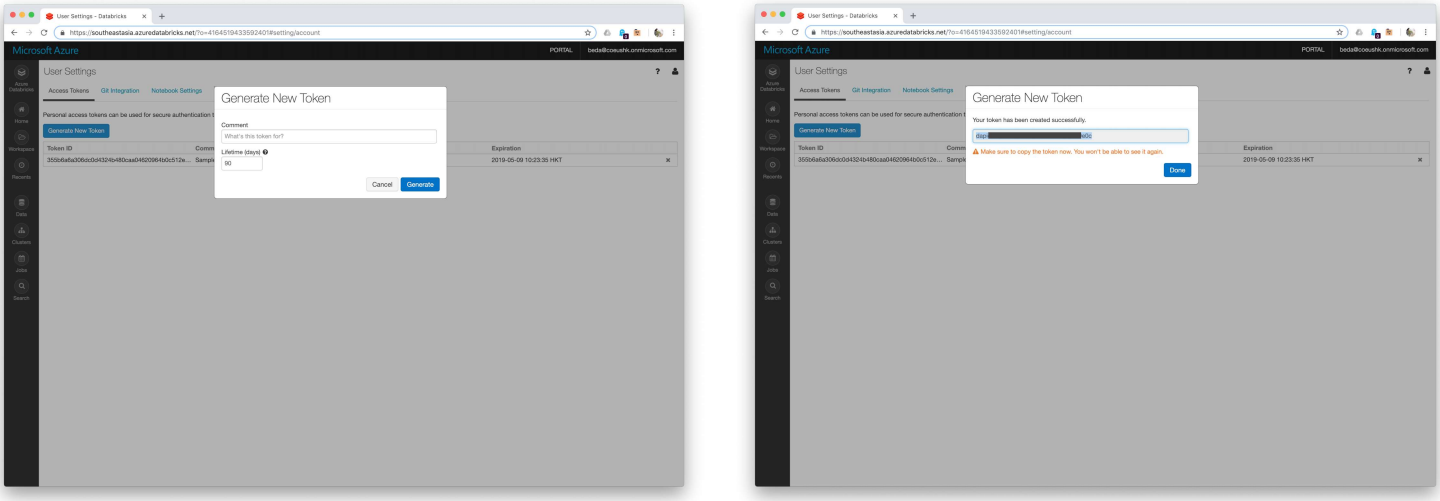
Step 2: Generate Azure Databricks API Token and store the token into Azure Key Vault

2–1. Access Azure Portal, look for the newly created resource group and Databricks, and launch Databricks Workspace as usual.



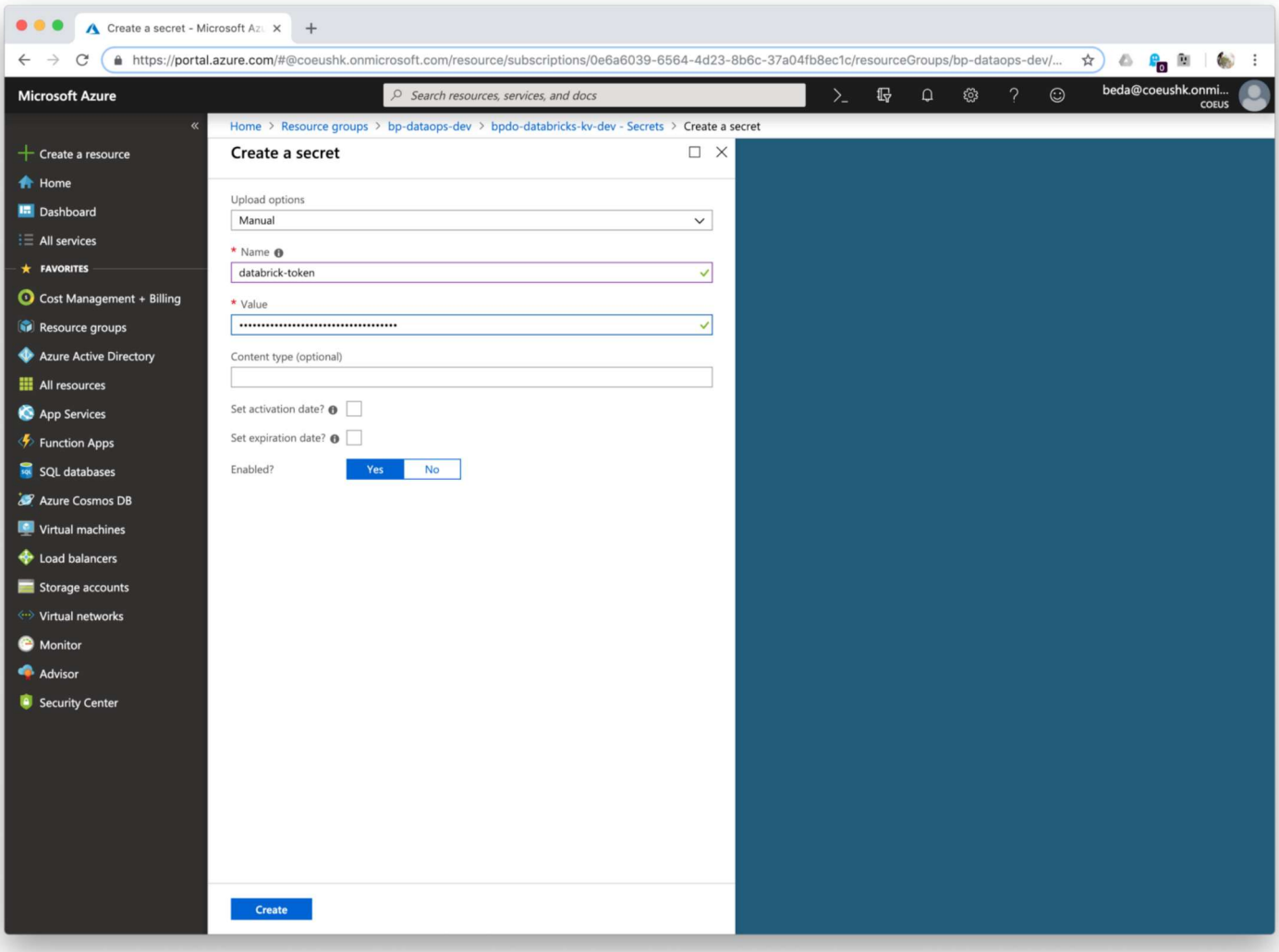
2–1 Access Databricks Workspace via Azure Portal

2–2. After logging into the workspace, click the **user icon** on the top right corner, select **User Settings**. Click **Generate New Token**, give it a meaningful comment and click **Generate**. We will use this token in our pipeline for Notebook deployment. Your token will only be displayed once, make sure you do not close the dialog or browser before you have copied it into key vault.



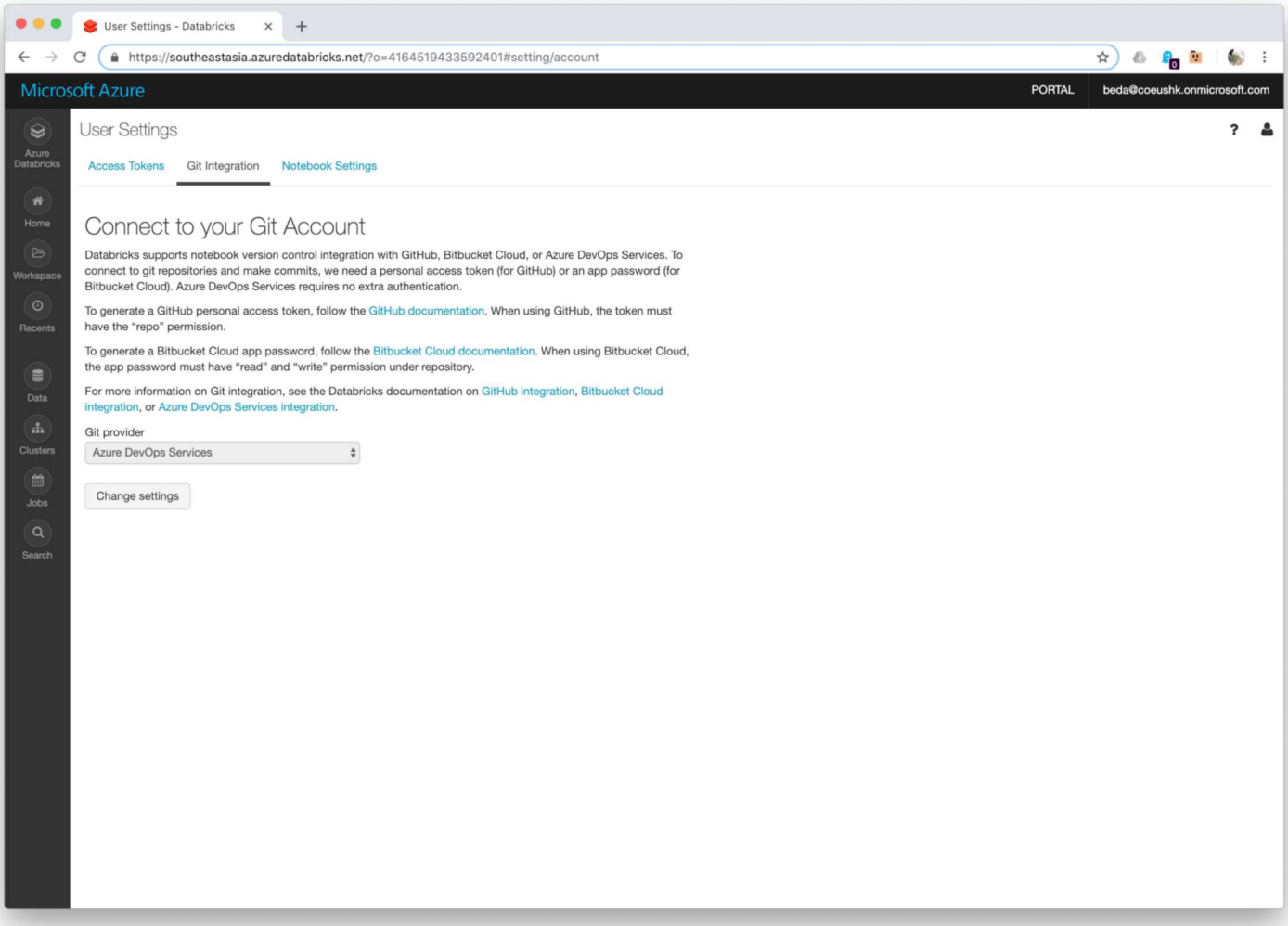
2–2 Generate new token

2–3. In another browser window, open Azure Portal, navigate to Azure Key Vault under the newly created Resource group. Access the **Secrets** tab, click **Generate/Import**. Set the ***Name*** as `databricks-token` , and copy the newly generated token into ***Value***. Click **Create** to save the token inside Azure Key Vault securely. Now you can safely close the



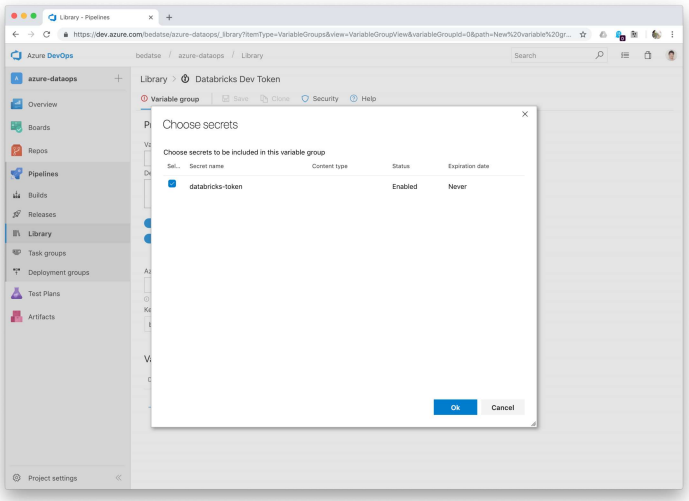
2–3 Save Databricks token into Azure Key Vault

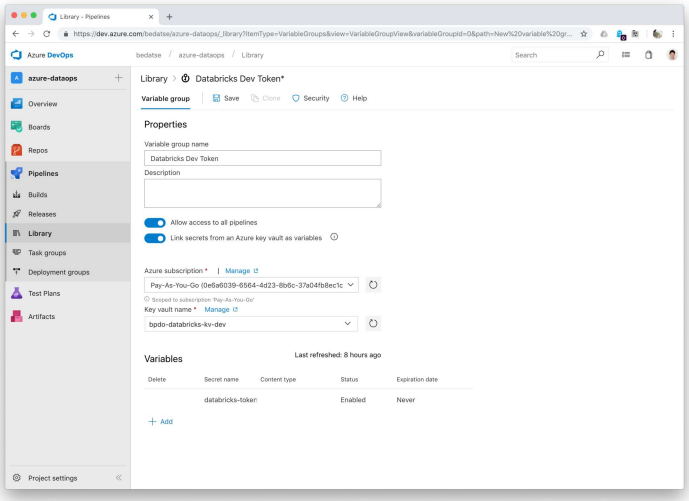
2–4. While we are in the Databricks workspace, also go to **Git Integration** tab and check ***Git provider*** setting, make sure it is set to **Azure DevOps Services**, or to the repository of your choice.



2–4 Ensure git integration settings is set to Azure DevOps Services

2–5. Go to Azure DevOps Portal, go to **Pipelines** > **Library**. Click **+Variable Group** to create new Variable Group. This time we are linking an Azure Key Vault into Azure DevOps as variable group. This allows Azure DevOps to obtain token from Azure Key Vault securely for deployment. Name the variable group as ***Databricks Dev Token***, select **Link secrets from an Azure key vault as variables**. Select the correct Azure subscription service connections and Key vault respectively. Click **+Add** and select *databricks-token* in the ***Choose secrets*** dialog. Click **Save**.

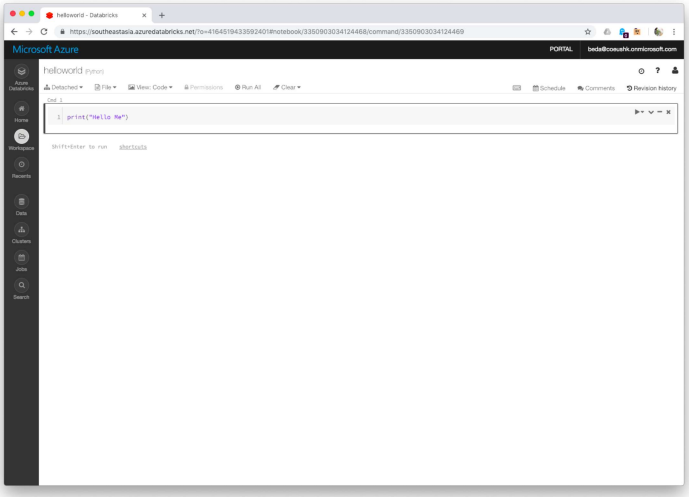


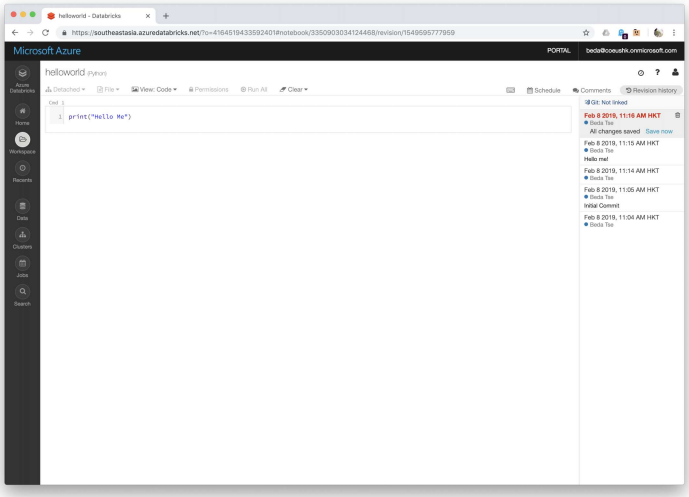


Step 3: Link your workbook development with Source code repository

Databricks workspaces integrate with git seamlessly as an IDE. We have in previous steps 2–4 set up the integration between Databricks and a source code repository. We can now link the workbook with the repository and commit into a repository directly.

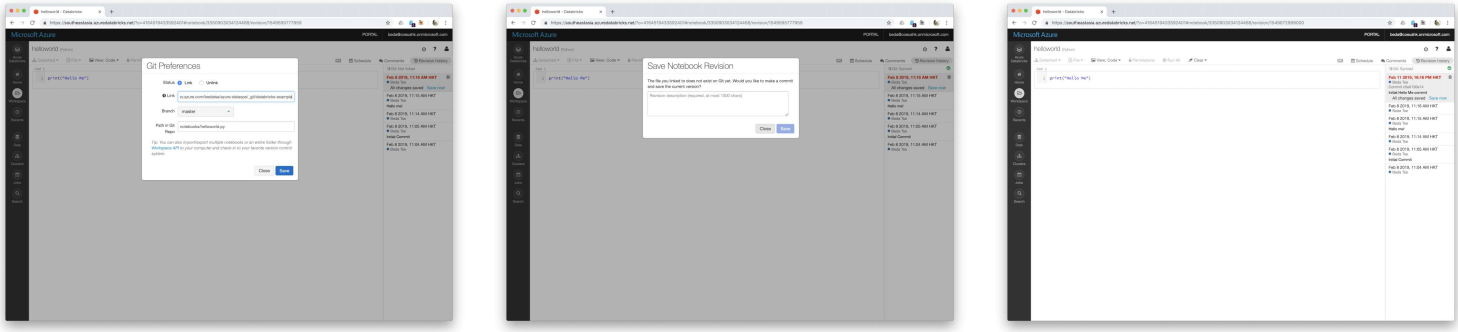
3–1. Open your notebook as usual, notice **Revision history** on the right top section of the screen. Click on **Revision history** to bring up the version history side panel.



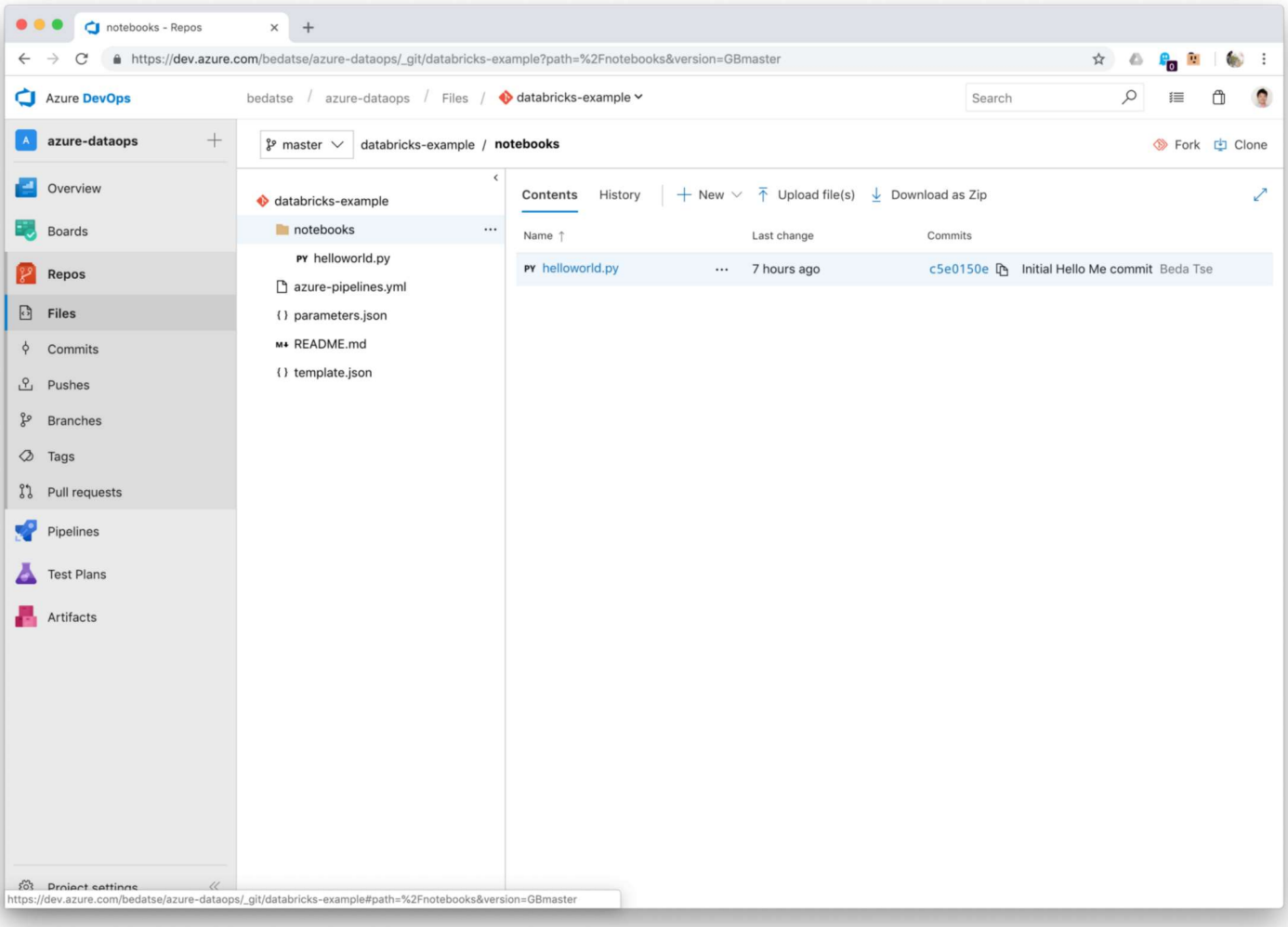


3–1 Version history side panel.

3–2. Click on Git: **Not Linked** to update ***Git Preferences***. Link your workbook to *Azure DevOps Repo*, which should be the URL of your git repository, and set the **Path in Git Repo** to the location which you want Databricks to save your notebook inside the repository.



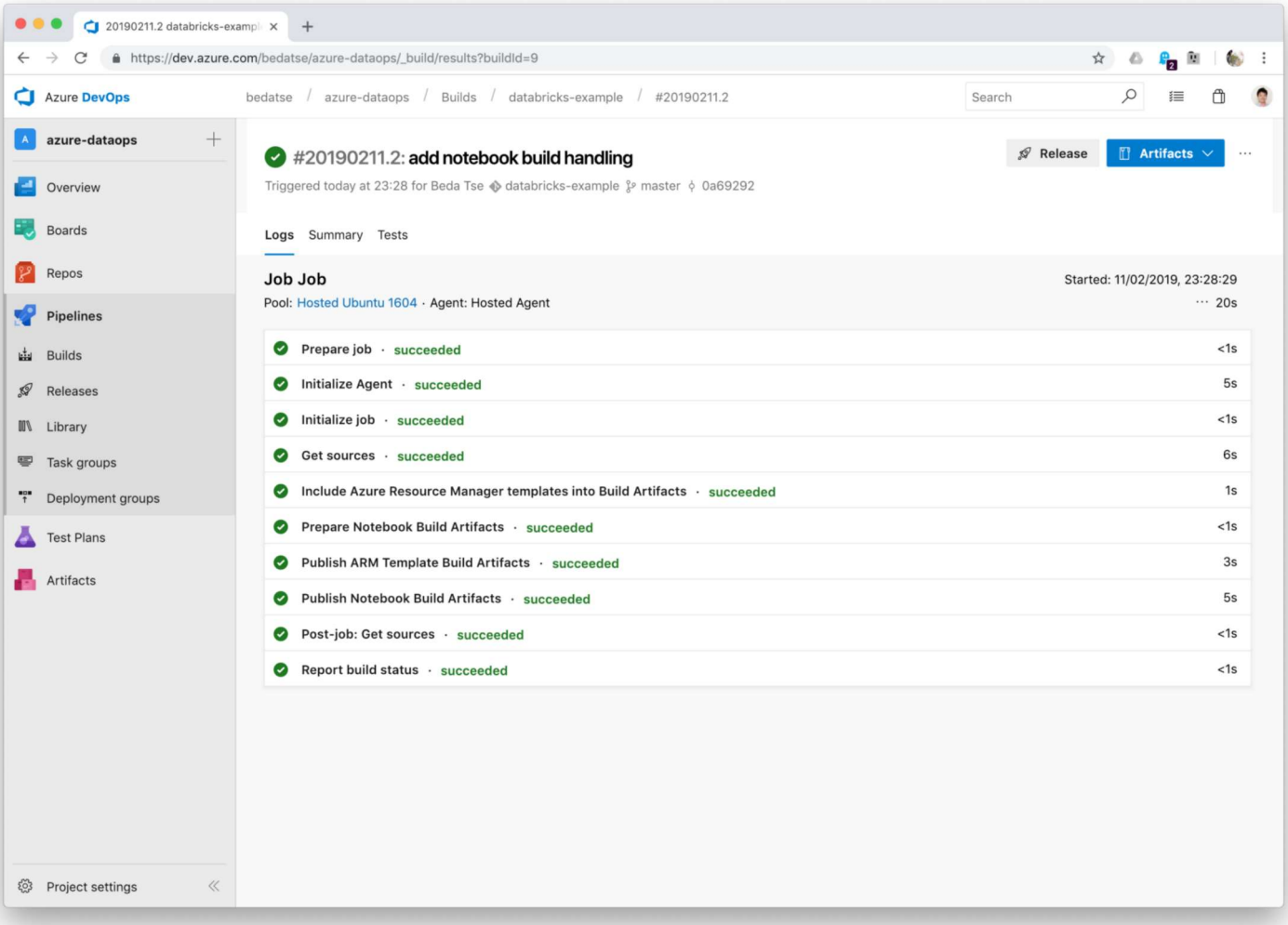
3–2 Link workbook with repository



3–2 Committed change pushed into git repo.

3–3. The committed change is pushed into git repository. What does that mean? That means it will trigger the build pipeline. With a little bit of further configuration, we can actually update the build pipeline to package this notebook into a deployable package, and use it to trigger a deployment pipeline. Now download the ***azure-pipelines.yml***

from this commit, replace the original ***azure-pipelines.yml*** from step 1–1, commit and push the change back to the repository.

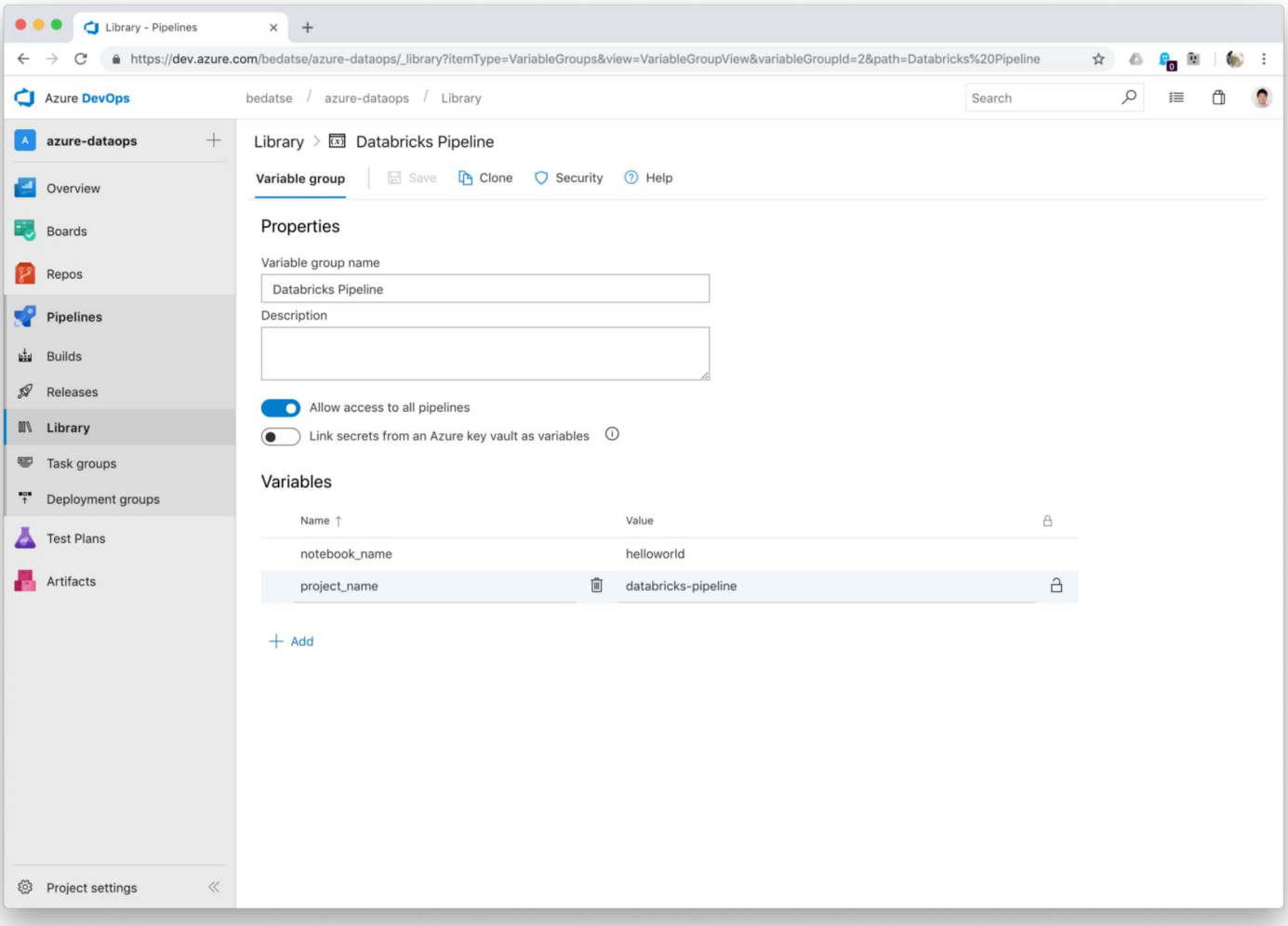


3–3 Build triggered by the pipeline update.

Step 4: Deploy the version controlled Notebook onto Databricks for automated tests

Since we have prepared our notebook build package, let us complete the flow by deploying it onto the Databricks that we have created and execute a run from the pipeline.

4–1. Go to **Pipelines** > **Library**, edit the project based Variable group ***Databricks Pipeline***, we need to specify a variable here such that the release pipeline will pickup from the variable which notebook to deploy. Add `notebook_name` variable with value `helloworld` . Click **Save** after it is done.

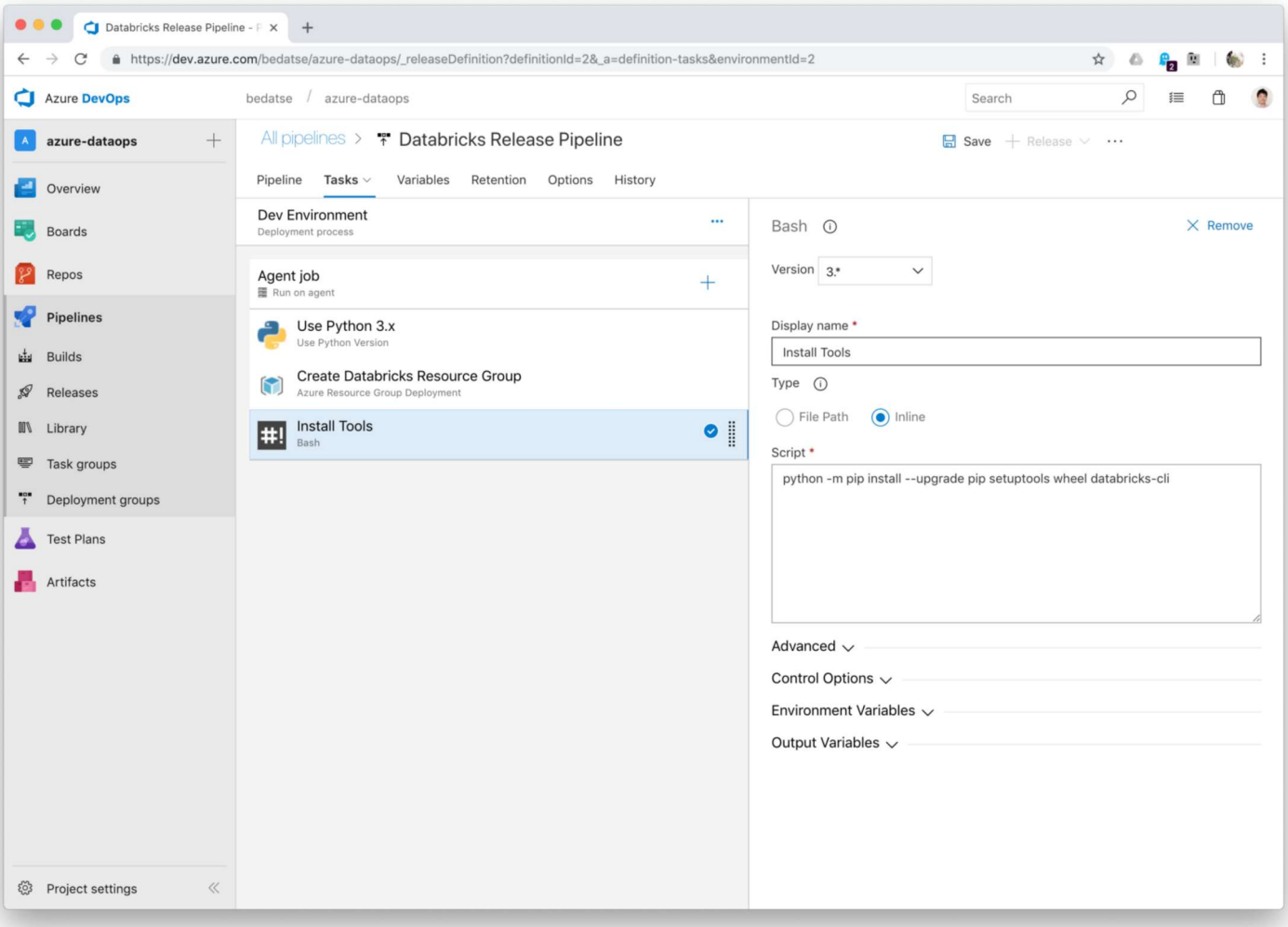


4–1 Update Variable group with notebook_name variable

4–2. Go to **Pipelines** > **Releases**, select ***Databricks Release Pipeline***, Click **Edit**. Navigate to the ***Tasks*** tab. Add ***Use Python Version*** task and drag it above the original ***Create Databricks Resource Group*** task. No further configuration is needed.'

4–3. Add Bash Task at the end of the job. Rename it to *Install Tools*. Select ***Type*** as **Inline**, copy the following scripts to the ***Script*** text area. This is to install the needed python tools for deploying notebook onto Databricks via command line interface.

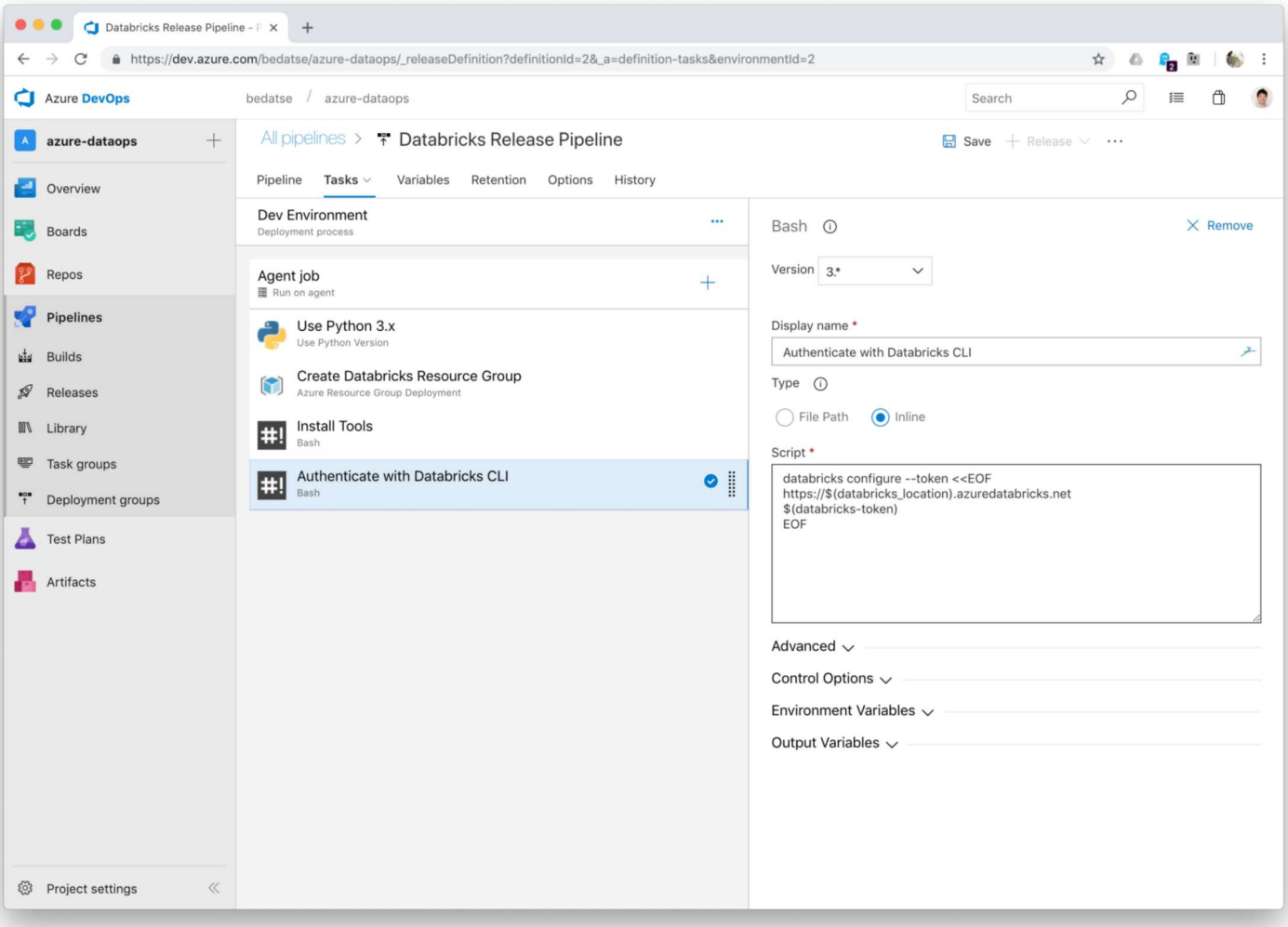
```
python -m pip install --upgrade pip setuptools wheel databricks-cli
```



4–3 Configure Install Tools Task

4–4. Add Bash Task at the end of the job. Rename it to *Authenticate with Databricks CLI*. Select ***Type*** as **Inline**, copy the following scripts to the ***Script*** text area. The variable `databricks_location` is obtained from variable group defined inside the pipeline, while `databricks-token` is obtained from variable group linked with Azure Key Vault.

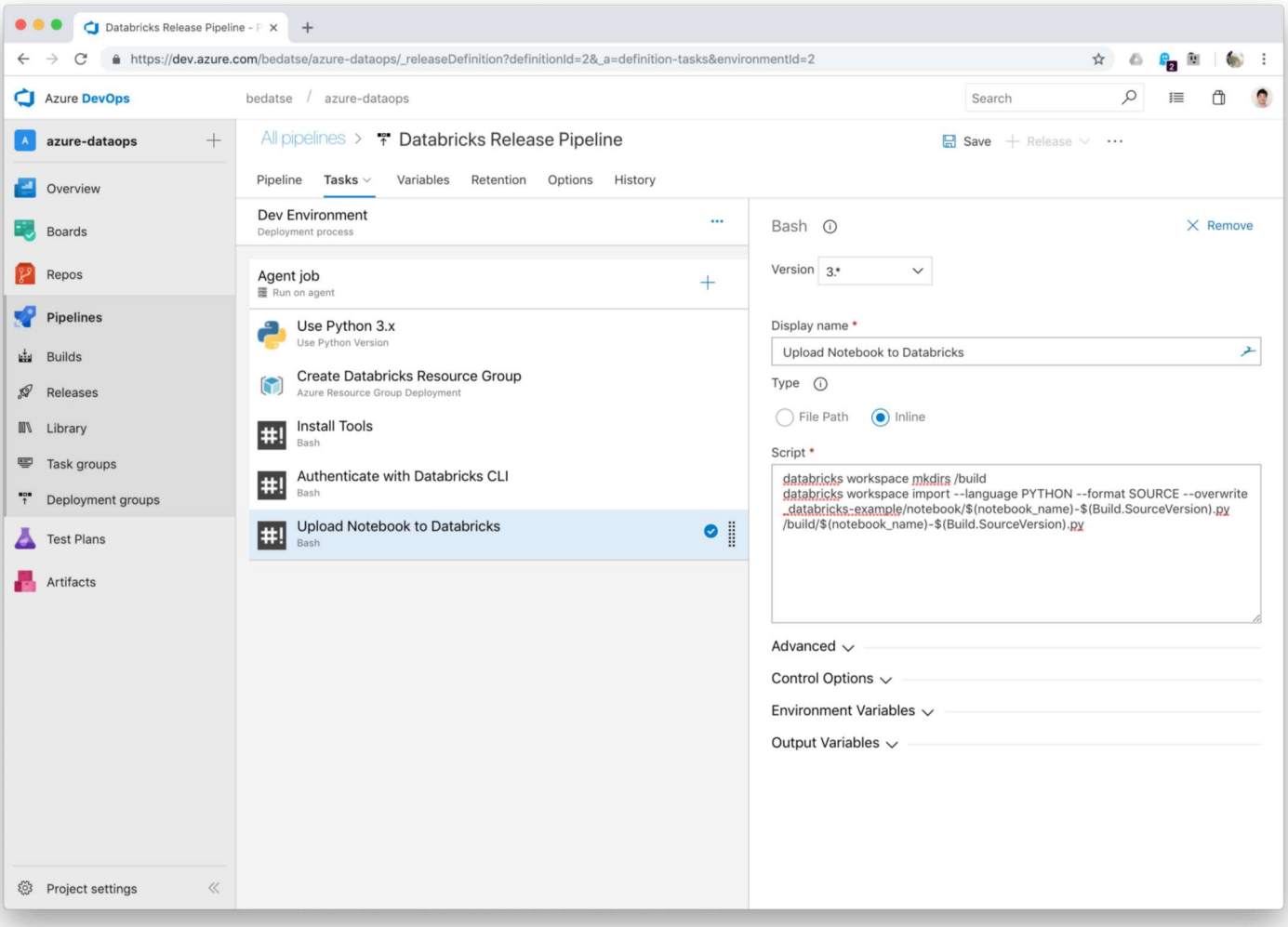
```
databricks configure --token <<EOF
https://$(databricks_location).azuredatabricks.net
$(databricks-token)
EOF
```



4–4 Configure CLI Authentication Task

4–5. Add Bash Task at the end of the job. Rename it to *Upload Notebook to Databricks*. Select ***Type*** as **Inline**, copy the following scripts to the ***Script*** text area. The variable `notebook_name` is retrieved from the release scoped variable group.

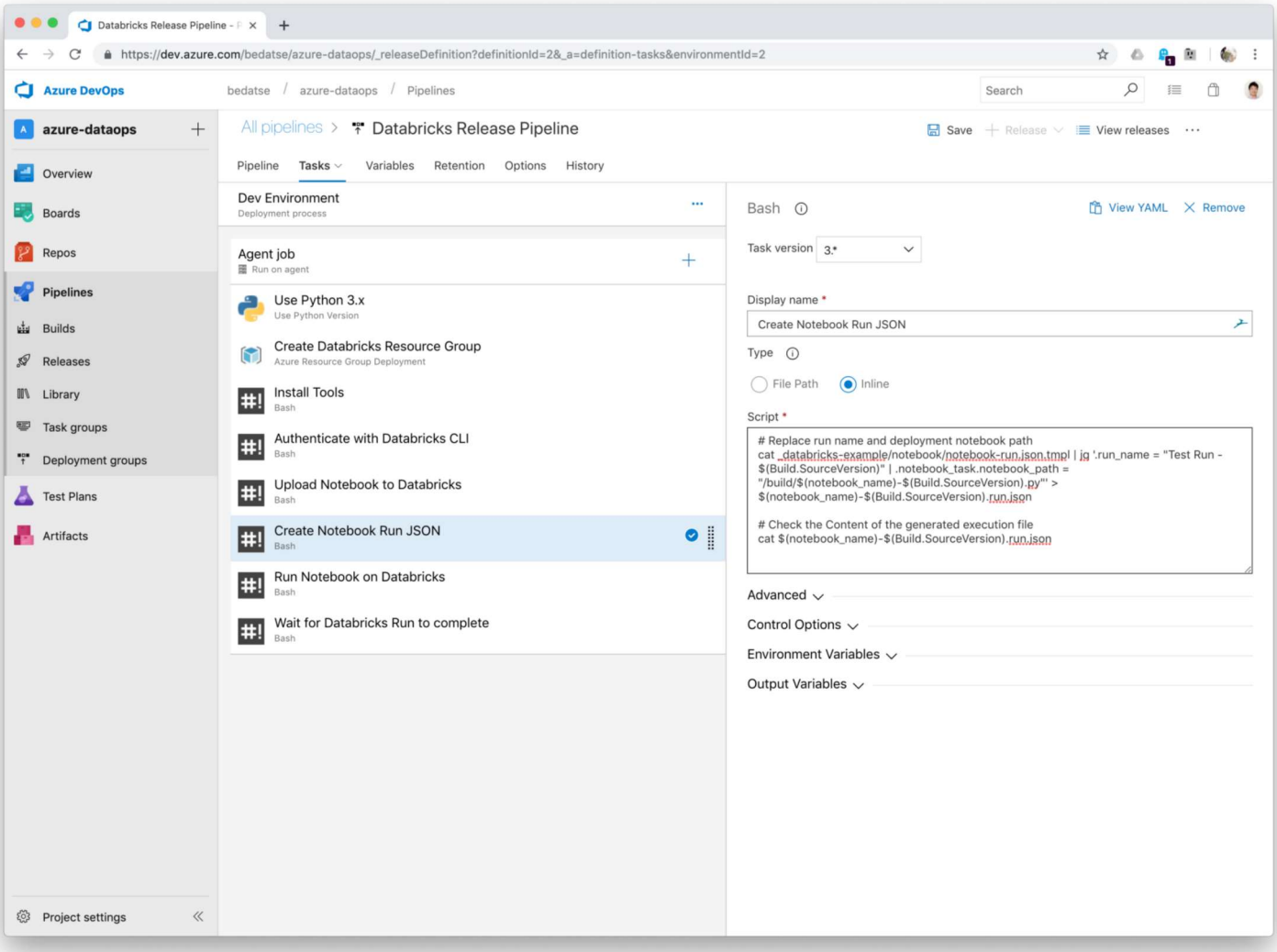
```
databricks workspace mkdirs /build
databricks workspace import --language PYTHON --format SOURCE --
overwrite _databricks-
example/notebook/$(notebook_name)-$(Build.SourceVersion).py
/build/$(notebook_name)-$(Build.SourceVersion).py
```



4–5 Configure Upload Notebook Task

4–6. Add Bash Task at the end of the job. Rename it to *Create Notebook Run JSON*. Select ***Type*** as **Inline**, copy the following scripts to the ***Script*** text area. This is to prepare a job execution configuration for the test run, using the template `notebook-run.json.tmpl`.

```
# Replace run name and deployment notebook path
cat _databricks-example/notebook/notebook-run.json.tmpl | jq
'.run_name = "Test Run - $(Build.SourceVersion)" |
.notebook_task.notebook_path =
"/build/$(notebook_name)-$(Build.SourceVersion).py"' >
$(notebook_name)-$(Build.SourceVersion).run.json

# Check the Content of the generated execution file
cat $(notebook_name)-$(Build.SourceVersion).run.json
```



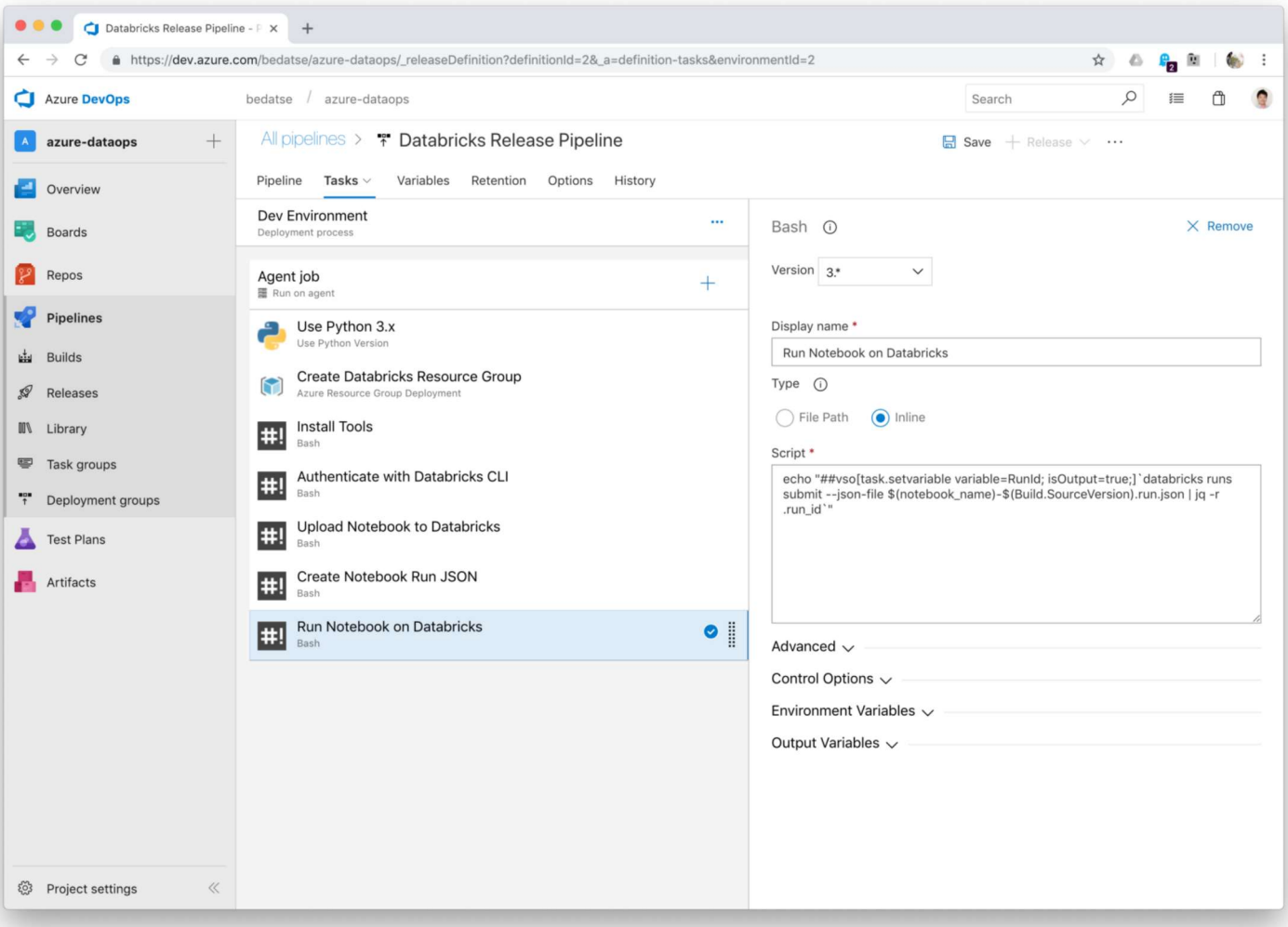
4–6 Configure Notebook Run JSON Creation task

4–7. Add Bash Task at the end of the job. Rename it to *Run Notebook on Databricks*. Select ***Type*** as **Inline**, copy the following scripts to the ***Script*** text area. This is to execute the notebook prepared in the Build pipeline, i.e. committed by you thru the Databricks UI, via Job Cluster.

```
echo "##vso[task.setvariable variable=RunId;
isOutput=true;]`databricks runs submit --json-file
$(notebook_name)-$(Build.SourceVersion).run.json | jq -r .run_id`"
```

You might have noticed there is weird template here with `##vso[task.setvariable variable=RunId; isOutput=true;]` . This is to save the `run_id` from the output of the

`databricks runs submit` command into Azure DevOps as variable `RunId` , such that we can reuse that run id in next steps.

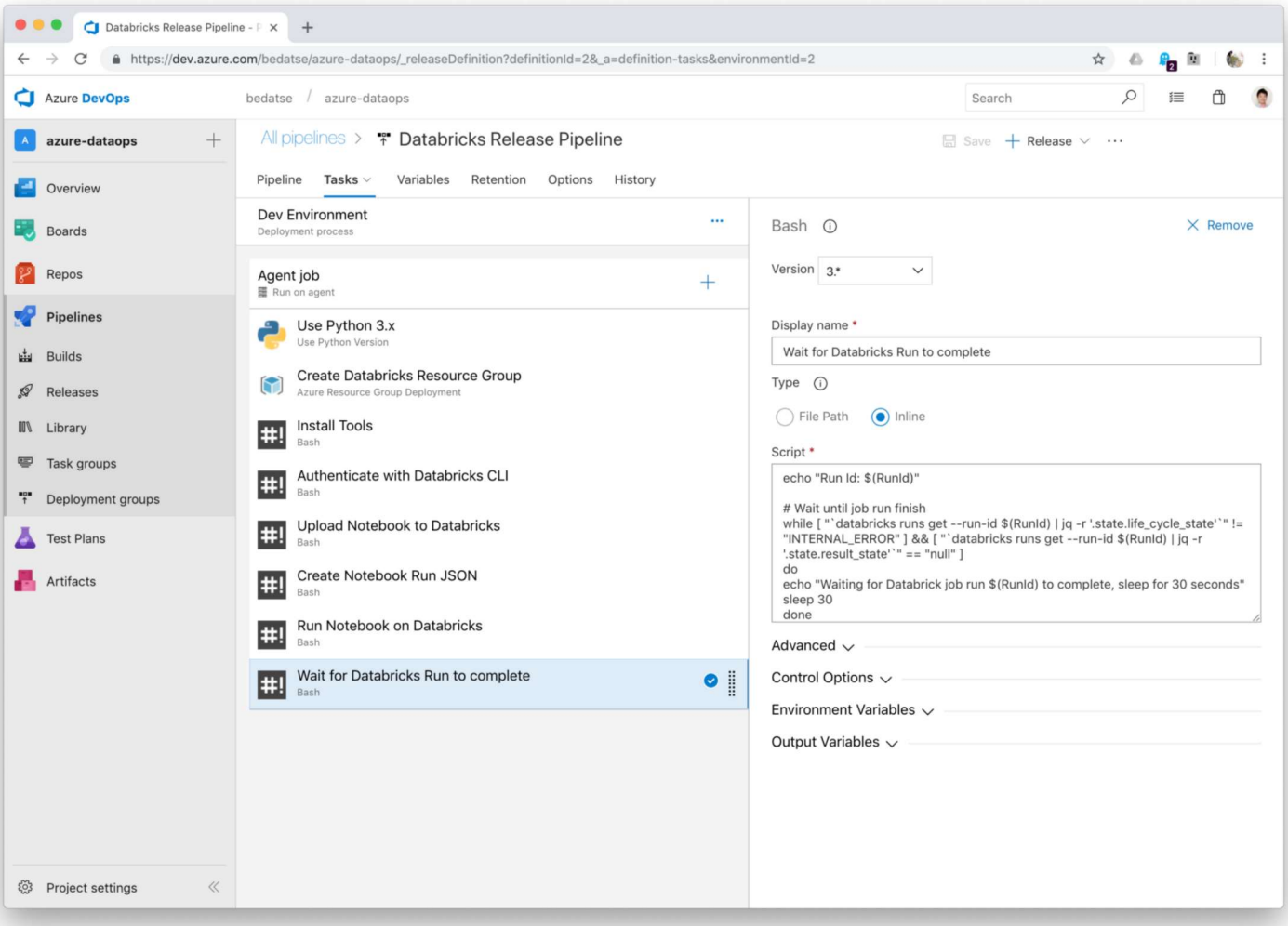


4–7 Configure Notebook Execution Task

4–8. Add Bash Task at the end of the job. Rename it to *Wait for Databricks Run to complete*. Select ***Type*** as **Inline**, copy the following scripts to the ***Script*** text area. This is to wait for the previously executing Databricks job and get the execution state from the run result.

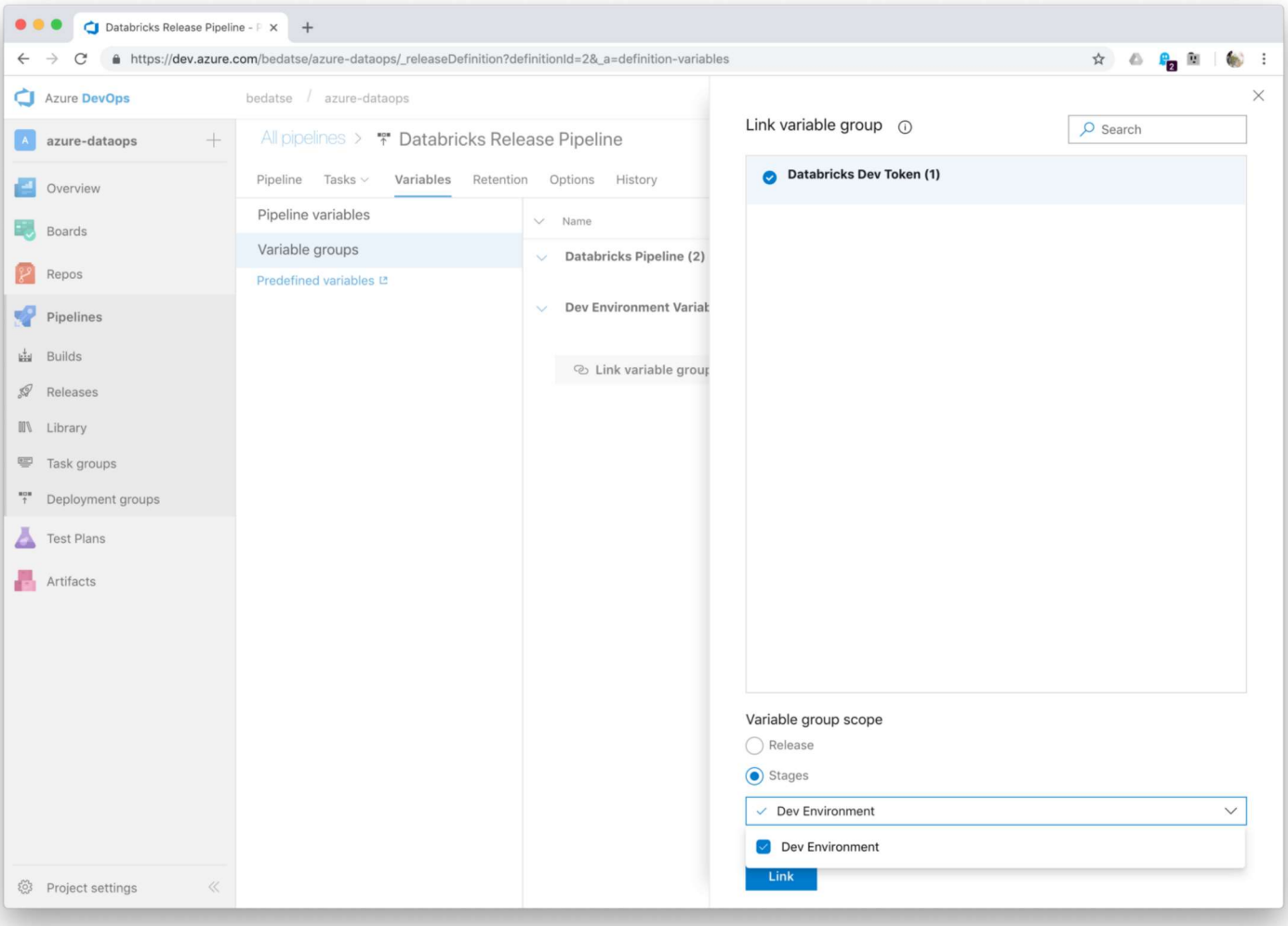
```
echo "Run Id: $(RunId)"

# Wait until job run finish
while [ "`databricks runs get --run-id $(RunId) | jq -r
'.state.life_cycle_state'`" != "INTERNAL_ERROR" ] && [ "`databricks
runs get --run-id $(RunId) | jq -r '.state.result_state'`" == "null"
```

```
]
do
echo "Waiting for Databrick job run $(RunId) to complete, sleep for
30 seconds"
sleep 30
done

# Print Run Results
databricks runs get --run-id $(RunId)

# If not success, report failure to Azure DevOps
if [ "`databricks runs get --run-id $(RunId) | jq -r
'.state.result_state'`" != "SUCCESS" ]
then
echo "##vso[task.complete result=Failed;]Failed"
fi
```



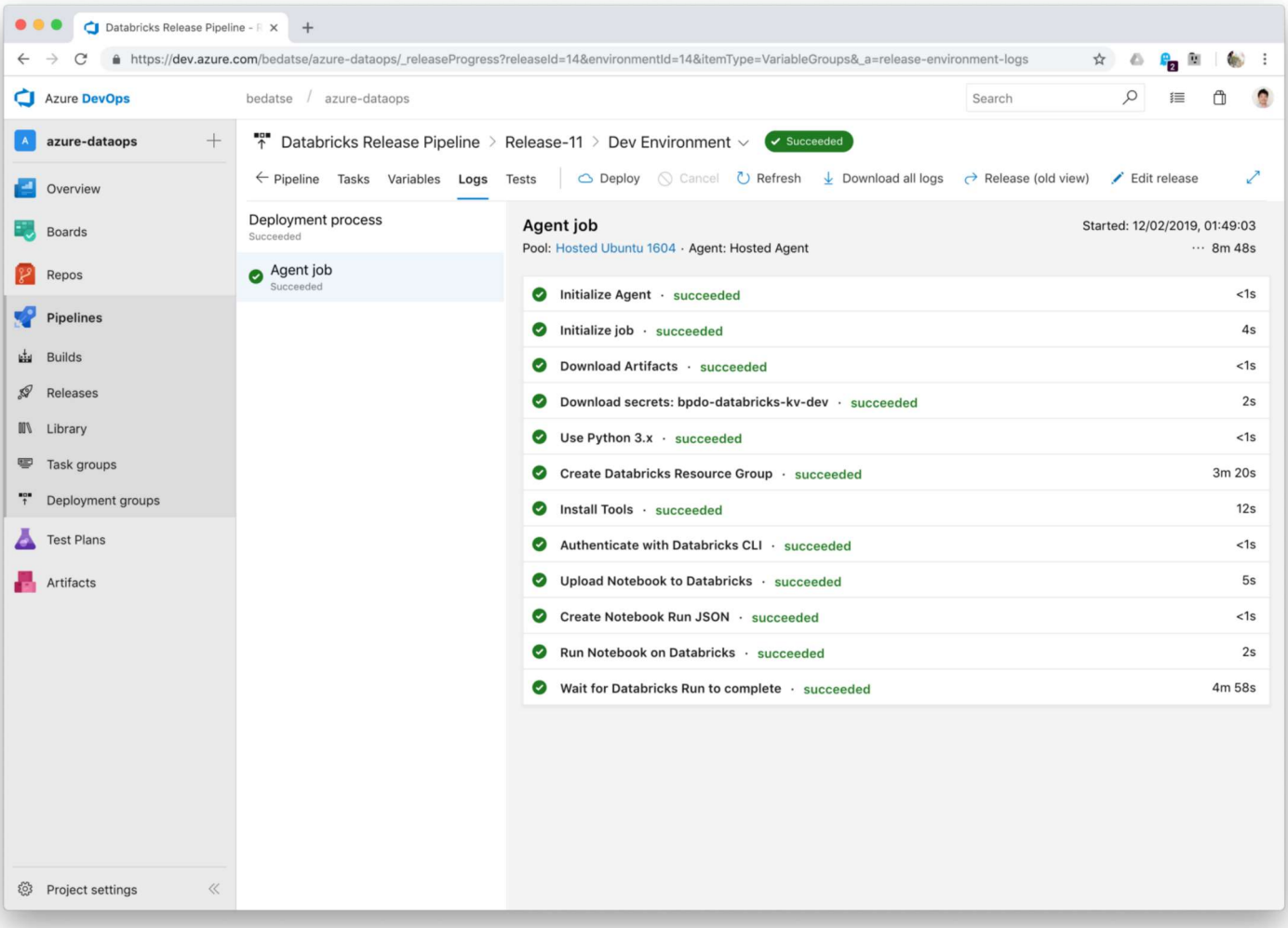
4–8 Configure Databricks Run task

4–9. Remember we have added Databricks token from Azure Key vault? It is now to put it in use. Access the **Variables** Tab, click ***Variable groups***, Link the variable group ***Databricks Dev Token*** with ***Dev Environment*** Stage.



4–9 Link the Token variable group with Environment

4–10. **Save** the Release Pipeline, and ***create a release*** to test the new pipeline.

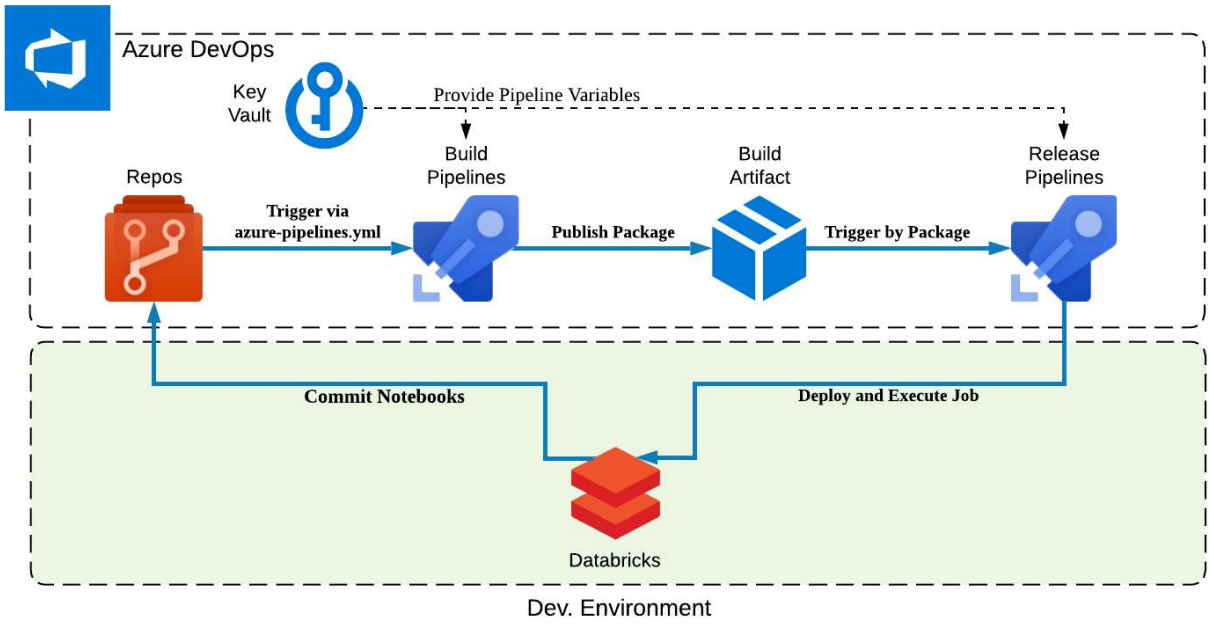


4–10 Deployment and execution result

Tada! Now your notebook is being deployed back to your Development environment, and successfully executed via a Job cluster!

Why do we want to do that? It is because you would like to test if the notebook can be executed on a cluster other than the interactive cluster you have been developing your notebook, ensuring your notebook is portable.

Let's recap what we have done



1. We have set up Databricks and Azure Key Vault provisioning via Azure Resource Manager Template.
2. We have set up Git integration with Databricks.
3. We have set up preliminary build steps and publish notebook as build artifacts.
4. We have been using Azure Key Vault for securely managing deployment credentials.
5. We have set up automated deployment and job execution flow with Databricks, which the job execution can be served as a very simple deployment test.

Edit on 3 March 2019

We have found that the key vault cannot be created automatically with the current ARM template, this tutorial will be updated in future. For now, please manually create the key vault for the first time in the resource group.

What's Next?

In our next posts, we will add Data Factory into the picture, provisioning sandbox data factory via ARM Template, and setting up Data Factory CI/CD properly.

Stay tuned!

Azure Databricks DevOps Ci Cd Pipeline Databricks Azure Key Vault