**Working with YAML file in Azure pipeline to automate the CI & CD.**

Up to now we configured the Continuous integration (CI) and continuous deployment (CD) manually in Azure DevOps. That means we done CI/CD pipeline in GUI format.

But we can also automate the CICD pipeline by configuring a single YAML file.

In Azure DevOps, a **YAML file is used to define your pipeline as code (Pac)**. This means instead of setting up CI/CD pipelines manually through the Azure DevOps GUI, you write a text-based **.yml** file to describe exactly how your application should be built, tested, and deployed.

Or

In Azure DevOps, a **YAML (YAML Ain’t Markup Language)** file is used to define the configuration for a CI/CD pipeline in a simple, readable, text-based format. Instead of clicking through the Azure DevOps UI to set up build and release pipelines, you declare your pipeline logic, steps, triggers, and environment in an **.yml** file.

**Benefits of using YAML pipelines in Azure DevOps:**

**1️. Pipeline as Code (PaC)**

* Pipelines are stored as code in your repository.
* Every change to the pipeline is version-controlled.
* Teams can collaborate on pipeline changes using pull requests, code reviews, and branches.

**2️. Consistency & Repeatability**

* Same YAML pipeline runs the same way in any environment or for any developer.
* Reduces human error compared to manually setting up GUI-based pipelines.

**3️. Easier Collaboration**

* Since the pipeline definition lives in the repo, it’s easy for team members to review, suggest changes, and track history.
* You can manage pipeline updates via pull requests.

**4️. Flexibility & Customization**

* YAML gives you control over jobs, stages, triggers, conditions, templates, variables, and agent pools.
* You can create modular pipelines using **templates** and **parameters** for reuse across multiple projects.

**5️. Better CI/CD Integration**

* Seamlessly integrates with your version control system.
* Enables **Continuous Integration (CI)** and **Continuous Deployment (CD)** automatically on code commits, pull requests, or scheduled triggers.

**6️. Portability**

* Easily move your pipeline definition to another project or repository.
* Standardizes the DevOps process across teams and projects.

**7️. Improved Auditability**

* Every pipeline change is traceable through your version control history.
* Helps with auditing for compliance and debugging issues.

**8️. Supports Complex Workflows**

* YAML pipelines can handle multi-stage deployments, approvals, environment checks, rollback strategies, and conditional tasks.
* Clean, hierarchical structure to define complex CI/CD processes.

**9️. Cloud Native**

* YAML pipelines work well with cloud-hosted agents (like Azure’s hosted runners) or your own self-hosted agents.
* Makes scaling easier for enterprise-grade applications.

**Primary uses:**

| **📌 Purpose** | **📋 Description** |
| --- | --- |
| **Pipeline as Code** | Define your build, test, and deployment process in code, version-controlled alongside your application code. |
| **Automation** | Automate the entire CI/CD workflow — from code push to build to deployment without manual steps. |
| **Consistency** | Ensure the same pipeline configuration is applied across environments, branches, or projects. |
| **Portability** | Since it's a text file, it’s easy to clone, reuse, and share across projects or teams. |
| **Customization & Conditions** | Use variables, parameters, conditions, and templates to customize workflows based on environment, branch, or other logic. |

**Standard syntax and structure of a YAML file for Azure DevOps pipelines:**

A basic Azure Pipelines YAML file is made up of the following key sections:

# Comments start with a hash (#)

trigger: # What events will start the pipeline

pool: # Which agent to run jobs on (Microsoft-hosted or self-hosted)

variables: # Custom variables you can define and reuse

stages: # Optional: break the pipeline into stages like Build, Test, Deploy, Dev, prod

jobs: # A collection of steps that run together on the same agent

steps: # The actual tasks or scripts to run

**Key Syntax Rules**

✅ **Indentation matters**

* Use 2 spaces per level (no tabs)
* Child items are indented under their parent

✅ **Dashes - are used for lists**

* Example: multiple stages, jobs, or steps

✅ **Variables are referenced like this** → $(variableName)

✅ **Tasks use task: <TaskName>@<Version>**

✅ **Scripts are defined as:**

**Example:** - script: echo "Hello"

displayName: 'Greet'

**Example:** Simple Azure DevOps YAML pipeline to install Node.js.

trigger:

branches:

include:

- main

pool:

vmImage: 'ubuntu-latest'

steps:

- task: NodeTool@0

inputs:

versionSpec: '18.x'

displayName: 'Install Node.js'

- script: |

node --version

npm --version

displayName: 'Check Node.js and npm version'

- script: |

npm install

displayName: 'Run npm install'

Fig: azure-pipelines.yaml file

Now let’s perform a Task practically to automate the CI/CD pipeline for Terraform code of VM build in Azure DevOps.

* 1. Create a New Organization within it create a new project.
  2. Push the Terraform code to Version control (Azure Repos) of the project.
  3. Write the YAML file (azure-pipelines.yaml)
  4. Save and run the azure-pipelines.yaml file.

Step1: Create the Organization (my-first-organization1) within it create a project

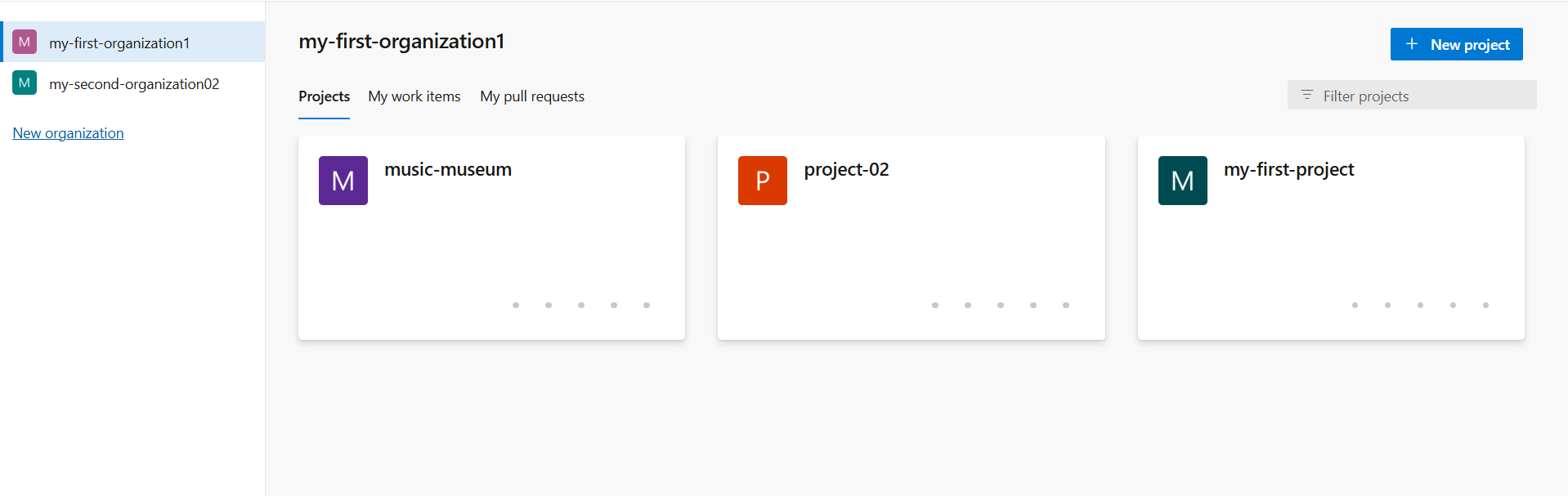


Fig: Organization (my-first-organization).

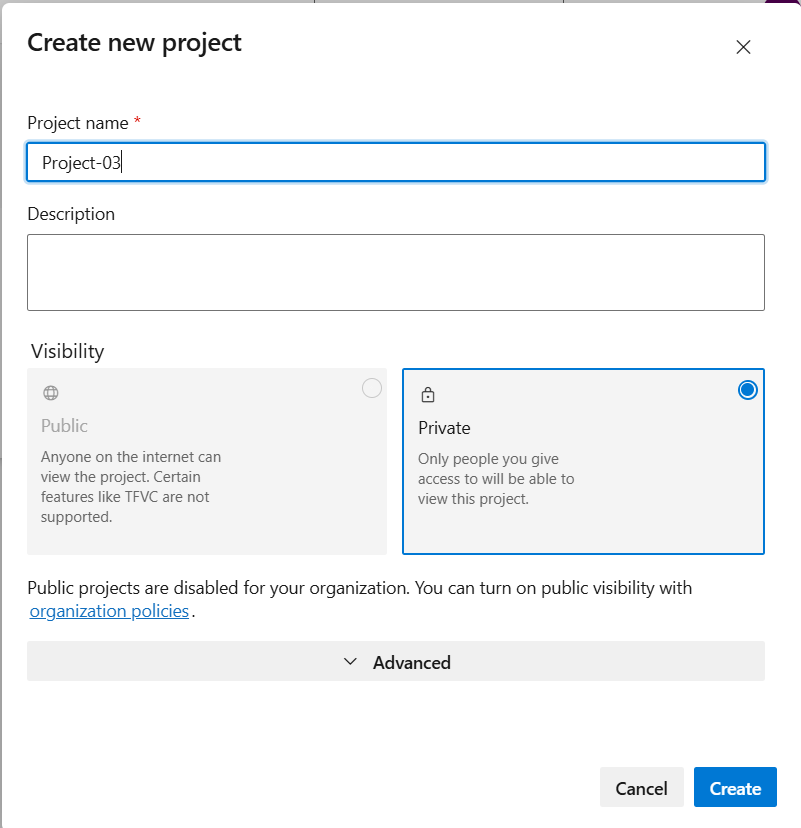


Fig: Project (project-03).

**Step2:** Push the local Terraform code (VM build) to the azure repository.

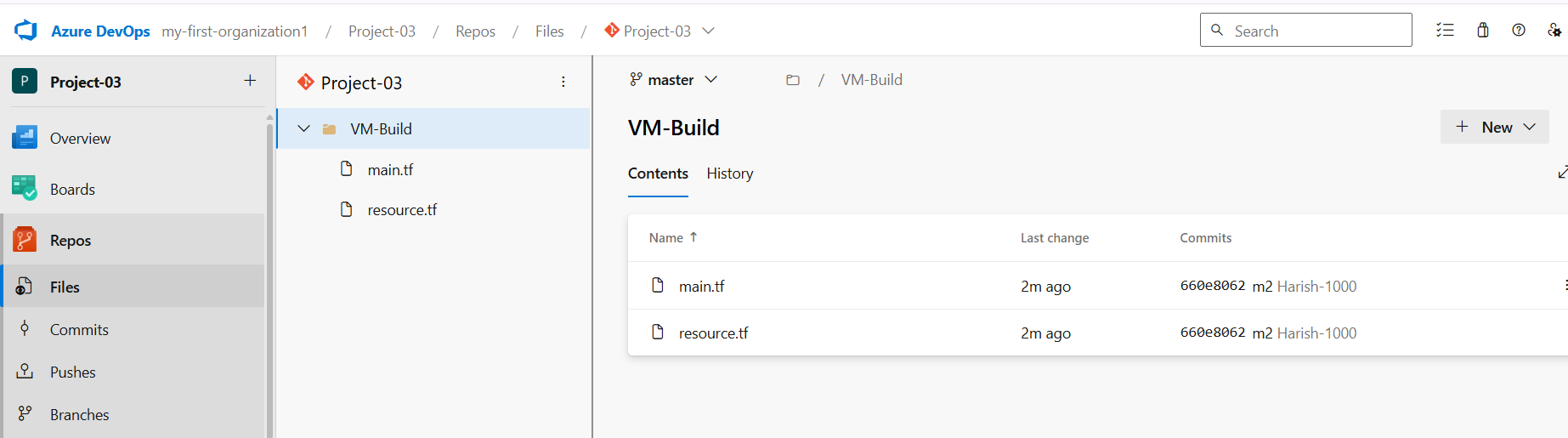


Fig: Terraform code is successfully pushed into Azure repo of project-03.

**Source folder:** VM-Build in azure repos.

**Step3:** Create a YAML file to automate the CI/CD pipeline in azure DevOps.

Organization (my-first-organization) 🡪project-03🡪pipeline🡪Create pipeline🡪Azure Repos Git (yaml)🡪Select repository🡪Started pipeline🡪 now configure the YAML file.

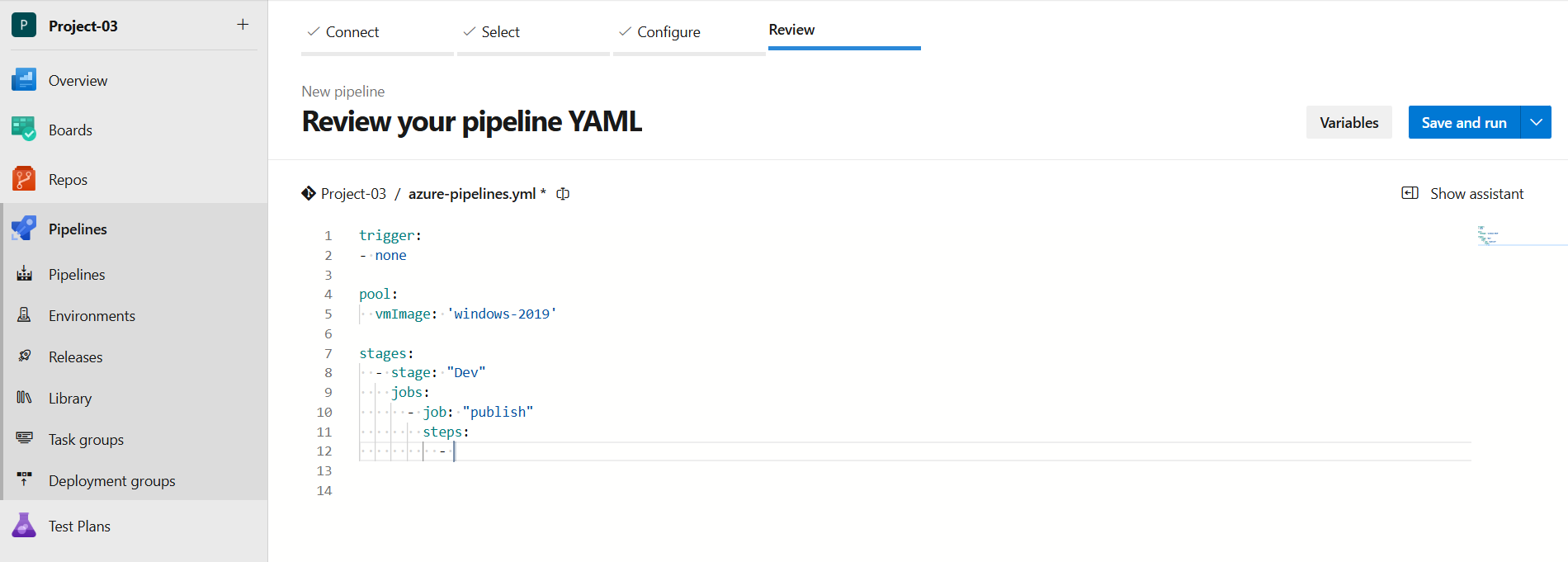
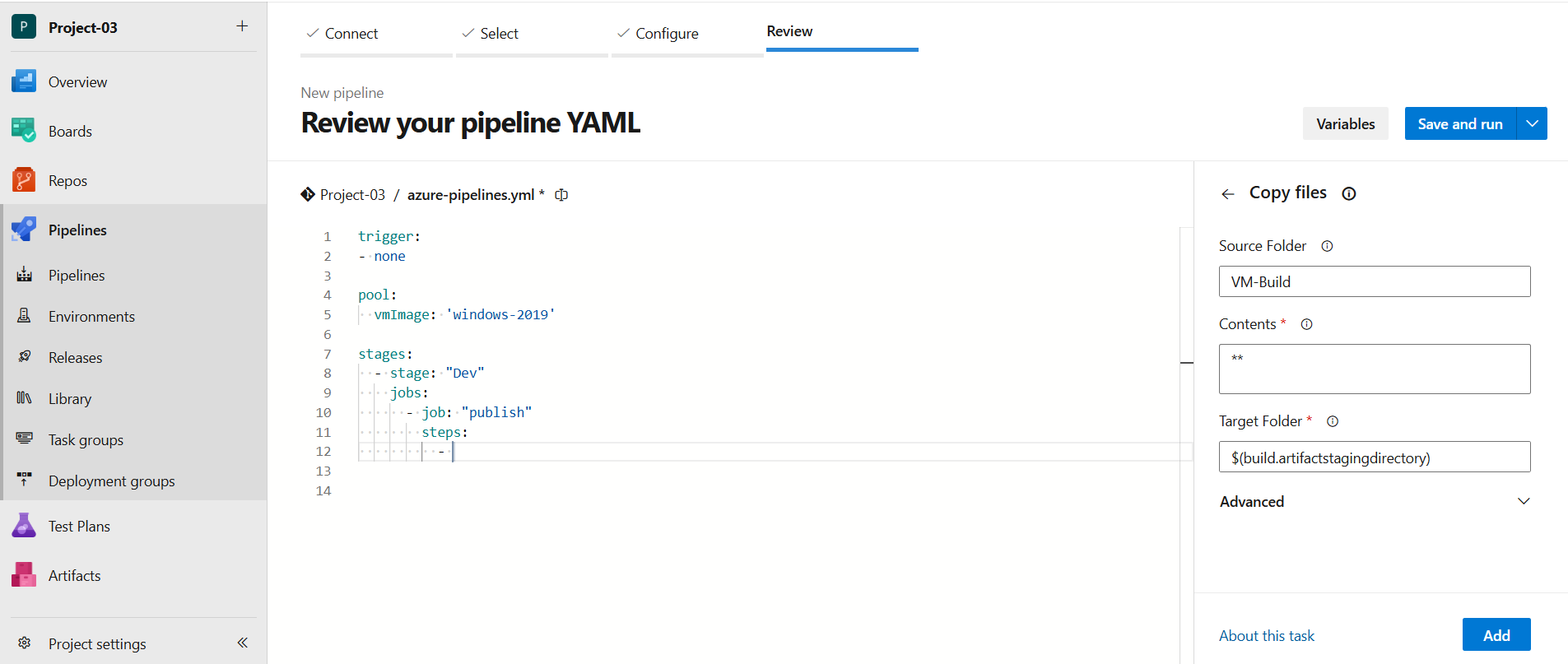


Fig: Provides assistance while configuring the YAML file.



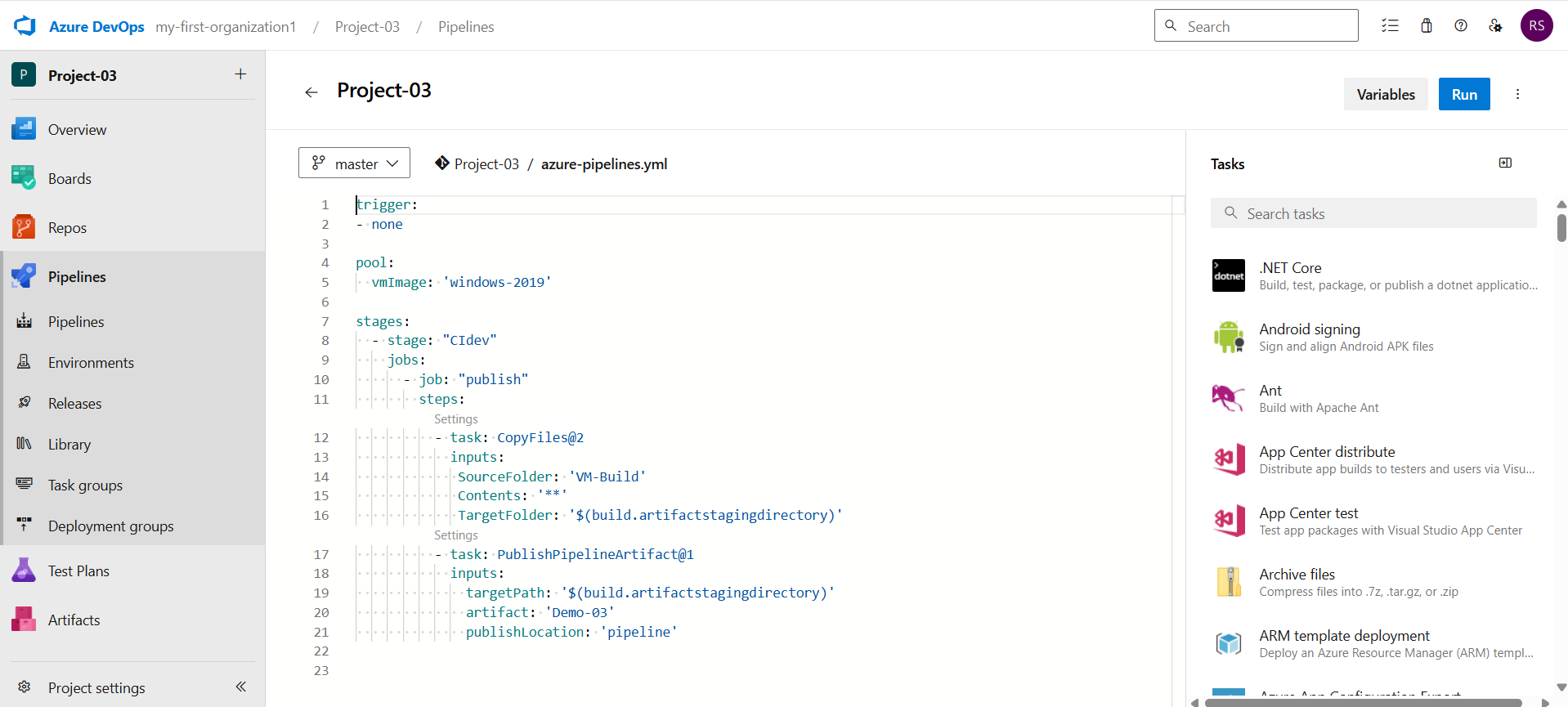


Fig: Up to now we configured the CI in the above YAML file using “Show assistance” option.

**Note:** For CD configuration in the YAML file first we have to create a Resource group, storage account and container in it. In order to store the state file in it.

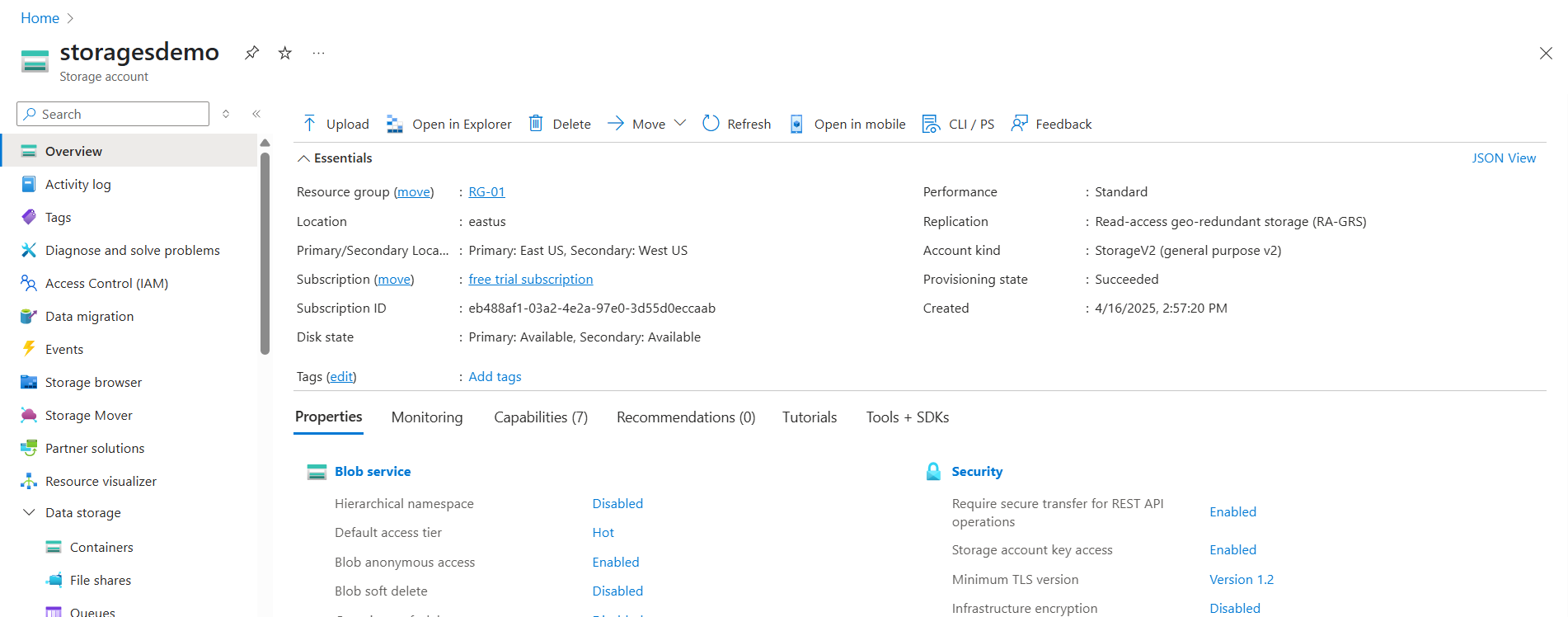


Fig: Storage account in resource group (RG-01).

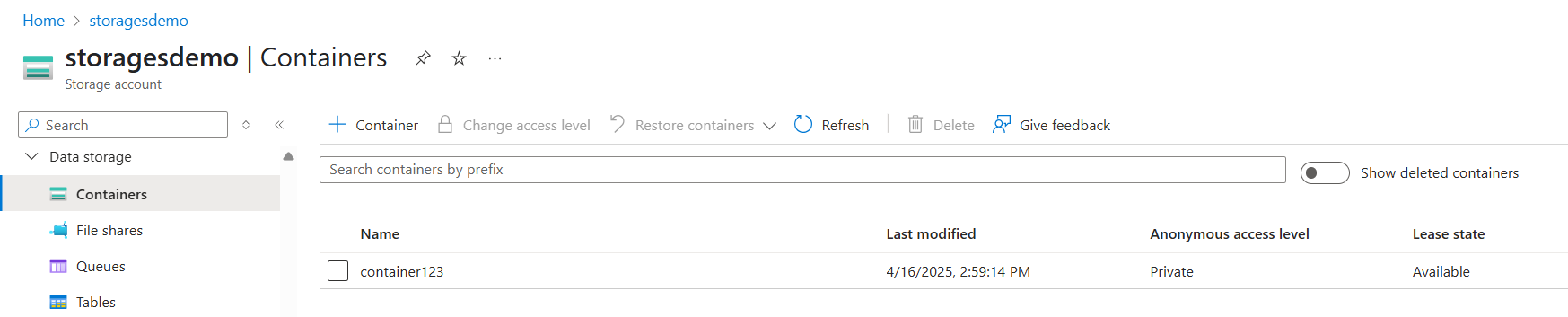


Fig: Container (container123) in storage account (storagedemo).

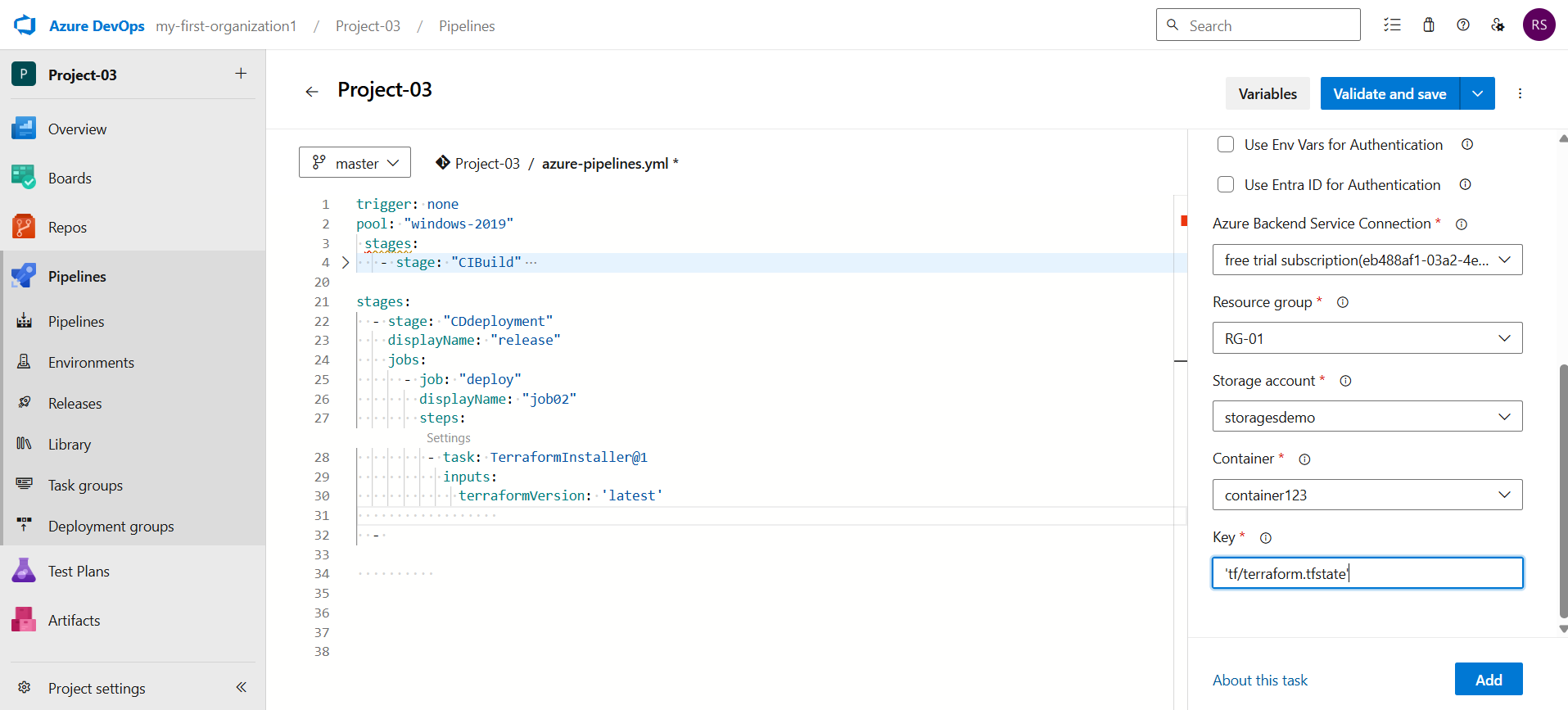


Fig: Configuring of YAML file with resource group, storage account and container.

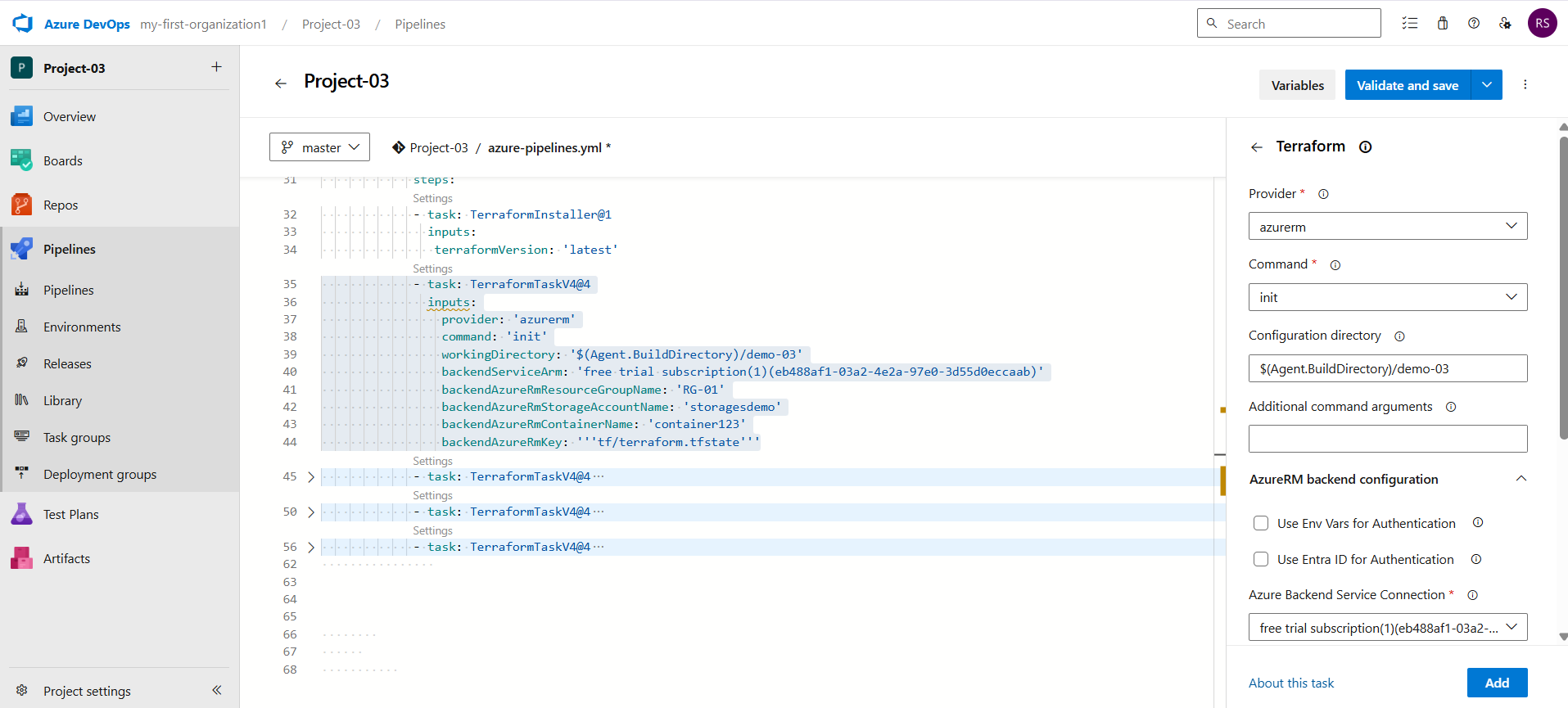


Fig: $(Agent.BuildDirectory)/demo-03

**$(Agent.BuildDirectory)**: is a predefined system variable in Azure DevOps pipelines.  
It represents the local directory path on the agent machine where the source code, build outputs, and artifacts for the current pipeline run are stored.

**demo-03:** It the directory at which we published arifacts.

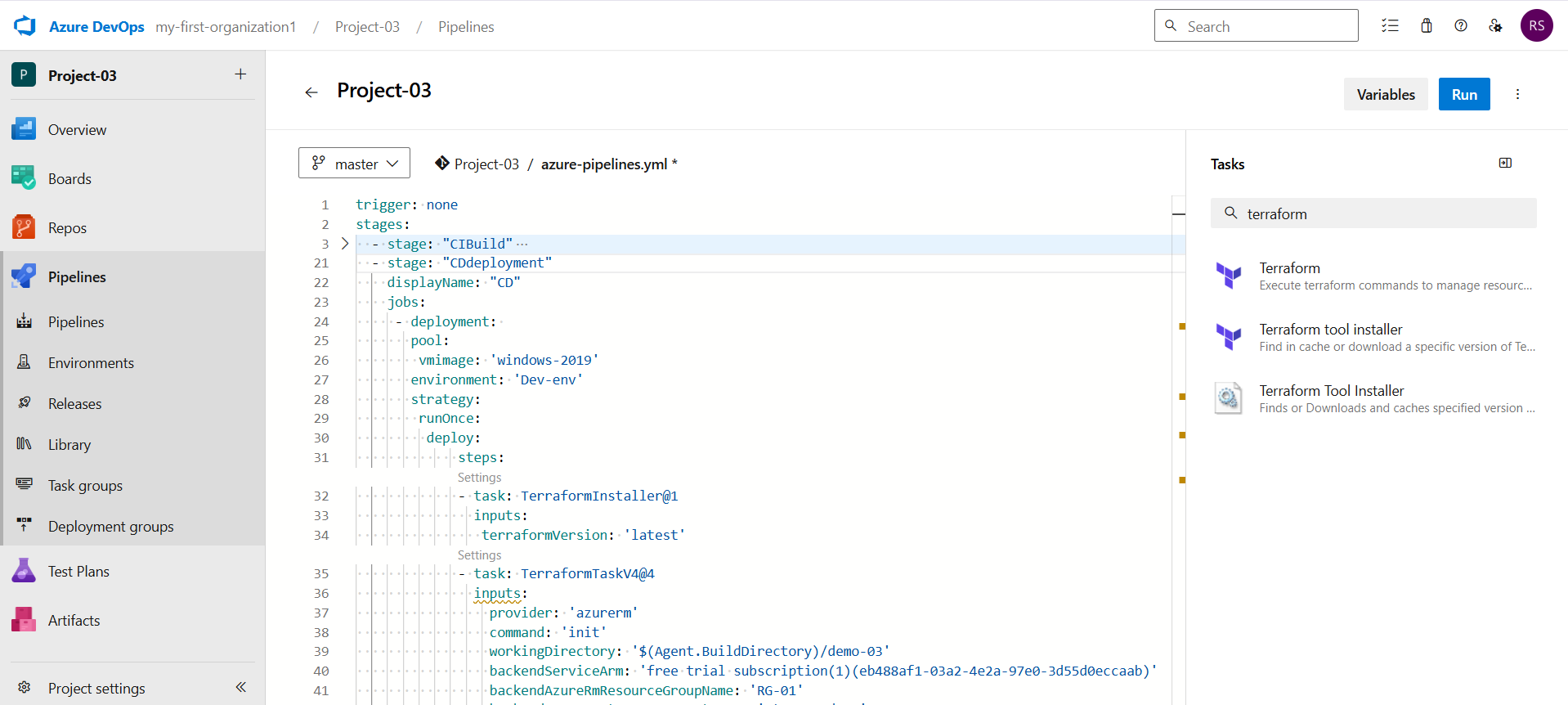


Fig: CD also configured in YAML file (azure-pipeline.yaml).

While configuring the CD in YAML file the

Now press the “**Run”** button and the YAML file will executes.

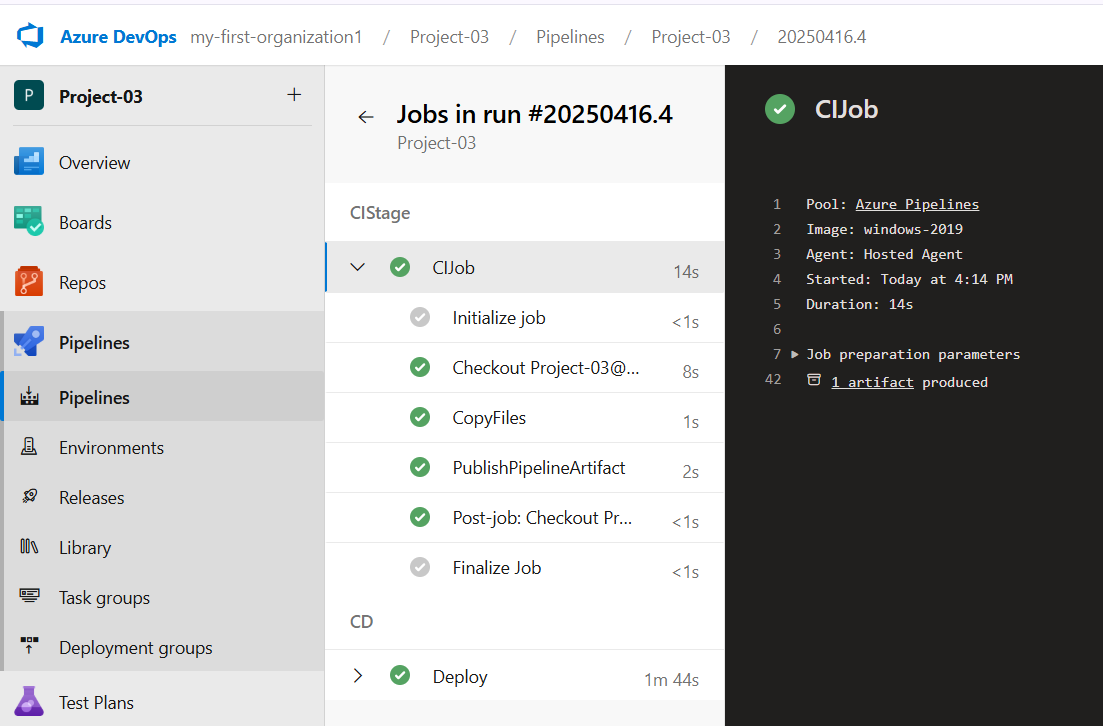


Fig: CI stage (CIstage) is successfully executed.

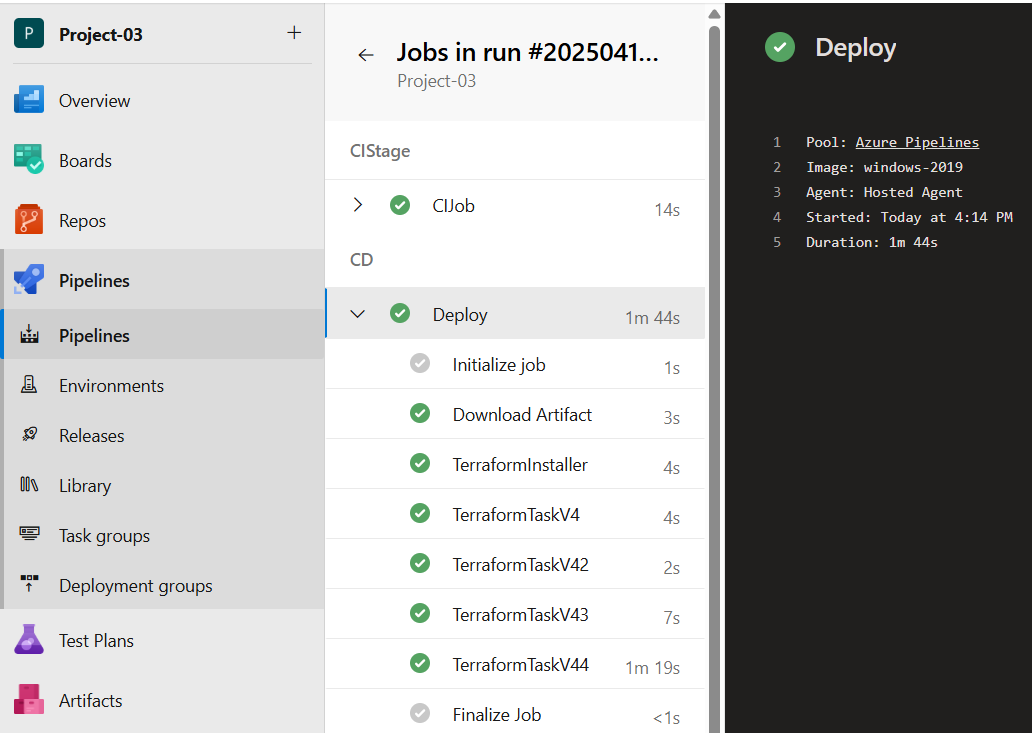


Fig: CD stage (CD) also deployed successfully.

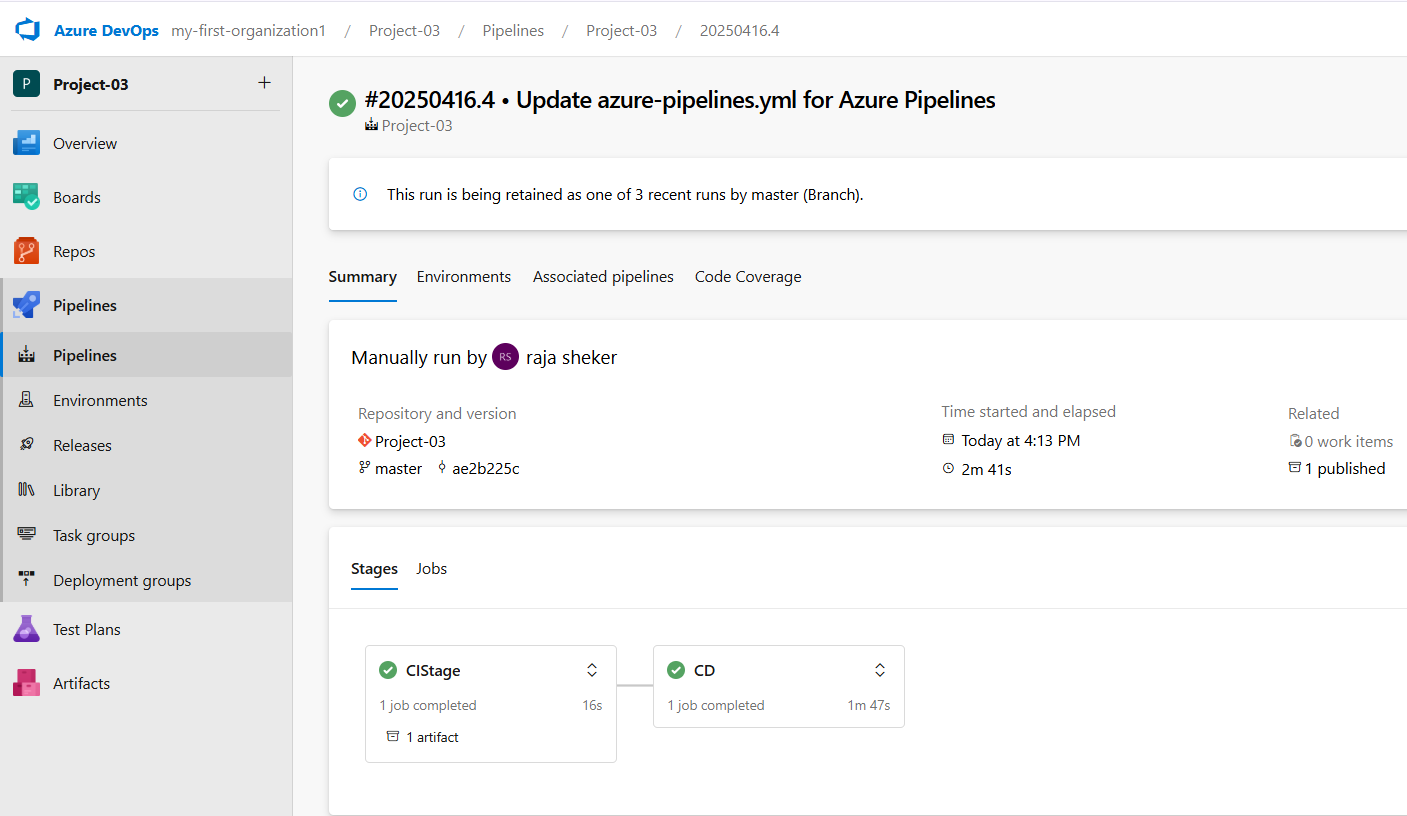


Fig: Both CI & CD stages are executed success fully by using a single YAML file (azure-pipeline.yam).

**For YAML file (azure-pipeline.yaml):** <https://github.com/sheriharish/YAML-file-to-automate-CI-CD.git>

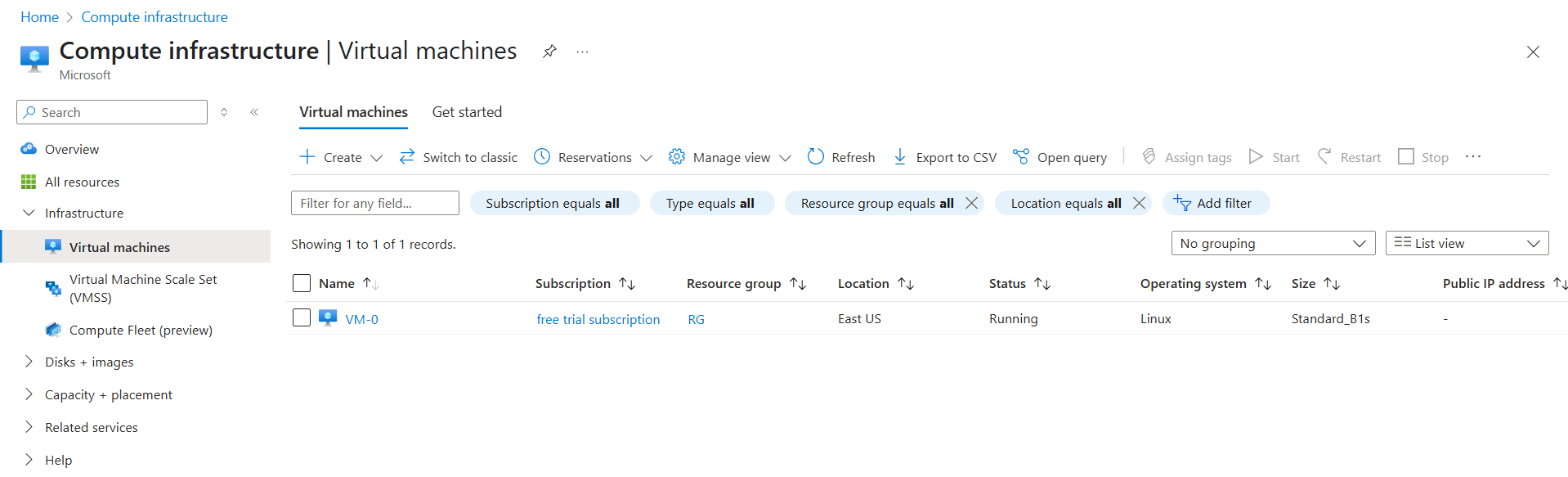


Fig: Virtual machine (VM-0).

The virtual machine is successfully created by deploying the terraform code in the azure pipeline using YAML file configuration.

**Note1:** By adding the multiple stages in CD like production, development, build, and QA environments, in YAML file we can deploy the application in different environment. But main thing is we can use single CI stage for multiple CD stages.

**Note2:** We can also add the stages (environments) by using GUI in Azure DevOps as shown below figure.

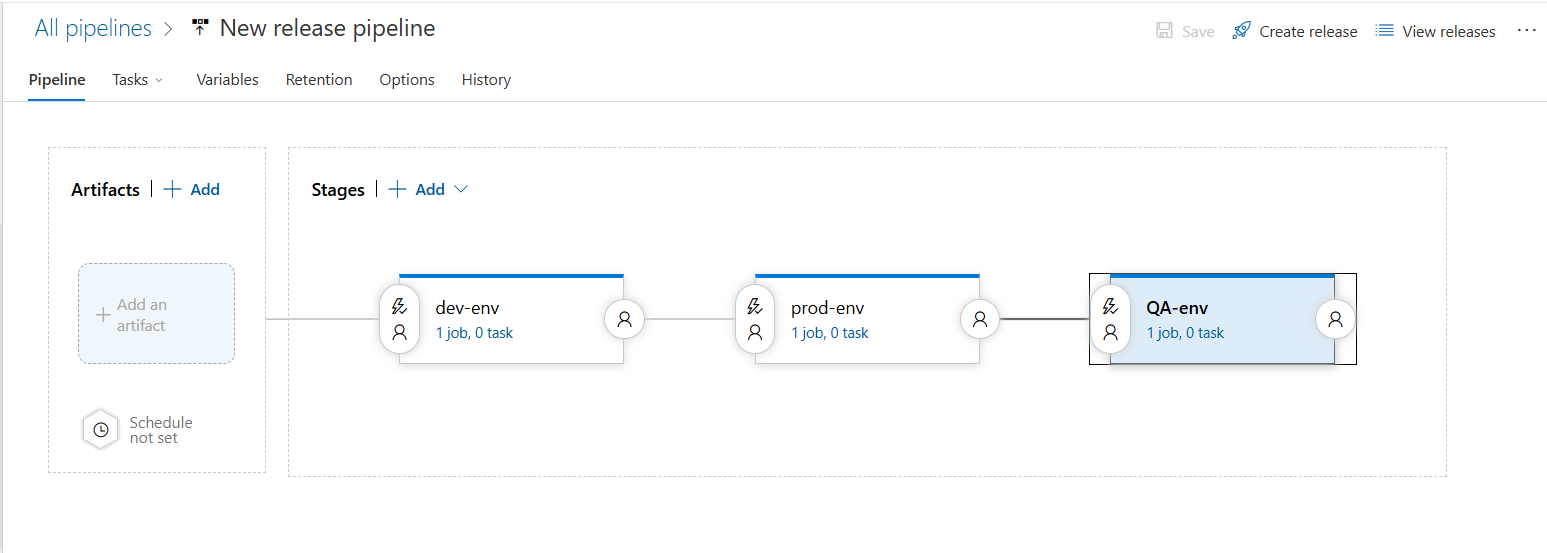


Fig: Adding of Different stages (dev-env, prod-env, and QS-env).

**Note:** A single CI stage is enough for multiple environments (dev, prod, QA) with in a single project.

From above figure the deployment of application is done in **prod-env** after deploying is completed in **Dev-env** that means if deployment is completedsuccessfully in dev-env then only the deployment process takes place in prod-env. Here deployment condition is dependent on one another.

But we can change this deployment conditions manually by using “pre-deployment conditions” in GUI of Azure DevOps release.

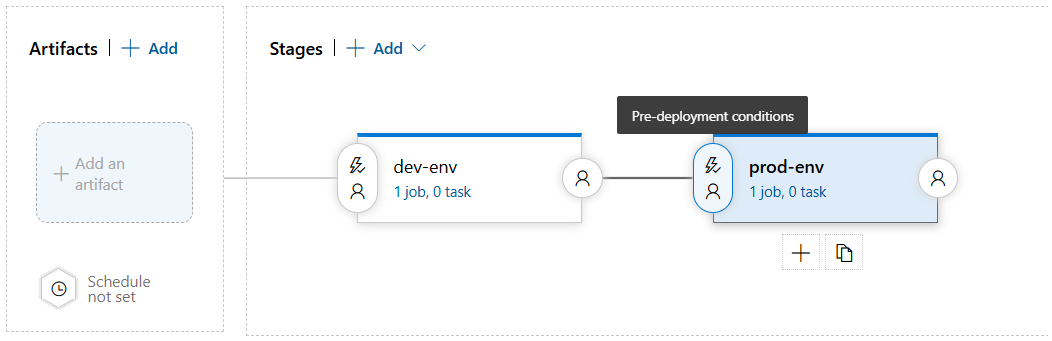


Fig: Pre-deployment condition.

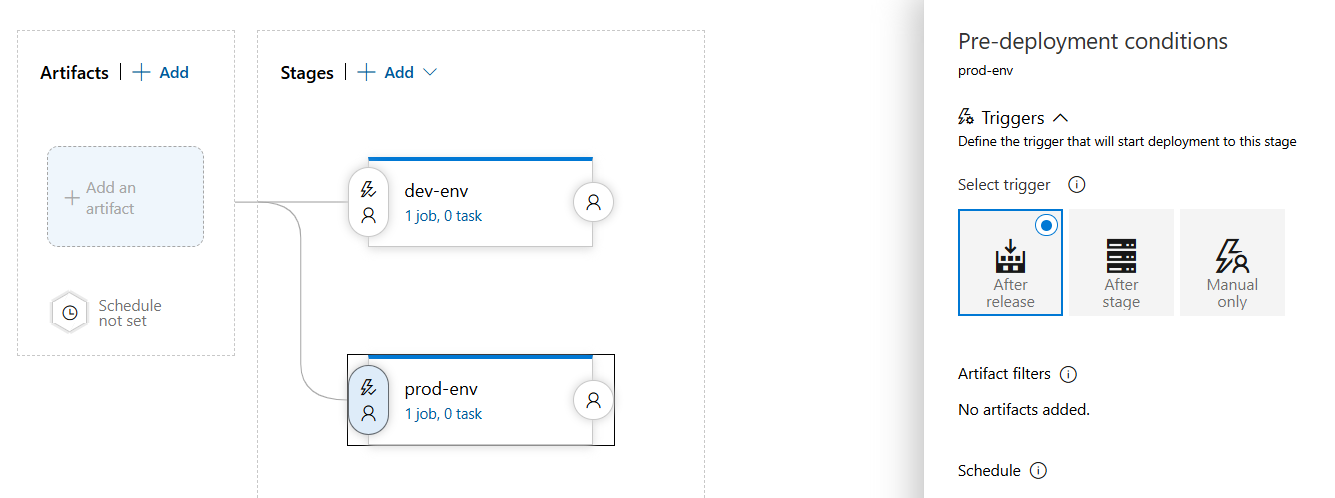


Fig: Deployment triggering after release.

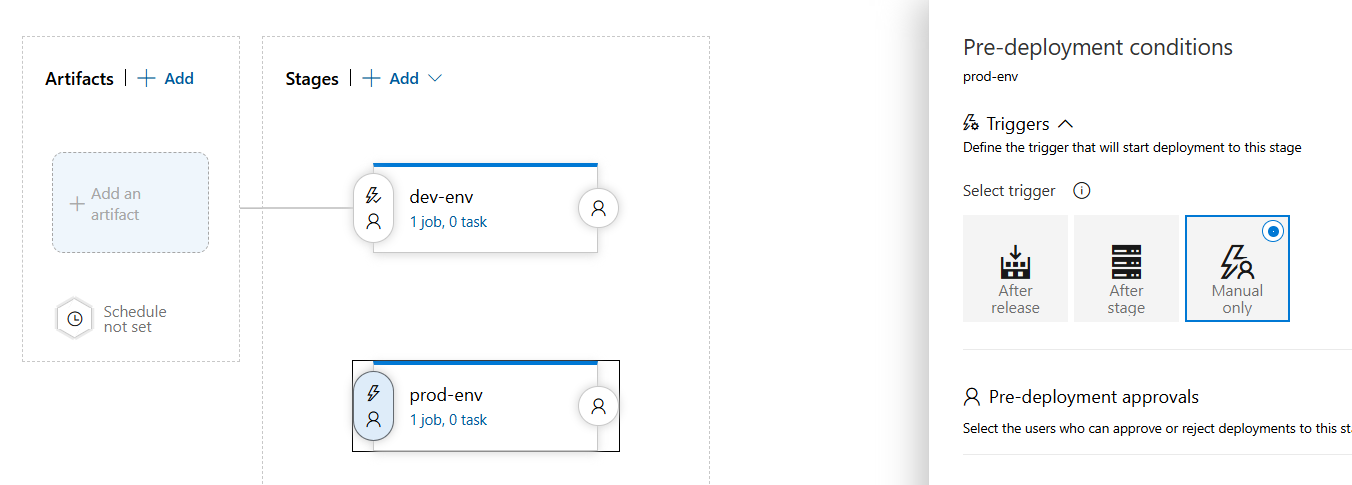


Fig: Deployment trigger in manual only.

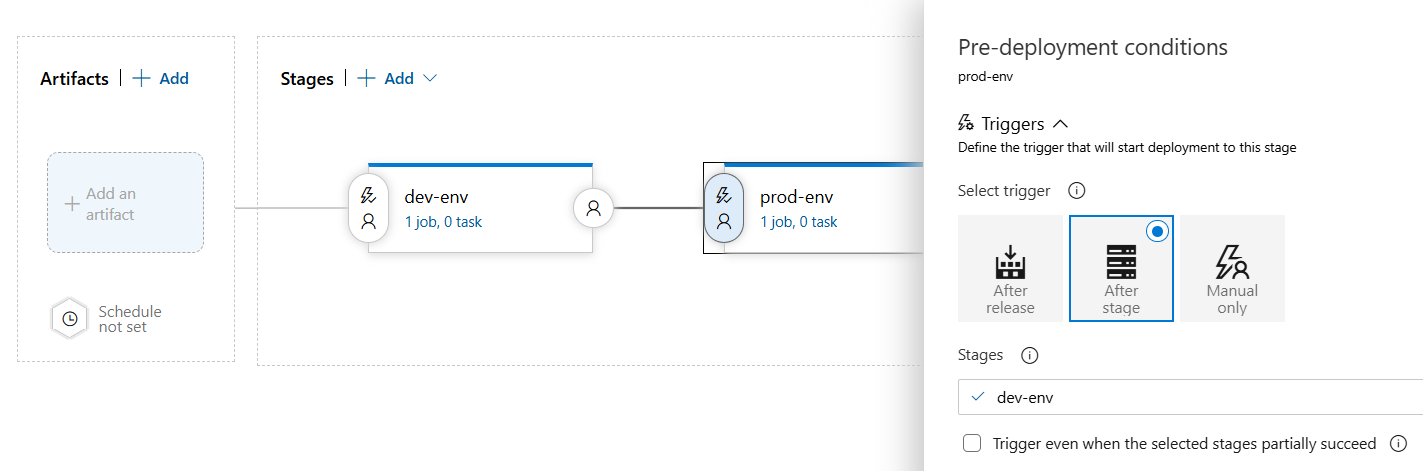


Fig: Deployment triggering after stage.

**Note:**

To deploy the application in the development environment right after pushing code to the Azure repository, you need to **enable continuous integration (CI)** by setting up a **trigger** in the pipeline. This ensures that the pipeline automatically runs whenever new code is pushed to the azure repository main/master branch as shown below figure.

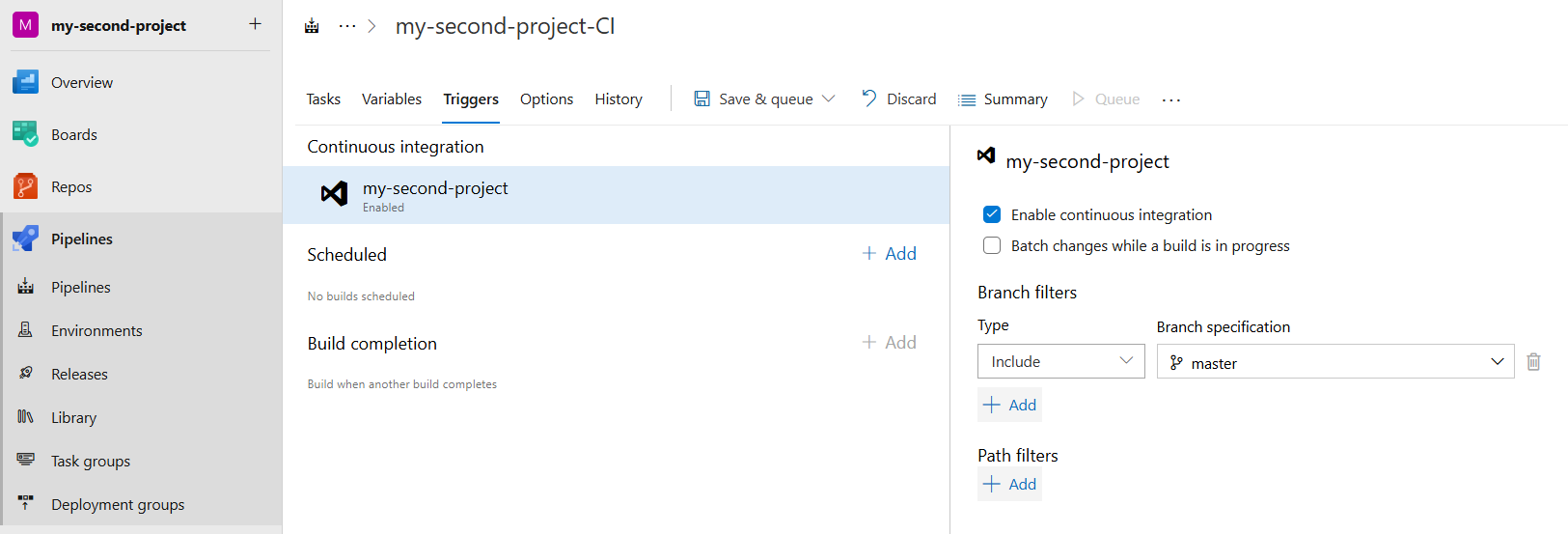


Fig: Enabling of continuous integration.

**Task Groups:**

A **Task Group** is like a reusable template for a set of tasks in your Azure pipeline. If you have a group of steps (tasks) that you use in multiple pipelines, instead of copying those tasks again and again, you can group them into a **Task Group** and reuse it wherever you need.

We use **Task Groups** to:

* Avoid repeating the same set of tasks in different pipelines.
* Make maintenance easier — if you need to change something, you only update it in one place (the Task Group) instead of every pipeline.
* Keep pipelines clean and organized by moving repeated or lengthy steps into a Task Group.
* Improve consistency — ensures the same steps run the same way across different projects or environments.

Let’s combine two tasks into one set practically:

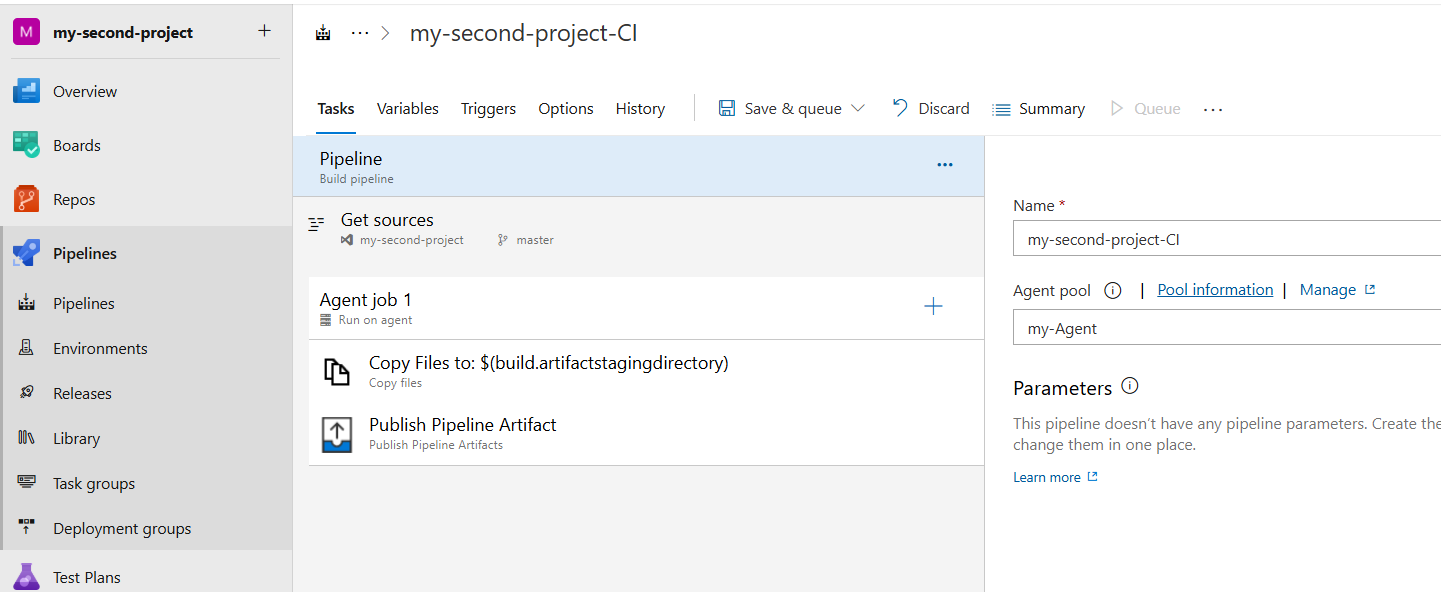


Fig: “Copy file” task and “publish pipeline artifacts” task in CI.

To group these two tasks (copy & publish), first select tasks which is to be grouped, and right click and select “create task group” as show in below figure.

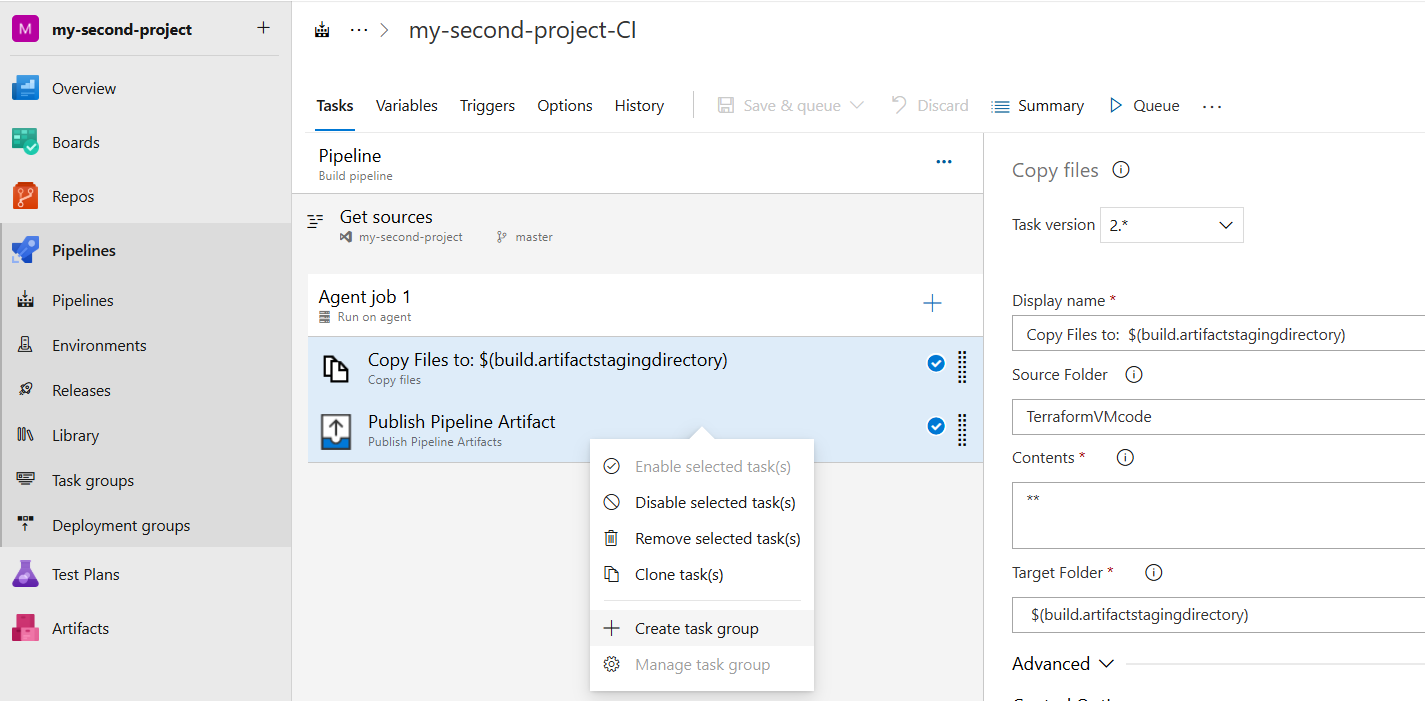


Fig: Create task group.

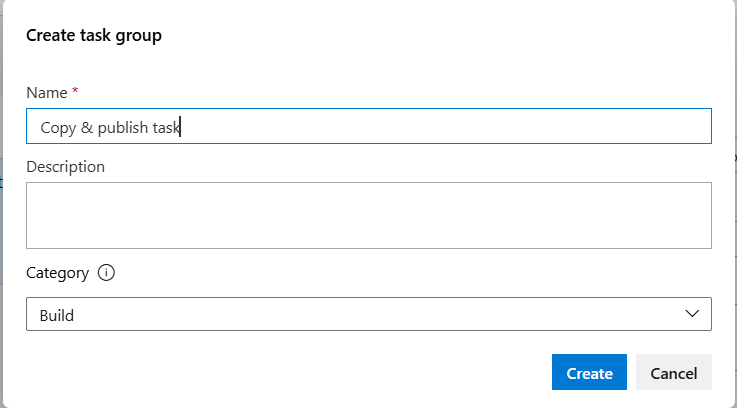


Fig: Name the task group.

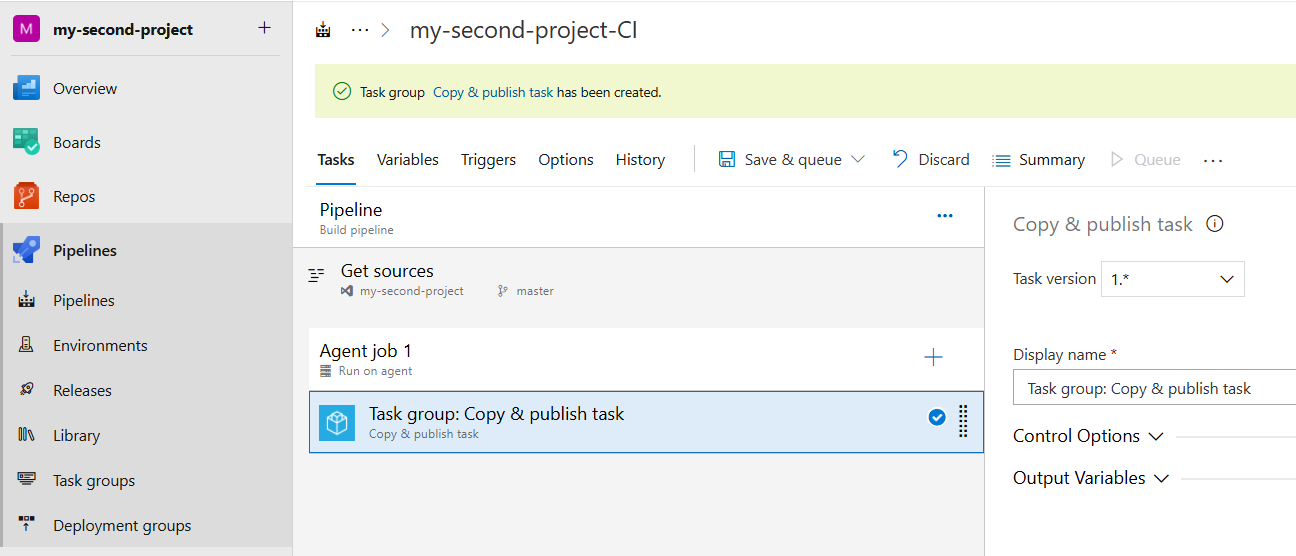


Fig: Task group with name “copy & publish task” is created successfully.

**Note:** This task group (copy & publish task) can be used within the Project only.

**Library:**

In Azure DevOps, a **Library** is a place where you can store **variables, secrets, and secure files** that you want to reuse across multiple pipelines and projects.

Think of it like a **centralized locker for important values** — so you don’t have to hard-code or repeat them in every pipeline.

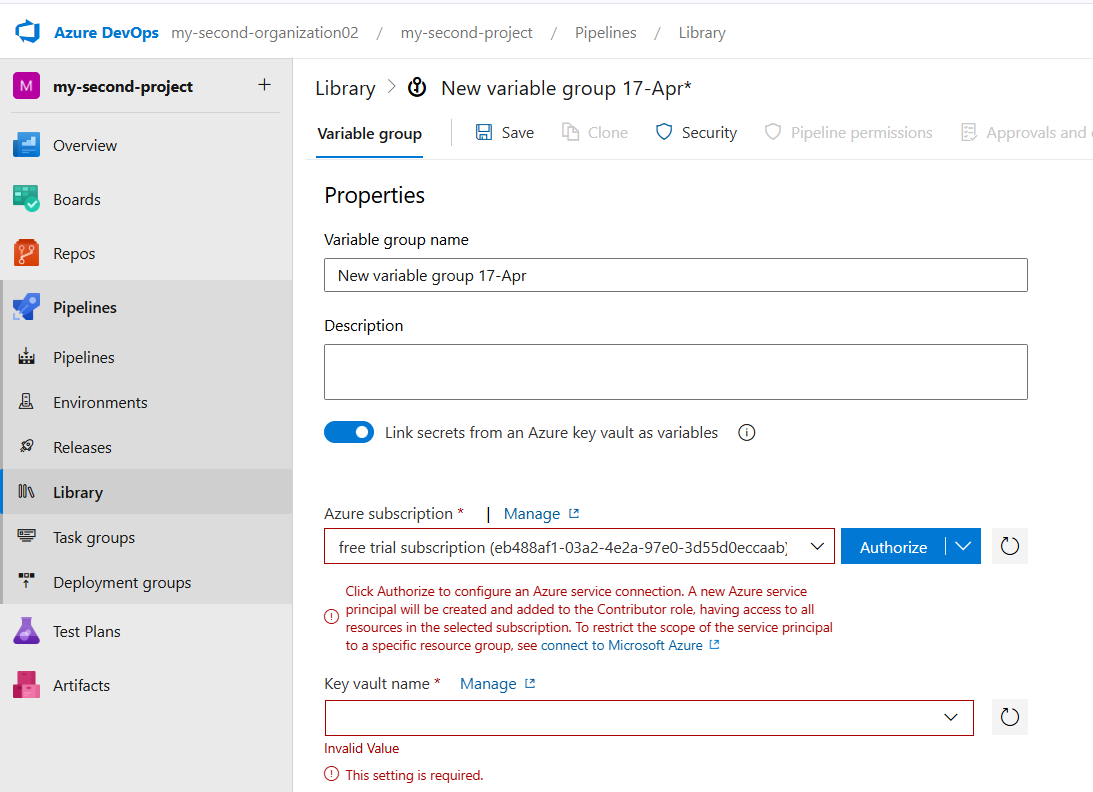
We use a **Library** to:

* **Store reusable variables** (like connection strings, environment names, API keys, or config values).
* **Securely store secrets** (like passwords, tokens, or certificates) using **Azure Key Vault integration** or secret variables.
* **Keep your pipelines clean** by avoiding hard-coded values inside YAML or classic pipelines.
* **Simplify management** — if a value changes, you update it in one place (the Library), and it automatically applies to every pipeline using it.

**Note:** By using this **Library** we can integrate our **Azure key vault** to the azure pipelines.

Let’s do it in practically:

Pipeline🡪Library🡪variable group🡪enable “link secrete from an azure key vault as variables🡪link azure subscription and key vault.



**Boards:**

**Boards** is a tool inside Azure DevOps that helps you **plan, track, and manage your work**. It’s like a digital whiteboard where your team can create, assign, and track tasks, bugs, features, and user stories while developing software.

It’s part of Azure DevOps Services made specifically for **project management and work tracking**.

Or

**Azure Boards** is a **work tracking tool** in Azure DevOps that helps teams plan, organize, and track work items (like tasks, bugs, and user stories) throughout the development lifecycle. It’s like a **digital whiteboard** for agile teams, supporting Scrum, Kanban, or custom workflows.

We use **Boards** to:

* 📋 **Track work items** like tasks, bugs, user stories, and features.
* 📝 **Plan sprints and backlogs** in agile or Scrum-based projects.
* 👥 **Assign work to team members** and track progress.
* 📊 **Visualize project status** using Kanban boards and dashboards.
* 🔄 **Connect work items to code changes, builds, and releases** for better traceability.

**Simple Example1:**

Let’s say your team is working on a new website:

* You create **user stories** for new features (like “Add login page”).
* Break them down into **tasks** (like “Design UI”, “Build API”, “Test login”).
* Track **bugs** if any issues appear.
* Plan and manage work in **sprints**.
* Use a **Kanban board** to move tasks through columns like *To Do*, *In Progress*, *Testing*, and *Done*.

**Example2:**

A software solution company receives a new project from a client.  
The client’s requirement is to build a dynamic website for their small business.

**📌 Step 1: Collecting Requirements**

The organization gathers all the client’s requirements and documents them as **work items in Azure Boards**.

**📌 Step 2: Choosing a Project Management Methodology**

Before starting development, the team decides which project management approach is suitable for the project — such as:

* **Agile**
* **Scrum**
* **Waterfall**

Each methodology has its own way of structuring and managing work in Azure Boards.

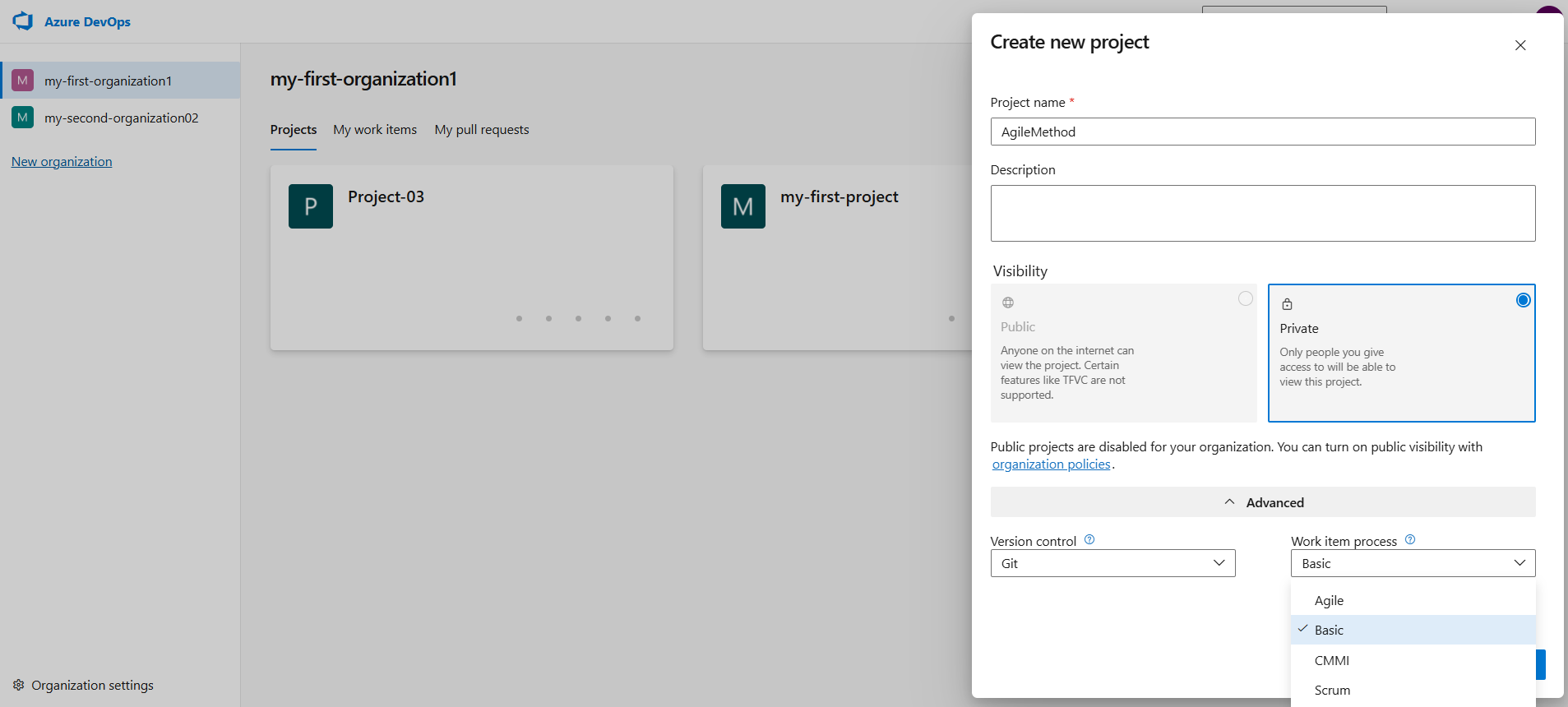
**📌 Step 3: Organizing Work Items Based on Methodology**

If the team chooses **agile methodology**, Azure Boards provides specific work item types to manage the workflow effectively:

* **Epic** — A large feature or goal, like *Build a dynamic business website*
* **User Story** — Smaller functionalities within an Epic, like *Create login page* or *Add product listing section*
* **Task** — Specific development or testing tasks, like *Design homepage layout*
* **Bug** — Issues found during development or testing
* **Test Case** — To validate features work as expected
* **Issue** — Any obstacles or concerns that may affect progress

These work items can be organized in **Boards**, moved through different states like *To Do*, *In Progress*, *Done*, and linked to sprints, builds, or deployments for full traceability.

Let’s Create a Board for agile methodology:



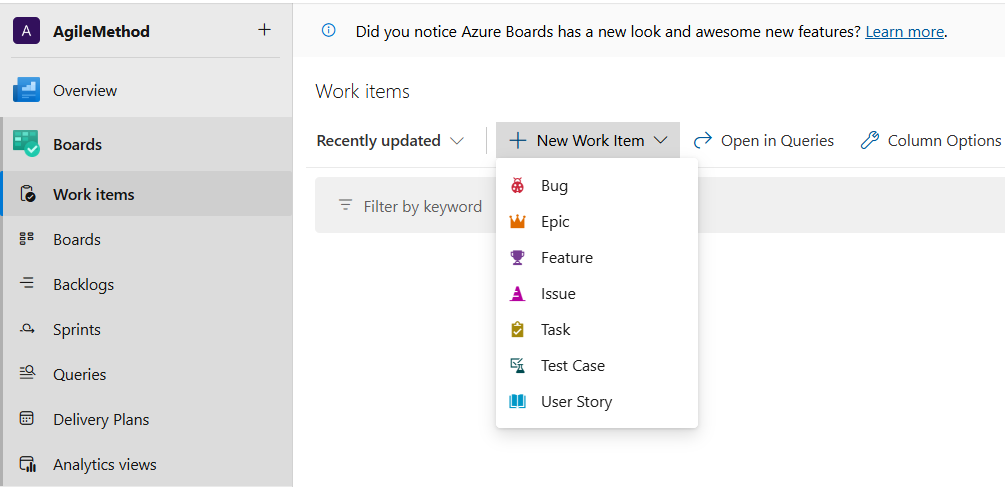
Fig: Selecting of Methodology.

Fig: Work items in **agile** methodology**.**

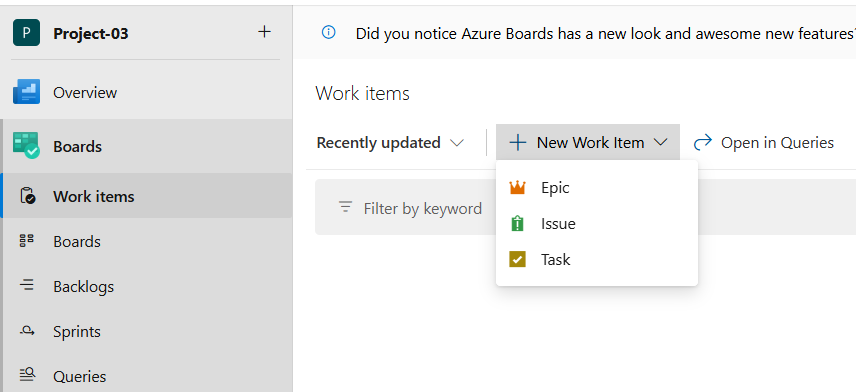


Fig: Work items in **Basic** methodology