Azure DevOps

By Khaja

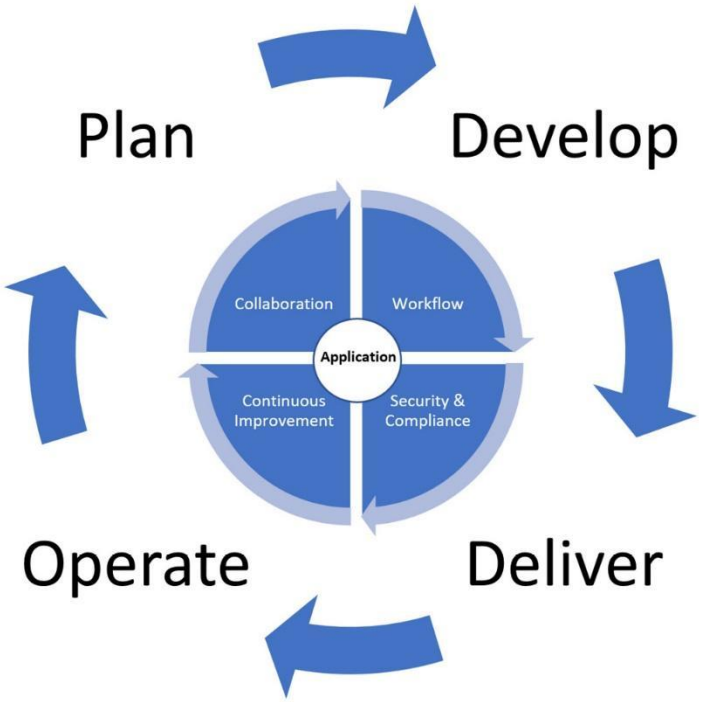
Table of Contents

The capacity to learn is a gift; the ability to learn is a skill; the willingness to learn is a choice.

Introducing Azure DevOps key concepts

Azure DevOps provides a wide variety of services for DevOps teams so that they can plan, work, collaborate on code development, and build and deploy software and services. Most DevOps teams rely on several tools and build custom toolchains for each phase in the application life cycle.

The following diagram shows the phases that are defined in the application life cycle:



PLAN

During the planning phase, teams can use Kanban boards and backlogs to define, track, and lay out the work that needs to be done in Azure Boards. They can also use GitHub for this. In GitHub, an issue can be created by suggesting a new idea or stating that a bug should be tracked. These issues can be organized and assigned to teams.

DEVELOP

The development phase is supported by Visual Studio Code and Visual Studio. Visual Studio Code is a cross-platform editor, while Visual Studio is a Windows- and Mac-only IDE. You can use Azure DevOps for automated testing and use Azure Pipelines to create automatic builds for building the source code. Code can be shared across teams with Azure DevOps or GitHub

DELIVER

The deliver phase is about deploying your applications and services to target environments. You can use Azure Pipelines to deploy code automatically to any Azure service or on-premises environments. You can use Azure Resource Manager templates or Terraform to spin up environments for your applications or infrastructure components. You can also integrate Jenkins and Spinnaker inside your Azure DevOps Pipelines.

OPERATE

In this phase, you implement full-stack monitoring for monitoring your applications and services. Keeping your applications and services secure is also part of this phase. To support the full life cycle of analyzing, designing, building, deploying, and maintaining software and infrastructure products and services, Azure DevOps provides integrated features that can be accessed through any web browser.

Discovering Azure DevOps services

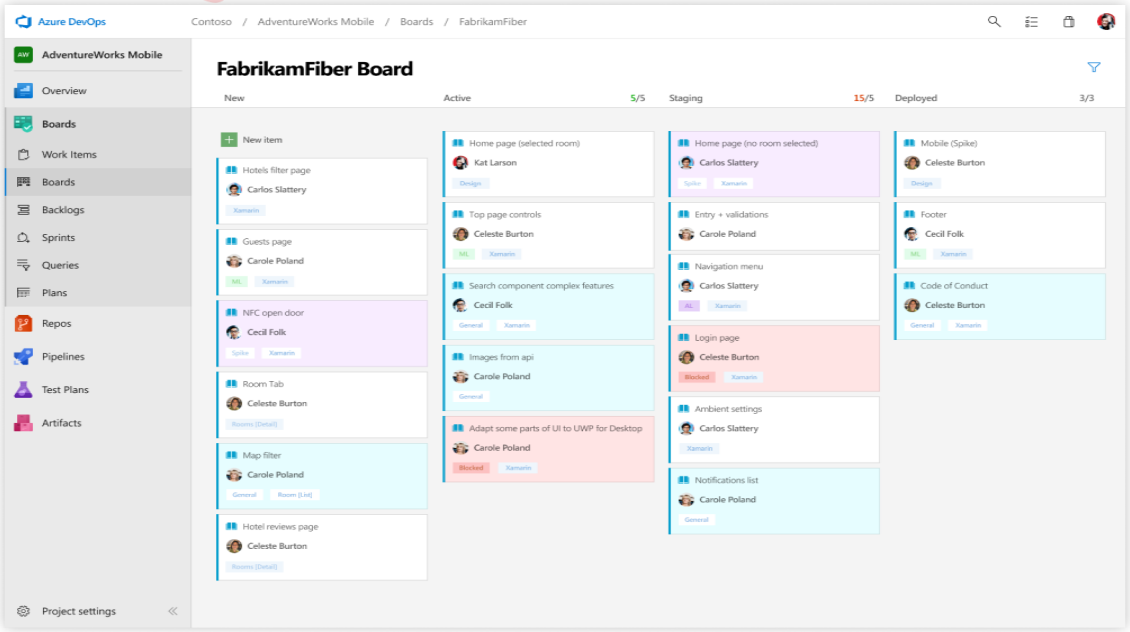
In this section, we are going to introduce the different services that are offered by Azure DevOps. These services can be used to support teams throughout the whole life cycle of realizing business value for customers.

AZURE BOARDS

Azure Boards can be used to plan, track, and discuss work across teams using the Agile planning tools that are available. Using Azure Boards, teams can manage their software projects. It also offers a unique set of capabilities, including native support for Scrum and Kanban. You can also create customizable dashboards, and it offers integrated reporting and integration with Microsoft Teams and Slack.

You can create and track user stories, backlog items, tasks, features, and bugs that are associated with the project using Azure Boards.

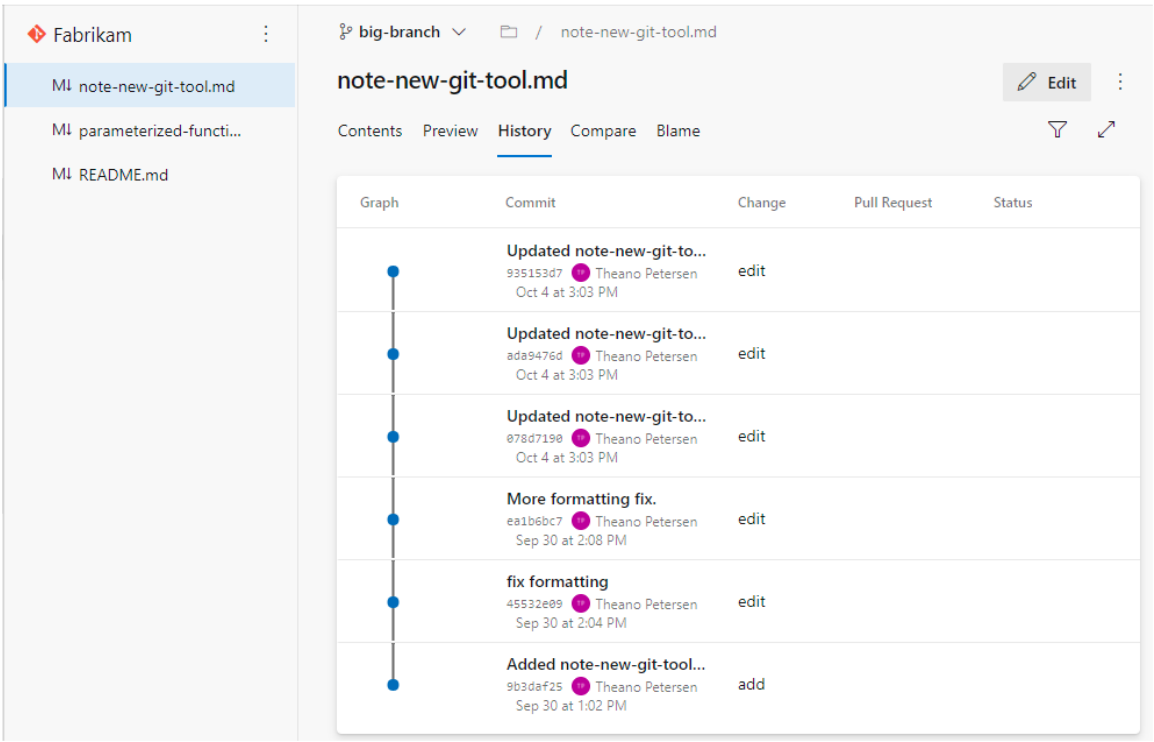
The following screenshot shows an example of an Azure Board:



AZURE REPOS

Azure Repos provides support for private Git repository hosting and for **Team Foundation Server Control** (**TFSC**). It offers a set of version control tools that can be used to manage the source code of every development project, large or small. When you edit the code, you ask the source control system to create a snapshot of the files. This snapshot is saved permanently so that it can be recalled later if needed.

Today, Git is the most used version control system among developers. Azure Repos offers standard Git so that developers can use the tools and clients of their choice, such as Git for Windows, Mac, third-party Git services, and tools such as Visual Studio and Visual Studio Code

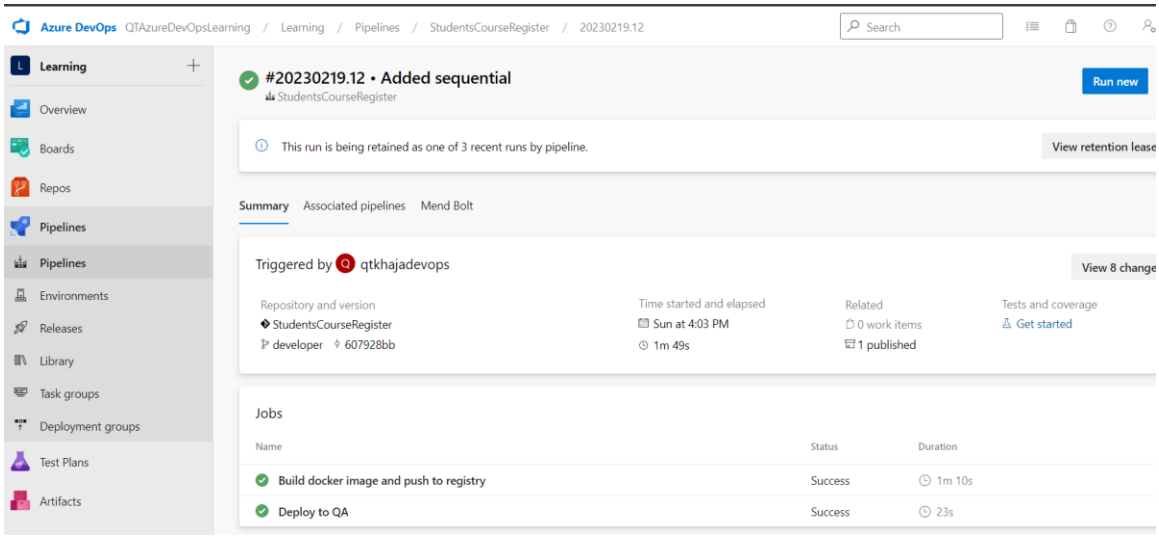


AZURE PIPELINES

You can use Azure Pipelines to automatically build, test, and deploy code to make it available to other users and deploy it to different targets, such as a **development, test, acceptance, and production** (**DTAP**) environment. It combines CI/CD to automatically build and deploy your code.

Before you can use Azure Pipelines, you should put your code in a version control system, such as Azure Repos. Azure Pipelines can integrate with a number of version control systems, such as Azure Repos, Git, TFVS, GitHub, GitHub Enterprise, Subversion, and Bitbucket Cloud. You can also use Pipelines with most application

types, such as Java, JavaScript, Node.js, Python, .NET, C++, Go, PHP, and XCode. Applications can be deployed to multiple target environments, including container registries, virtual machines, Azure services, or any on-premises or cloud target.

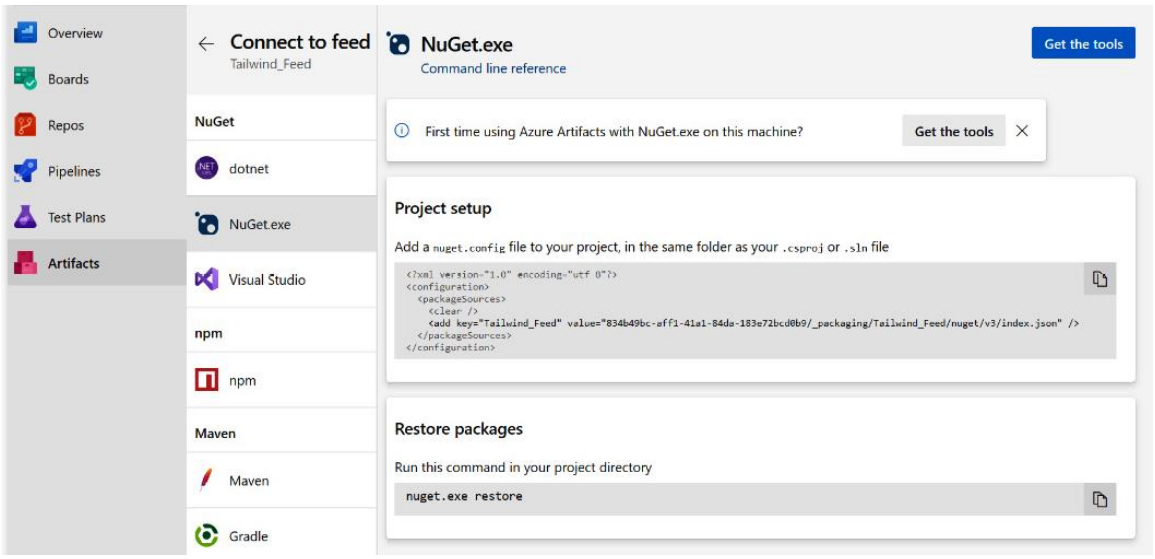


AZURE TEST PLANS

With Azure Test Plans, teams can improve their code quality using planned and exploratory services in Azure DevOps. Azure Test Plans offer features for planned manual testing, exploratory testing, user acceptance testing, and for gathering feedback from stakeholders. With manual testing, tests are organized into test plans and test suites by testers and test leads. Teams can begin testing from their Kanban boards or from the Work Hub directly. With user acceptance testing, the value that's delivered to meet customer requirements is verified. This is usually done by designated testers. Exploratory testing includes tests that are executed by the whole development team, including developers, product owners, and testers. The software is tested by exploring the software systems, without the use of test plans or test suites. Stakeholder feedback gathering is done outside the development team by marketing or sales teams. Developers can request feedback on their user stories and features from Azure DevOps. Stakeholders can then respond directly to the feedback item.

AZURE ARTIFACTS

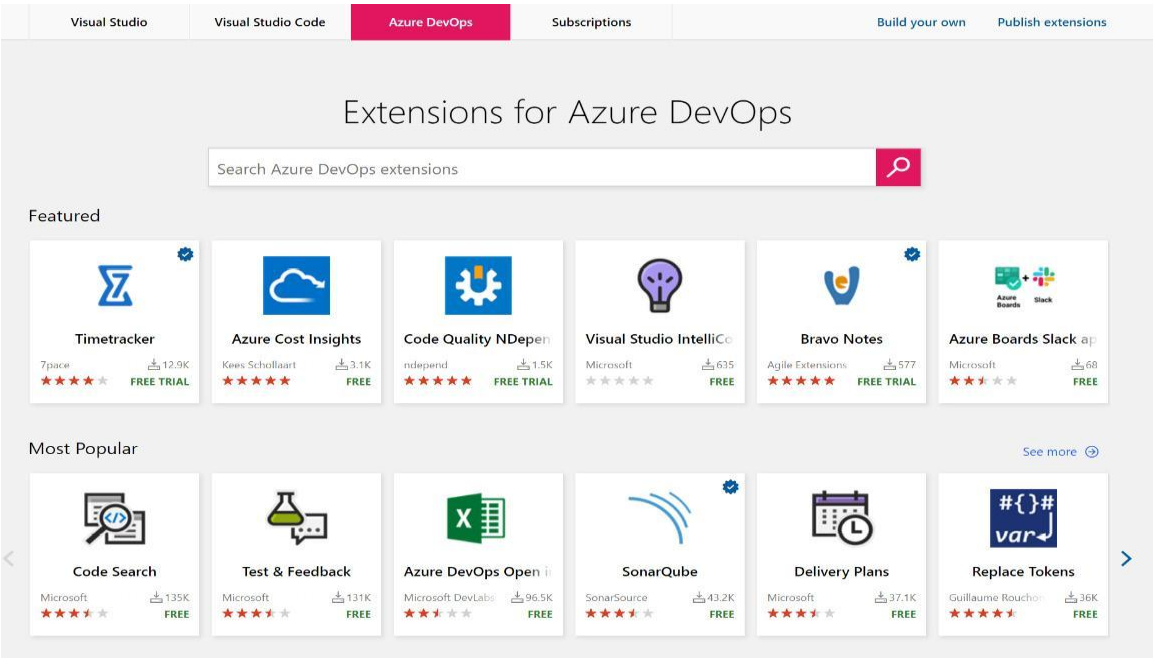
With Azure Artifacts, you can create and share NuGet, npm, Python, and Maven packages from private and public sources with teams in Azure DevOps. These packages can be used in source code and can be made available to the CI/CD pipelines. With Azure Artifacts, you can create multiple feeds that you can use to organize and control access to the packages.



EXTENSION MARKETPLACE

You can download extensions for Azure DevOps from the Visual Studio Marketplace. These extensions are simple add-ons that can be used to customize and extend your team's experience with Azure DevOps. They can help by extending the planning and tracking of work items, code testing and tracking, pipeline build and release flows, and collaboration among team members. The extensions are created by Microsoft and the community.

The following screenshot shows some of the extensions that can be downloaded from the marketplace:



Managing Projects with Azure DevOps Boards

The following topics will be covered in this chapter:

- Understanding processes and process templates
- Creating an organization
- Creating a project
- Creating and managing project activities

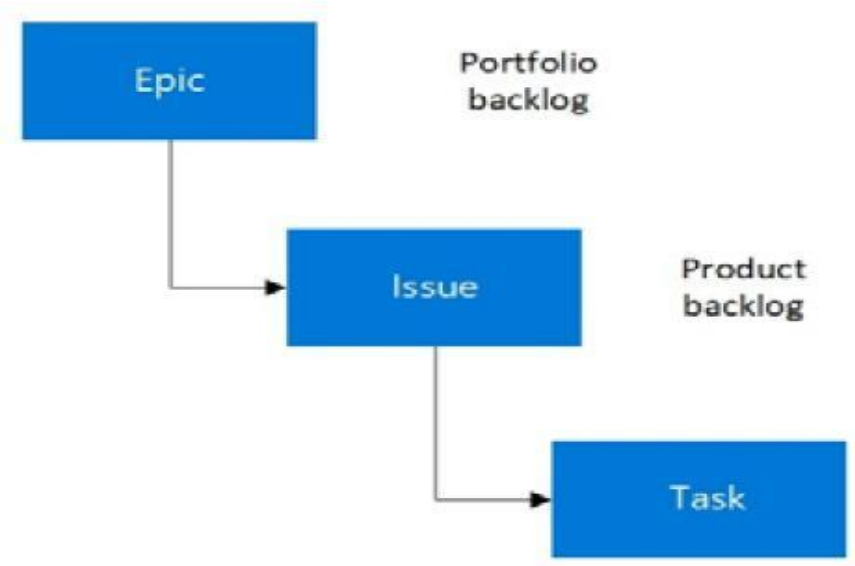
UNDERSTANDING PROCESSES AND PROCESS TEMPLATES

With Azure Boards, you can manage the work of your software projects. Teams need tools to support them that can grow and that are flexible. This includes native support for Scrum and Kanban, as well as customizable dashboards and integrated reporting capabilities and tools.

At the start of the project, teams must decide which process and process templates need to be used to support the project model that is being used. The process and the templates define the building blocks of the Work Item tracking system that is used in Azure Boards.

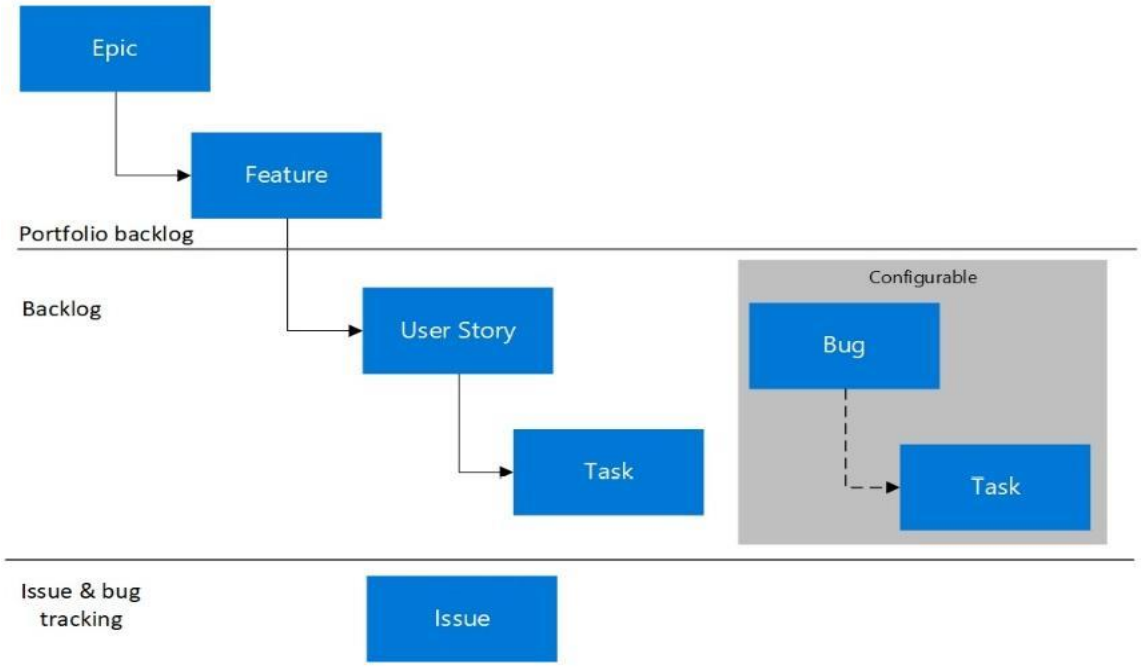
Azure DevOps supports the following processes:

- **Basic**: This is the simplest model that teams can choose. It uses **Epics**, **Issues**, and **Tasks** to track the work. These artifacts are created when you create a new basic project, as follows:

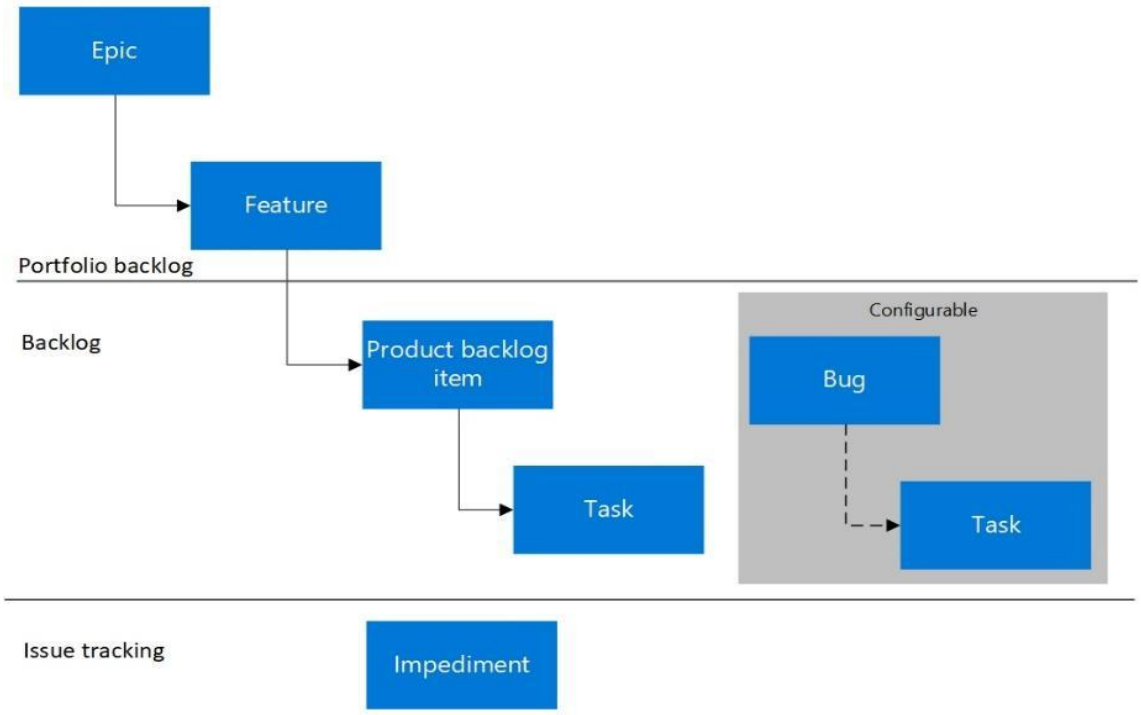


- **Agile**: Choose Agile when your team uses the Agile planning process. You can track different types of work, such as **Features**, **User Stories**, and **Tasks**. These artifacts are created when you create a new project using the Agile process. Development and test activities are tracked separately here, and Agile uses the Kanban board to track User Stories and bugs. You can also track them on the task board:

–



- **Scrum**: When your team is practicing the Scrum methodology, choose the Scrum process. This will create **Product backlog items** (**PBIs**), **Tasks**, **Bugs**, and more artifacts for the team. You can track artifacts using the Kanban board, or break PBIs and bugs down into tasks on the task board. The Scrum process is shown in the following diagram:

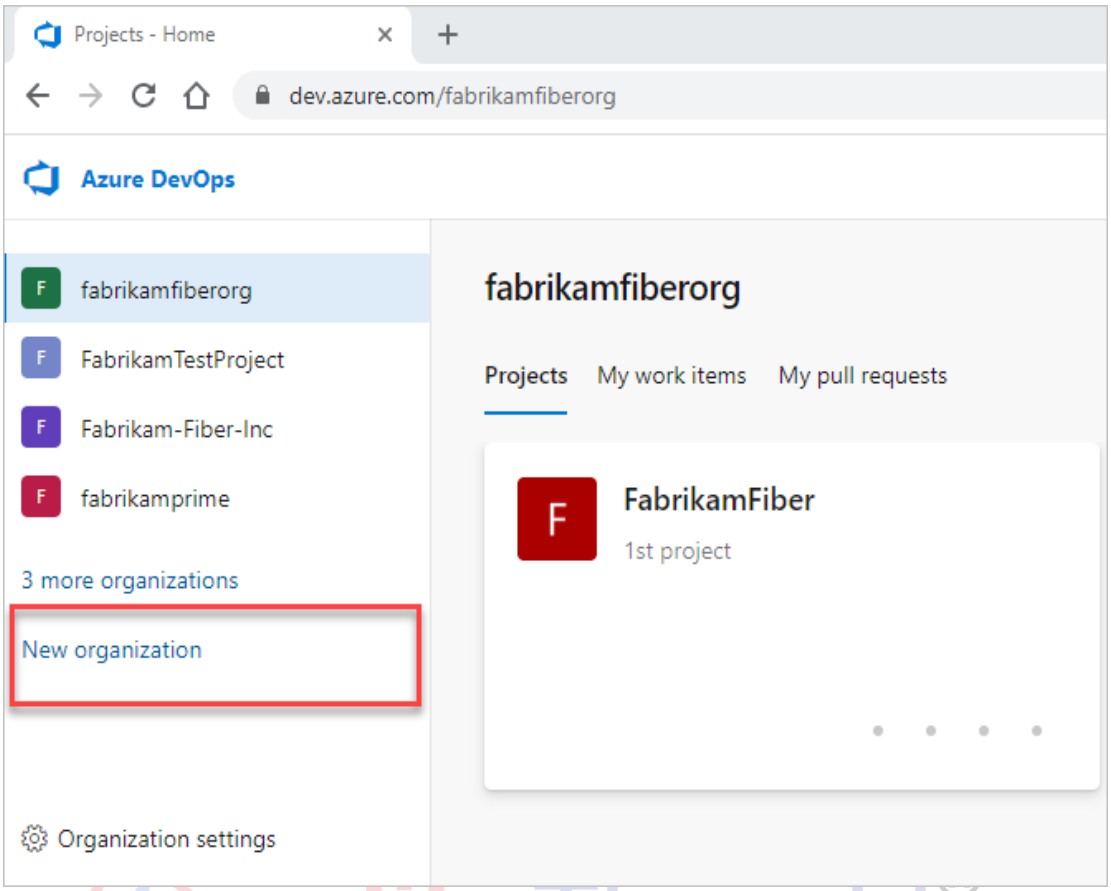


CREATING AN ORGANIZATION

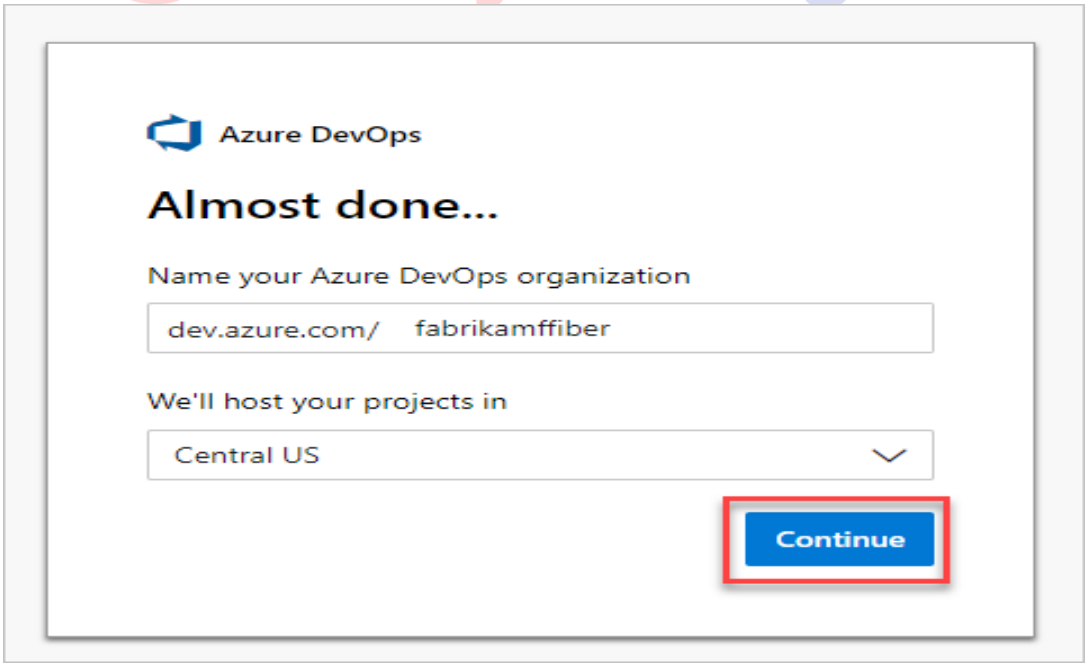
An organization in Azure DevOps is used to connect groups of related projects. You can plan and track your work here and collaborate with others when developing applications. From the organization level, you can also integrate with other services, set permissions accordingly, and set up continuous integration and deployment.

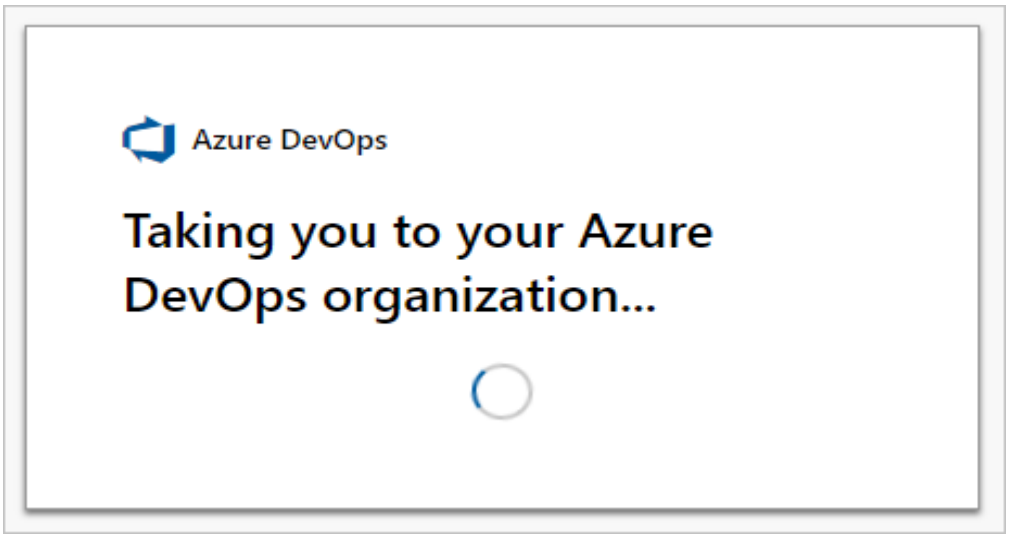
To create organization, follow the below steps

1. Open a web browser and navigate to https://dev.azure.com/.
2. Log in with your Microsoft account and from the left menu, click on **New organization**:



3. Confirm information, and then select **Continue**.



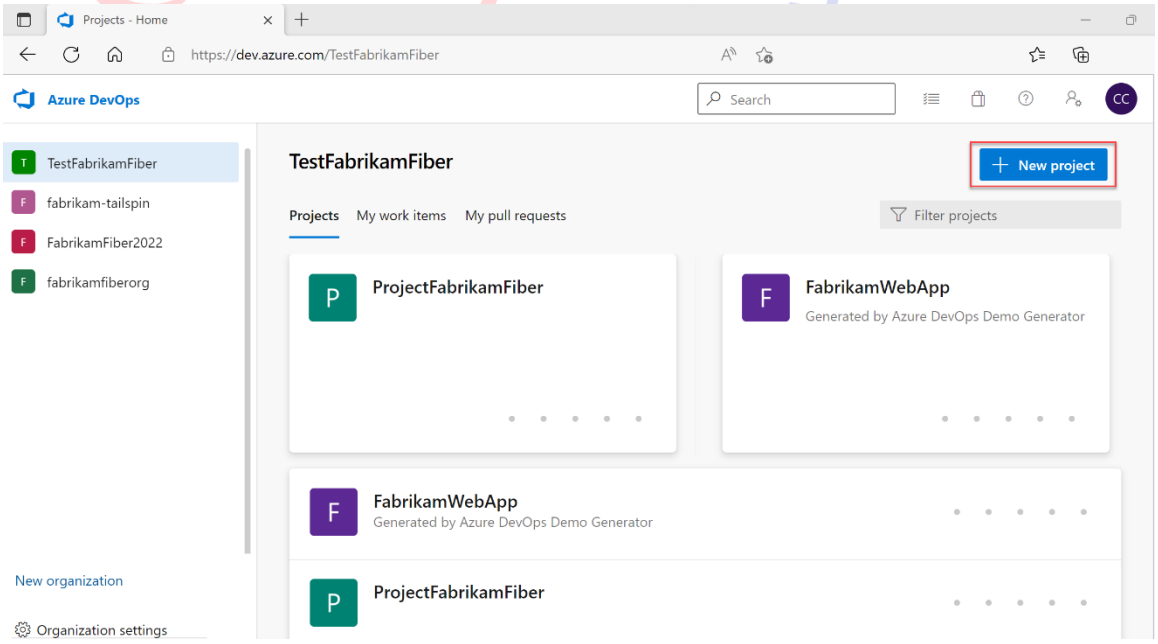


With that, the organization has been created. In the next section, we are going to learn how to add a new project to this organization.

CREATING A PROJECT

After creating a new organization, Azure DevOps automatically gives you the ability to create a new project. Perform the following steps:

- Sign in to your organization (https://dev.azure.com/{yourorganization}).
- Select New project.



- Enter the information as shown below

Create new project

Project name *

Fabrikam Test

Description

Test project for testing new features before roll out.

Visibility

Public

Anyone on the internet can view the project. Certain features like TFVC are not supported.

Private

Only people you give access to will be able to view this project.

Advanced

Version control

Git

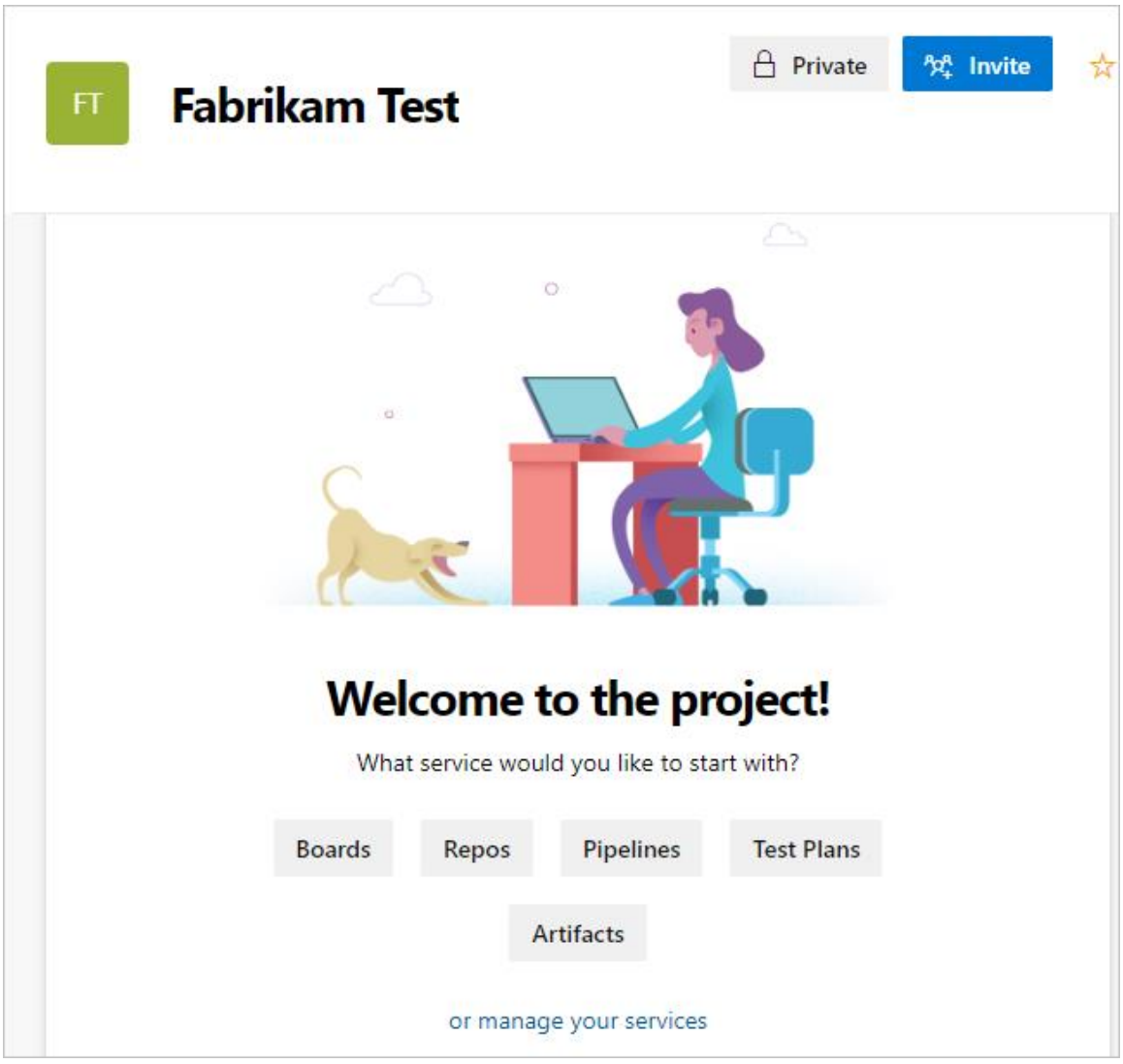
Work item process

Agile

Create Cancel

- When you choose public visibility, anyone on the internet can view your project. With private visibility, only users you give access to can view your project. For more information about public projects, see Create a public project in your organization. If the **Public** option isn't available, you need to change the policy.

- Select **Create**. Azure DevOps displays the project welcome page.



CREATING AND MANAGING PROJECT ACTIVITIES

Azure DevOps offers different project features that can be used by teams to manage their software development project, such as Work Items, backlogs, sprints, boards, and queries. These will be covered in the following sections.

Work Items

Teams use artifact Work Items to track all the work for a team. Here, you will describe what is needed for the software development project. You can track the features and the requirements, the code defects or bugs, and all other items. The

Work Items that are available to you are based on the process that was chosen when the project was created.

Work Items have three different states: **new**, **active**, and **closed**. During the development process, the team can update the items accordingly so that everyone has a complete picture of the work related to the project.

Refer the Microsoft official docs to create work item https://learn.microsoft.com/en-us/azure/devops/boards/work-items/view-add-work-items?view=azure-devops&tabs=browser

Backlogs

The product backlog is a roadmap for what teams are planning to deliver. By adding User Stories, requirements, or backlog items to it, you can get an overview of all the functionality that needs to be developed for the project.

From the backlog, Work Items can be easily reordered, prioritized, and added to sprints. Refer to the Microsoft official docs to create Backlogs https://learn.microsoft.com/en-us/azure/devops/boards/backlogs/create-your-backlog?view=azure-devops&tabs=agile-process

Boards

Another way to look at the different Work Items you have is by using boards. Each project comes with a preconfigured Kanban board that can be used to manage and visualize the flow of the work.

This board comes with different columns that represent different work stages. Here, you can get a comprehensive overview of all the work that needs to be done and what the current status of the Work Items is.

Refer to the Microsoft official docs to create Boards https://learn.microsoft.com/en-us/azure/devops/boards/boards/kanban-quickstart?view=azure-devops

Sprints

Iterations or **Sprints** are used to divide the work into a specific number of (mostly) weeks. This is based on the velocity that a team can handle; that is, the rate at which the team is burning the User Stories.

Queries

You can filter Work Items based on the filter criteria that you provide in Azure DevOps. This way, you can easily get an overview of all the Work Items that are in a particular type, state, or have a particular label. This can be done within a project, but also across different projects.

Source Code and Builds

In this section, Azure builds are covered as well as how to manage your source . code in Azure DevOps.

SOURCE CONTROL MANAGEMENT WITH AZURE DEVOPS

Source control management (SCM) is a vital part of every company that develops software professionally, but also for every developer that wants to have a safe way to store and manage their code.

When working in teams, it's absolutely necessary to have a safe central repository where all your code is stored. It's also necessary to have a system that guarantees that the code is safely shared between developers and that every modification is inspected and merged without raising conflicts

UNDERSTANDING AZURE DEVOPS PIPELINES

When adopting **Azure DevOps** in your organization, one of the main important decisions you must make is how to define the **pipeline** of your development process. A pipeline is a company-defined model that describes the steps and actions that a code base must support, from building to the final release phase. It's a key part of any DevOps architecture.

Technical requirements

To follow this chapter, you need to have the following:

- A valid organization in Azure DevOps
- An Azure subscription where you can create an Azure VM or a local machine on one of these environments so that you can install the build agent software
- Visual Studio or Visual Studio Code as your development environment
- Access to the following GitHub repository for cloning the project: https://github.com/Microsoft/PartsUnlimited

Implementing a CI/CD process

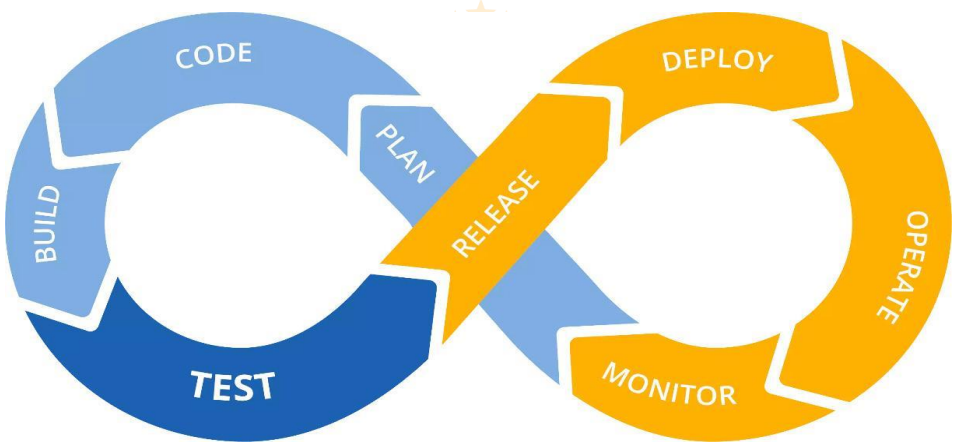
When adopting DevOps in a company, implementing the right DevOps tools with the right DevOps processes is essential and crucial. One of the fundamental flows in a DevOps implementation is the **continuous integration** (**CI**) and **continuous delivery** (**CD**) process, which can help developers build, test, and distribute a code base in a quicker, structured, and safer way.

CI is a software engineering practice where developers in a team integrate code modifications in a central repository a few times in a day. When a code modification integrated into a particular branch (normally with a pull request, as explained in the previous chapter), a new build is triggered in order to check the code and detect integration bugs quickly. Also, automatic tests (if available) are executed during this phase to check for breakages.

CD is the process that comes after the CI process. In this process, the output of the CI phase is packaged and delivered to the production stage without bugs. This is extremely helpful so that we always have a master branch that is tested, consistent, and ready to be deployed.

In DevOps, you can also have a **continuous deployment** process in place, where you can automate the deployment of your code modifications to the final production environments without manual intervention.

The typical DevOps CI/CD loop is represented in the following famous "loop" diagram:



A typical CI/CD pipeline implementation contains the following stages:

- **Commit stage**: Here, new code modifications are integrated into the code base and a set of unit tests are performed in order to check code integrity and quality.
- **Build stage**: Here, the code is automatically built and then the final results of the build process (artifacts) are pushed to the final registry.
- **Test stage**: The build code will be deployed to preproduction, where the final testing will be performed and then go to production deployment. Here, the code is tested by adopting alpha and beta deployments. The alpha deployment stage is where developers check the performance of their new builds and the interactions between builds. In the Beta

deployment stage, developers execute manual testing in order to double-check whether the application is working correctly.
- **Production deployment stage**: This is where the final application, after successfully passing all the testing requirements, goes live to the production stage.

There are lots of benefits of implementing a CI/CD process in your organizations. The main benefits are as follows:

- **Improved code quality and early bug detection**: By adopting automated tests, you can discover bugs and issues at an early stage and fix them accordingly.
- **Complete traceability**: The whole build, test, and deployment process is tracked and can be analyzed later. This guarantees that you can inspect which changes in a particular build are included and the impact that they can have on the final tests or release.
- **Faster testing and release phases**: Automating building and testing of your code base on every new commit (or before a release).

OVERVIEW OF AZURE PIPELINES

Azure Pipelines is a cloud service offered by the Azure platform that allows you to automate the building, testing, and releasing phases of your development life cycle (CI/CD). Azure Pipelines works with any language or platform, it's integrated in Azure DevOps, and you can build your code on Windows, Linux, or macOS machines

Azure Pipelines is free for public projects, while for private projects, you have up to 1,800 minutes' (30 hours) worth of pipelines for free each month. More information about pricing can be found here: https://azure.microsoft.com/en-us/pricing/details/devops/azure-devops-services/

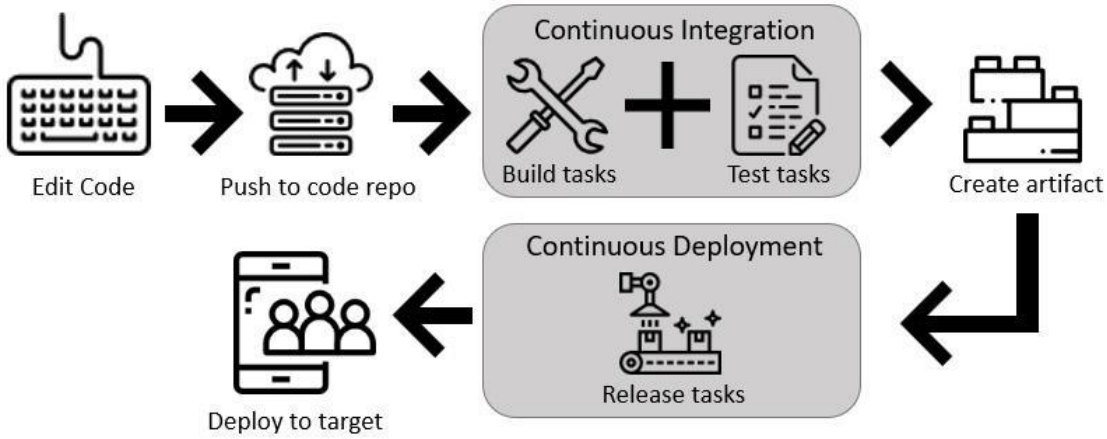
Some important feature of Azure Pipelines can be summarized as follows

- It's platform and language independent, which means you can build code on every platform using the code base you want.
- It can be integrated with different types of repositories (Azure Repos, GitHub, GitHub Enterprise, BitBucket, and so on).
- Lots of extensions (standard and community-driven) are available for building your code and for handling custom tasks.
- Allows you to deploy your code to different cloud vendors.
- You can work with containerized applications such as Docker, Azure Container Registry, or Kubernetes.

To use Azure Pipelines, you need the following:

- An organization in Azure DevOps, where you can create public or private projects
- A source code stored in a version control system (such as Azure DevOps Repos or GitHub)

Azure Pipelines works with the following schema:



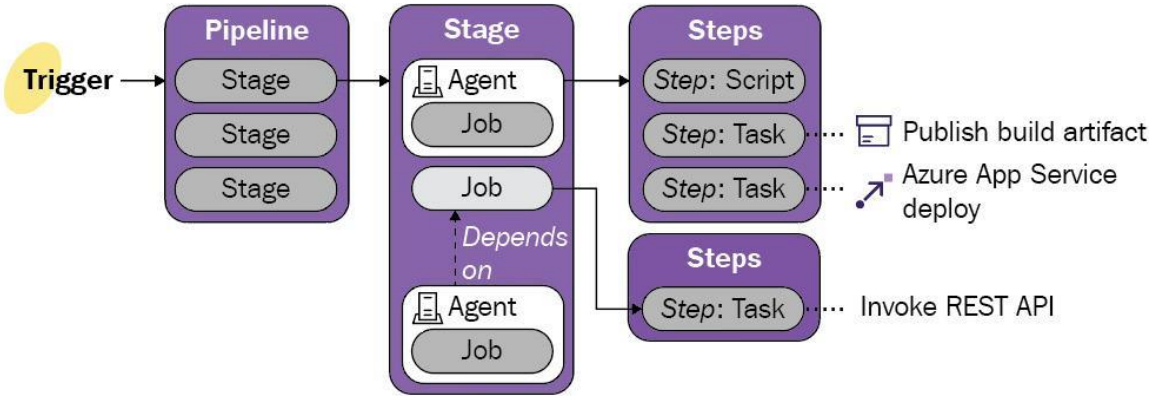
When your code is committed to a particular branch inside a repository, the **build pipeline** engine starts, build and test tasks are executed, and if all is successfully completed, your app is built and you have the final output (artifact). You can also create a **release pipeline** that takes the output of your build and releases it to the target environment (staging or production).

To start using Azure Pipelines, you need to create a **pipeline**. A pipeline in Azure DevOps can be created in the following two ways:

- **Using the Classic interface**: This allows you to select some tasks visually from a list of possible tasks. You only need to fill in the parameters for these tasks.
- **Using a scripting language called YAML**: The pipeline can be defined by creating a YAML file inside your repository with all the needed steps.

Using the classic interface can be easier initially, but remember that many features are only available on YAML pipelines. A YAML pipeline definition is a file, and this can be versioned and controlled just like any other file inside a repository. You can easily move the pipeline definition between projects (this is not possible with the Classic interface).

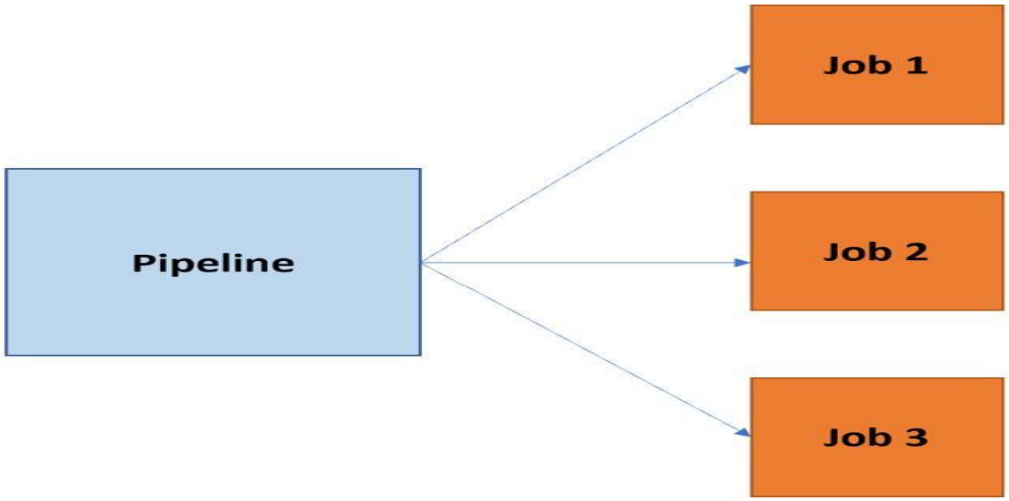
An Azure Pipeline can be represented as follows (courtesy of Microsoft):



A pipeline starts from a **trigger** (a manual trigger, a push inside a repository, a pull request, or a schedule). A pipeline is normally composed of one or more **stages** (logical separation of concerns in a pipeline, such as building, testing, deployment, and so on; they can run in parallel), and each stage contains one or more **jobs** (a set of steps that can also run in parallel). Every pipeline contains at least one stage if you don't explicitly create it. Each job runs on an **agent** (service or piece of software that executes the job). Every step is composed of a **task** that performs some action on your code (sequentially). The final output of a pipeline is an **artifact** (collection of files or packages published by the build).

When creating a pipeline, you need to define a set of jobs and tasks for automating your builds (or multi-phased builds). You have native support for testing integration, release gates, automatic reporting, and so on.

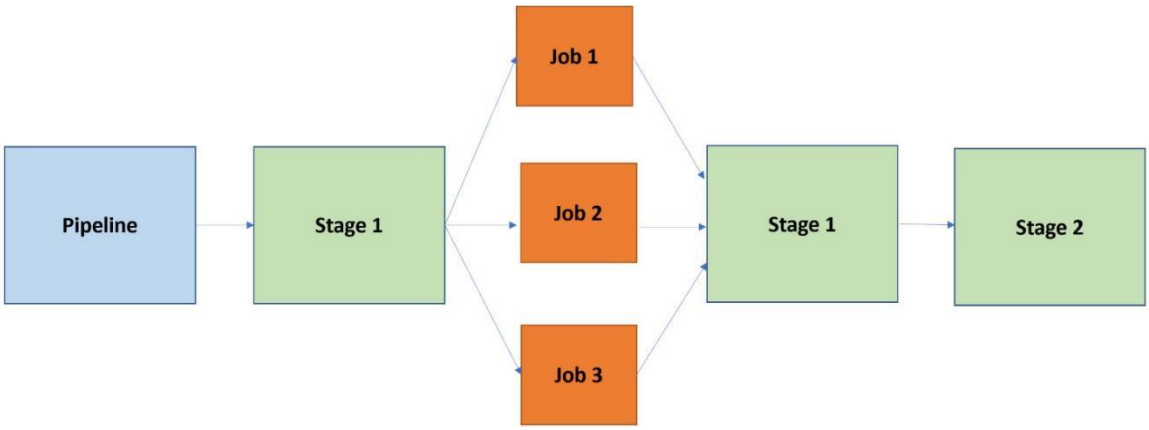
When defining multiple jobs within a pipeline, these jobs are executed in parallel. A pipeline that contains multiple jobs is called a **fan-out** scenario:



A pipeline with multiple jobs in a single stage can be represented as follows:

```
pool:
  vmImage: 'ubuntu-latest'
jobs:
- job: job1
  steps:
  - bash: echo "Hello!"
  - bash: echo "I'm job 1"
- job: job2
  steps:
  - bash: echo "Hello again…"
  - bash: echo "I'm job 2"
```

If you're using stages when defining your pipeline, this is what is called a fan-out/fan-in scenario:



Here, each stage is a fan-in operation, where all the jobs in the stage (which can consist of multiple tasks that run in sequence) must be finished before the next stage can be triggered (only one stage can be executing at a time).

UNDERSTANDING BUILD AGENTS

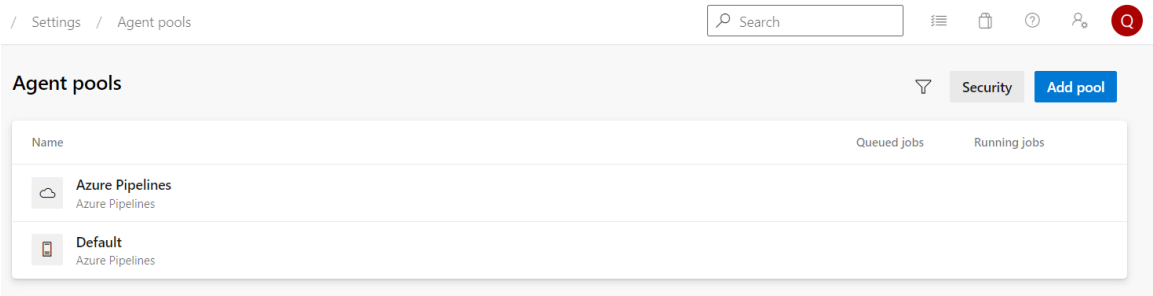
To build and deploy your code using Azure Pipelines, you need at least one agent. An agent is a service that runs the jobs defined in your pipeline. The execution of these jobs can occur directly on the agent's host machine or in containers.

When defining agents for your pipeline, you have essentially two types of possible agents:

- **Microsoft-hosted agents**: This is a service totally managed by Microsoft and it's cleared on every execution of the pipeline (on each pipeline execution, you have a fresh new environment).
- **Self-hosted agents**: This is a service that you need to set up and manage by yourself. This can be a custom virtual machine on Azure or a custom on-premise machine inside your infrastructure. In a self-hosted agent, you can install all the software you need for your builds, and this is persisted on every pipeline execution. A self-hosted agent can be on Windows, Linux, macOS, or in a Docker container.

Microsoft-hosted agents

Microsoft-hosted agents is the simplest way to define an agent for your pipeline. Azure Pipelines provides a Microsoft-hosed agent pool by default called **Azure Pipelines**:



By selecting this agent pool, you can create different virtual machine types for executing your pipeline. At the time of writing, the available standard agent types are as follows:

Image	Classic Editor Agent Specification	YAML VM Image Label
Windows Server 2022 with Visual Studio 2022	*windows-2022*	`windows-latest` OR `windows-2022`
Windows Server 2019 with Visual Studio 2019	*windows-2019*	`windows-2019`
Ubuntu 22.04	*ubuntu-22.04*	`ubuntu-latest` OR `ubuntu-22.04`
Ubuntu 20.04	*ubuntu-20.04*	`ubuntu-20.04`
Ubuntu 18.04 (deprecated starting 8/8/2022 and unsupported by 4/1/2023)	*ubuntu-18.04*	`ubuntu-18.04`
macOS 12 Monterey	*macOS-12*	`macOS-latest` OR `macOS-12`
macOS 11 Big Sur	*macOS-11*	`macOS-11`
macOS X Catalina 10.15 (deprecated starting 5/31/2022 and unsupported by 4/1/2023)	*macOS-10.15*	`macOS-10.15`

Each of these images has its own set of software automatically installed. You can install additional tools by using the pre-defined Tool Installer task in your pipeline definition. More information can be found here: https://learn.microsoft.com/en-us/azure/devops/pipelines/tasks/reference/?view=azure-pipelines#tool-tasks

When you create a pipeline using a Microsoft-hosted agent, you just need to specify the name of the virtual machine image to use for your agent from the preceding table. As an example, this is the definition of a hosted agent that's using Windows Server 2019 with a Visual Studio 2019 image:

```
- job: Windows
  pool:
    vmImage: 'windows-latest'
```

When using a Microsoft-hosted agent, you need to remember the following:

- You cannot sign in on the agent machine
- The agent runs on a Standard DS2v2 Azure Virtual Machine and you cannot increase that capacity.
- It runs as an administrator user on the Windows platform and as a *passwordless sudo* user on the Linux platform.
- The Microsoft-hosted agent runs in the same Azure geography as your Azure DevOps organization, but it's not guaranteed that it will run in the same region too (an Azure geography contains one or more regions).

Self-hosted agents

While Microsoft-hosted agents are a SaaS service, self-hosted agents are private agents that you can configure as per your needs by using Azure virtual machines or directly using your on-premise infrastructure. You are responsible for providing all the necessary software and tools to execute your pipeline and you're responsible for maintaining and upgrading your agent.

A self-hosted agent can be installed on the following platforms:

- Windows
- Linux
- macOS
- Docker

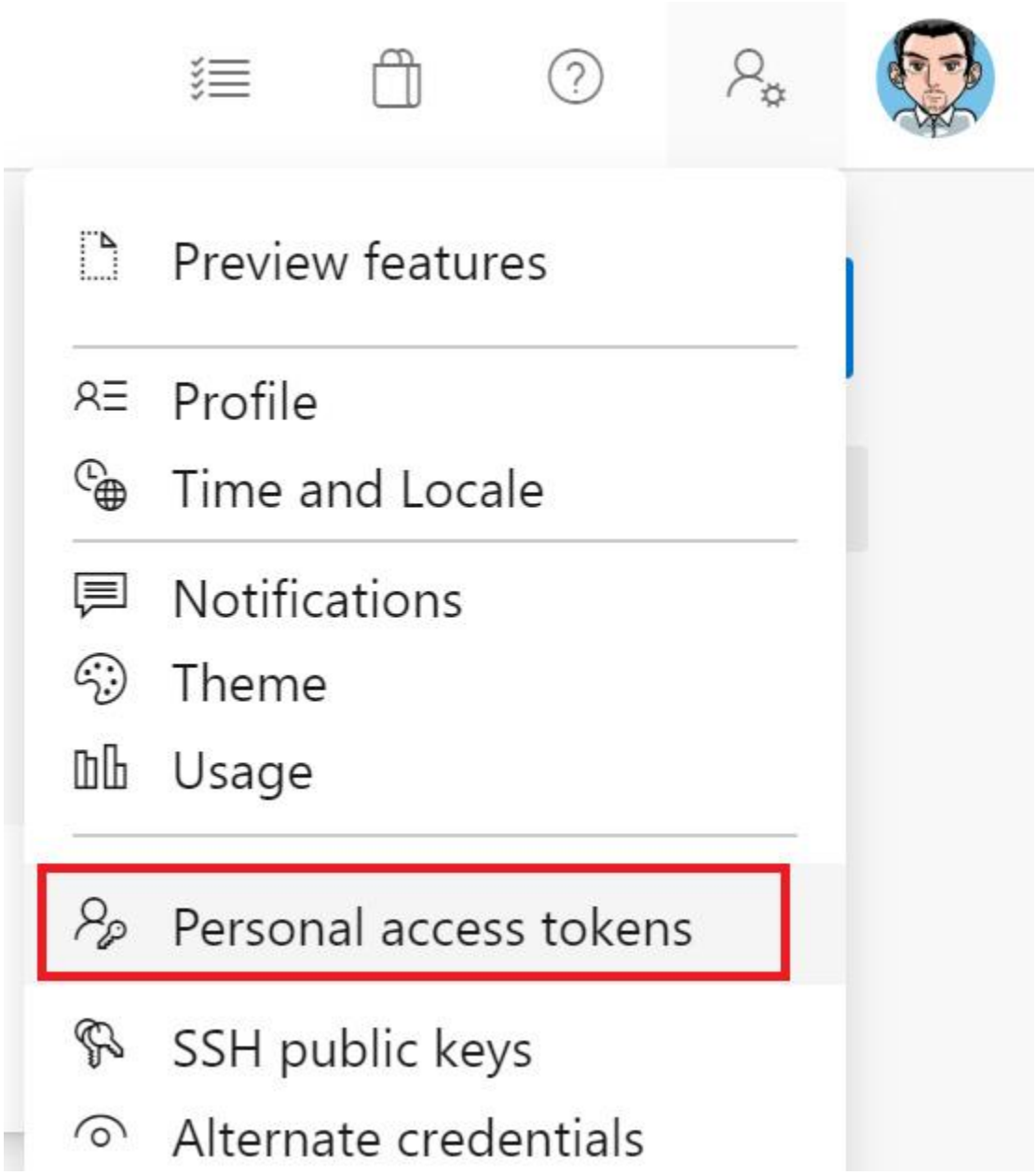
Creating a self-hosted agent involves completing the following activities:

- Prepare the environment
- Prepare permissions on Azure DevOps
- Download and configure the agent
- Start the agent

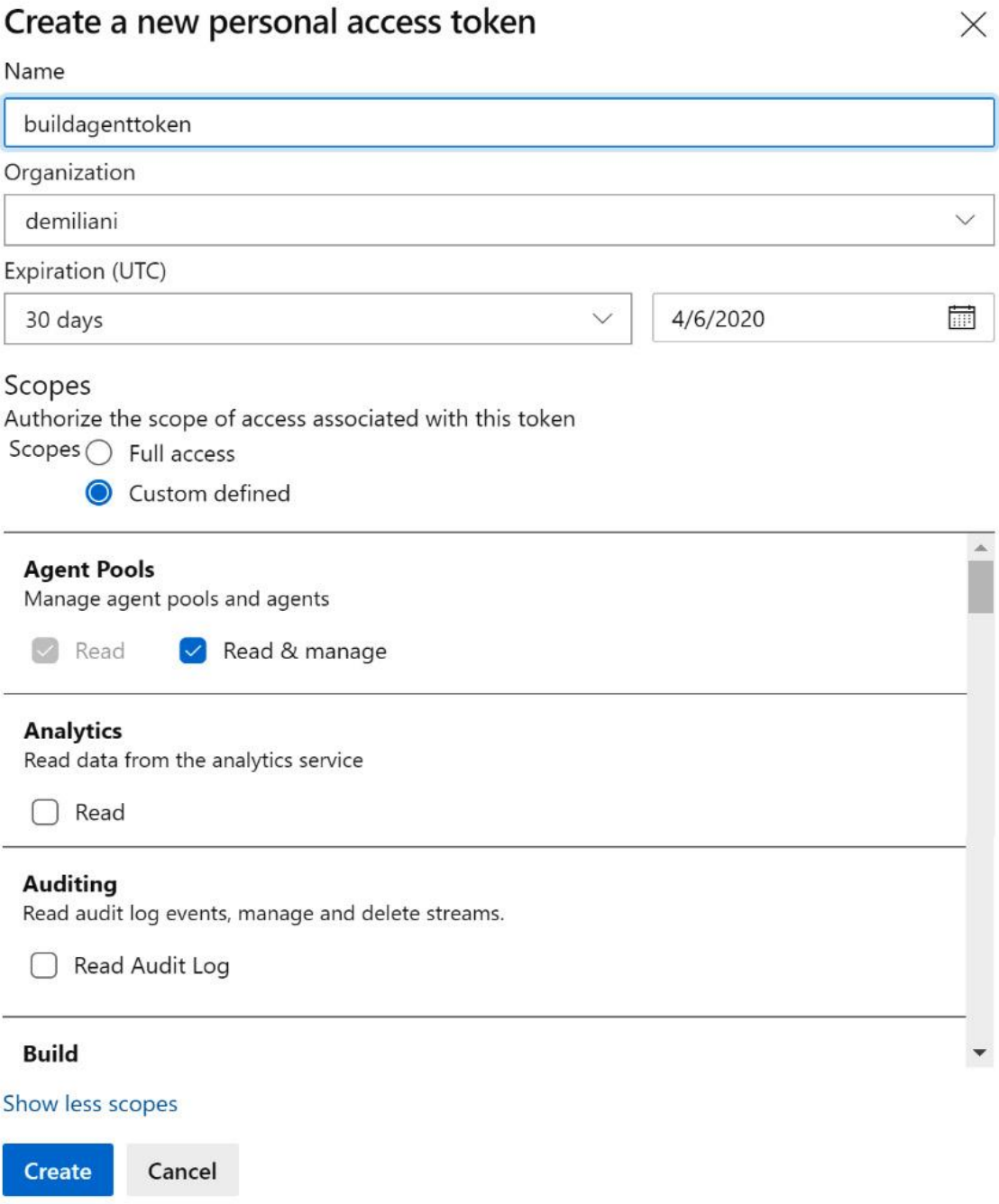
Creating a self-hosted Windows agent

A self-hosted Windows agent is used to build and deploy applications built on top of Microsoft's platforms (such as .NET applications, Azure cloud apps, and so on) but also for other types of platforms, such as Java and Android apps.

The first step to perform when creating an agent is to register the agent in your Azure DevOps organization. To do so, you need to sign into your DevOps organization as an administrator and from the **User Settings** menu, click on **Personal access tokens**:



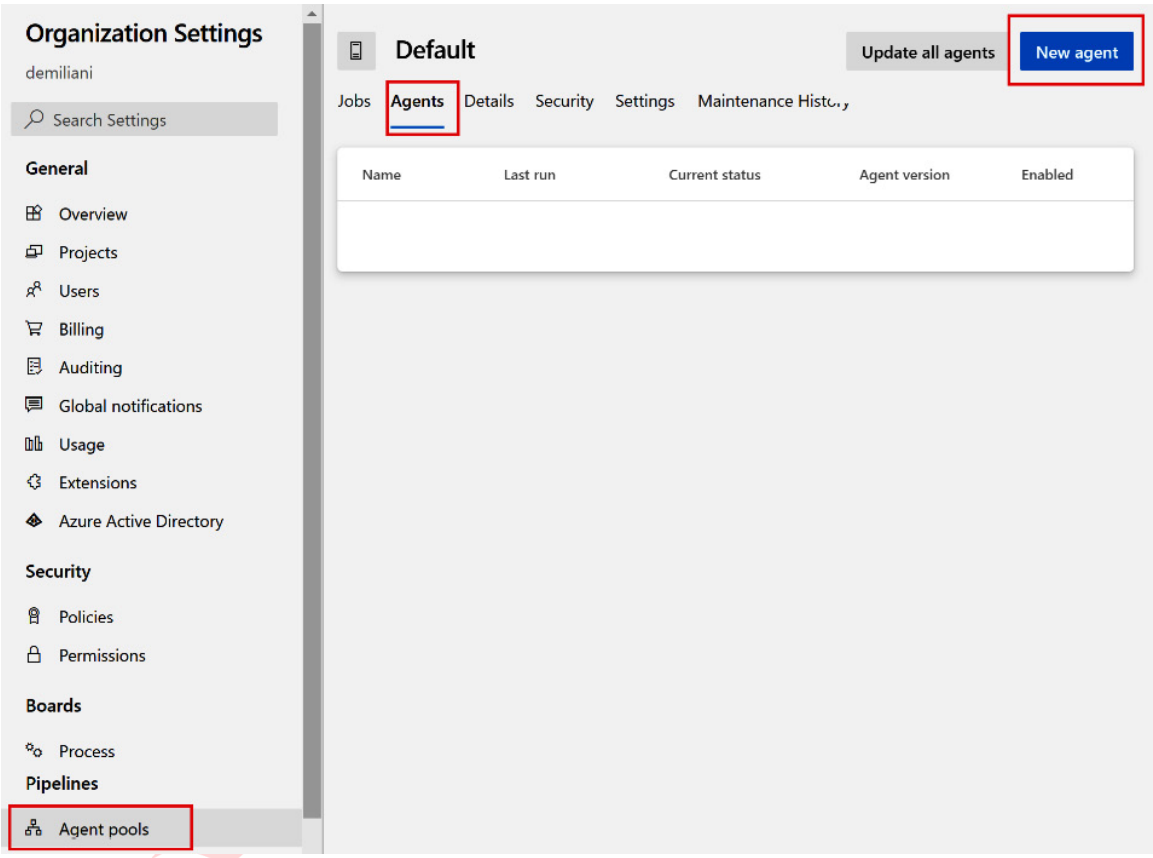
Here, you can create a new personal access token for your organization with an expiration date and with full access or with a custom defined access level (if you select the custom defined scope, you need to select the permission you want for each scope). To see the complete list of available scopes, click on the **Show all scopes** link at the bottom of this window:



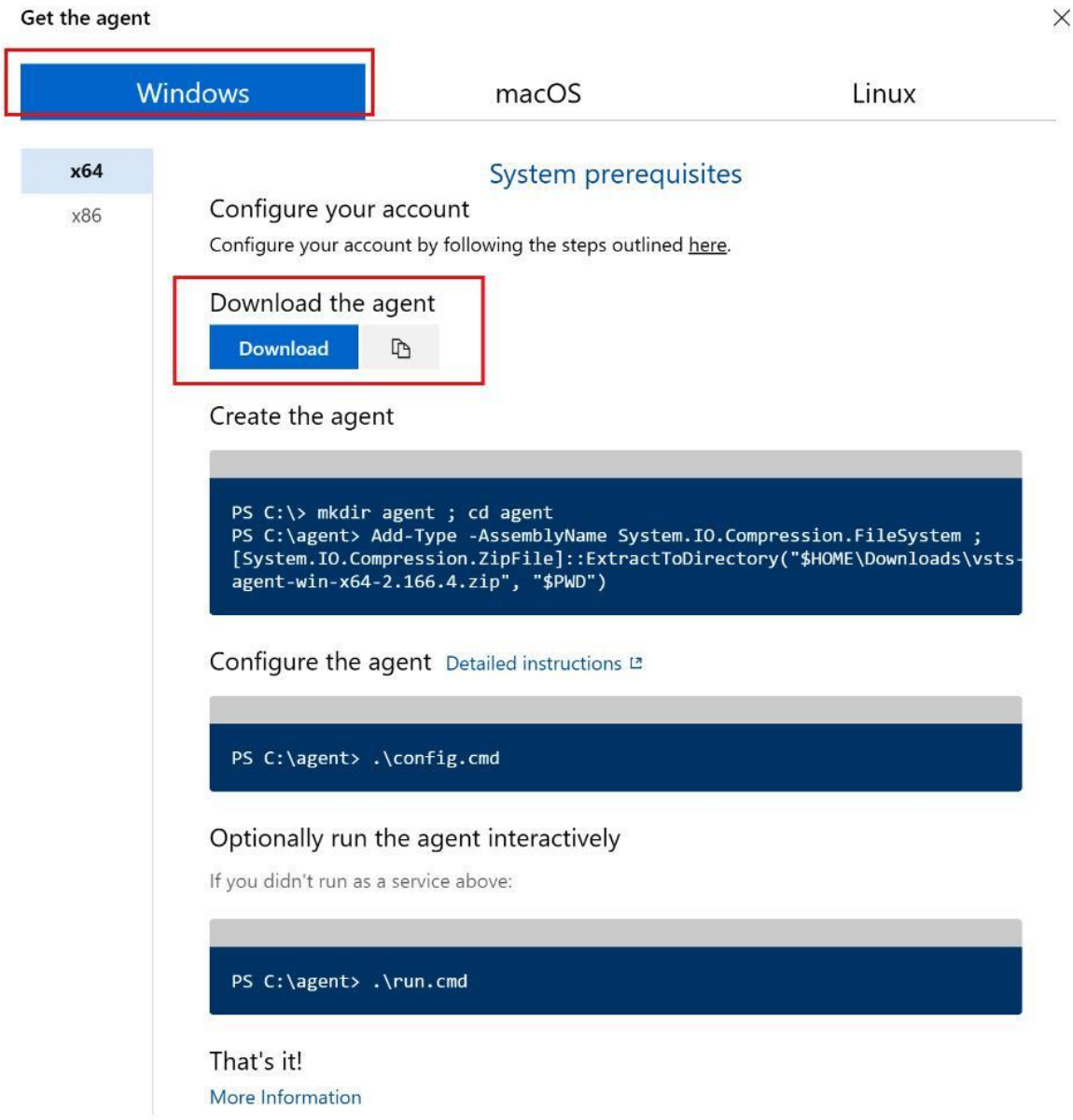
Please check that the **Agent Pools** scope has the **Read & manage** permission enabled.

When finished, click on **Create** and then copy the generated token before closing the window (it will only be shown once).

Now, you need to download the agent software and configure it. From **Organization Settings** | **Agent Pools**, select the **Default** pool and from the **Agents** tab, click on **New agent**:



The **Get the agent** window will open. Select **Windows** as the target platform, select **x64** or **x86** as your target agent platform (machine) accordingly, and then click on the **Download** button:



This procedure will download a package (normally called **vsts-agent-win-x64-2.166.4.zip**). You need to run this package (**config.cmd**) on the agent machine (an Azure VM or your on-premise server, which will act as an agent for your builds)

The setup will ask you for the following:

- The URL of your Azure DevOps organization (https://dev.azure.com/{your-organization})
- The personal access token to use (created previously)

When running the agent (interactively or as a service), it's recommended to run it as a service if you want to automate builds.

After inserting these parameters, the setup registers the agent:

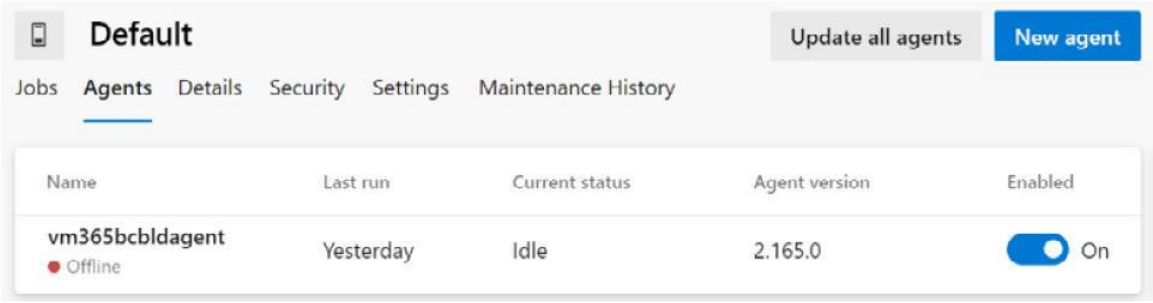
```
Enter authentication type (press enter for PAT) >
Enter personal access token > ********************************************************
Connecting to server ...

>> Register Agent:
```

To register the agent, you need to insert the agent pool, the agent name, and the work folder (you can leave the default value as-is).

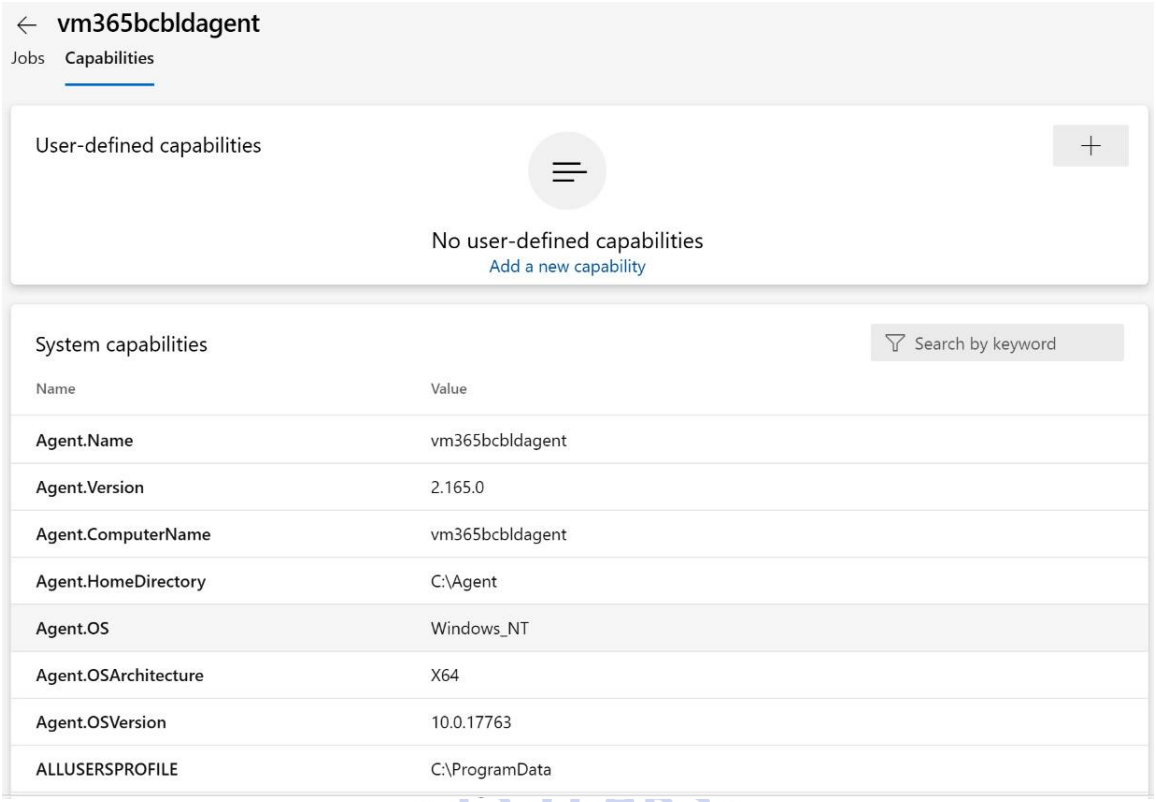
Finally, you need to decide whether your agent must be executed *interactively* or *as a service*. As we mentioned previously, running the agent as a service is recommended, but in many cases, the interactive option can be helpful because it gives you a console where you can see the status and running UI tests.

In both cases, please be aware of the user account you select for running the agent. The default account is the built-in Network Service user, but this user normally doesn't have all the needed permissions on local folders. Using an administrator account can help you solve a lot of problems.

If the setup has been completed successfully, you should see a service running on your agent machine and a new agent that pops up on your agent pool in Azure DevOps:



If you select the agent and then go to the **Capabilities** section, you will be able to see all its capabilities (OS version, OS architecture, computer name, software installed, and so on):



The agent's capabilities can be automatically discovered by the agent software or added by you (user-defined capabilities) if you click on the **Add a new capability** action. Capabilities are used by the pipeline engine to redirect a particular build to the correct agent according to the required capabilities for the pipeline (demands).

When the agent is online, it's ready to accept your code build, which should be queued.

Remember that you can also install multiple agents on the same machine (for example, if you want the possibility to execute core pipelines or handle jobs in parallel), but this scenario is only recommended if the agents will not share resources.

When to use a Microsoft-hosted or a self-hosted agent

Microsoft-hosted agents are normally useful when you have a standard code base and you don't need particular software or environment configuration to build your code. If you're in this scenario, using a Microsoft-hosted agent is recommended because you don't have to worry about creating environments.

Self-hosted agents are the way to go when you need a particular environment configuration, when you need a particular piece of software or tools installed on

the agent, and when you need more power for your builds. Self-hosted agents are also the way to go when you need to preserve the environment between each run of your builds. A self-hosted agent is normally the right choice when you need to have better control of your agent or you wish to deploy your build to on-premise environments (not accessible externally). It also normally allows you to save money.

An overview of YAML, the scripting language that allows you to define a pipeline.

OVERVIEW OF THE YAML LANGUAGE

YAML, an acronym for **YAML Ain't Markup Language**, is a human-readable scripting language used for data serialization and normally used for handling configurations definitions for applications. It can be considered a superset of JSON.

YAML uses indentation for handling the structure of the object's definitions, and it's insensitive to quotation marks and braces. It's simply a data representation language and is not used for executing commands.

With Azure DevOps, YAML is extremely important because it allows you to define a pipeline by using a script definition instead of a graphical interface (that cannot be ported between projects).

The official YAML website can be found here: https://yaml.org/

A YAML structure is based on key-value elements:

Key: Value # This is a comment

In the following sections, we'll learn how to define objects in YAML.

Scalars

As an example, the following are scalar variables that have been defined in YAML:

```
Number: 1975
quotedText: "some text description"
notQuotedtext: strings can be also without quotes
boolean: true
nullKeyValue: null
```

You can also define multi-line keys by using **?**, followed by a space, as follows:

```
? |
  This is a key
```

that has multiple lines
: and this is its value

Collections and lists

This is a YAML definition for a collection object:

Cars:
- Fiat
- Mercedes
- BMW

You can also define nested collections:

- Drivers:
  name: Stefano Demiliani
  age: 45
  Driving license type:
  - type: full car license
    license id: ABC12345
    expiry date: 2025-12-31

Dictionaries

You can define a **Dictionary** object by using YAML in the following way:

CarDetails:
  make: Mercedes
  model: GLC220
  fuel: Gasoline

Document structure

YAML uses three dashes, **---**, to separate directives from document content and to identify the start of a document. As an example, the following YAML defines two documents in a single file:

---# Products purchased
- item   : Surface Book 2
  quantity: 1
- item   : Surface Pro 7
  quantity: 3
- item   : Arc Mouse
  quantity: 1
Product out of stock

- item : Surface 4
- item : Microsoft Trackball

Complex object definition

As an example of how to define a complex object in YAML, the following is the representation used for an **Invoice** object:

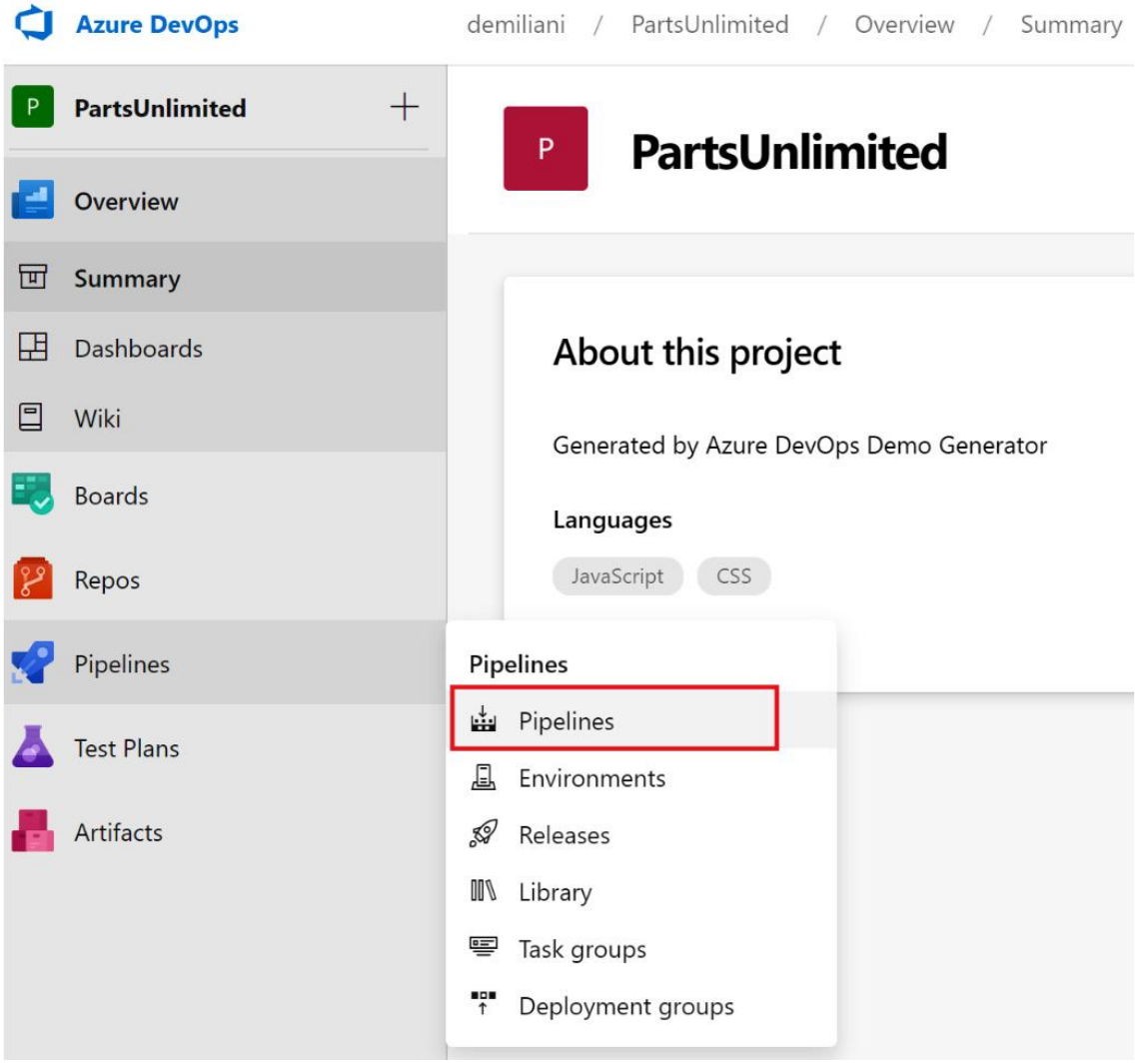
```
---
# A sample yaml file
company: qualitythought.in
domain:
 - devops
 - devsecops
tutorial:
  - yaml:
      name: "YAML Ain't Markup Language"
      type: awesome
      born: 2001
  - json:
      name: JavaScript Object Notation
      type: great
      born: 2001
  - xml:
      name: Extensible Markup Language
      type: good
      born: 1996
author: omkarbirade
published: true
```

CREATING A BUILD PIPELINE WITH AZURE DEVOPS

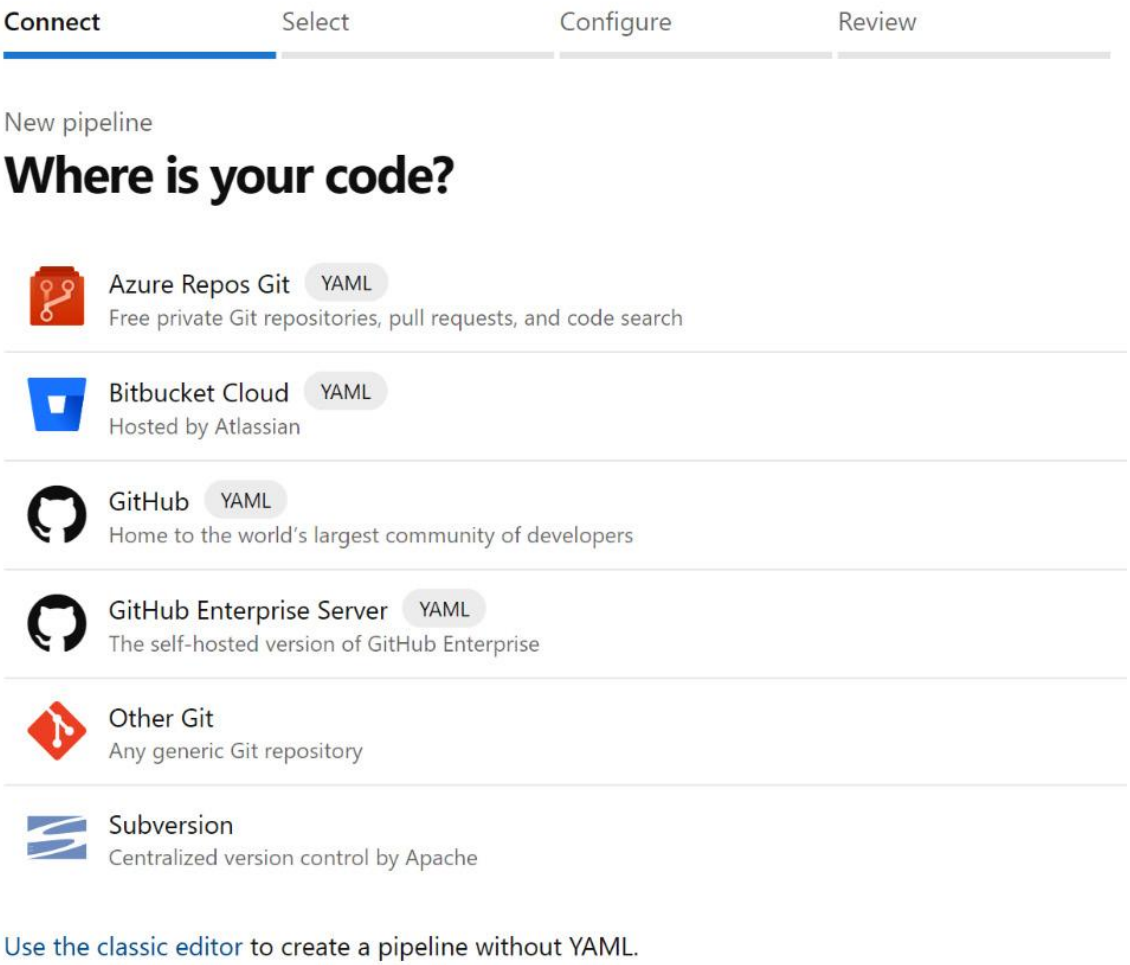
Having a build pipeline in place is a fundamental step if you want to implement continuous integration for your code (having your code automatically built and tested on every commit).

The prerequisite to creating a build pipeline with Azure DevOps is obviously to have some code stored inside a repository.

To create a build pipeline with Azure DevOps, you need to go to the **Pipelines** hub and select the **Pipelines** action:



From here, you can create a new build pipeline by selecting the **New pipeline** button. When pressed, you will see the following screen, which asks you for a code repository:



This screen is extremely important. From here, you can start creating a build pipeline in two possible ways (described previously):

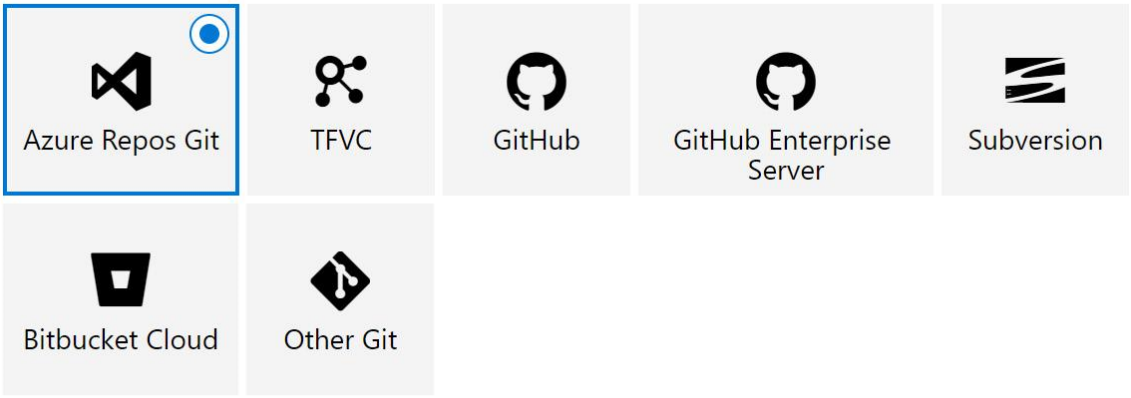
- Using a YAML file to create your pipeline definition. This is what happens when you select the repository in this window.
- Using the classic editor (graphical user interface). This is what happens when you click on the **Use the classic editor** link at the bottom of this page.

Pipeline definition with the classic editor

The classic editor permits you to define a build pipeline for your project graphically by selecting pre-defined actions.

When you click on the **Use the classic editor** link, you need to select the repository where your code is stored (**Azure Repos Git**, **GitHub**, **GitHub Enterprise Server**, **Subversion**, **TFVC**, **Bitbucket Cloud**, or **Other Git**) and the branch that the build pipeline will be connected to:

Select a source



Team project

PartsUnlimited

Repository

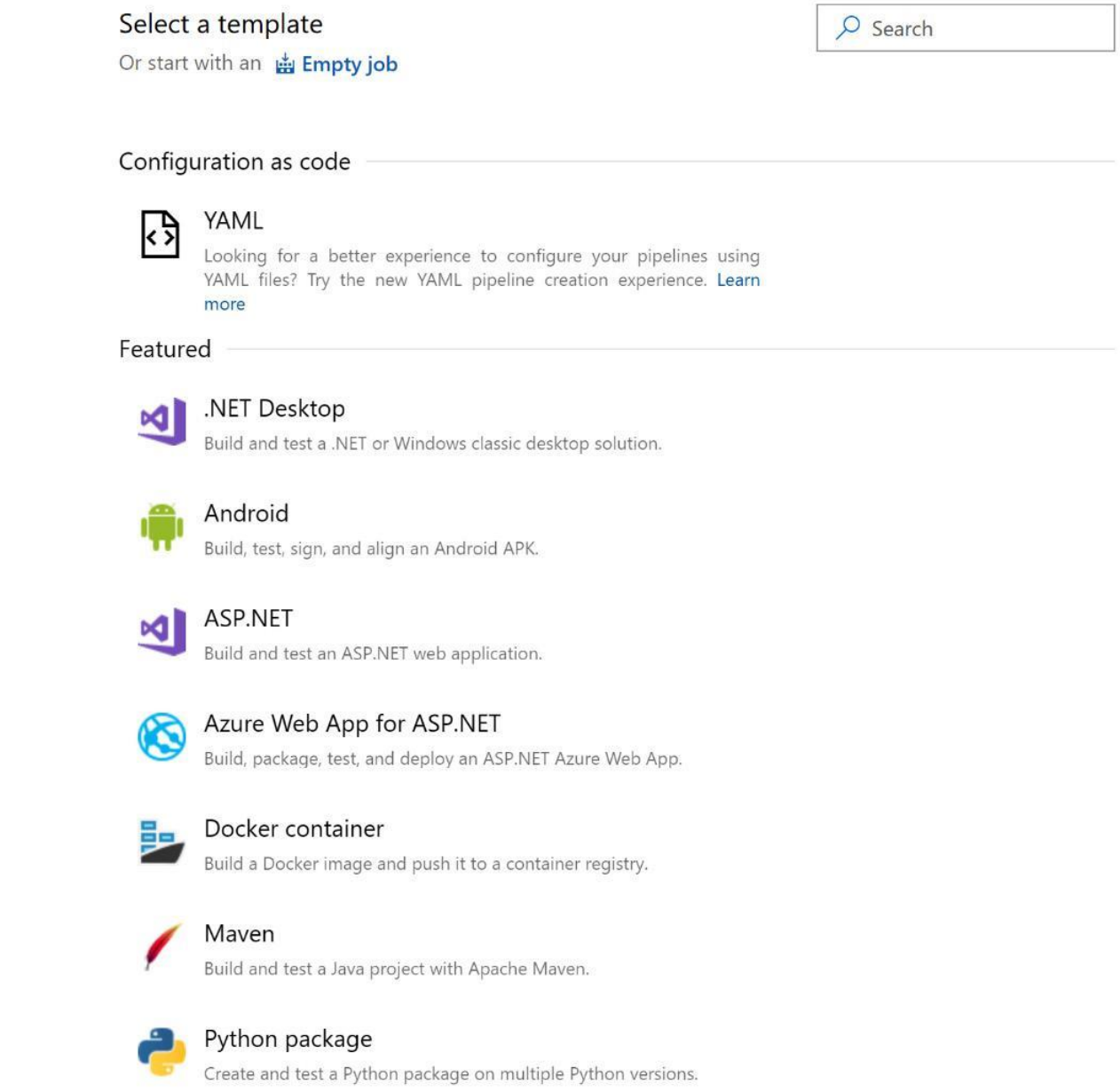
PartsUnlimited

Default branch for manual and scheduled builds

master

Continue

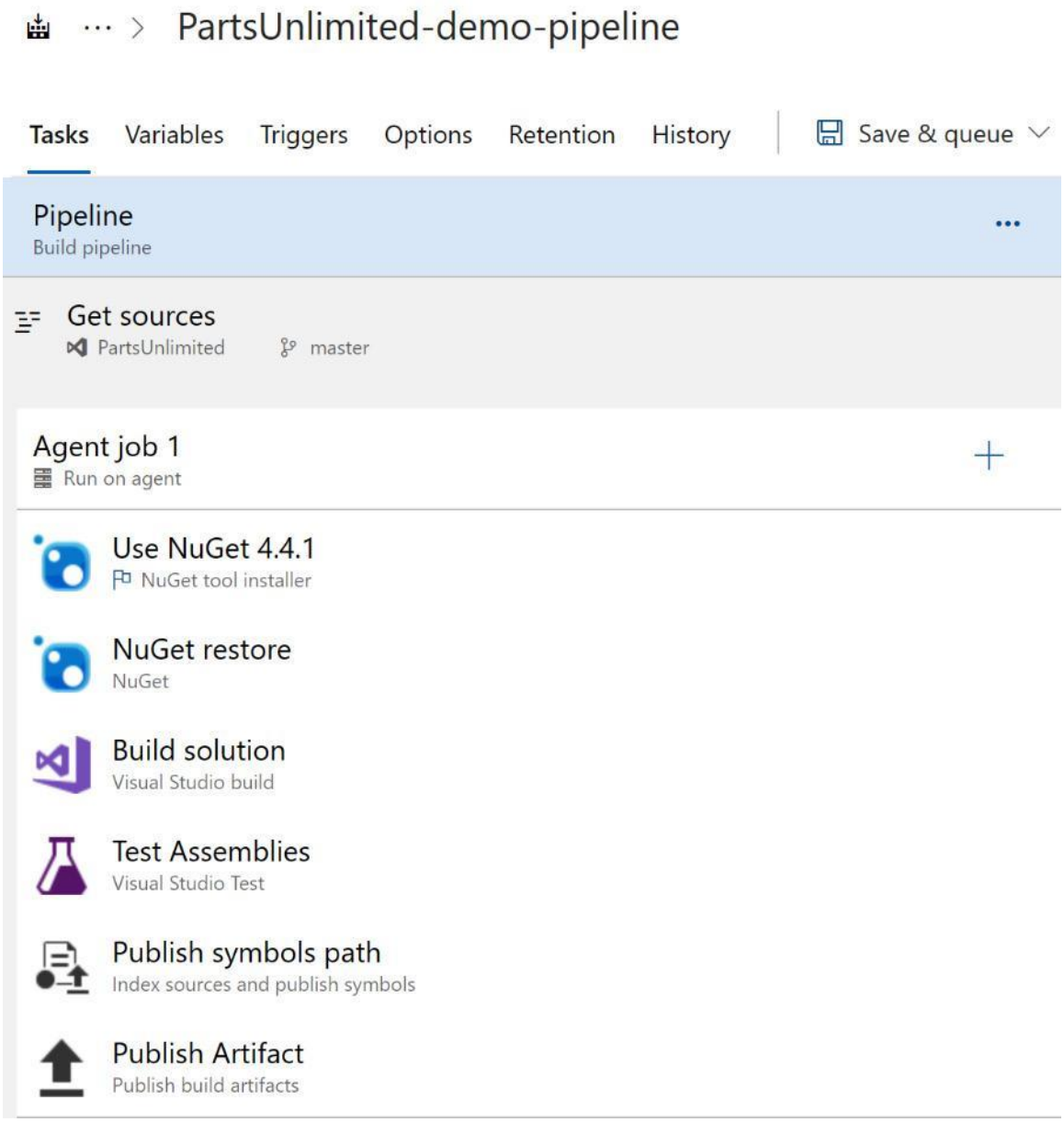
Then, you need to choose a template for the kind of app you're building. You have a set of predefined templates to choose from (that you can customize later), but you can also start from an empty template:



If predefined templates fit your needs, you can start by using them; otherwise, it's recommended to create a custom pipeline by selecting the actions you need.

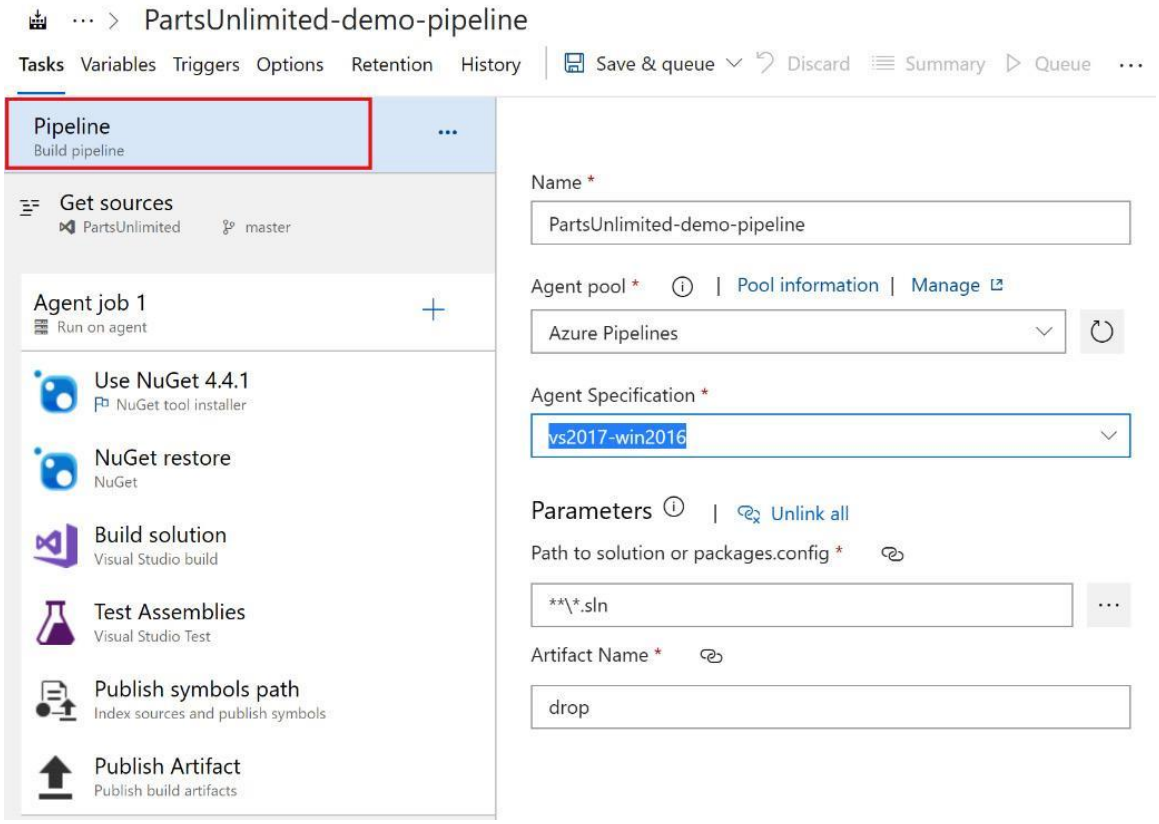
Here, my application that's stored in the Azure DevOps project repository is an ASP.NET web application (an e-commerce website project called **PartsUnlimited**; you can find the public repository at the following URL: https://github.com/Microsoft/PartsUnlimited), so I've selected the ASP.NET template.

When selected, this is the pipeline template that will be created for you automatically:

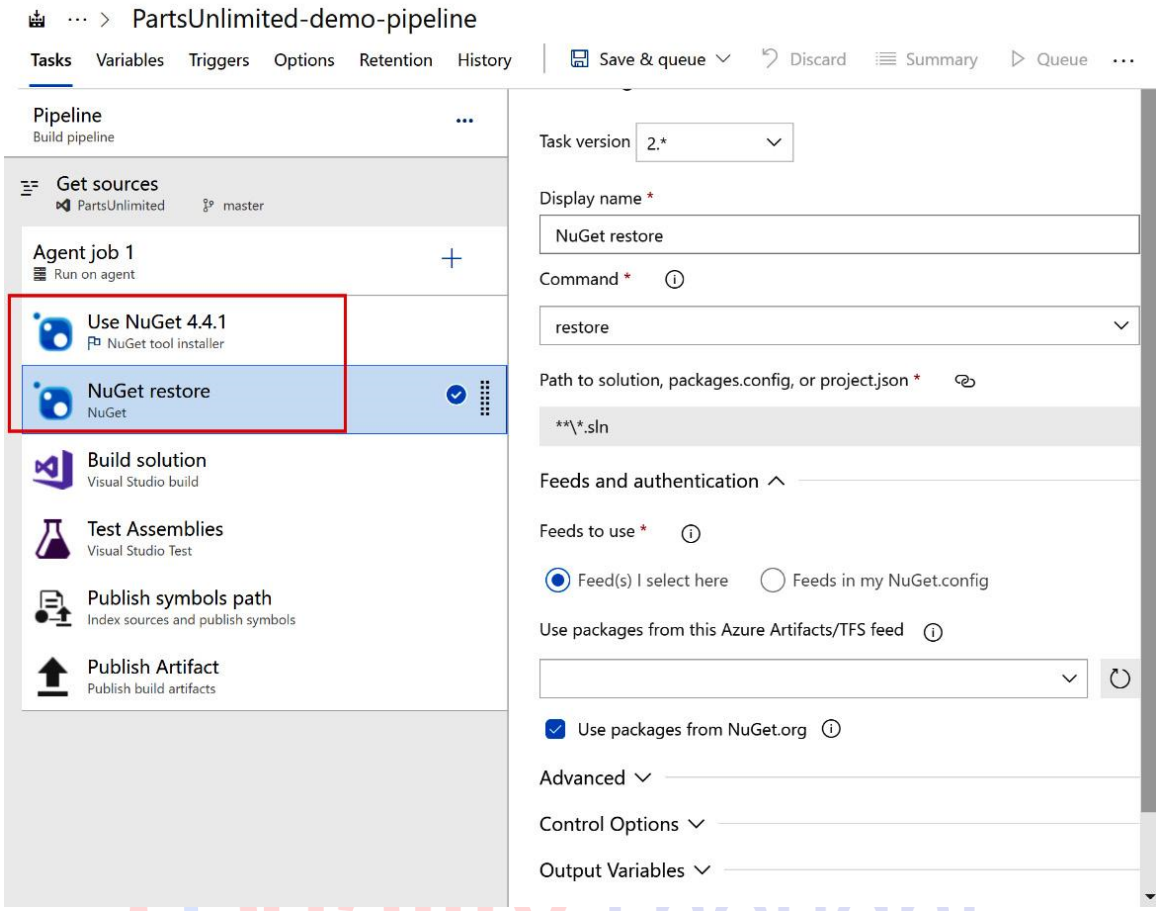


Let's check every section of the pipeline in detail.

The pipeline (here, this is called **PartsUnlimited-demo-pipeline**) runs on a Microsoft-hosted agent (Azure Pipelines agent pool) based on the **vs2017-win2016** template (Windows Server 2016 with Visual Studio 2017), as shown in the following screenshot:



The agent job starts by installing the NuGet package manager and restoring the required packages for building the project in the selected repository. For these actions, the pipeline definition contains the tasks that you can see in the following screenshot:



Then, there's a task for building the solution:

PartsUnlimited-demo-pipeline

Tasks | Variables | Triggers | Options | Retention | History | Save & queue | Discard | Summary | Queue ...

Pipeline
Build pipeline

Get sources
PartsUnlimited master

Agent job 1
Run on agent

Use NuGet 4.4.1
NuGet tool installer

NuGet restore
NuGet

Build solution
Visual Studio build

Test Assemblies
Visual Studio Test

Publish symbols path
Index sources and publish symbols

Publish Artifact
Publish build artifacts

Task version 1.*

Display name *
Build solution

Solution *
***.sln

Visual Studio Version
Latest

MSBuild Arguments
/p:DeployOnBuild=true /p:WebPublishMethod=Package ckage
/p:PackageAsSingleFile=true /p:SkipInvalidConfigurations=true rations=true
/p:PackageLocation="$(build.artifactstagingdirectory)\\" ry)\\"

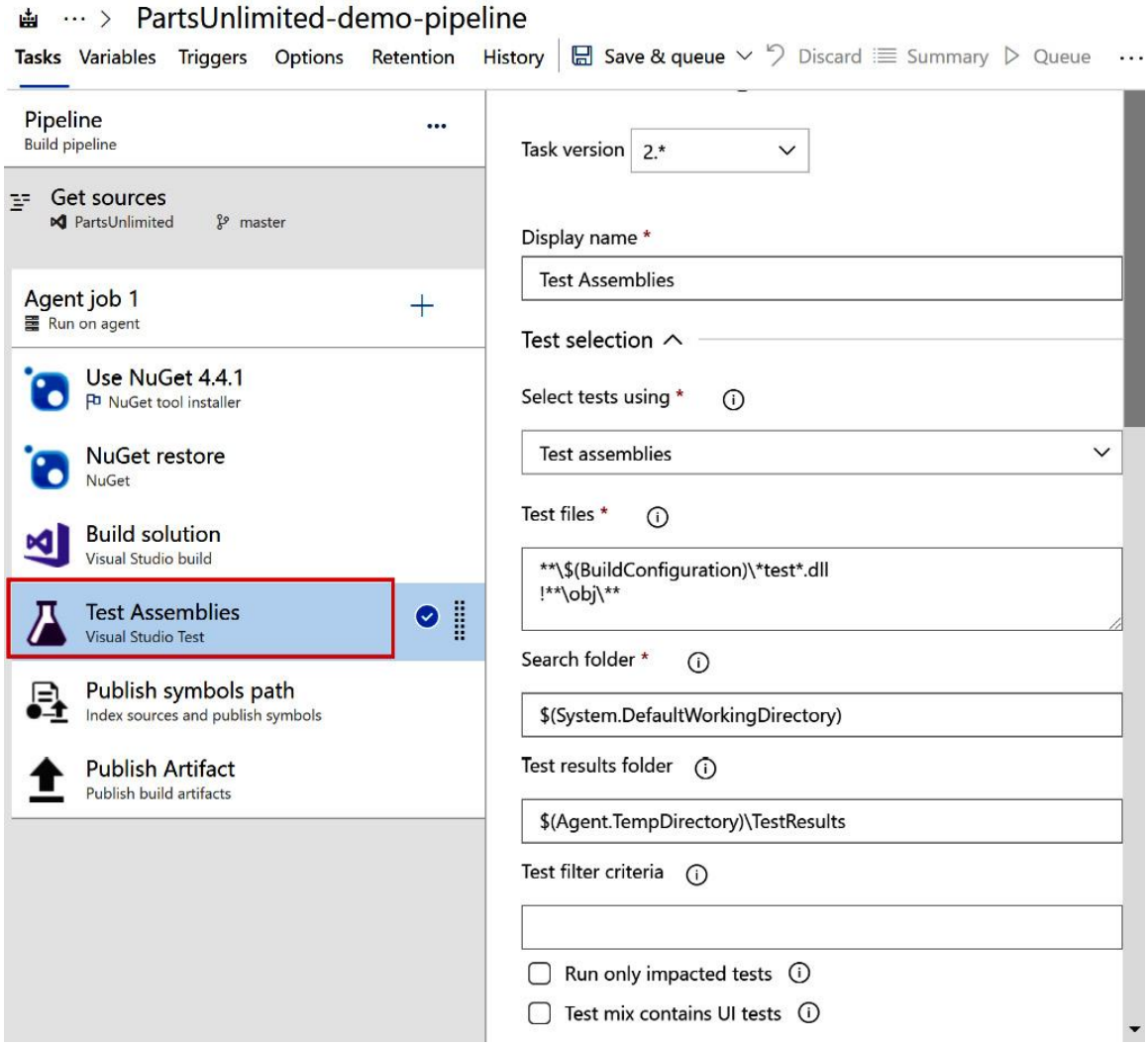
Platform
$(BuildPlatform)

Configuration
$(BuildConfiguration)

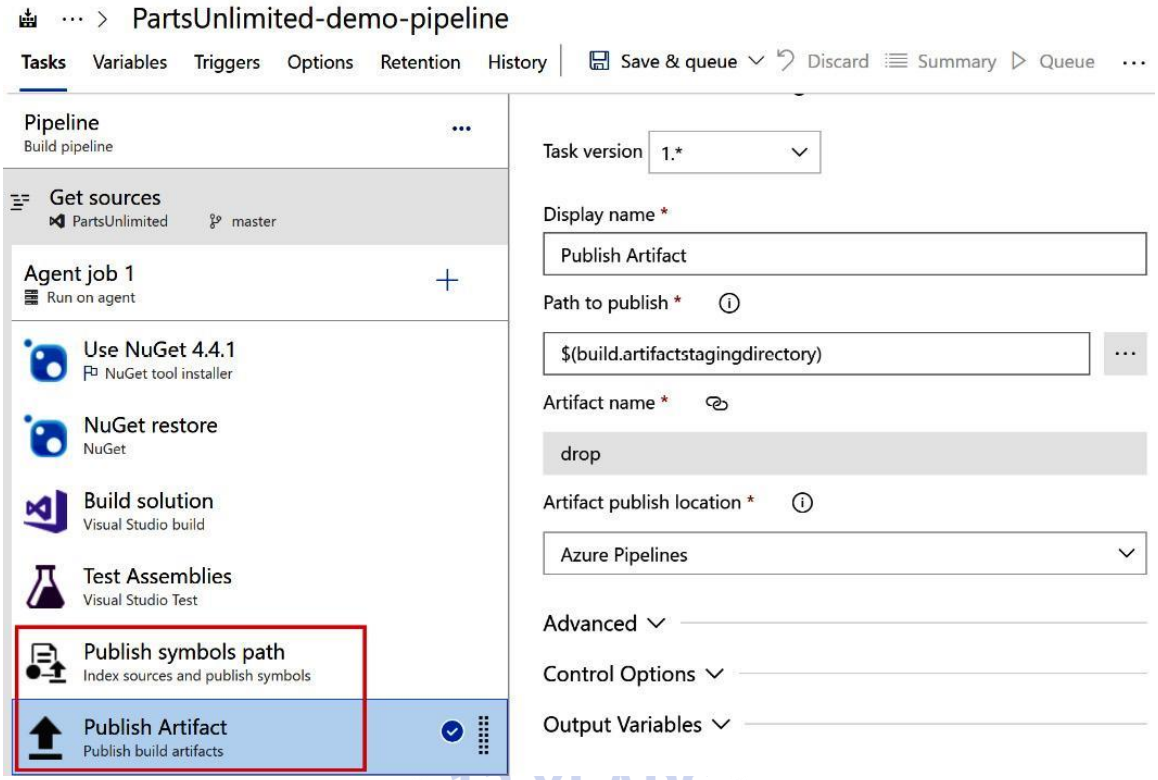
Clean

Advanced

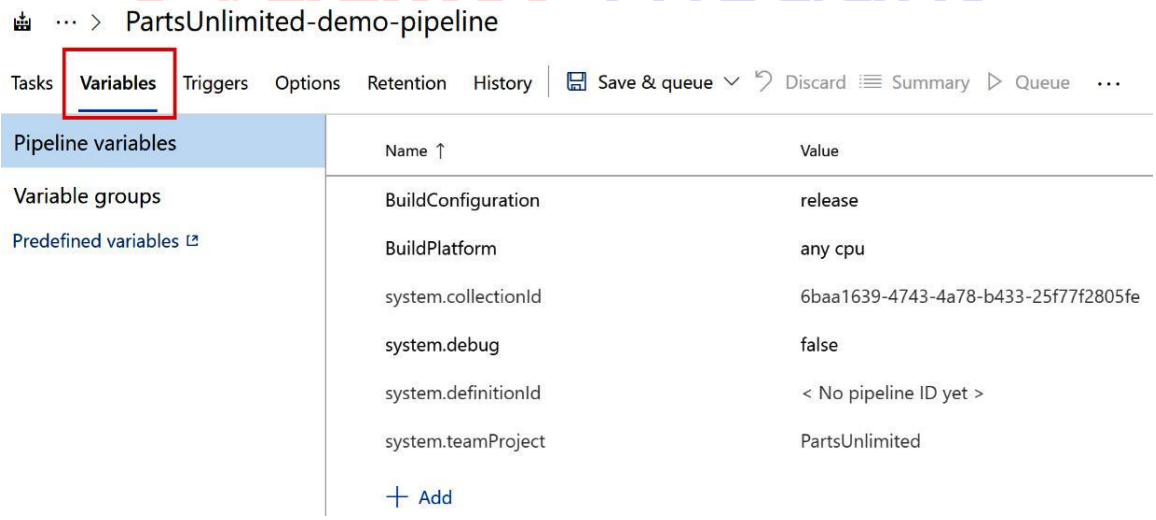
Control Options



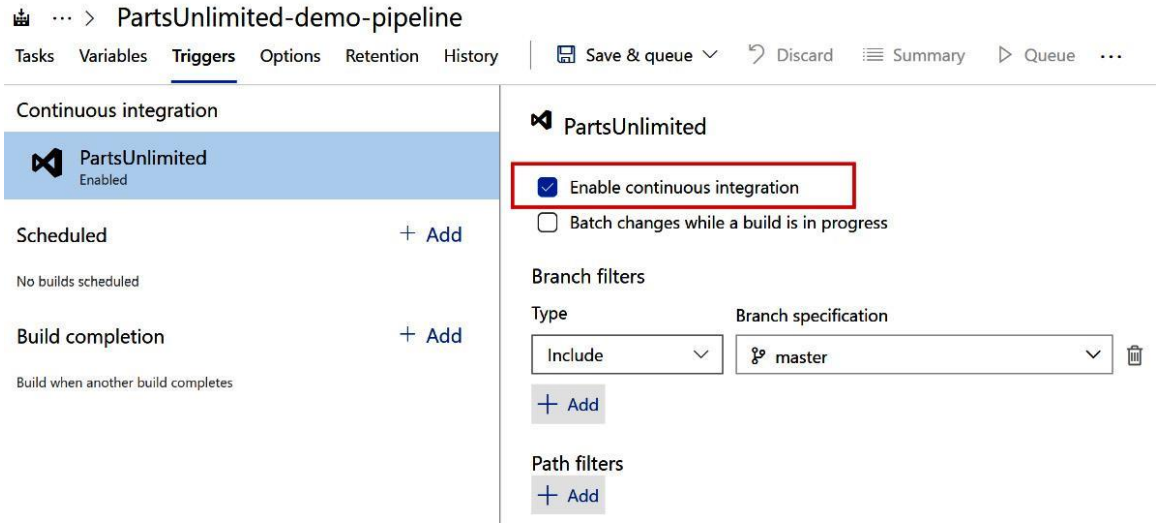
The last steps are for publishing the sources of the build process as artifacts (output of the build):



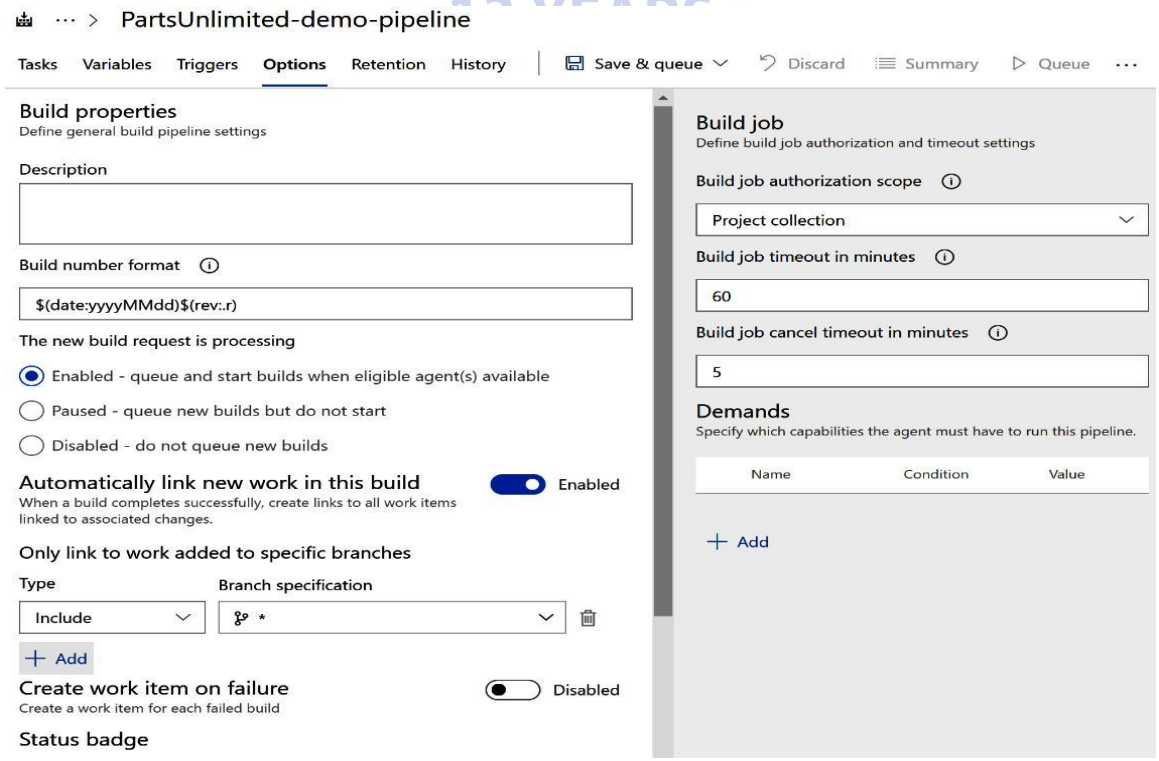
If you select the **Variables** tab, you will see that there are some parameters that are used during the build process. Here, you can create your own variables to use inside the pipeline if needed:



The next section is called **Triggers**. Here, you can define what triggers start your pipeline. By default, no triggers are published initially, but here, you can enable CI to automatically start your pipeline on every commit on the selected branch:



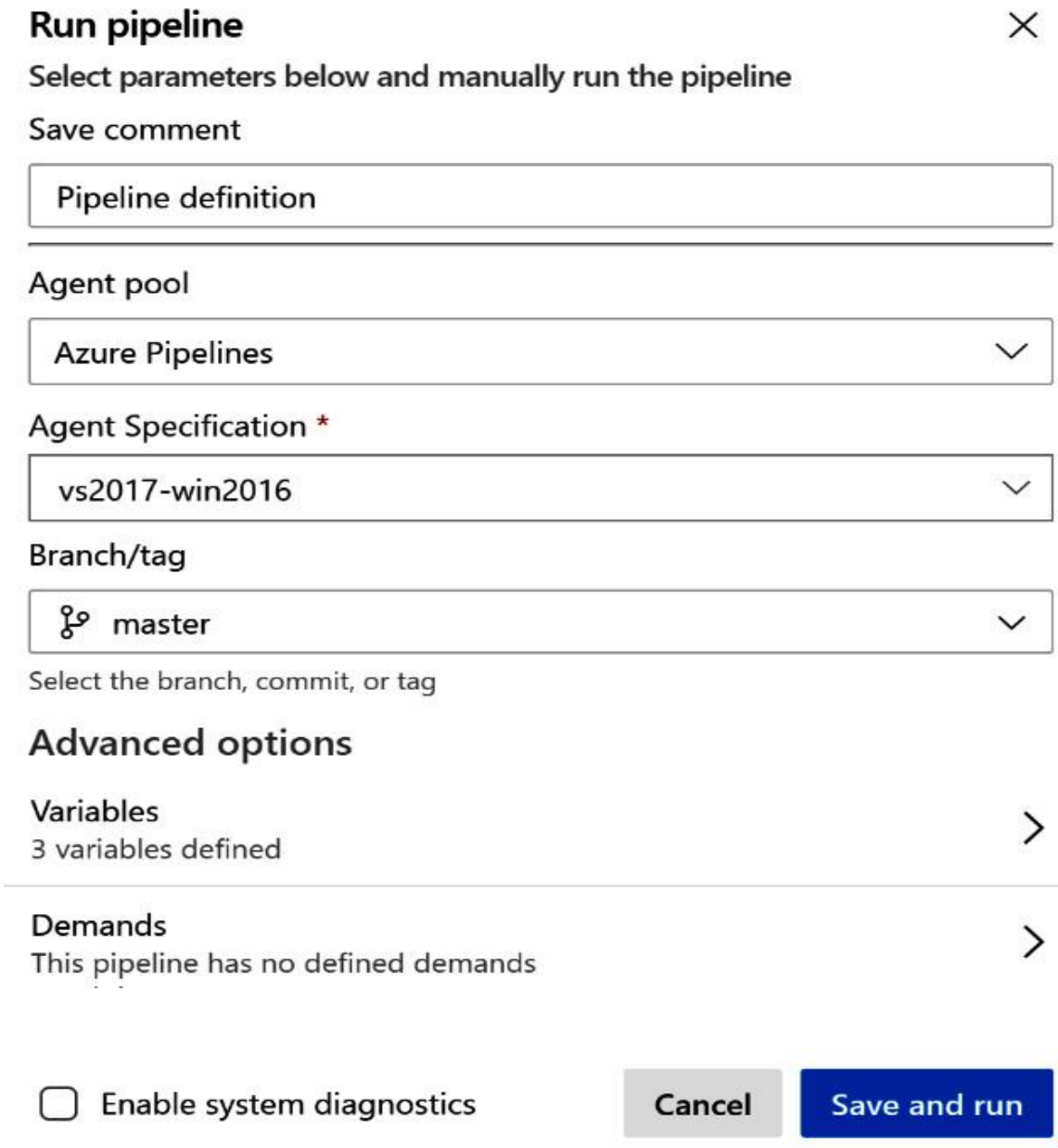
In the **Option** tab, you can set some options related to your build definition. For example, here, you can create links to all the work items so that they're linked to associated changes when a build completes successfully, create work items on failure of a build, set the status badge for your pipeline, specify timeouts, and so on:



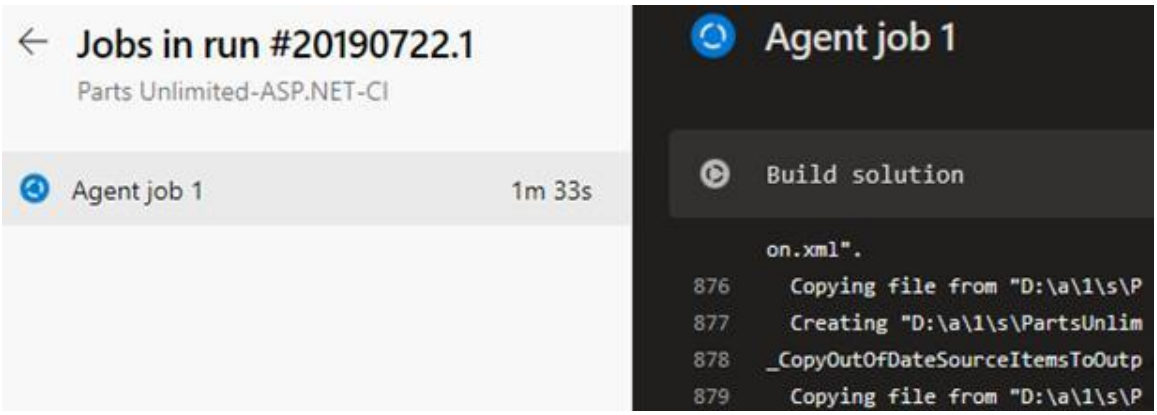
The **Retention** tab, on the other hand, is used for configuring the retention policy for this specific pipeline (how many days to keep artifacts for, the number of days to keep runs and pull requests for, and so on). Doing this will override the

general retention settings. We'll talk about them later in the *Retention of builds* section.

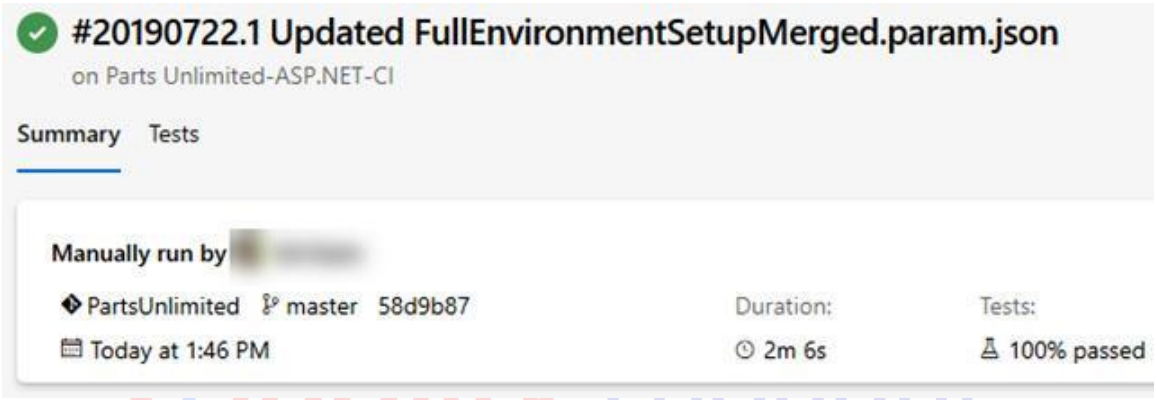
Once you've finished defining the pipeline, you can click **Save & queue** to save your definition. By clicking on **Save and run**, the pipeline will be placed in a queue and wait for an agent:



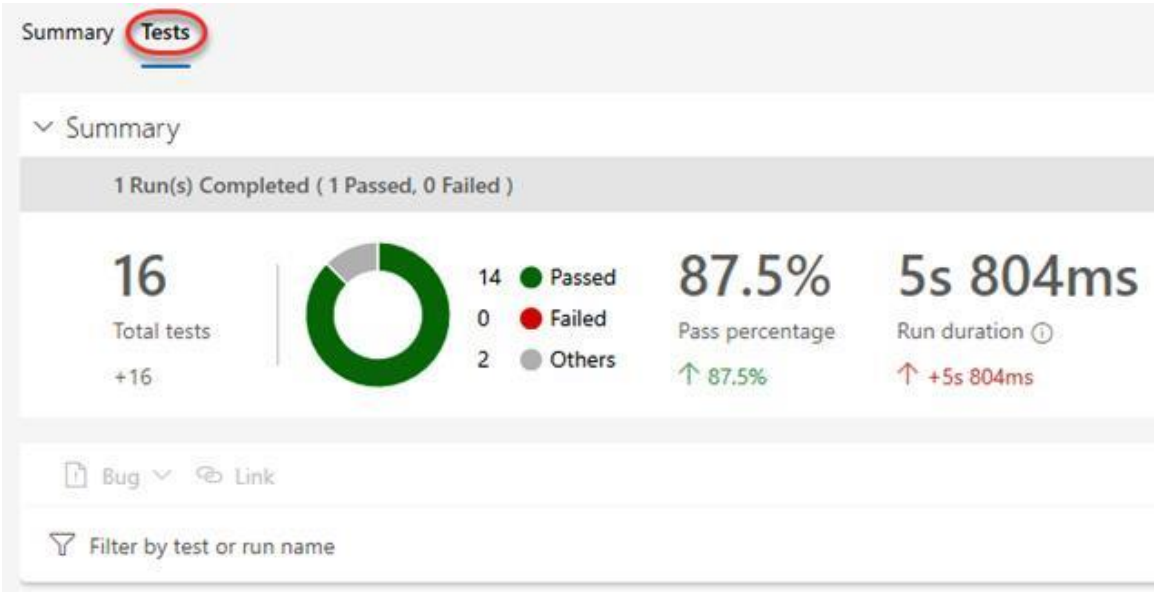
When the agent is found, the pipeline is executed and your code is built:



You can follow the execution of each step of the pipeline and see the related logs. If the pipeline ends successfully, you can view a summary of its execution:



You can also select the **Tests** tab to review the test execution status:

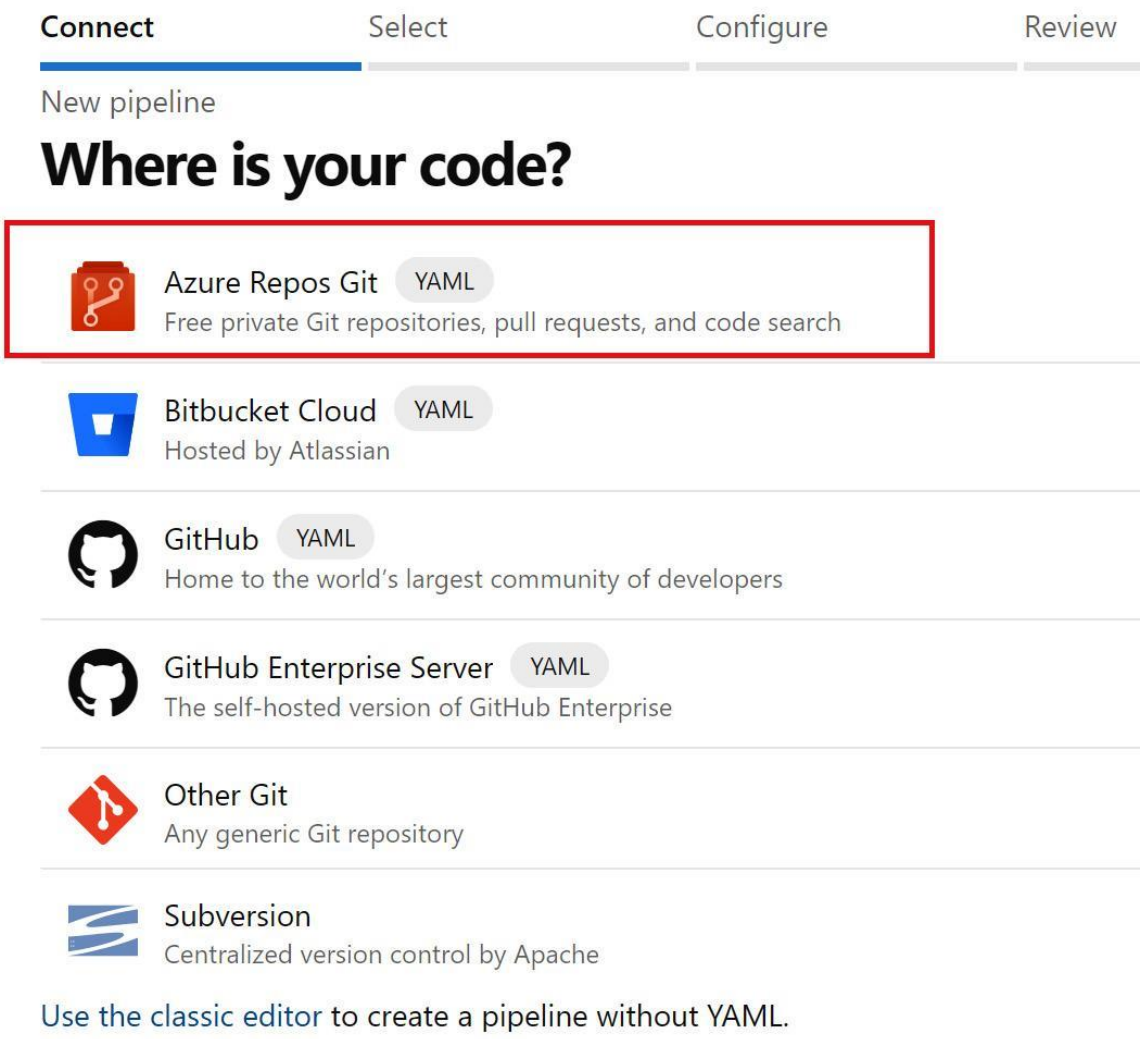


YAML PIPELINE DEFINITION

As previously explained, when you start creating a build pipeline with Azure DevOps, the wizard creates a YAML-based pipeline by default.

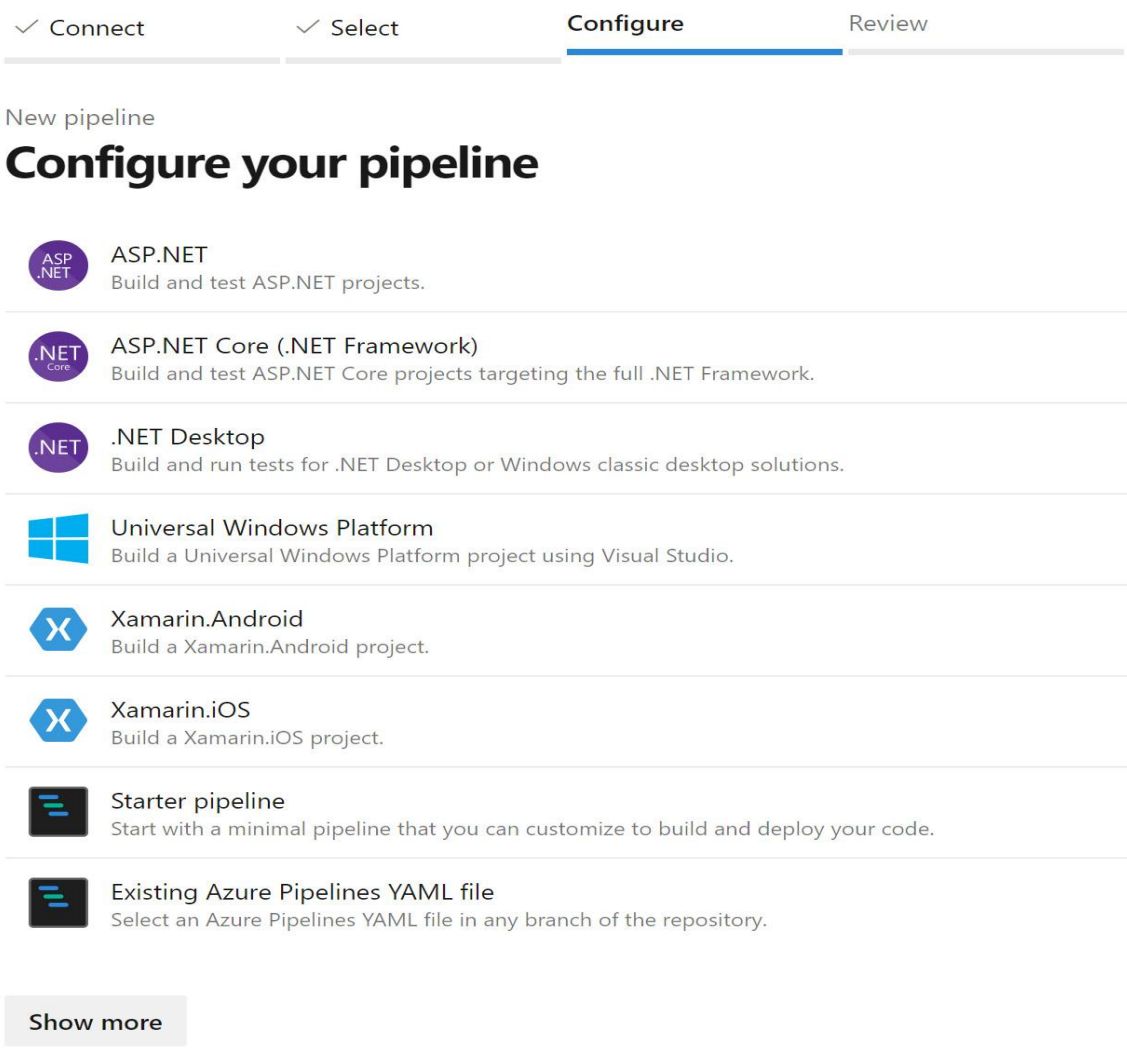
To start creating a YAML pipeline, go to the **Pipeline** section in Azure DevOps and click on **New Pipeline**.

Here, instead of selecting the classic editor (as we did in the previous section), just select the type of repository where your code is located (**Azure Repos Git**, **GitHub**, **BitBucket**, and so on):

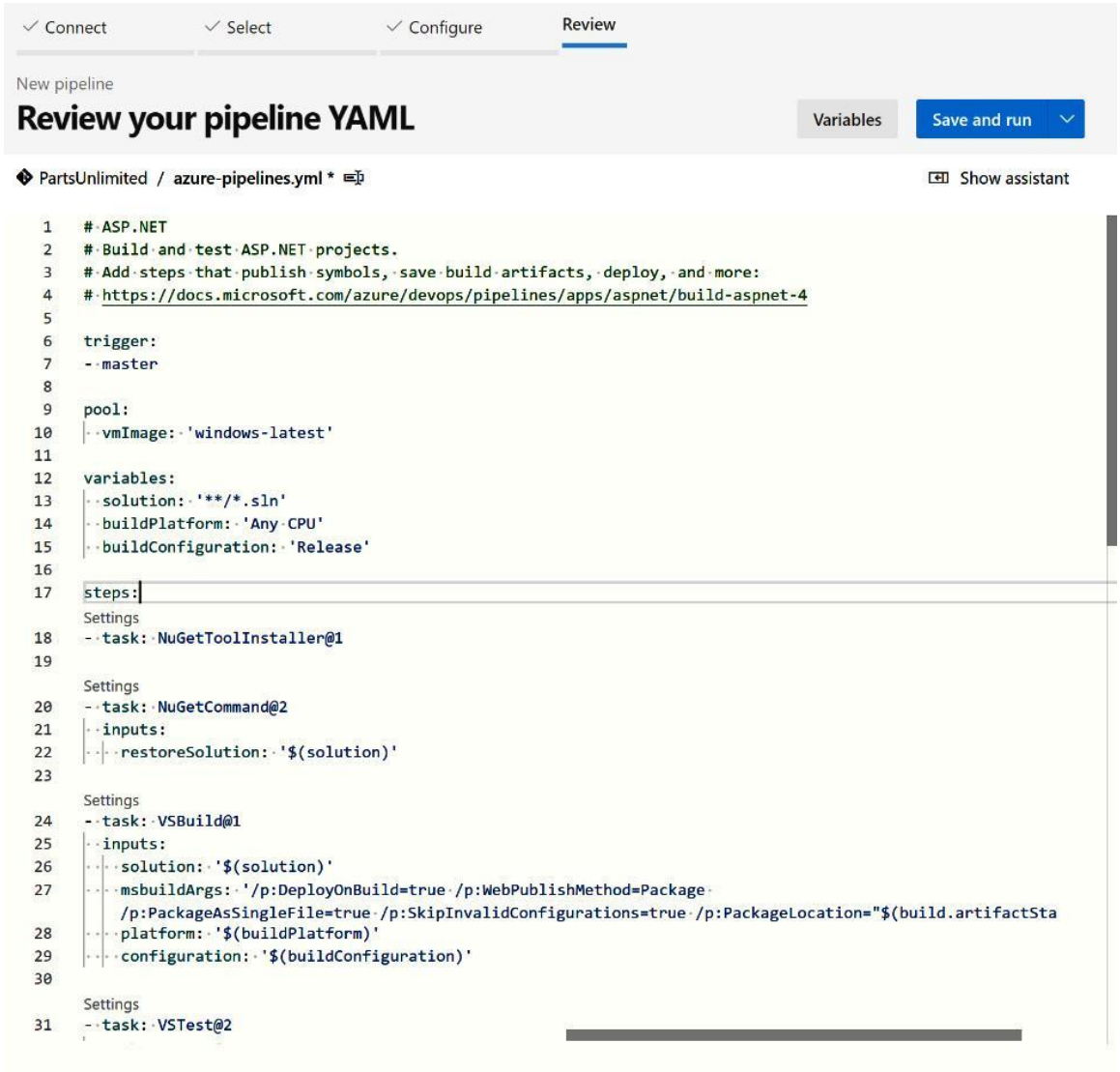


Then, select your repository from the available repositories list:

The system now analyzes your repository and proposes a set of available templates according to the code stored in the repository itself. You can start from a blank YAML template or you can select a template. Here, I'm selecting the ASP.NET template:



The system creates a YAML file (called **azure-pipelines.yml**), as shown in the following screenshot:



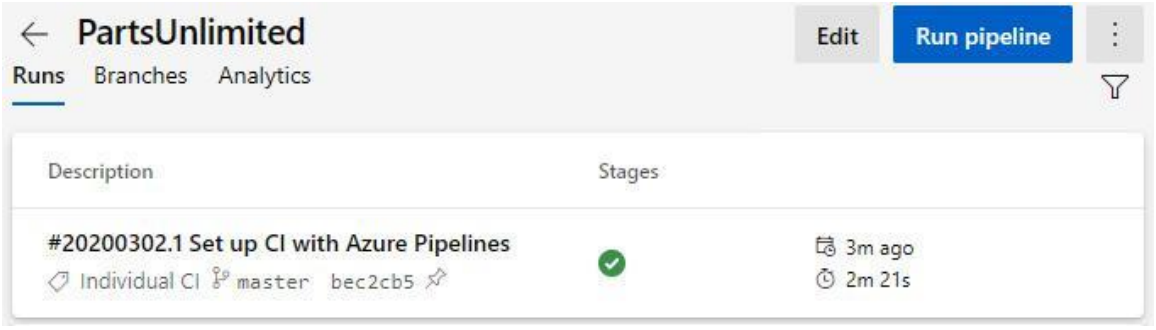
The generated YAML definition contains a set of tasks, just like in the previous example, but here, these tasks are in their YAML definition. The complete generated file is as follows:

```
---
trigger:
  - master
pool:
  vmImage: windows-latest
variables:
  solution: "**/*.sln"
```

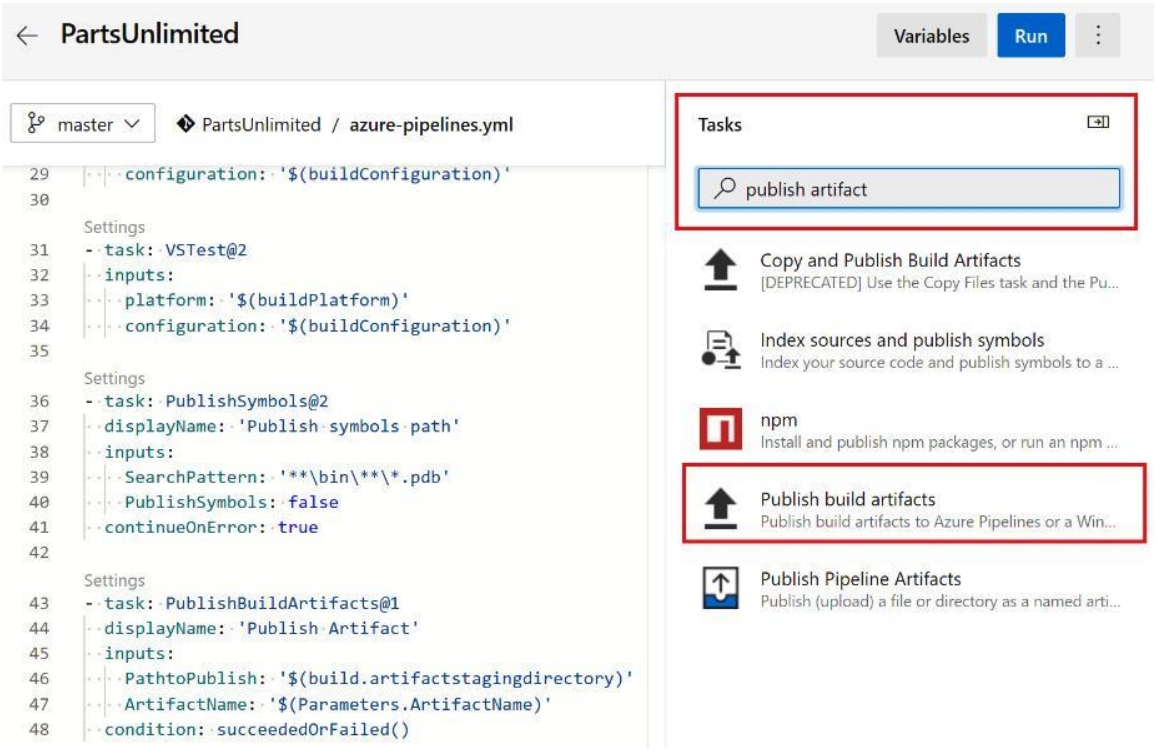
```
  buildPlatform: Any CPU
  buildConfiguration: Release
steps:
  - task: NuGetToolInstaller@1
  - task: NuGetCommand@2
    inputs:
      restoreSolution: $(solution)
  - task: VSBuild@1
    inputs:
      solution: $(solution)
      msbuildArgs: /p:DeployOnBuild=true /p:WebPublishMethod=Package
        /p:PackageAsSingleFile=true /p:SkipInvalidConfigurations=true
        /p:PackageLocation="$(build.artifactStagingDirectory)"
      platform: $(buildPlatform)
      configuration: $(buildConfiguration)
  - task: VSTest@2
    inputs:
      platform: $(buildPlatform)
      configuration: $(buildConfiguration)
  - task: PublishSymbols@2
    displayName: Publish symbols path
    inputs:
      SearchPattern: "**\\bin\\**\\*.pdb"
      PublishSymbols: false
    continueOnError: true
  - task: PublishBuildArtifacts@1
    displayName: Publish Artifact
    inputs:
      PathtoPublish: $(build.artifactstagingdirectory)
      ArtifactName: $(Parameters.ArtifactName)
    condition: succeededOrFailed()
```

As you can see, the YAML file contains the trigger that starts the pipeline (here, this is a commit on the master branch), the agent pool to use, the pipeline variables, and the sequence of each task to execute (with its specific parameters).

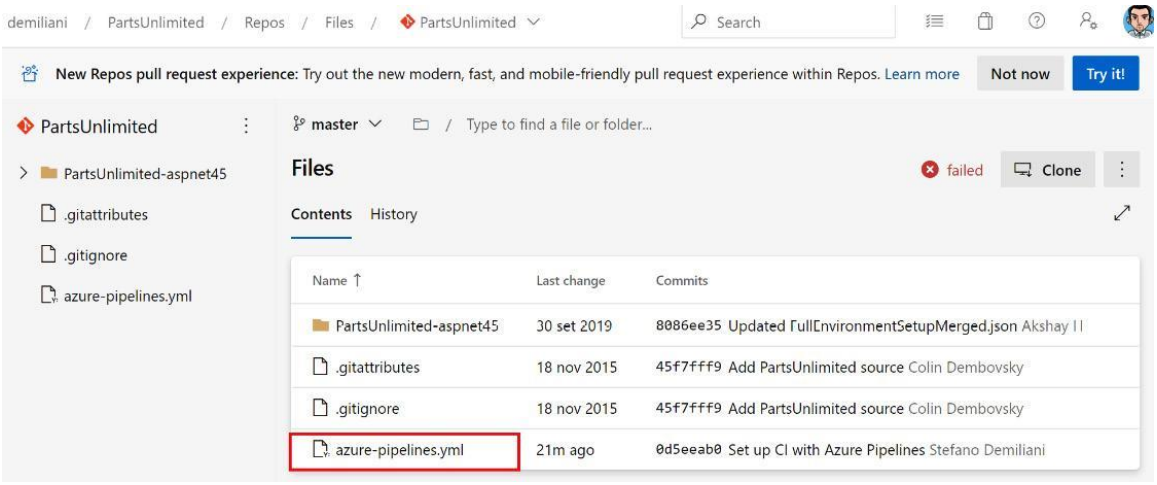
Click on Save and run as shown in the previous screenshot to queue the pipeline and have it executed. The following screenshot shows the executed YAML pipeline.



To add new tasks, it's useful to use the assistant tool on the right of the editor frame. It allows you to have a **Tasks** list where you can search for a task, fill in the necessary parameters, and then have the final YAML definition:



When you choose to create a pipeline with YAML, Azure DevOps creates a file that's stored in the same repository that your code is stored in:



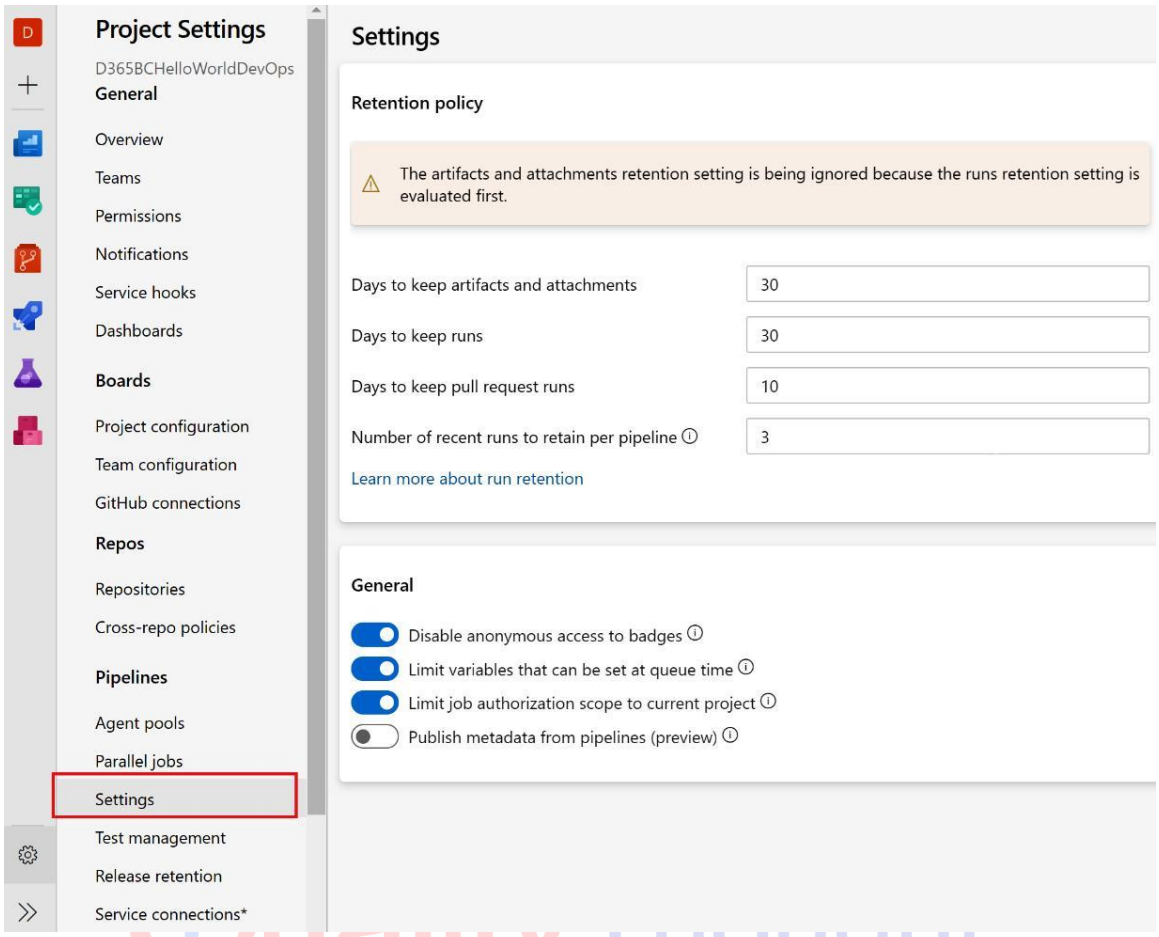
This file is under source control and versioned on every modification.

For a complete reference to the YAML schema for a pipeline, I suggest following this link: https://learn.microsoft.com/en-us/azure/devops/pipelines/yaml-schema/?view=azure-pipelines&tabs=schema%2Cparameter-schema.&viewFallbackFrom=azure-devops

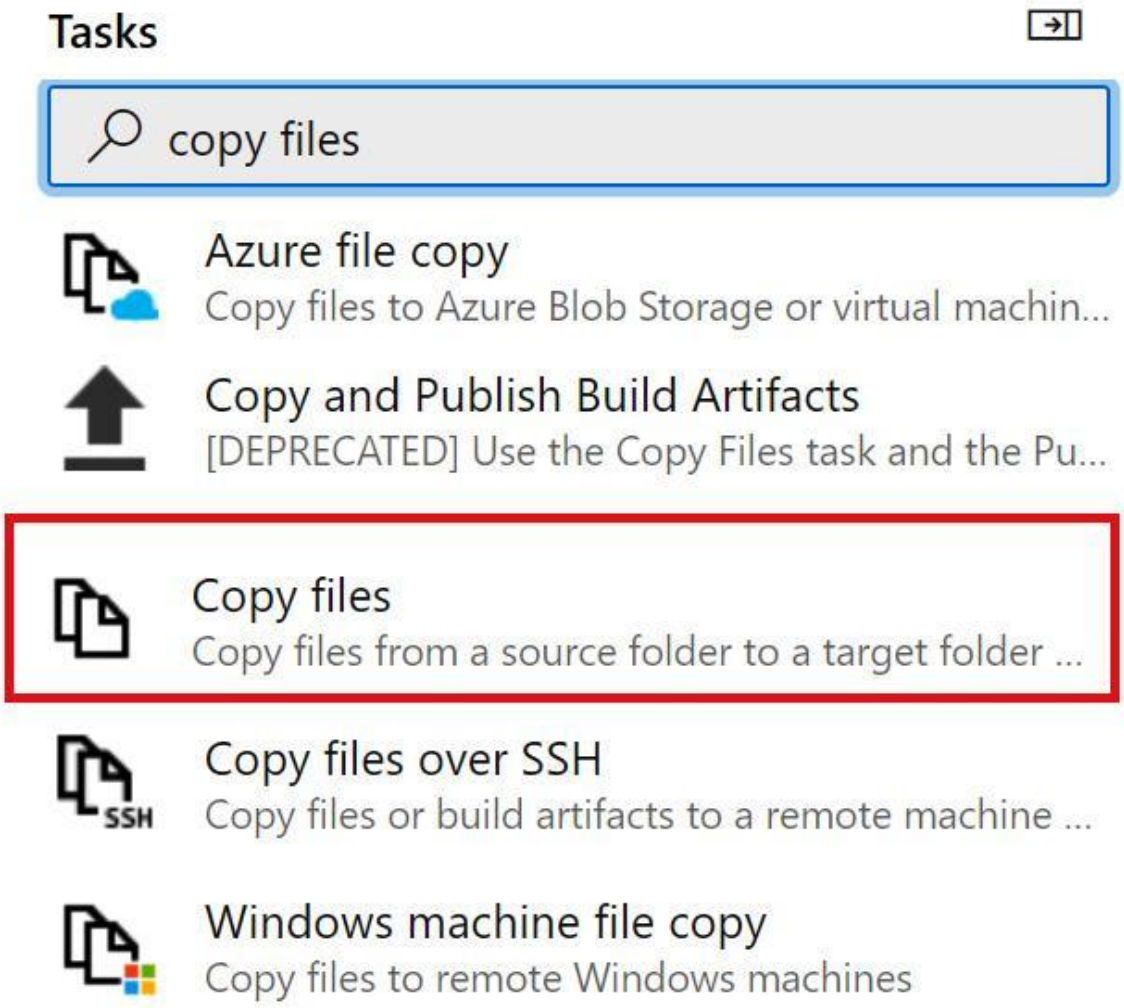
Retention of builds

When you run a pipeline, Azure DevOps logs each step's execution and stores the final artifacts and tests for each run.

Azure DevOps has a default retention policy for pipeline execution of 30 days. You can change these default values by going to **Project settings** | **Pipelines** | **Settings**:



You can also use the **Copy files** task to store your build and artifacts data in external storage so that you can preserve them for longer than what's specified in the retention policy:



The YAML definition for this task is as follows:

```
- task: CopyFiles@2
  displayName: 'Copy files to shared network'
  inputs:
    SourceFolder: '$(Build.SourcesDirectory)'
    Contents: '**'
    TargetFolder: '\\networkserver\storage\$(Build.BuildNumber)'
```

Remember that any data saved as artifacts with the **Publish Build Artifacts** task is periodically deleted.

More information about the **Copy files** task can be found here: https://learn.microsoft.com/en-us/azure/devops/pipelines/tasks/reference/copy-files-v2?view=azure-pipelines&tabs=yaml&viewFallbackFrom=azure-devops

Multi-stage pipeline

As we explained previously, you can organize the jobs in your pipeline into . **stages**. **Stages** are logical boundaries inside a pipeline flow (units of works that you can assign to an agent) that allow you to isolate the work, pause the pipeline, and execute checks or other actions. By default, every pipeline is composed of one stage, but you can create more than one and arrange those stages into a dependency graph.

The basic YAML definition of a multi-stage pipeline is as follows:

```
stages:
- stage: Build
  jobs:
  - job: BuildJob
    steps:
    - script: echo Build!
- stage: Test
  jobs:
  - job: TestOne
    steps:
    - script: echo Test 1
  - job: TestTwo
    steps:
    - script: echo Test 2
- stage: Deploy
  jobs:
  - job: Deploy
    steps:
    - script: echo Deployment
```

As an example of how to create a multi-stage pipeline with YAML, let's look at a pipeline that builds code in your repository (with .NET Core SDK) and publishes the artifacts as NuGet packages. The pipeline definition is as follows. The pipeline uses the **stages** keyword to identify that this is a multi-stage pipeline.

In the first stage definition (**Build**), we have the tasks for building the code:

```
trigger:
- master
stages:
- stage: 'Build'
  variables:
    buildConfiguration: 'Release'
  jobs:
```

```
- job:
  pool:
    vmImage: 'ubuntu-latest'
  workspace:
    clean: all
  steps:
  - task: UseDotNet@2
    displayName: 'Use .NET Core SDK'
    inputs:
      packageType: sdk
      version: 2.2.x
      installationPath: $(Agent.ToolsDirectory)/dotnet
  - task: DotNetCoreCLI@2
    displayName: "NuGet Restore"
    inputs:
      command: restore
      projects: '**/*.csproj'
  - task: DotNetCoreCLI@2
    displayName: "Build Solution"
    inputs:
      command: build
      projects: '**/*.csproj'
      arguments: '--configuration (buildConfiguration)'
```

Here, we installed the .NET Core SDK by using the **UseDotnet** standard task template that's available in Azure DevOps (more information can be found here: https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/tool/dotnet-core-tool-installer?view=azure-devops). After that, we restored the required NuGet packages and built the solution.

Now, we have the task of creating the release version of the NuGet package. This package is saved in the packages/release folder of the artifact staging directory. Here, we will use **nobuild = true** because in this task, we do not have to rebuild the solution again (no more compilation):

```
- task: DotNetCoreCLI@2
  dispalyName: 'Create NuGet Package - Release Version'
  inputs:
    command: pack
    packDirectory: '$(Build.ArtifactStagingDirectory)/packages/releases'
    arguments: '--configuration $(buildConfiguration)'
    nobuild: true
```

As the next step, we have the task of creating the prerelease version of the NuGet package. In this task, we're using the **buildProperties** option to add the build number to the package version (for example, if the package version is 2.0.0.0 and the build number is 20200521.1, the package version will be 2.0.0.0.20200521.1). Here, a build of the package is mandatory (for retrieving the build ID):

```
- task: DotNetCoreCLI@2
  displayName: 'Create NuGet Package - Prerelease Version'
  inputs:
    command: pack
    buildProperties: 'VersionSuffix="$(Build.BuildNumber)"'
    packDirectory: '$(Build.ArtifactStagingDirectory)/packages/prereleases'
    arguments: '--configuration $(buildConfiguration)'
```

The next task publishes the package as an artifact:

```
- publish: '$(Build.ArtifactStagingDirectory)/packages'
  artifact: 'packages'
```

Next, we need to define the second stage, called PublishPrereleaseNuGetPackage. Here, we skip the checkout of the repository and the download step downloads the packages artifact that we published in the previous build stage. Then, the NuGetCommand task publishes the prerelease package to an internal feed in Azure DevOps called Test:

```
- stage: 'PublishPrereleaseNuGetPackage'
  displayName: 'Publish Prerelease NuGet Package'
  dependsOn: 'Build'
  condition: succeeded()
  jobs:
  - job:
    pool:
      vmImage: 'ubuntu-latest'
    steps:
    - checkout: none
    - download: current
      artifact: 'packages'
    - task: NuGetCommand@2
      displayName: 'Push NuGet Package'
      inputs:
        command: 'push'
        packagesToPush: '$(Pipeline.Workspace)/packages/prereleases/*.nupkg'
```

```
      nuGetFeedType: 'internal'
      publishVstsFeed: 'Test'
```

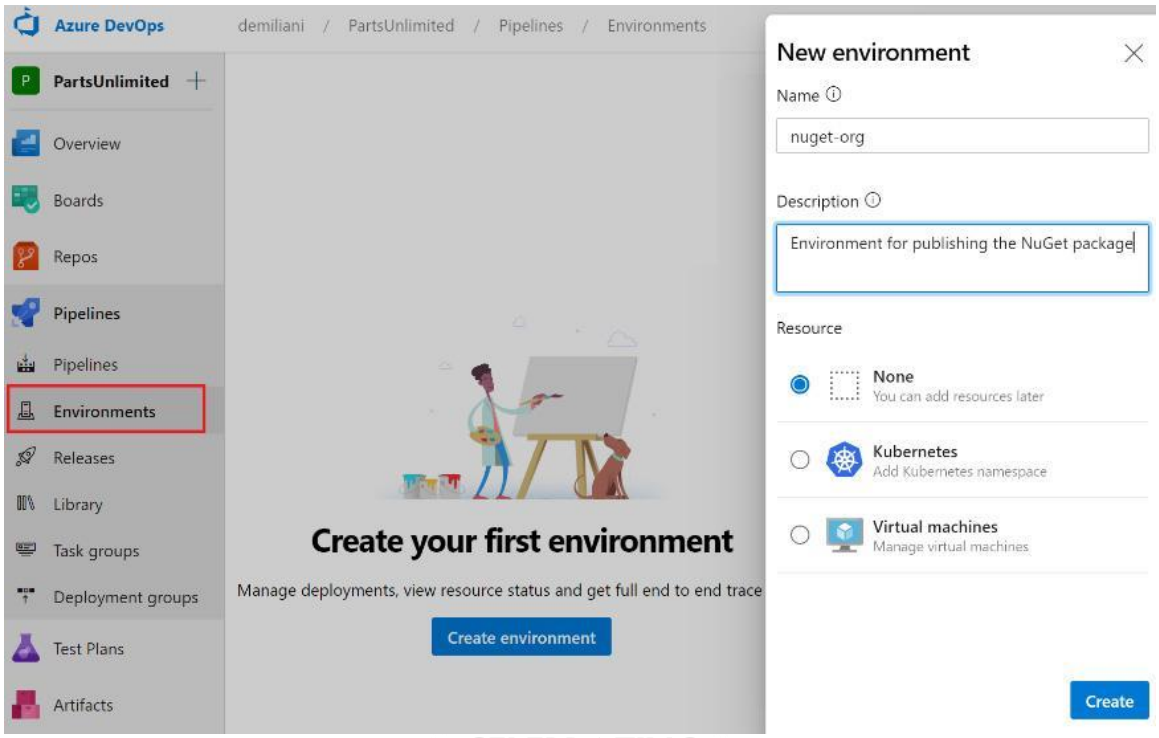
Now, we have to define the third stage, called **PublishReleaseNuGetPackage**, which creates the release version of our package for NuGet:

```
- stage: 'PublishReleaseNuGetPackage'
  displayName: 'Publish Release NuGet Package'
  dependsOn: 'PublishPrereleaseNuGetPackage'
  condition: succeeded()
  jobs:
  - deployment:
    pool:
      vmImage: 'ubuntu-latest'
    environment: 'nuget-org'
    strategy:
     runOnce:
       deploy:
        steps:
        - task: NuGetCommand@2
          displayName: 'Push NuGet Package'
          inputs:
            command: 'push'
            packagesToPush: '$(Pipeline.Workspace)/packages/releases/*.nupkg'
            nuGetFeedType: 'external'
            publishFeedCredentials: 'NuGet'
```

This stage uses a deployment job to publish the package to the configured environment (here, this is called **nuget-org**). An environment is a collection of resources inside a pipeline.

In the **NuGetCommand** task, we specify the package to push and that the feed where we're pushing the package to is external (**nuGetFeedType**). The feed is retrieved by using the **publishFeedCredentials** property, set to the name of the service connection we created.

For this stage, we have created a new environment:



Once the environment has been created, in order to publish it to NuGet, you need to create a new service connection by going to **Project Settings** | **Service Connections** | **Create Service Connection**, selecting **NuGet** from the list of available service connection types, and then configuring the connections according to your NuGet account:

New NuGet service connection

Authentication method

- (●) ApiKey
- ( ) External Azure DevOps Server
- ( ) Basic Authentication

Feed URL

https://api.nuget.org/v3/index.json

URL for the feed. This will generally end with 'index.json'. For nuget.org, use https://api.nuget.org/v3/index.json

Authentication

ApiKey

ApiKey (only for push).

Details

Service connection name

NuGet

Description (optional)

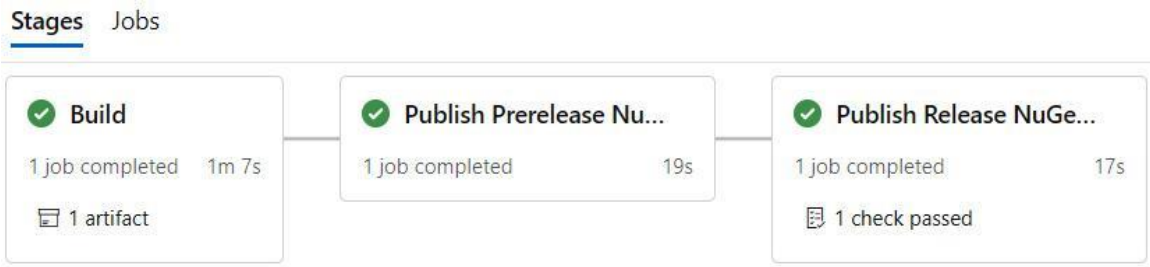
Security

- [x] Grant access permission to all pipelines

Learn more
Troubleshoot

Back Save

With that, we have created a multi-stage build pipeline. When the pipeline is executed and all the stages terminate successfully, you will see a results diagram that looks as follows:



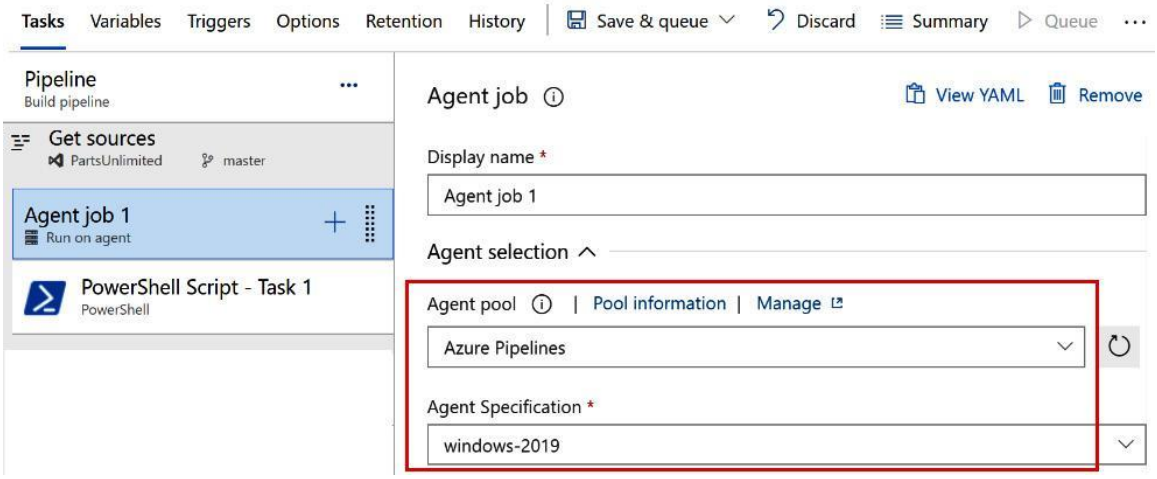
Now that we have understood what a multi-stage pipeline is, we'll create some pipelines.

EXECUTING JOBS IN PARALLEL IN AN AZURE PIPELINE

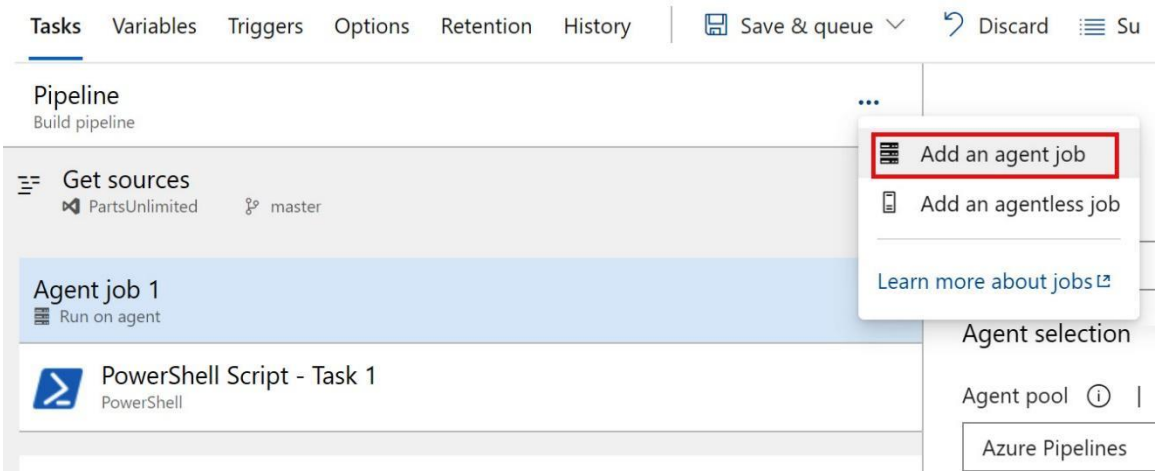
Within an Azure Pipeline, you can also execute jobs in parallel. Each job can be independent of other jobs and can also be executed on a different agent. This will allow you to speed up your build time and improve your pipeline's performance.

As an example of how to handle parallel jobs in a pipeline, consider a simple pipeline where you have to execute three PowerShell scripts called **Task 1**, **Task 2**, and **Final Task**. **Task 1** and **Task 2** can be executed in parallel, while **Final Task** can only be executed when the previous two tasks are completed.

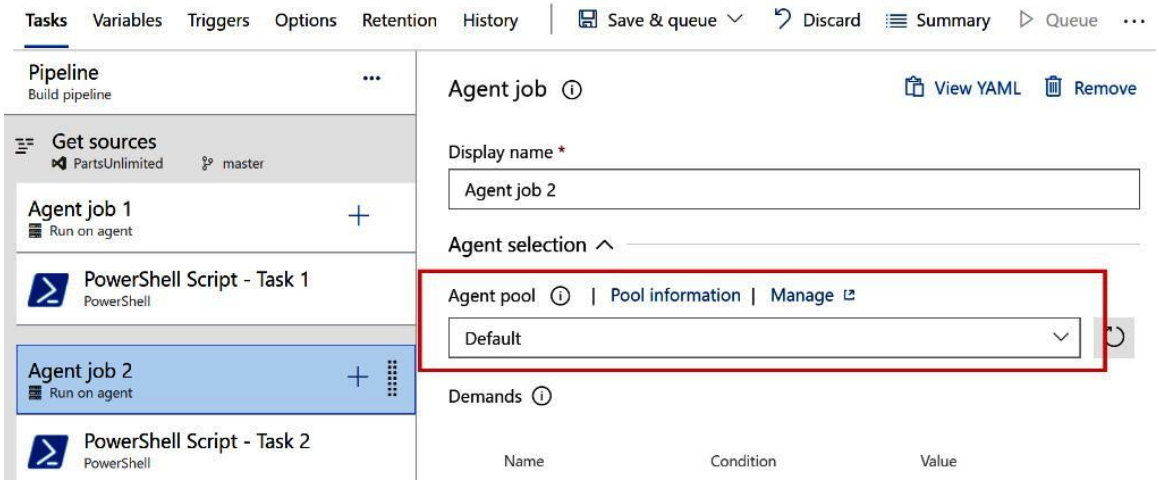
When you start creating a new pipeline (I'm using the classic editor here for simplicity), Azure DevOps creates an agent job (here, this is called **Agent Job 1**). You can add your task to this agent. By selecting the agent job, you can specify the agent pool where this task runs. Here, I want this task to be executed on a Microsoft-hosted agent pool:



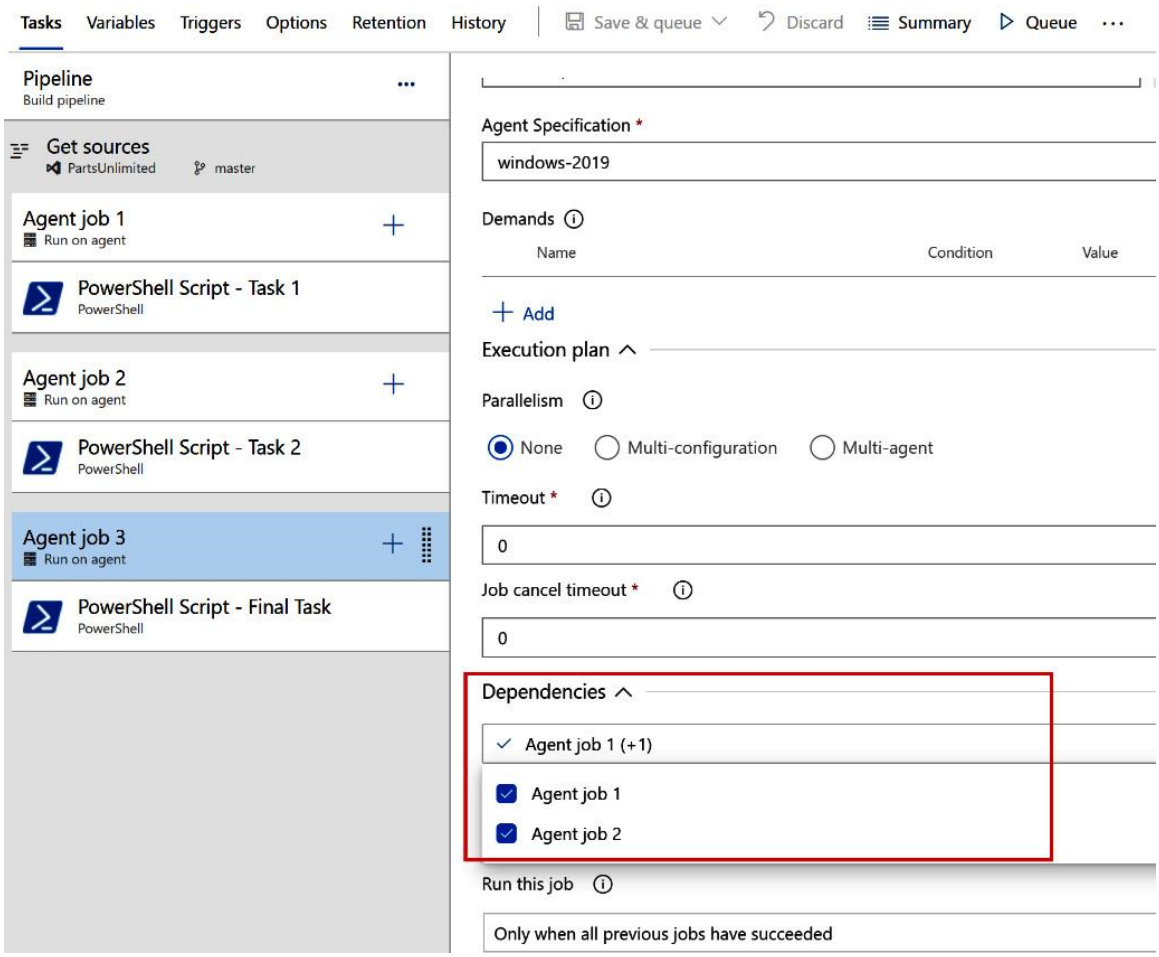
Then, to add a new agent pool to your pipeline (for executing the other task independently), click the three dots beside the pipeline and select **Add an agent job**:



Now, we'll add a second agent job (here, this is called **Agent job 2**) that runs on a self-hosted agent. This job will execute the **Task 2** PowerShell script:



Finally, we'll add a new agent job (here, this is called **Agent Job 3)** to execute the **Final Task** that will run on a Microsoft-hosted agent. However, this job has dependencies from **Agent Job 1** and **Agent Job 2**:



In this way, the first two tasks start in parallel and the final job will wait until the two previous tasks are executed.

AGENTS ON AZURE CONTAINER INSTANCES

If standard Microsoft-hosted agents don't fit your needs (requirements, performance, and so on), there's also the possibility to create a self-hosted agent for Azure DevOps that runs inside a Docker container on the **Azure Container Instances** (**ACI**) service.

You can create a build agent running on Azure Container Instances by using a custom image or by reusing one of Microsoft's available images.pipe

To create a build agent running on ACI, you need to create a **personal access token** for your Azure DevOps organization. To do so, from your Azure DevOps organization home page, open the user settings (top-right corner) and select **Personal access tokens**.

When you have the personal access token for your agent, you can create an agent on ACI by executing the following command from the Azure CLI (after connecting to your Azure subscription):

```
az container create -g RESOURCE_GROUP_NAME -n CONTAINER_NAME --image mcr.microsoft.com/azure-pipelines/vsts-agent --cpu 1 --memory 7 --environment-variables VSTS_ACCOUNT=AZURE_DEVOPS_ACCOUNT_NAME VSTS_TOKEN=PERSONAL_ACCESS_TOKEN VSTS_AGENT=AGENT_NAME VSTS_POOL=Default
```

Here, we have the following:

- **RESOURCE_GROUP_NAME** is the name of your resource group in Azure where this resource will be created.
- **CONTAINER_NAME** is the name of the ACI container.
- **AZURE_DEVOPS_ACCOUNT_NAME** is the name of your Azure DevOps account.
- **PERSONAL_ACCESS_TOKEN** is the personal access token you created previously.
- **AGENT_NAME** is the name of the build agent that you want to create. This will be displayed on Azure DevOps.

In this command, there are also other two important parameters:

- **--image** is used to select the name of the Azure Pipelines image for creating your agent, as described here: https://hub.docker.com/_/microsoft-azure-pipelines-vsts-agent.
- **VSTS_POOL** is used to select the agent pool for your build agent.

Remember that you can start and stop an ACI instance by using the **az container stop** and **az container start** commands. This can help you save money.

USING CONTAINER JOBS IN AZURE PIPELINES

If you're using Windows or Linux agents, you can also run a job inside a container (in an isolated way from the host). To run a job inside a container, you need to have Docker installed on the agent and your pipeline must have permission to access the Docker daemon.

As an example, this is a YAML definition for using a container job in a Windows pipeline:

```
pool:
  vmImage: 'windows-2019'
container: mcr.microsoft.com/windows/servercore:ltsc2019
steps:
- script: date /t
  displayName: Gets the current date
- script: dir
  workingDirectory: $(Agent.BuildiDirectory)
  displayName: list the content of a folder
```

As we mentioned previously, to run a job inside a Windows container, you need to use the **windows-2019** image pool. It's required that the kernel version of the host and the container match, so here, we're using the **ltsc2019** tag to retrieve the container's image.

For a Linux-based pipeline, you need to use the **ubuntu-20.04** image:

```
pool:
  vmImage: 'ubuntu-20.04'
container: ubuntu:20.04
steps:
- script: printenv
```

Running Quality Tests in a Build Pipeline

INTRODUCTION TO UNIT TESTING

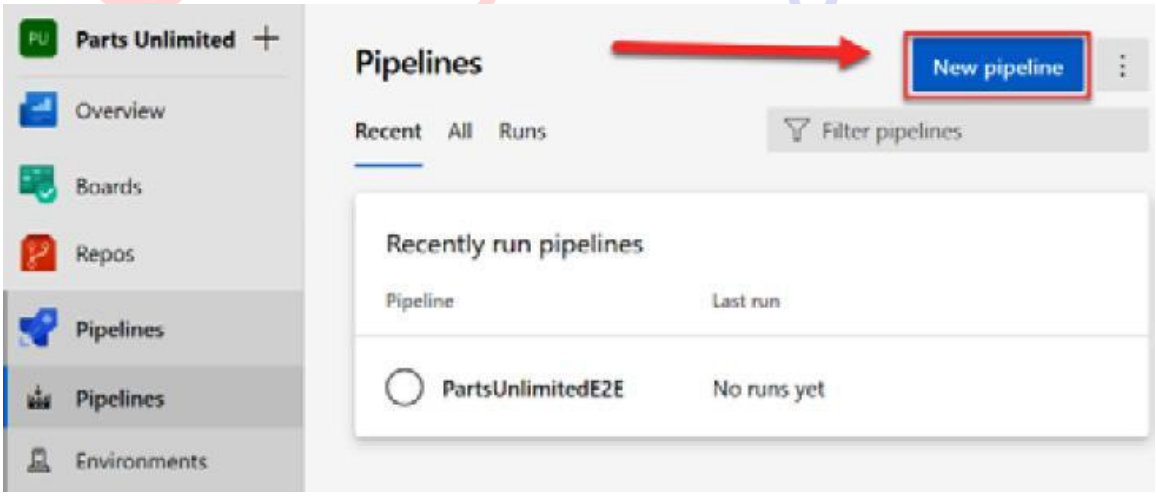
With unit testing, you break up code into small pieces, called units, that can be tested independently from each other. These units can consist of classes, methods, or single lines of code. The smaller the better works best here. This will give you a better view of how your code is performing and allows tests to be run fast.

In most cases, unit tests are written by the developer that writes the code. There are two different ways of writing unit tests: before you write the actual production code, or after. Most programmers write it afterwards, which is the traditional way of doing things, but if you are using **test-driven development** (**TDD**), you will typically write them beforehand. Unit testing will also make code documentation easier. It encourages better coding practices and you can leave code pieces to describe the code's functionality behind. Here, you will focus more on updating a system of checks.

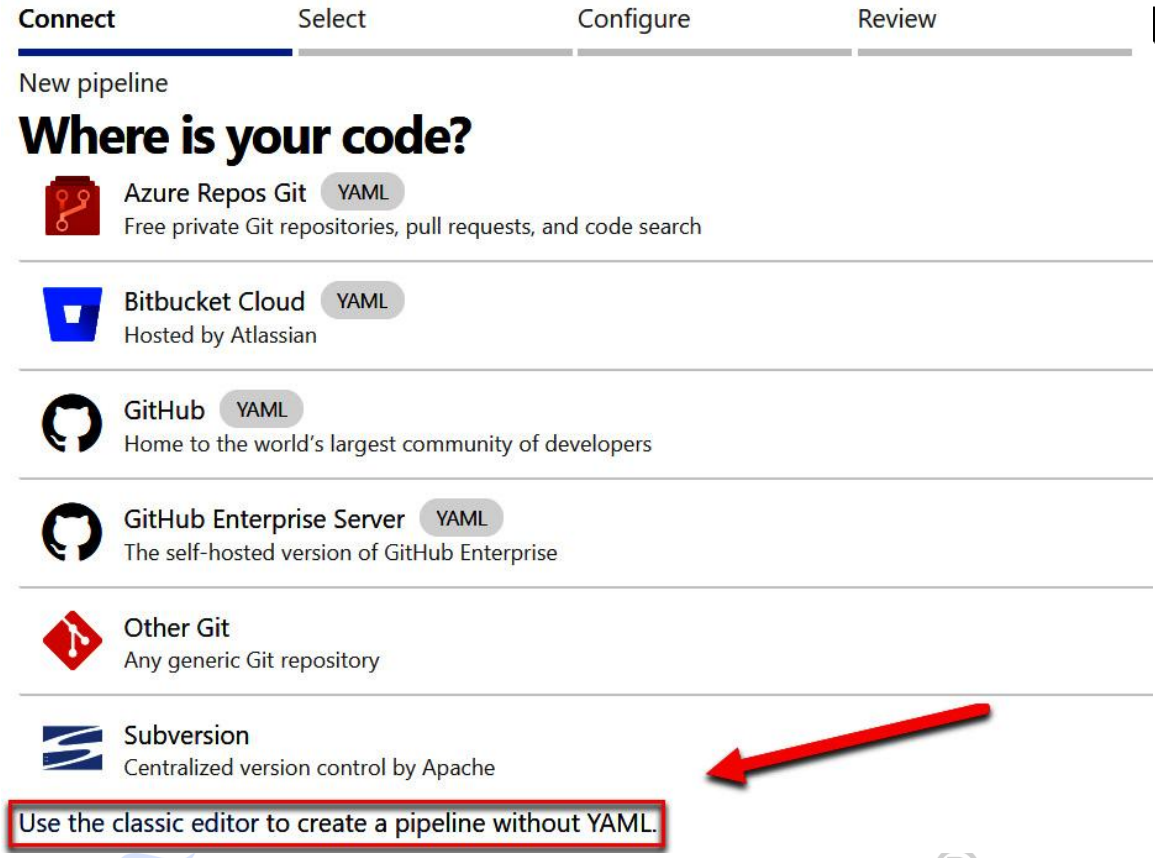
CREATING THE PIPELINE WITH TESTS

To create the pipeline, we need to go back to Azure DevOps. From there, follow these steps:

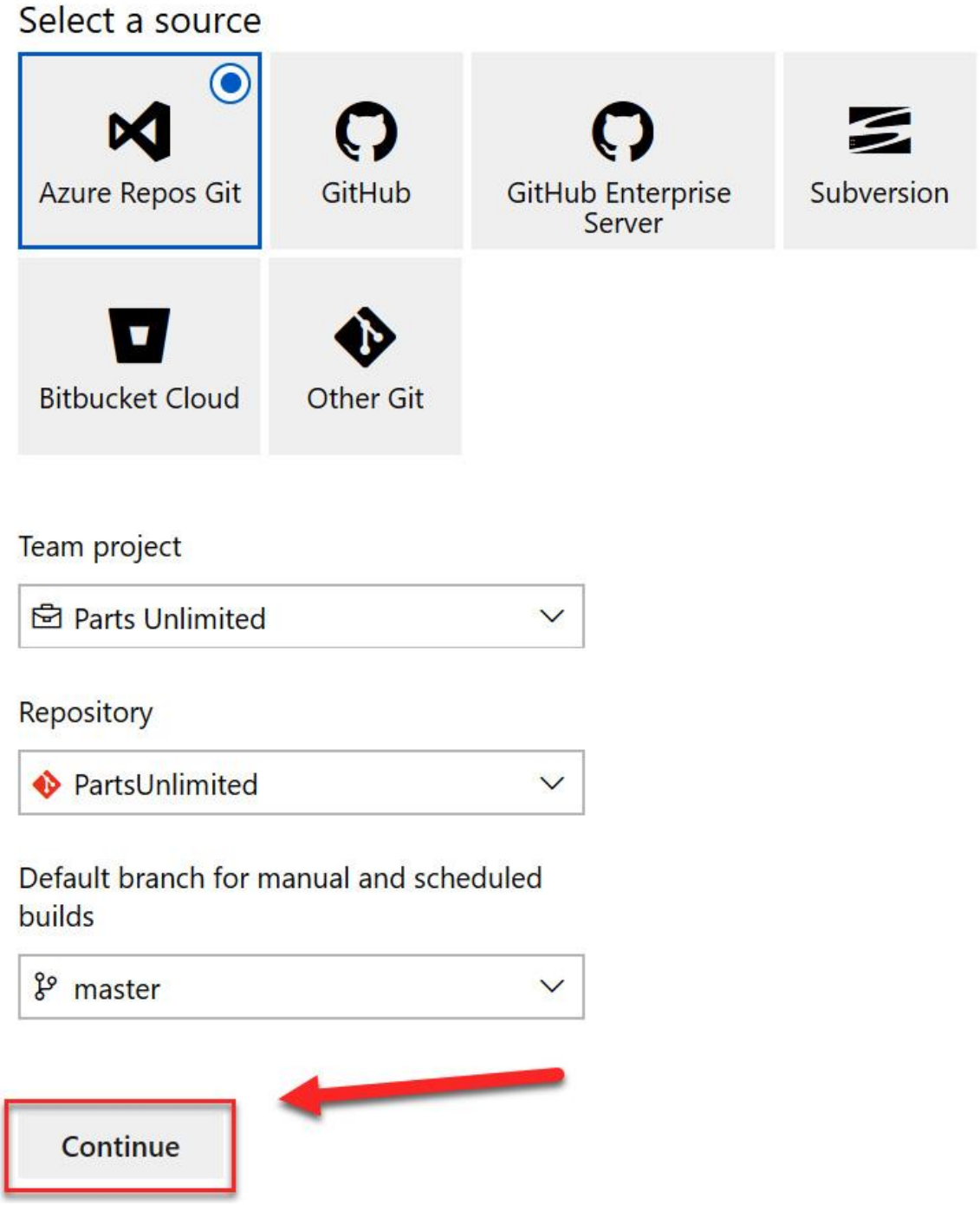
- From the left-hand menu, select **Pipelines**.
- At the top-right of the screen, find and click **New**



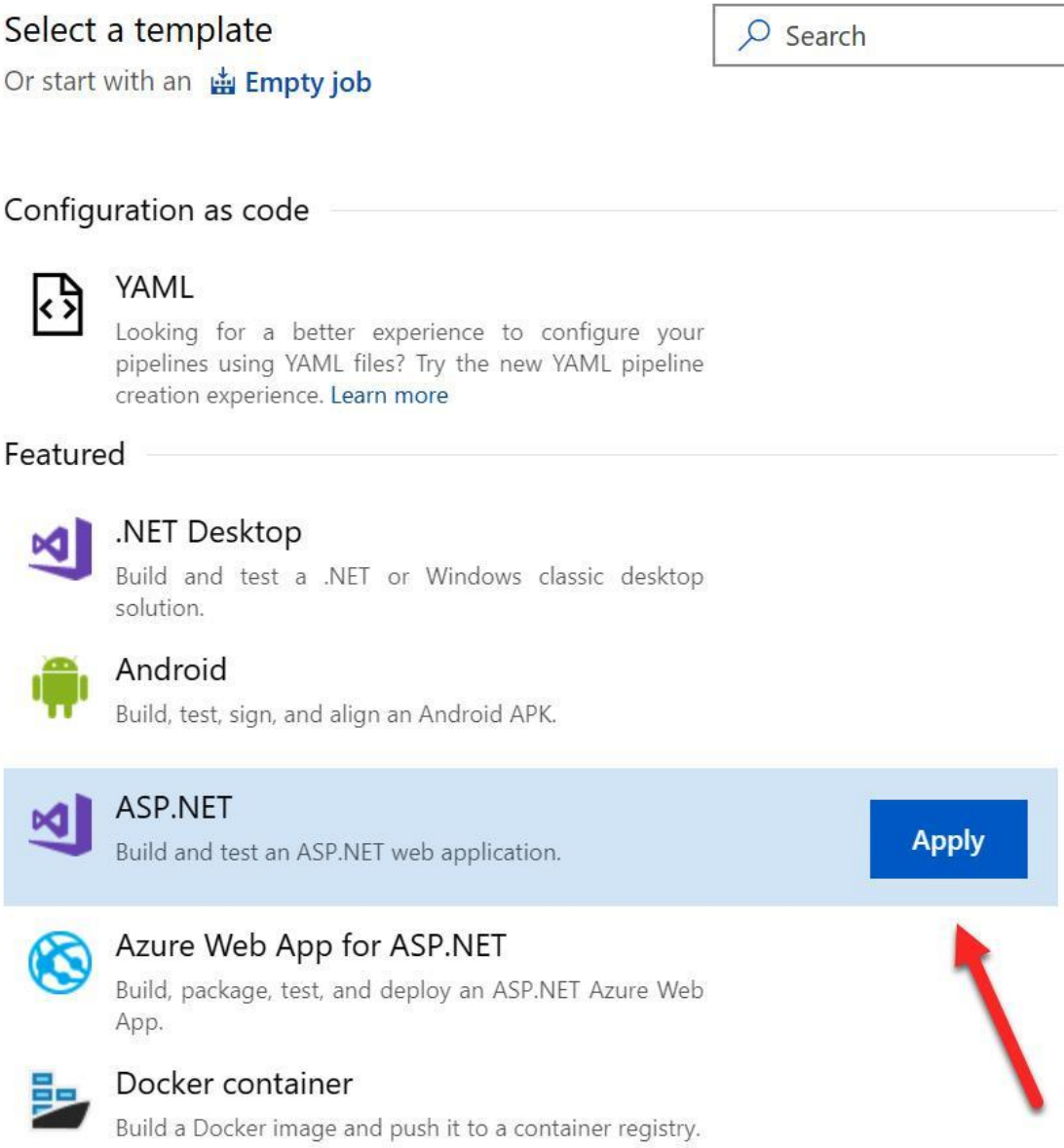
- The wizard for creating a build pipeline will appear. On the first screen, select **Use the classic editor** to create a pipeline using the designer:



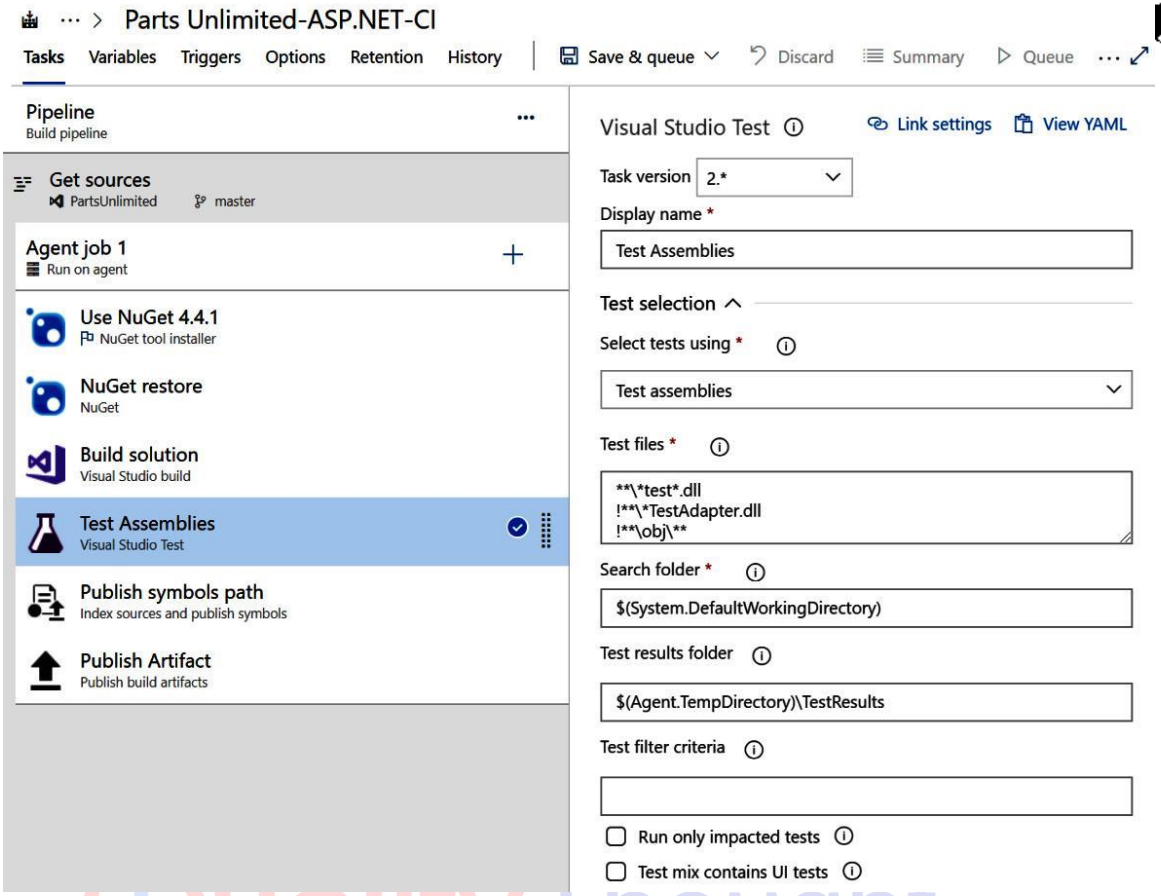
- On the next screen, make sure that **Azure Repos Git** is selected. Keep the default settings as they are and click **Continue**:



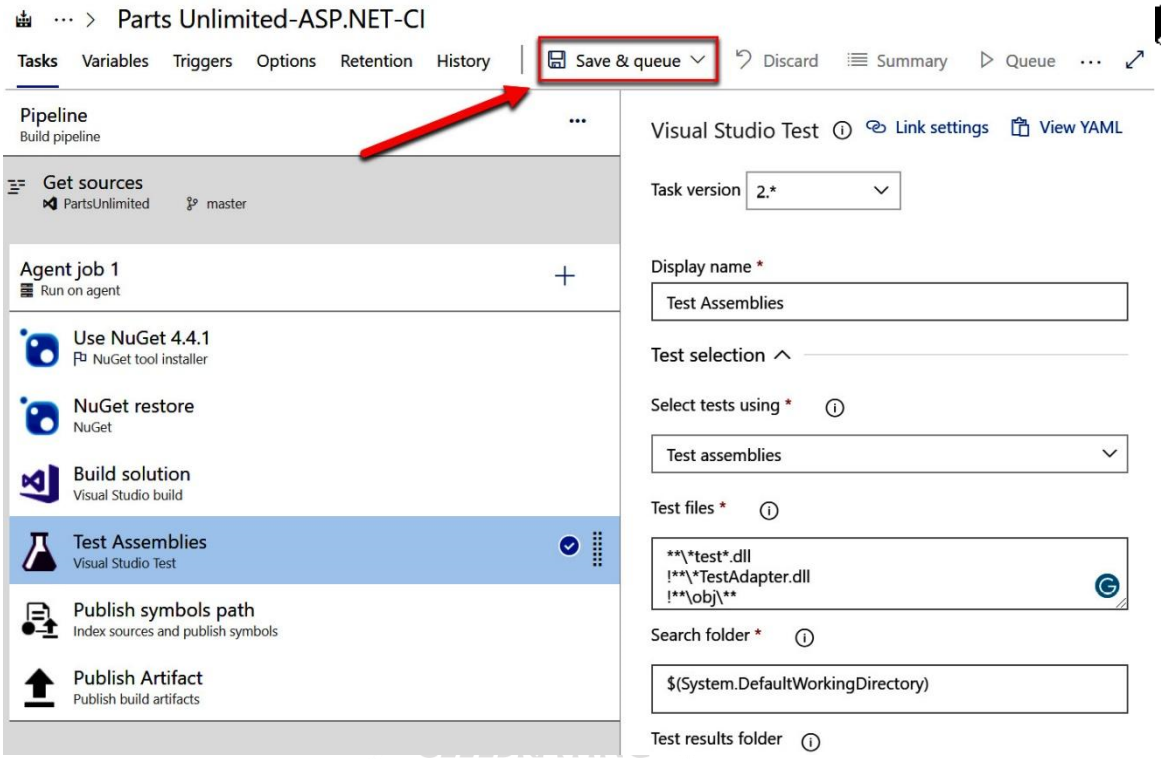
- Next, we need to select a template. Select **ASP.NET** from the overview and click **Apply**:



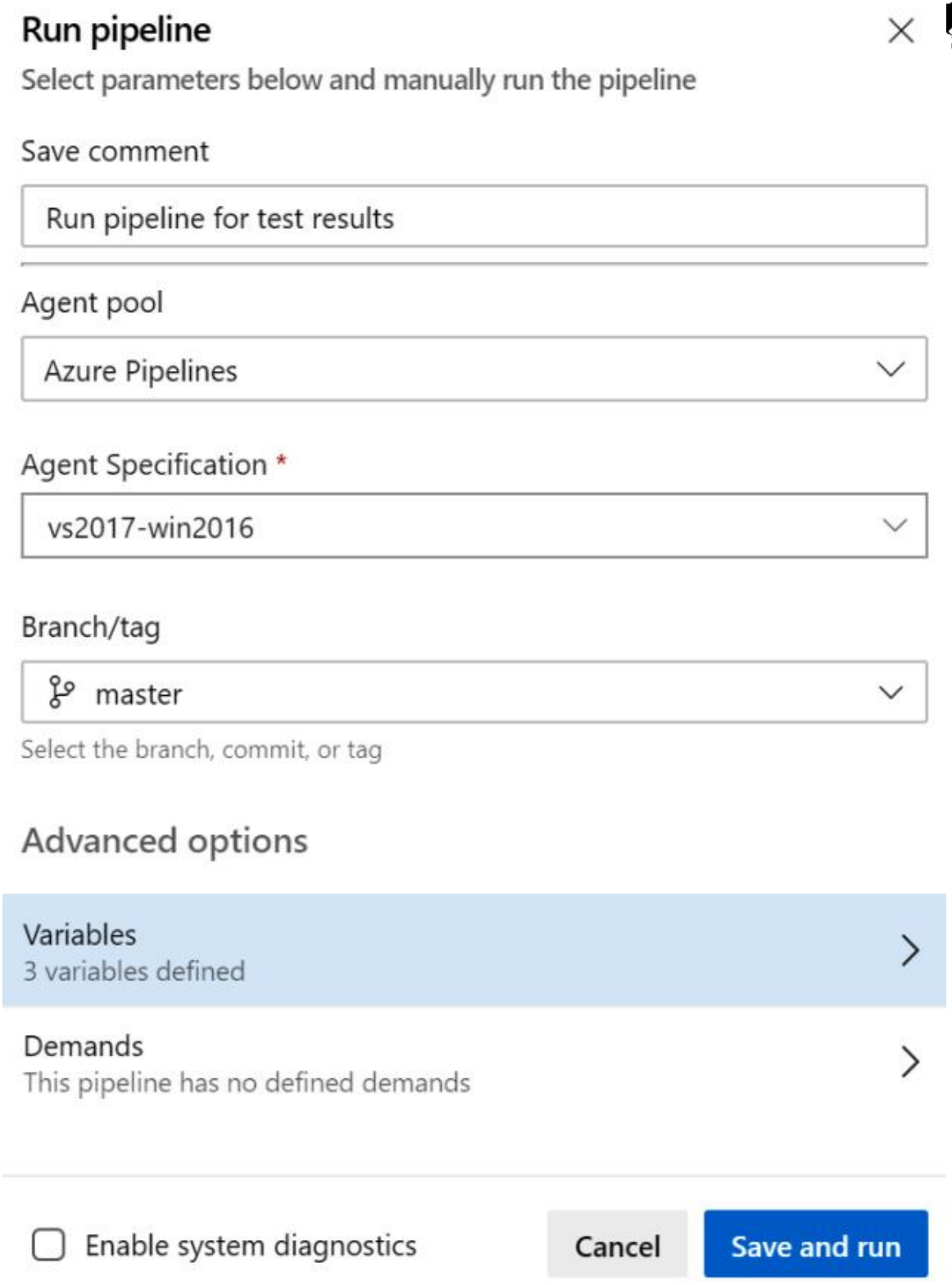
- With that, the pipeline will be created. Various tasks are added to the pipeline by default. We are going to use these tasks here. For this demo, we are going to focus on the **Test Assemblies** task. Click on this task and make sure that version **2** is selected. Under **Test selection**, you will see the following settings:



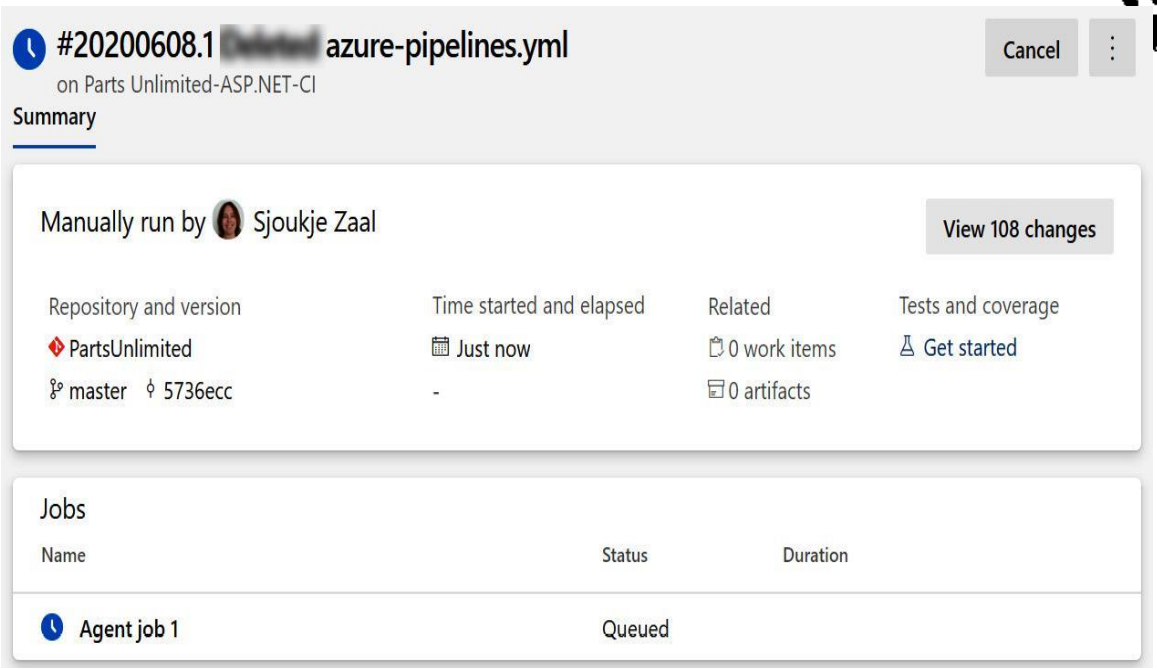
- By default, **Test assemblies** will be selected under **Select tests using**. Keep that selected. Since we want to run our unit tests automatically, this is the option we need to choose. Unit tests are usually stored inside an assembly.
- Also, by default, there are some assemblies already filled in. You can make changes to them if needed. For this demo, we will keep the default settings as they are because the task looks for assemblies in different folders that have **test** in them. Our test project is called **PartsUnlimited.UnitTests**, so this will be picked up by the task.
- The search folder is the folder that's used to search for test assemblies. In this case, this is the default working directory.
- The **test results** folder is where test results are stored. The results directory will always be cleaned before the tests are run.
- We are now ready to run the test. Click on **Save & queue** from the top menu and then again on the **Save & queue** sub-menu item to execute the build pipeline:



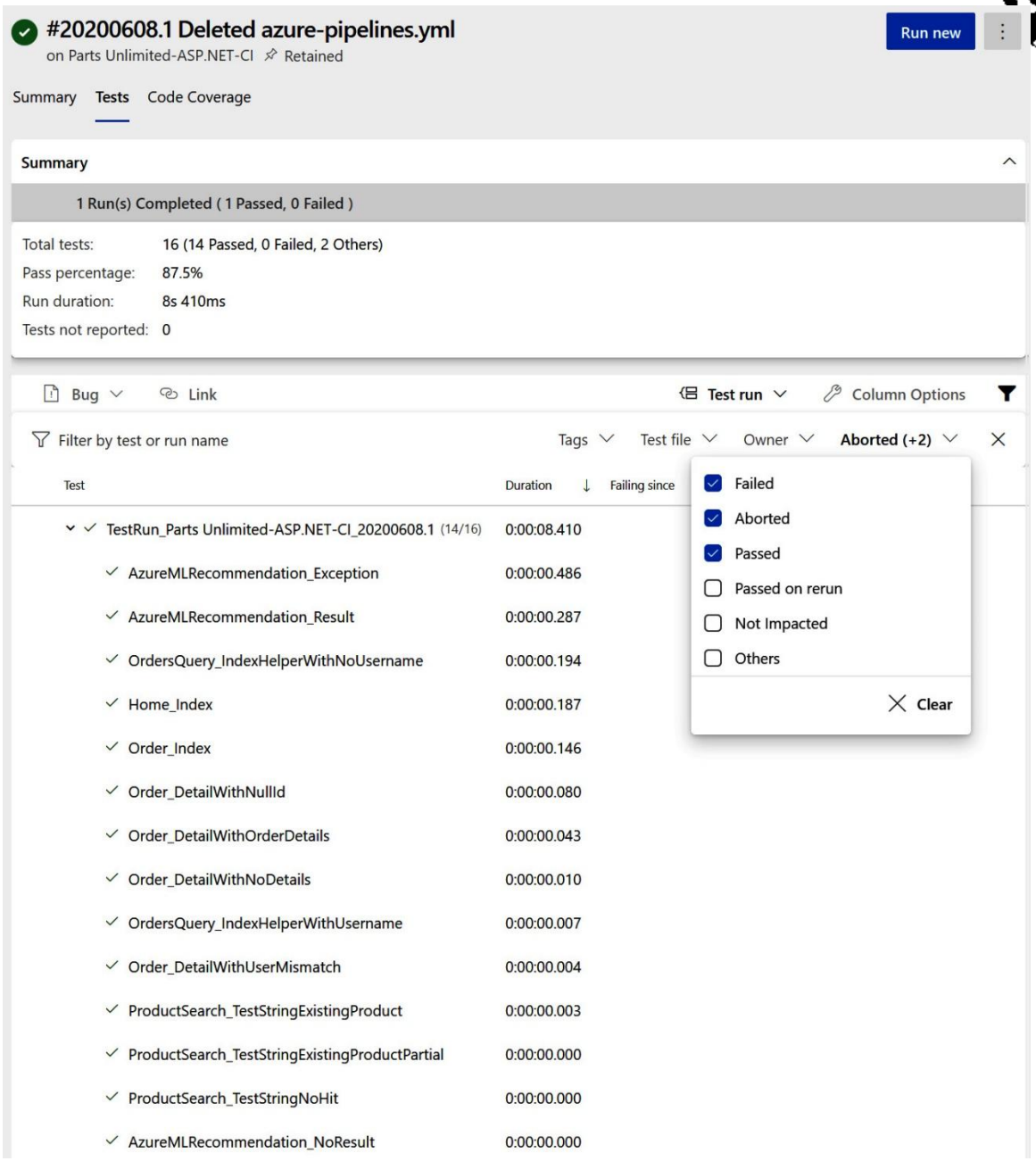
- The wizard for running the pipeline will open. Here, you can specify a comment and then select an **Agent Pool**, **Agent Specification**, and which **Branch/tag** you would like to use:



- Click **Save and run** to queue the pipeline. The overview page of the job will be displayed, which is where you can view the status of the execution:



- After a couple of minutes, the pipeline will have completed. From the right-top menu, under **Tests and coverage**, you will be able to see the pass percentage for the tests for this build. You can click on this to navigate to the test results (alternatively, you can navigate to it by clicking **Tests** from the top-left menu:
- On the **Tests** screen, you will see the number of tests you have, as well as the tests that passed and failed. You can also see the duration of the run from here.
- At the bottom of the screen, you can filter by specific tests. For instance, you can filter for tests that have been **Passed**, **Failed**, and **Aborted**:



In this demonstration, we have created a build pipeline that includes automatic unit testing for our source code.

INTRODUCTION TO CODE COVERAGE TESTING

With code coverage testing, you can measure what source code for an application is going to be tested. Code coverage testing measures how many

lines, blocks, and classes are executed while automated tests, such as unit tests, are running.

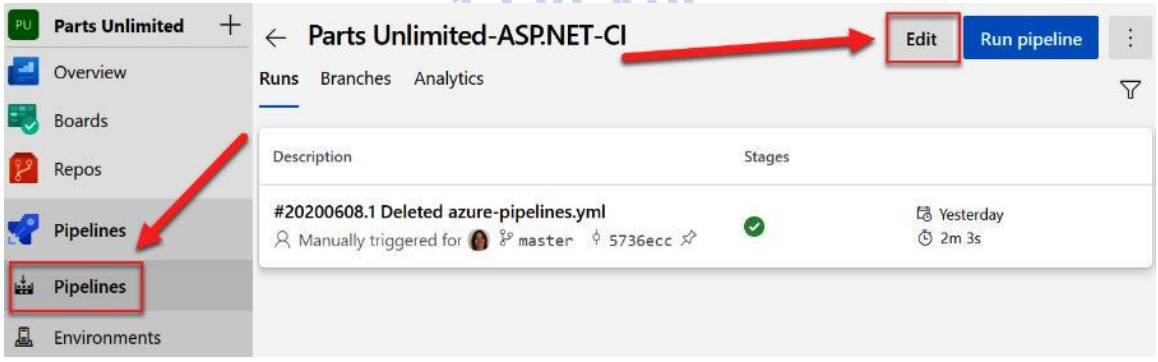
The more code that's tested, the more confident teams can be about their code changes. By reviewing the outcome of the code coverage tests, teams can identify what code is not covered by these tests. This information is very helpful as it reduces test debt over time.

Azure DevOps supports code coverage testing from the build pipeline. The **Test Assemblies** task allows us to collect code coverage testing results. There is also a separate task, called **Publish Code Coverage Results**, that can also publish these results. This task offers out-of-the-box support for popular coverage results formats such as Cobertura and JaCoCo.

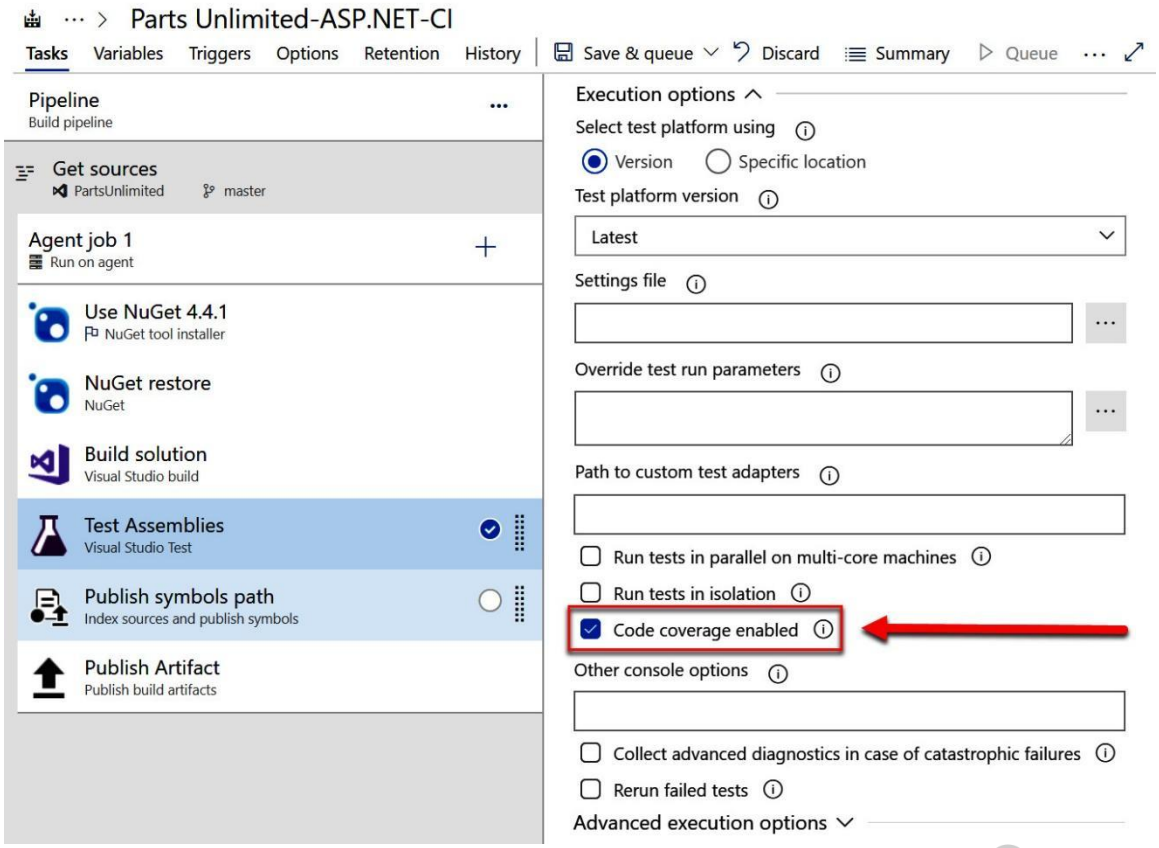
Performing code coverage testing

To perform code coverage testing, we need to open the build pipeline that we created in the previous demo. Let's get started:

- With the build pipeline open, select the **Edit** button in the right-hand corner:



- Navigate to the **Test Assemblies** task to open the settings. Under **Execution** settings, check the **Code coverage enabled** box:



- Now, **Save and queue** the build, specify a save comment, and wait until the pipeline is fully executed. The Visual Studio Test task creates an artifact that contains **.coverage** files that can be downloaded and used for further analysis in Visual Studio.

- After executing the pipeline, on the overview page of the build, select **Code Coverage** from the top menu and click on **Download code coverage results**. A file with the **.coverage** extension will be downloaded to your local filesystem.

- Double-click the downloaded file so that it opens in Visual Studio. From here, you can drill down into the different classes and methods to get an overview of the test results:

Code Coverage Results

20200609.2.release.any cpu.16.coverage

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
20200609.2.release.any cpu.16...	1753	80.67%	420	19.33%
fabrikamfiber.seleniumtests...	74	100.00%	0	0.00%
partsunlimited.dll	1620	89.95%	181	10.05%
PartsUnlimited	242	100.00%	0	0.00%
PartsUnlimited.Api	24	100.00%	0	0.00%
PartsUnlimited.Areas.Ad...	4	100.00%	0	0.00%
PartsUnlimited.Areas.Ad...	153	100.00%	0	0.00%
PartsUnlimited.Controlle...	255	75.00%	85	25.00%
PartsUnlimited.Models	278	100.00%	0	0.00%
PartsUnlimited.ProductS...	10	50.00%	10	50.00%
PartsUnlimited.Recomm...	2	50.00%	2	50.00%
PartsUnlimited.Security	28	100.00%	0	0.00%
PartsUnlimited.Utils	624	89.14%	76	10.86%
PartsUnlimited.ViewMo...	0	0.00%	8	100.00%
partsunlimited.unittests.dll	59	19.80%	239	80.20%
PartsUnlimited.UnitTests...	0	0.00%	64	100.00%
HomeControllerTests	0	0.00%	52	100.00%
OrdersControllerTests	0	0.00%	12	100.00%
PartsUnlimited.UnitTests...	59	25.21%	175	74.79%
MockDataContext	23	16.43%	117	83.57%
AddProductOrde...	0	0.00%	16	100.00%
CreateProduct(in...	0	0.00%	19	100.00%
Dispose()	1	100.00%	0	0.00%
Entry(object)	2	100.00%	0	0.00%
InitOrders()	0	0.00%	57	100.00%
InitProducts()	0	0.00%	21	100.00%
MockDataContex...	0	0.00%	4	100.00%
SaveChangesAsy...	20	100.00%	0	0.00%
MockHttpContext	2	7.69%	24	92.31%
TestDbAsyncEnumer...	4	33.33%	8	66.67%
TestDbAsyncEnumer...	2	18.18%	9	81.82%
TestDbAsyncQueryP...	12	75.00%	4	25.00%
TestDbSet<TEntity>	16	55.17%	13	44.83%

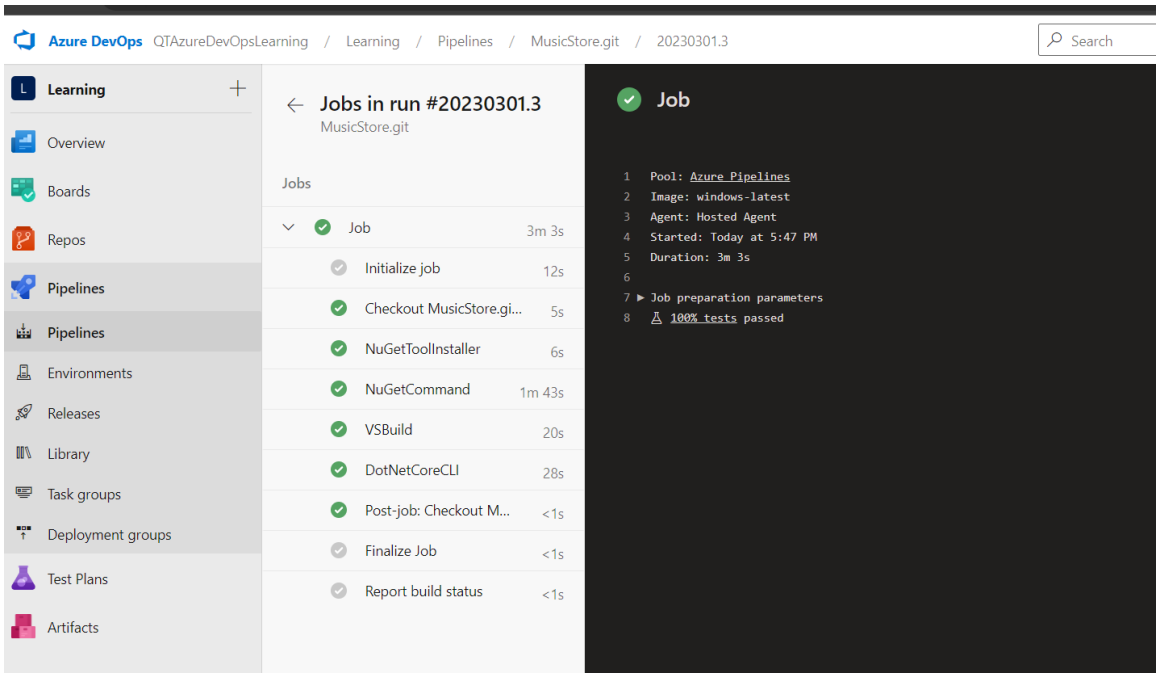
PUBLISHING TEST RESULTS AND CODE COVERAGE VIA YAML PIPELINE

- Sample Yaml for a project music store

```
# ASP.NET Core (.NET Framework)
# Build and test ASP.NET Core projects targeting the full .NET Framework.
# Add steps that publish symbols, save build artifacts, and more:
# https://docs.microsoft.com/azure/devops/pipelines/languages/dotnet-core

trigger:
```

```
- main

pool:
  vmImage: 'windows-latest'

variables:
  solution: '**/*.sln'
  buildPlatform: 'Any CPU'
  buildConfiguration: 'Release'

steps:
- task: NuGetToolInstaller@1

- task: NuGetCommand@2
  inputs:
    restoreSolution: '$(solution)'

- task: VSBuild@1
  inputs:
    solution: '$(solution)'
    msbuildArgs: '/p:DeployOnBuild=true /p:WebPublishMethod=Package /p:Pack
ageAsSingleFile=true /p:SkipInvalidConfigurations=true /p:DesktopBuildPackageL
ocation="$(build.artifactStagingDirectory)\WebApp.zip" /p:DeployIisAppPath="D
efault Web Site"'
    platform: '$(buildPlatform)'
    configuration: '$(buildConfiguration)'

- task: DotNetCoreCLI@2
  inputs:
    command: 'test'
    projects: 'MusicStoreTest/MusicStoreTest.csproj'
```

Result will be as shown below



Hosting Your Own Azure Pipeline Agent

We will first look at the types of pipeline agents available and then dive into the technical specifications of setting up the agent pools. We will also look at how you can use VM scale sets for large-scale Azure DevOps projects.

We'll be covering the following topics:

- Azure pipeline agent overview
- Understanding the types of agents in Azure Pipelines
- Planning and setting up your own pipeline agent in Azure
- Updating your Azure pipeline to use your self-hosted agent
- Using containers as your self-hosted agents

AZURE PIPELINE AGENT OVERVIEW

An Azure pipeline agent is the component responsible for executing the tasks defined in the pipeline definition. This agent typically runs inside a VM or container and includes the pre-requisites required for the pipeline to run successfully.

In most cases, you'll need to have an agent in order to run the pipeline. As your project size and the number of developers grows, you will need to have more agents to support the scale.

Each execution of a pipeline initiates a job on one of the agents, and one agent can only run one job at a time. Azure pipeline agents can be hosted in the cloud or on-premises in one of the following compute infrastructures:

- Server or client host (physical or virtual)
- Containers

Azure pipeline agents are grouped into **agent pools**. You can create as many agent pools as you require.

UNDERSTANDING THE TYPES OF AGENTS IN AZURE PIPELINES

Azure Pipelines offers two types of agents:

- Microsoft-hosted agents
- Self-hosted agents

Microsoft-hosted agents

Microsoft-hosted agents are fully managed VMs, deployed and managed by Microsoft. You can choose to use a Microsoft-hosted agent with no additional pre-requisites or configurations. Microsoft-hosted agents are the simplest and are available at no additional cost.

Every time you execute a pipeline, you get a new VM for running the job, and it's discarded after one use.

Self-hosted agents

Self-hosted agents are servers owned by you, running in any cloud platform or data center owned by you. Self-hosted agents are preferred due to various reasons, including security, scalability, and performance.

You can configure your self-hosted agent to have the dependencies pre-installed, which will help you decrease the time for your pipeline execution.

Choosing between a Microsoft-hosted agent and self-hosted agents depends on various factors, such as the following:

- The size of the code base
- The number of developers
- The build and release frequency

- The dependencies and packages required for the build process
- Security and compliance requirements
- Performance

If your code base is small and the build pipeline is optimized, it's better to use Microsoft-hosted agents as it won't take much time to download all the dependencies on the fly. However, if you have a large code base with numerous amounts of dependencies, using a self-hosted agent will be a better option as you can eliminate various build pre-creation tasks from the pipeline by configuring them in your self-hosted environment in advance. Self-hosted agents would be the only option in the case of highly secure and customized build pipelines that interact with other services running in your network. If you need more CPU and memory than what is provided with Microsoft-hosted agents, you can use self-hosted agents with your customized sizing.

It is recommended to start with Microsoft-hosted agents and move to self-hosted at a later stage when the Microsoft-hosted agents become a bottleneck in your build and release process.

PLANNING AND SETTING UP YOUR SELF-HOSTED AZURE PIPELINE AGENT

In order to use a self-hosted agent with Azure Pipelines, you will need to set up a machine and configure it for your pipeline requirements. Typically, you would choose an OS version best suited for your project, considering the framework, libraries, and build tools compatibility.

For the purpose of this demonstration, we'll be setting up a VM in Azure and will configure it to use a self-hosted agent. You can choose to host your agent server in any cloud or on-premises environment.

Choosing the right OS/image for the agent VM

The first decision you take while setting up the VM is choosing the OS/image for the server depending on your target deployment. If you are deploying in an on-premises environment, you may just select one of the supported OS versions (such as Windows Server 2016) and install the necessary software. In the case of cloud deployments, you have multiple options provided in the form of images, which come in various combinations of OS version and pre-installed tools.

It is advised that you have the agent VM specifications planned with your developers to have them best suited for your project needs. Here is a recommended approach:

- Identify whether your application is built to run on Windows, Linux, or macOS. If it's cross-platform, choose the one that runs it best and has support for the build tools you're using.

- List down the underlying frameworks and external libraries/components used with their versions.
- Select the latest version of the OS version from the top-level OS selected in step 1.
- Identify whether it is compatible and supported by the original equipment manufacturers (OEMs) for all the dependencies listed in step 2.
- Keep going one version down at a time and select the one that is compatible for all the required dependencies for your project.

Based on the specifications identified in this process, you can choose to start with a vanilla OS and install your required frameworks and build tools, or choose a pre-created image in the cloud.

OS support and pre-requisites for installing an Azure Pipelines agent

Azure supports various OS versions to use as a self-hosted agent; based on the OS you choose, there is a set of pre-requisites you'll need to complete before you can install the Azure Pipelines agent on your host.

Supported OSes

The following list shows the supported OSes:

- Windows-based:
    a) Windows 7, 8.1, or 10 (if you're using a client OS)
    b) Windows Server 2008 R2 SP1 or higher (Windows Server OS)
- Linux-based:
    a) CentOS 7, 6
    b) Debian 9
    c) Fedora 30, 29
    d) Linux Mint 18, 17
    e) openSUSE 42.3 or later
    f) Oracle Linux 7
    g) Red Hat Enterprise Linux 9,8, 7, 6
    h) SUSE Enterprise Linux 12 SP2 or later
    i) Ubuntu 20.04,18.04, 16.04

Pre-requisite software

Based on the OS you choose, you will have to install the following pre-requisites. before you can set up the host as an Azure pipeline agent:

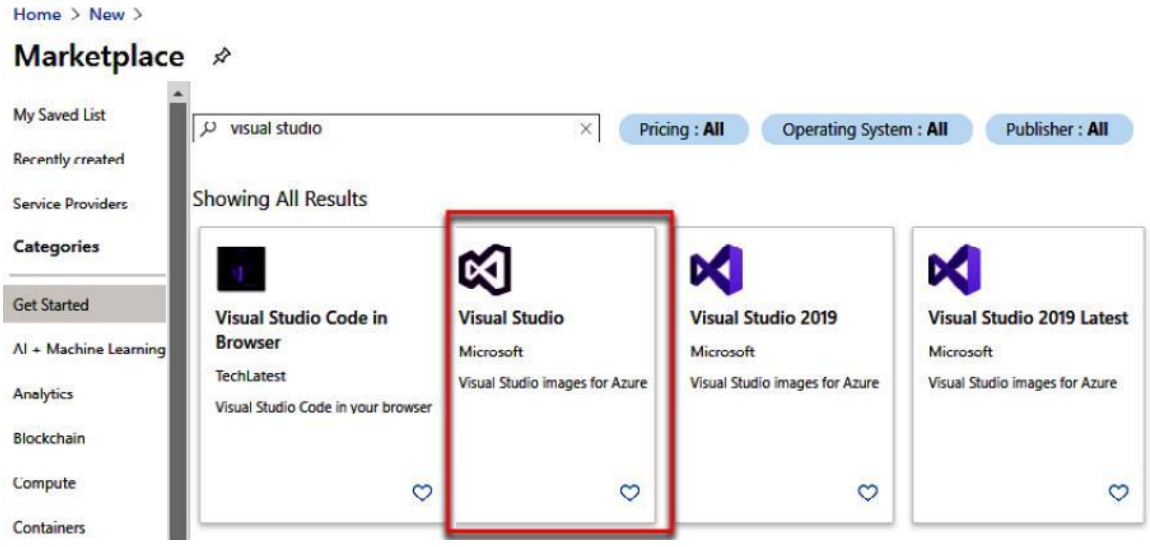
- Windows-based:
  - PowerShell 3.0 or higher
  - .NET Framework 4.6.2 or higher
- Linux/ARM/macOS-based:
  - Git 2.9.0 or higher.
  - RHEL 6 and CentOS 6 require installing the specialized RHEL.6-x64 version of the agent.

CREATING A VM IN AZURE FOR YOUR PROJECT

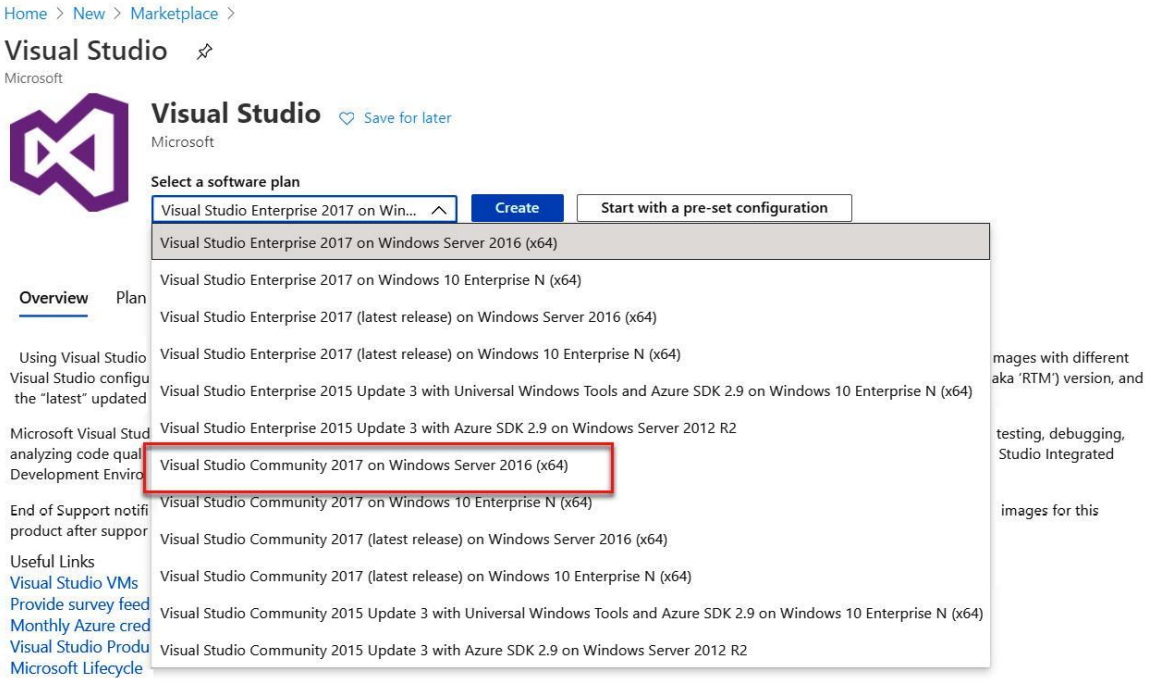
The **PartsUnlimited** project is built using .NET Framework 4.5 and Visual Studio as the primary IDE tool. You can review that by browsing through the repository in the **PartsUnlimited** project in your Azure DevOps.

Looking at that, our best bet would be to use a Visual Studio-based server OS. Let's look in Azure to explore our options here:

- Log in to the Azure portal and click **+ Create a resource**.
- Search for **Visual Studio** and select the **Visual Studio images for Azure** option:



- Now, you'll be able to select from the various combinations available. We'll go with **Visual Studio community 2017 on Windows Server 2016(x64)**:



- Click **Create** to start creating a VM. Choose the required subscription, resource group, and other settings based on your preference.
- In further pages, you can modify the settings to use a pre-created virtual network, as well as customize the storage settings and other management aspects. Please review the documentation to explore more on VM creation in Azure.
- Log in to the VM upon creation and install the required pre-requisites.

Setting up the build agent

In this section, we'll configure the newly created VM to use as a self-hosted pipeline agent.

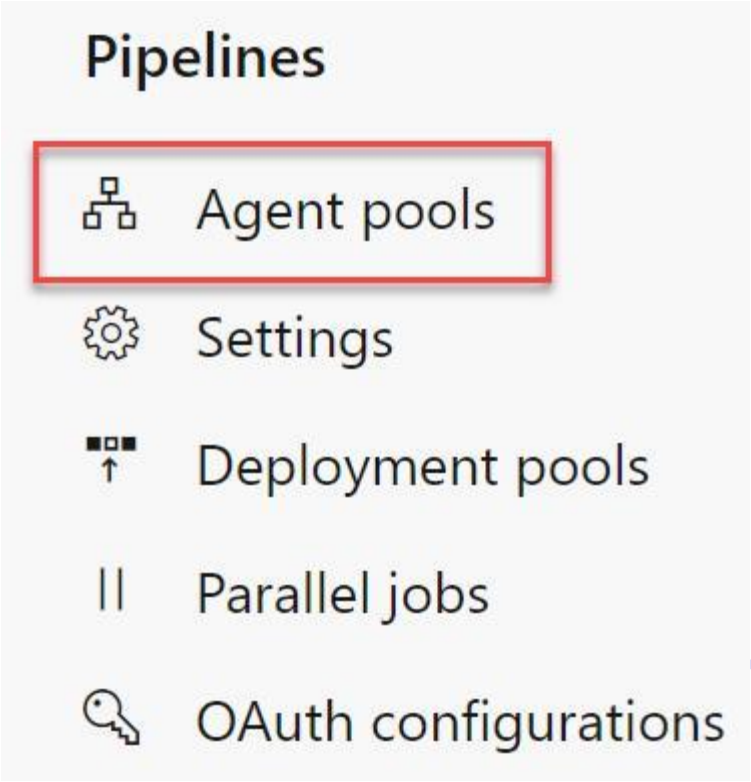
Setting up the agent pool in Azure DevOps

You can organize your agents in Azure DevOps as **agent pools**. **Agent pools** are a collection of your self-hosted agents; they help you organize and manage the agents at the pool level, rather than managing them individually.

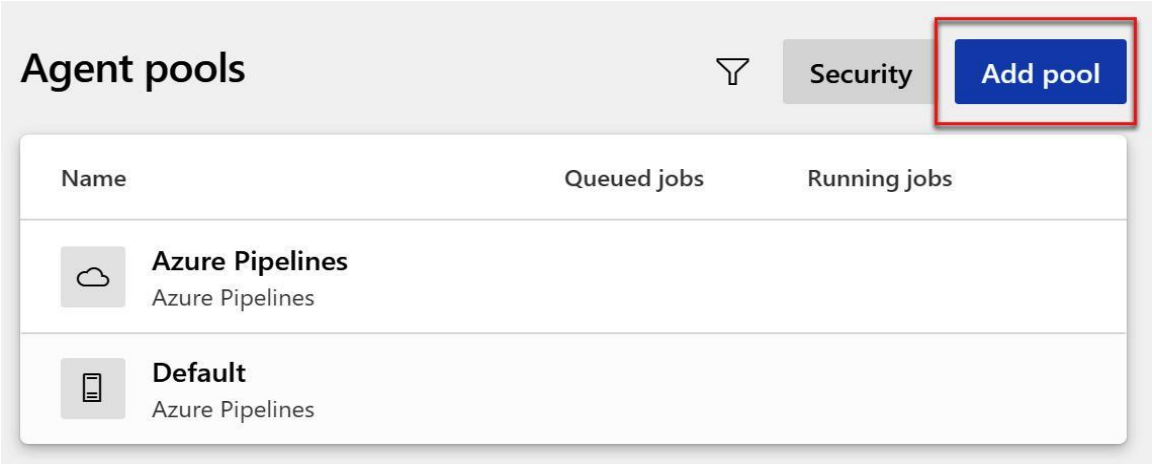
Agent pools are managed at the organization level and can be used by multiple projects at the same time. Let's create an **agent pool** to get started:

- Log in to Azure DevOps and click on **Organization settings**:

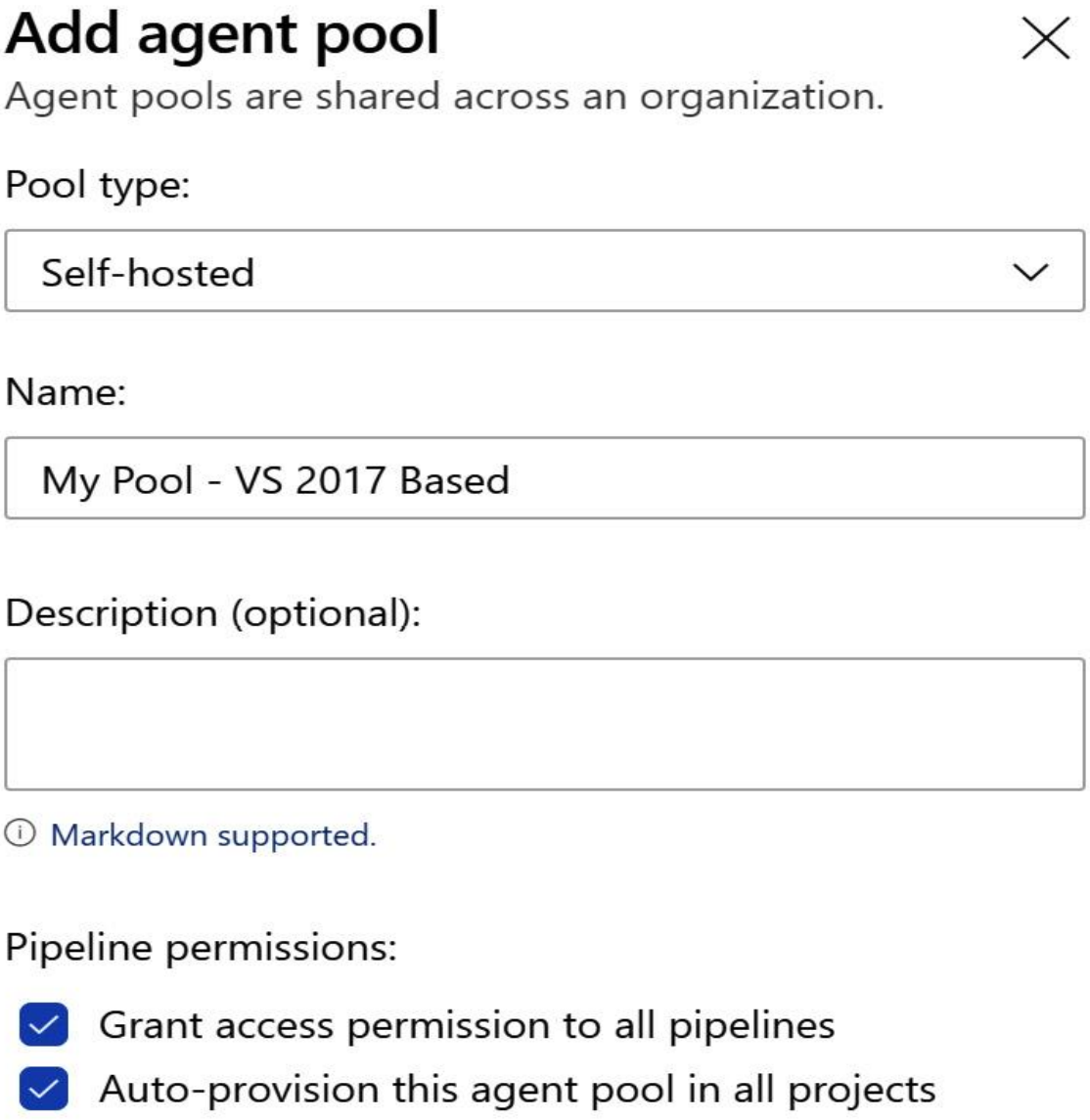
- Click on **Agent pools** under **Pipelines**:



- You will see that there are two default agent pools created. Click on **Add pool** to create a new pool:



- Provide the pool type, name, description, and pipeline permissions. Under the permissions option, you can choose to make this pool available to all pipelines and projects at once. Click **Create** once you're ready:
    - --**Pool type**: **Self-hosted**
    - --**Name** and **Description**: Give a meaningful name that you can use to refer to later:

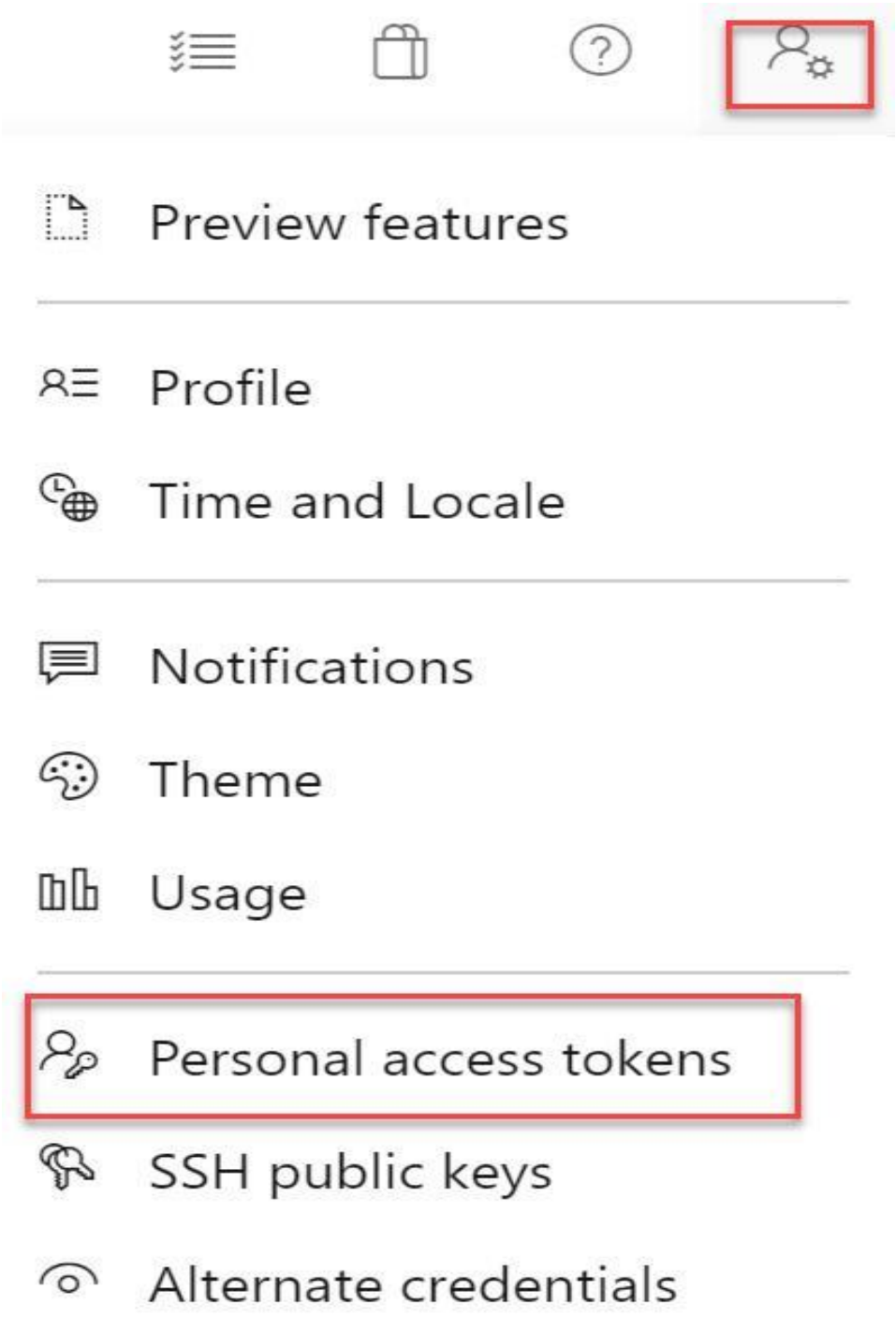


- Your agent pool should be listed under **Agent pools** now.
- We will now set up an access token for the agent VM to be able to authenticate with Azure DevOps.

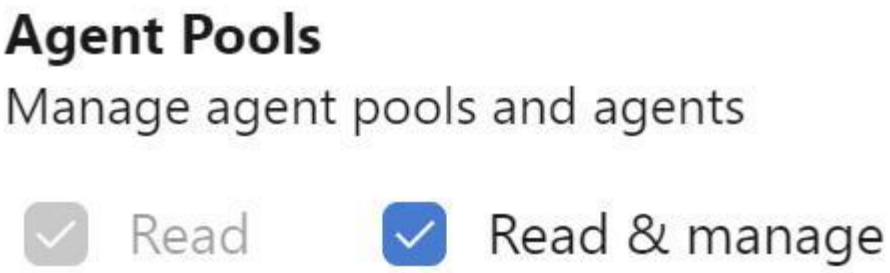
Setting up an access token for agent communication

In this task, you will create a personal access token that will be used by the Azure Pipelines agent to communicate with your Azure DevOps organization:

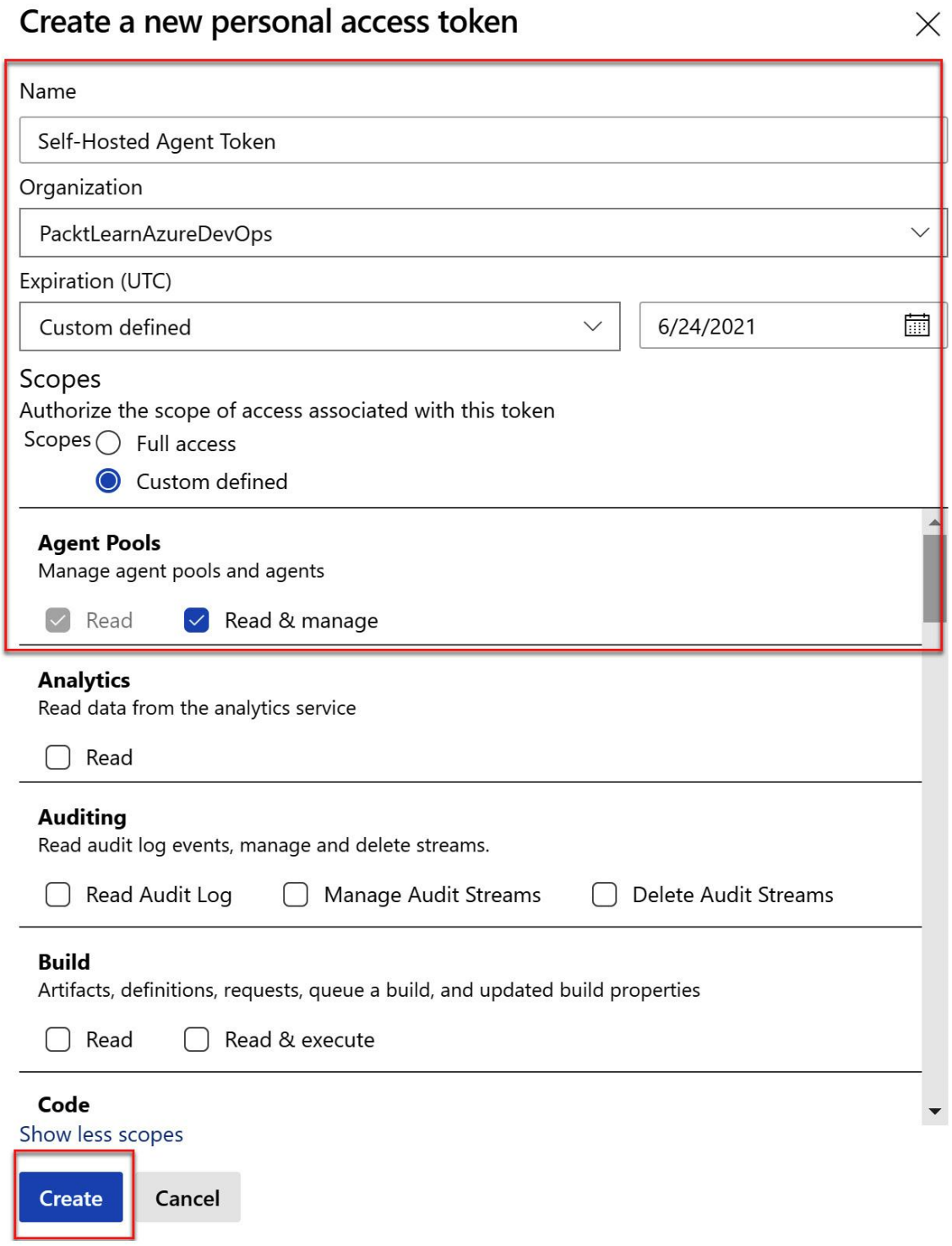
- Sign in to your Azure DevOps organization with the admin user account.
- Go to your user profile and click **Personal access tokens**:



- Click on **New token**.
- Provide the token specifications as given here:
    - **Name**: **Self-Hosted Agent Token**.
    - **Organization**: Your Azure DevOps organization.
    - **Expiration**: You can choose a date as per your choice. This is only for a one-time setup; you do **not** need to re-configure the agent once this token expires.
- **Scope**: **Custom defined**.
- On **Scope**, it is recommended to only give the permissions required to manage the agents. Click on **Show all scope** and select both the **Read** and **Read & manage** permissions:



- Review all the settings and click **Create**:

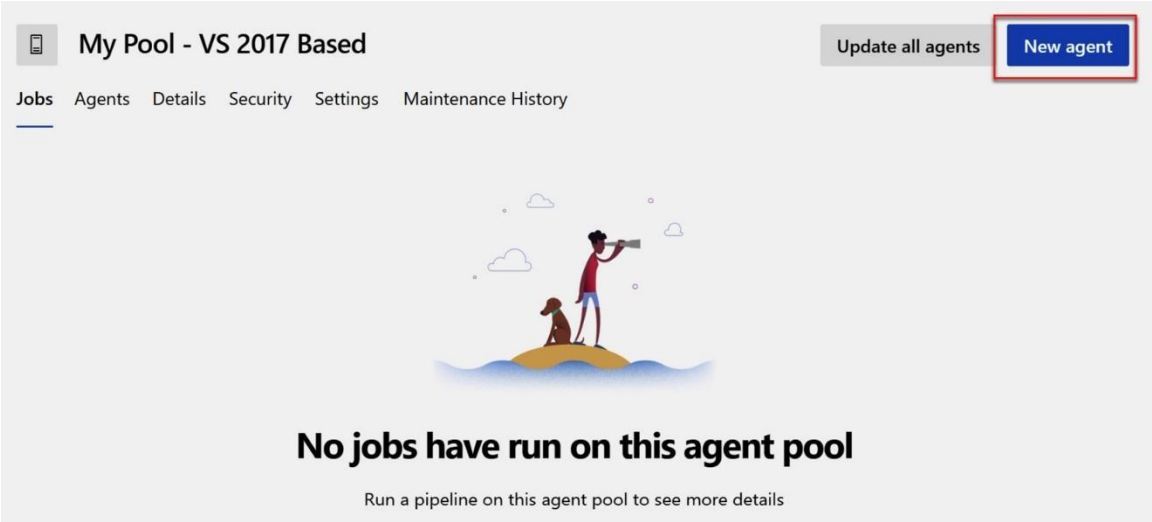


- Once you click **Create**, Azure DevOps will display the personal access token. Please copy the token and save it in a secure location. If you happen to lose this token, you must create a new token for setting up new agents. We will use this token when setting up the Azure Pipelines agent.

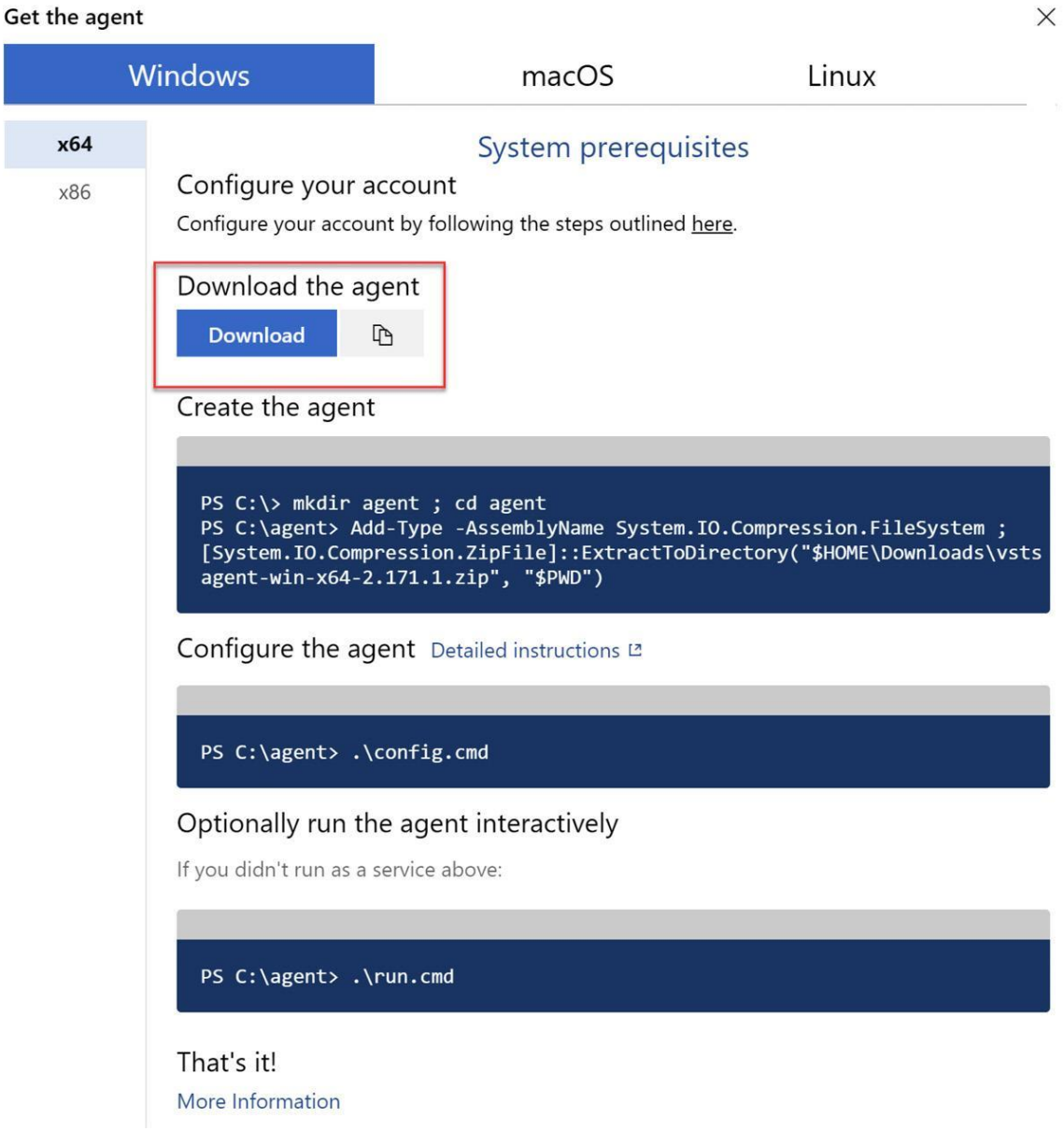
- Now that we have completed the agent pool setup in Azure DevOps, we can start installing the agent in the VM we created earlier.

Installing Azure Pipelines agents

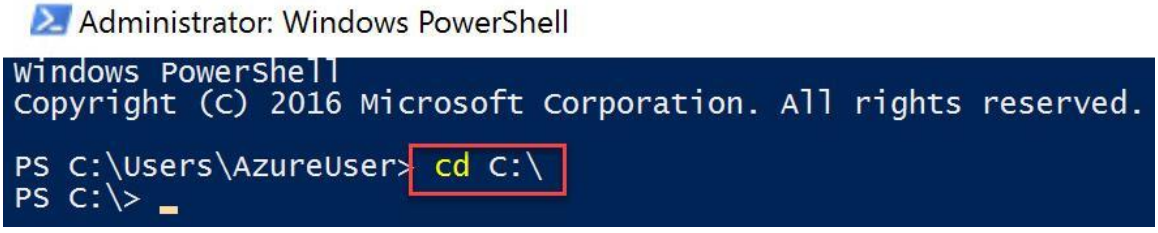
- You are now ready to install the Azure Pipelines agent on your VMs that you created earlier. Let's download the Azure Pipelines agent. Before you start, please log in to the VM created earlier using **Remote desktop**:
- In your Azure DevOps account, browse to **Organization Settings** > **Agent Pools**.
- Select your newly created agent pool and click on **New agent**:



- On the next page, you can download the agent installer based on the OS and architecture (x64/x32). In our example, we're using a Windows Server 2016-based VM. We'll choose Windows and the x64 architecture. You can also copy the download URL and use it to download the agent directly inside your self-hosted agent machine. You can also choose to follow the installation steps given in the Azure DevOps portal based on the OS for your agent machine:



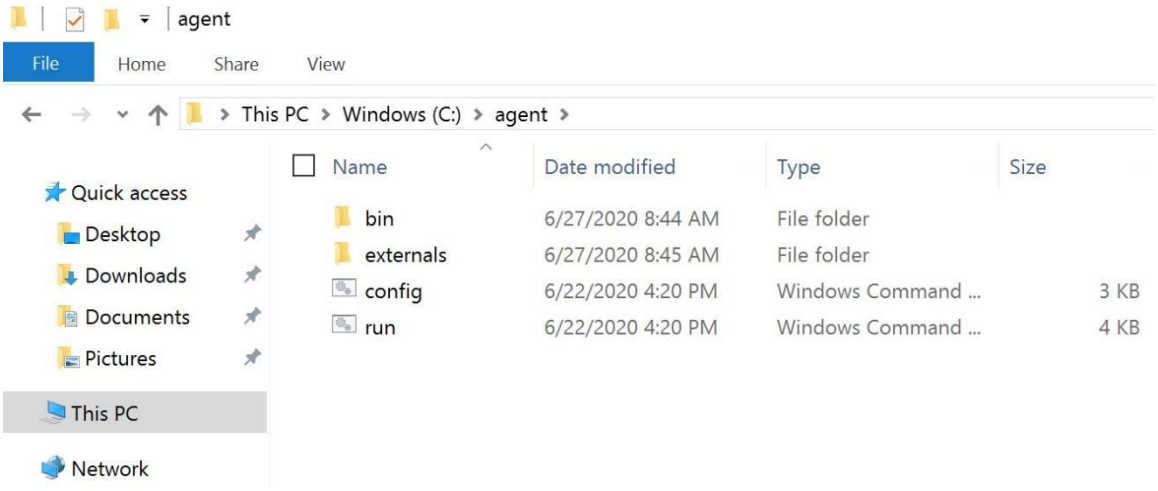
- Launch an elevated PowerShell window and change to the **C:** directory root by running the **cd C:** command:



- Run the following PowerShell commands to create an agent directory on the **C** drive and extract the agent files to the new directory. Please note that you may have to change the filename/path depending on the version of the agent you've downloaded and the directory where you saved the downloaded file:

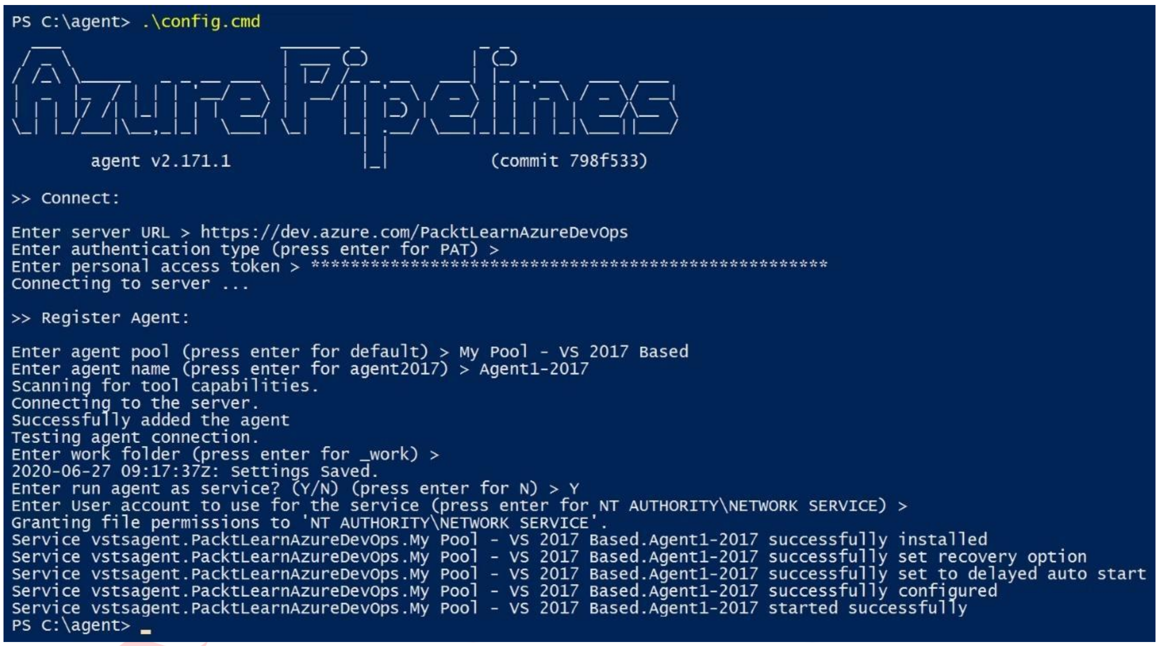
```
mkdir agent ; cd agent
Add-Type -AssemblyName System.IO.Compression.FileSystem ;
[System.IO.Compression.ZipFile]::ExtractToDirectory("$HOME\Downloads\vsts-agent-win-x64-2.171.1.zip", "$PWD")
```

- It will take a minute or two to extract the files. Please browse to the new directory once it's completed. You should see files as displayed in the following screenshot:

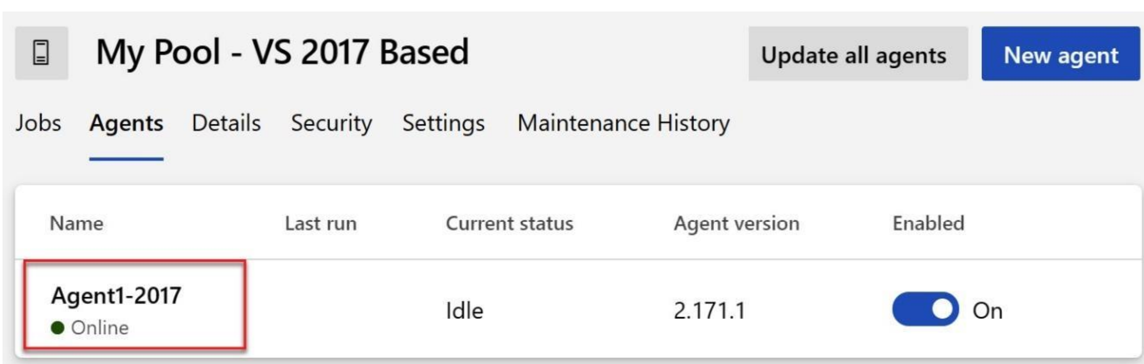


- You can run the Azure pipeline agent in two modes:
- --**Run Once**: This will run the agent manually using the **run** batch file stored in the agent directory. Your agent will stop responding to pipelines if you stop the interactive authentication.
- --**Run as Service**: In this version, you configure the agent to run as a Windows service that will remain online all the time and auto-start on reboot. This is the recommended setup for production scenarios.
- Let's configure the agent to run as a service. In your PowerShell window, run **.\config.cmd**.
- This will ask a series of questions about your Azure DevOps organization.
- Enter your Azure DevOps organization as the server URL. Typically, this would be https://dev.azure.com/YourOrganizationName.
- Press *Enter* to select **PAT** (**Personal access token**) as the authentication mechanism for Azure DevOps.
- Provide your personal access token generated earlier.

- Provide the agent pool name you created earlier.
- Provide a name for this agent.
- Provide a working directory for the agent to choose as default.
- Finally, press *Y* and hit *Enter* to configure to run the agent as a Windows service.
- You can accept the default account to run the service.
- This will complete the agent setup; at the end, you should see a message stating that the services are started successfully:


PS C:\agent> .\config.cmd
agent v2.171.1
(commit 798f533)
>> Connect:
Enter server URL > https://dev.azure.com/PacktLearnAzureDevOps
Enter authentication type (press enter for PAT) >
Enter personal access token > **
Connecting to server ...
>> Register Agent:
Enter agent pool (press enter for default) > My Pool - VS 2017 Based
Enter agent name (press enter for agent2017) > Agent1-2017
Scanning for tool capabilities.
Connecting to the server.
Successfully added the agent
Testing agent connection.
Enter work folder (press enter for _work) >
2020-06-27 09:17:37Z: Settings Saved.
Enter run agent as service? (Y/N) (press enter for N) > Y
Enter User account to use for the service (press enter for NT AUTHORITY\NETWORK SERVICE) >
Granting file permissions to 'NT AUTHORITY\NETWORK SERVICE'.
Service vstsagent.PacktLearnAzureDevOps.My Pool - VS 2017 Based.Agent1-2017 successfully installed
Service vstsagent.PacktLearnAzureDevOps.My Pool - VS 2017 Based.Agent1-2017 successfully set recovery option
Service vstsagent.PacktLearnAzureDevOps.My Pool - VS 2017 Based.Agent1-2017 successfully set to delayed auto start
Service vstsagent.PacktLearnAzureDevOps.My Pool - VS 2017 Based.Agent1-2017 successfully configured
Service vstsagent.PacktLearnAzureDevOps.My Pool - VS 2017 Based.Agent1-2017 started successfully
PS C:\agent>

- Now, if you look under your agent pool in the Azure DevOps portal, you should see this agent listed:



- You now have a ready-to-use, self-hosted agent for your Azure pipelines! This self-hosted agent can be used to run your Azure pipeline jobs, and you can add as many agents as you want in a similar fashion. You may have various types of hosted agents in one pool; the appropriate agent

for the job is automatically selected based on the pipeline requirements, or you can select an agent specifically at execution time.

- Typically, in a large environment, you'd pre-create an image of an agent server so that it is faster to provision additional agents whenever needed. In the next section, we will update our pipeline to leverage this newly set-up self-hosted agent. Please refer to this documentation if you wish to use a Linux-based hosted agent: https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/v2-linux?view=azure-devops.
- Refer to the following link for macOS-based agents: https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/v2-osx?view=azure-devops.

PREPARING YOUR SELF-HOSTED AGENT TO BUILD THE PARTS UNLIMITED PROJECT

Before we can start using the self-hosted agent, we must prepare it to support building our sample project, **PartsUnlimited**. The **PartsUnlimited** project is built using Visual Studio leveraging .NET Framework, Azure development tools and .NET Core, Node.js, and so on. In order to use our self-hosted agent for building the solution, we must install the required dependencies prior to running the pipeline jobs:

- Log in to your self-hosted agent VM.
- Download the Visual Studio build tools with this link: https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=16.
- This will launch Visual Studio Installer.
- Select **ASP.Net and Web Development** and **Azure Development**.
- Click **Modify**. This will start installing the required framework and tools.
- Once this is completed, please download and install .NET Core 2.2. You can download it from this link: https://dotnet.microsoft.com/download/dotnet-core/thank-you/sdk-2.2.110-windows-x64-installer.
- You can find all the .NET downloads here: https://dotnet.microsoft.com/download
- Install Azure PowerShell by running the following commands in an elevated PowerShell window:

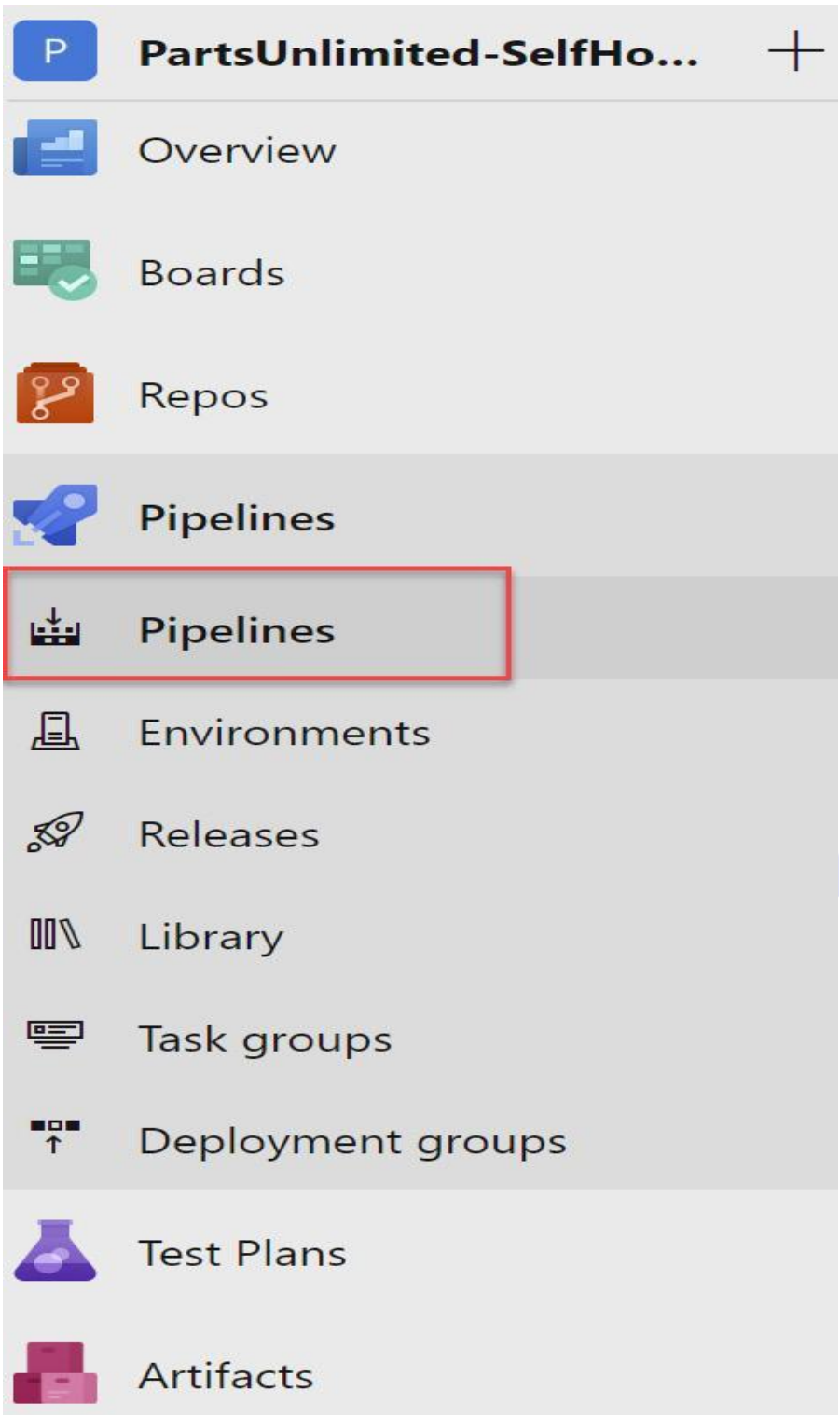
```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
Install-Module AzureRM -AllowClobber
```

- Install Node.js version 6.x from https://nodejs.org/download/release/v6.12.3. You can download the file named **node-v6.12.3-x64.msi** and install it using the interactive installer.

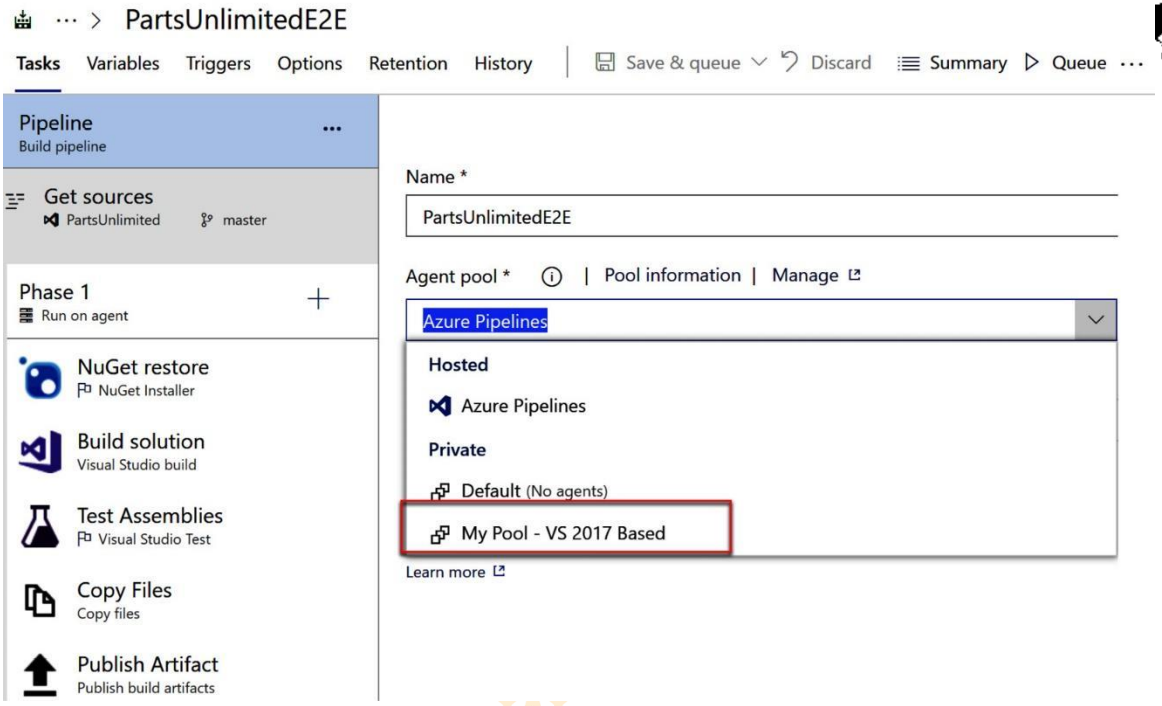
RUNNING THE AZURE PIPELINE

In this task, we'll now run the pipeline job to build the **PartsUnlimited** solution using our own self-hosted agents:

- Log in to the **Azure DevOps** portal and browse to the **PartsUnlimited** project.
- Browse to **Pipelines**:

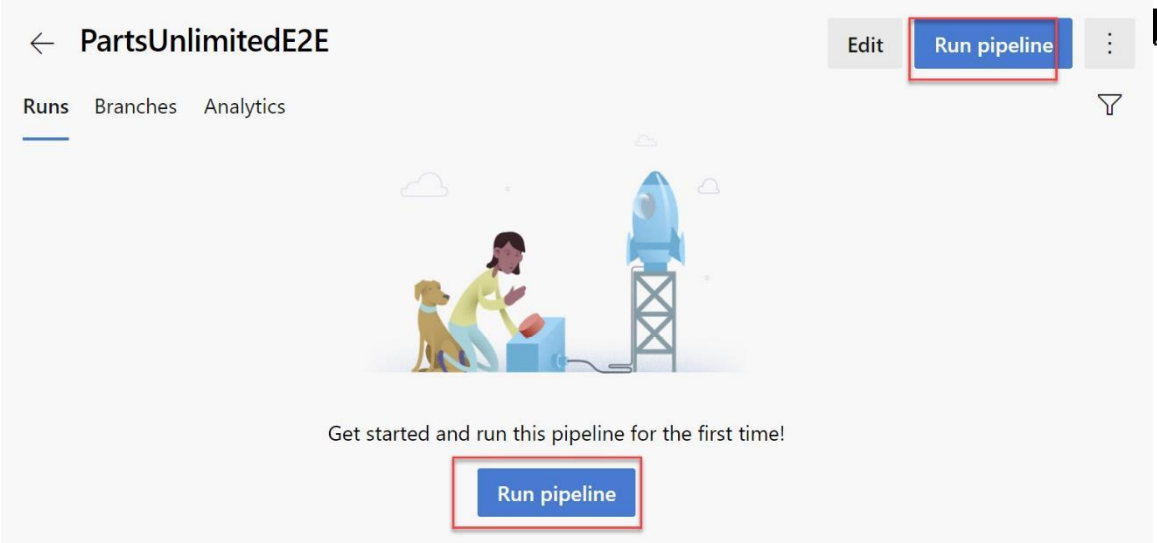


- Open the pre-created pipeline named **PartsUnlimitedE2E**.
- Click on **Edit Pipeline**.
- In the pipeline, change the **agent pool** to your newly created agent pool:



- Save the pipeline. You can also choose to run it after saving by selecting **Save & queue**:

- Now, we are ready to execute the pipeline. Click on **Run pipeline**:



- Under **Agent pool**, change the pool name to the agent pool you configured in the previous section and click **Run**:

Run pipeline

Select parameters below and manually run the pipeline

Agent pool

My Pool - VS 2017 Based

Branch/tag

master

Select the branch, commit, or tag

Advanced options

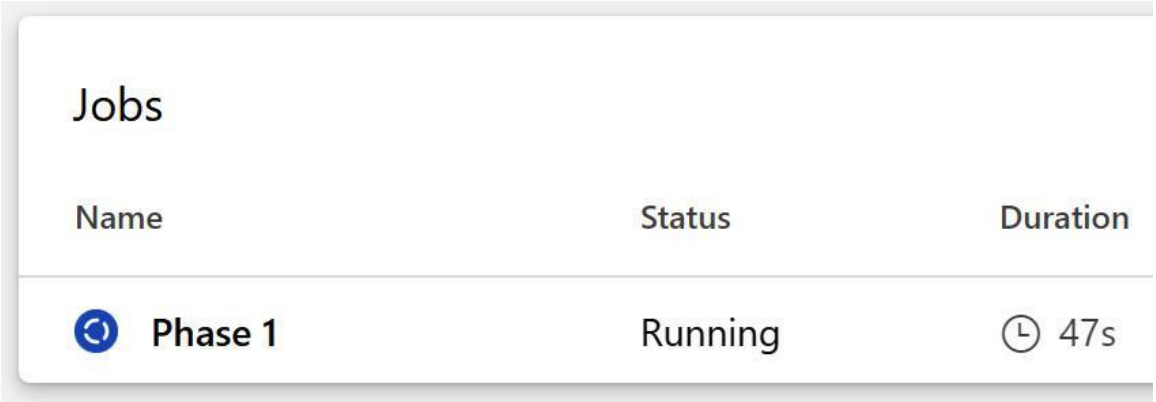
Variables
3 variables defined

Demands
This pipeline has no defined demands

Enable system diagnostics

Cancel Run

- This will start executing the pipeline job on your self-hosted agent. This may take a few minutes to complete:



- You can click on the job name to view the logs in real time.

Artifacts and Deployments

USING ARTIFACTS WITH AZURE DEVOPS

In the previous chapter, we covered how to host build agents in Azure Pipelines. In this chapter, we are going to cover how to use artifacts with Azure DevOps. We will begin by explaining what artifacts are. Then, we will look at how to create them in Azure DevOps, as well as how to produce the artifact package from a built pipeline. Next, we are going to cover how to deploy the feed using a release pipeline. Then, we are going to cover how to set the feed permissions and how to consume the package in Visual Studio.

The following topics will be covered in this chapter:

- Introducing Azure Artifacts
- Creating an artifact feed with Azure Artifacts
- Producing the package using a build pipeline
- Publishing the package to the feed from a build pipeline
- Configuring the feed permissions from the feed settings
- Consuming the package in Visual Studio from the Artifacts feed

INTRODUCING AZURE ARTIFACTS

It is likely that every developer has used a third-party or open source package in their code to add extra functionalities and speed up the development process of

their application. Using popular, pre-built components that have been used and tested by the community will help you get things done more easily.

Functionalities, scripts, and code that have been built by various teams in your organization are often reused by other teams and in different software development projects. These different artifacts can be moved into a library or package so that others can benefit from this.

There are different ways to build and host these packages. For instance, you can use NuGet for hosting and managing packages for the Microsoft Development platform or npm for JavaScript packages, Maven for Java, and more. Azure Artifacts offers features so that you can share and reuse packages easily. In Azure Artifacts, packages are stored in feeds. A feed is a container that allows you to group packages and control who has access to them.

You can store packages in feeds that have been created by yourself or other teams, but it also has built-in support for **upstream sources**. With upstream sources, you can create a single feed to store both the packages that your organization produces and the packages that are consumed from remote feeds, such as NuGet, npm, Maven, Chocolatey, RubyGems, and more.

It is highly recommended to use Azure Artifacts as the main source for publishing internal packages and remote feeds. This is because it allows you to keep a comprehensive overview of all the packages being used by the organization and different teams. The feed knows the provenance of all the packages that are saved using upstream resources; the packages are saved into the feed even when the original source goes down or the package is deleted. Packages are versioned, and you typically reference a package by specifying the version of the package that you want to use in your application.

Many packages allow for unrestricted access, without the need for users to sign in. However, there are packages that require us to authenticate by using a username and password combination or access token. Regarding the latter, access tokens can be set to expire after a given time period.

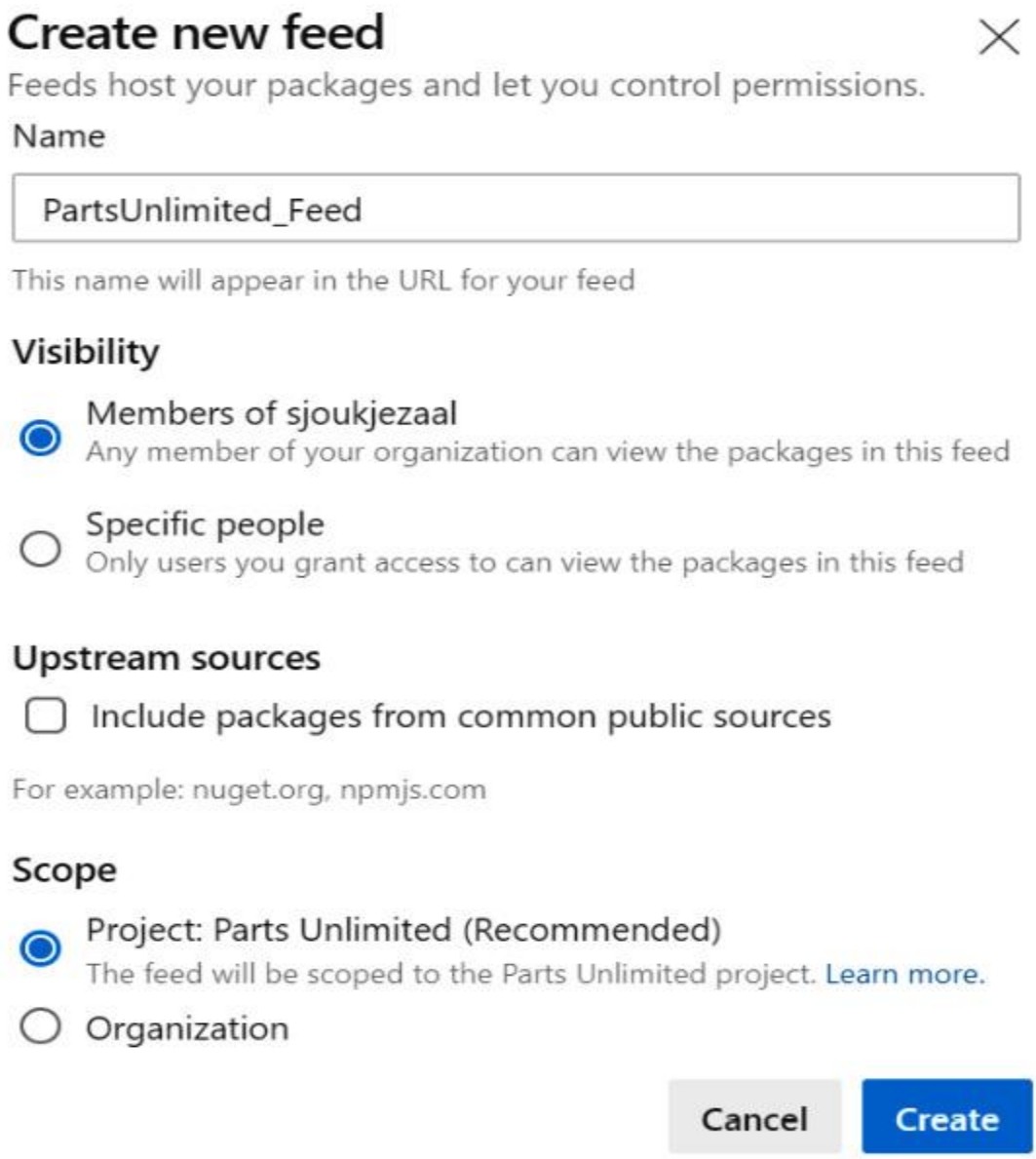
In the next section, we are going to look at how to create an **Artifact Feed** in Azure DevOps.

CREATING AN ARTIFACT FEED WITH AZURE ARTIFACTS

In this demo, we are going to create an artifact feed in Azure Artifacts. Packages are stored in feeds, which are basically organizational constructs that allow us to group packages and manage their permissions. Every package type (NuGet, npm, Maven, Python, and Universal) can be stored in a single feed.

For this demonstration, we are going to use our **PartsUnlimited** sample project again and add a new artifact feed to the project. To do this, perform the following steps:

- Open a web browser and navigate to https://dev.azure.com/.
- Log in with your Microsoft account and from the left menu, select **Artifacts**. Then, click the **+ Create Feed** button.
- In the **Create new feed** dialog box, add the following values (make sure that **Upstream sources** is disabled; we are not going to use packages from remote feeds in this chapter):



- Click the **Create** button.

With that, we have created a new feed so that we can store our packages. In the next section, we are going to produce a package using a build pipeline.

PRODUCING THE PACKAGE USING A BUILD PIPELINE

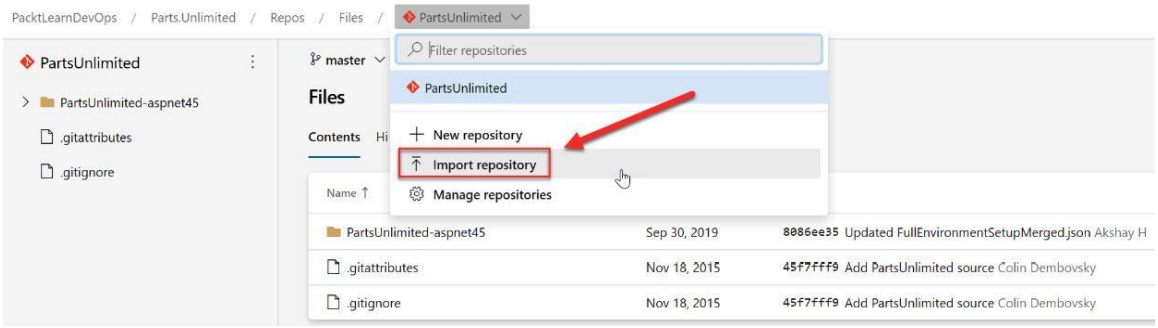
Now that we have created our feed, we are going to create a build pipeline that automatically creates a package during the build of the project. For this example, you can use the sample project provided in this book's GitHub repository. This sample project consists of all the models from the **PartsUnlimited** project. We are going to add all the models to a package and distribute it from Artifacts. This way, you can easily share the data model across different projects.

The first step is to import the GitHub repository into the **PartsUnlimited** organization in Azure DevOps.

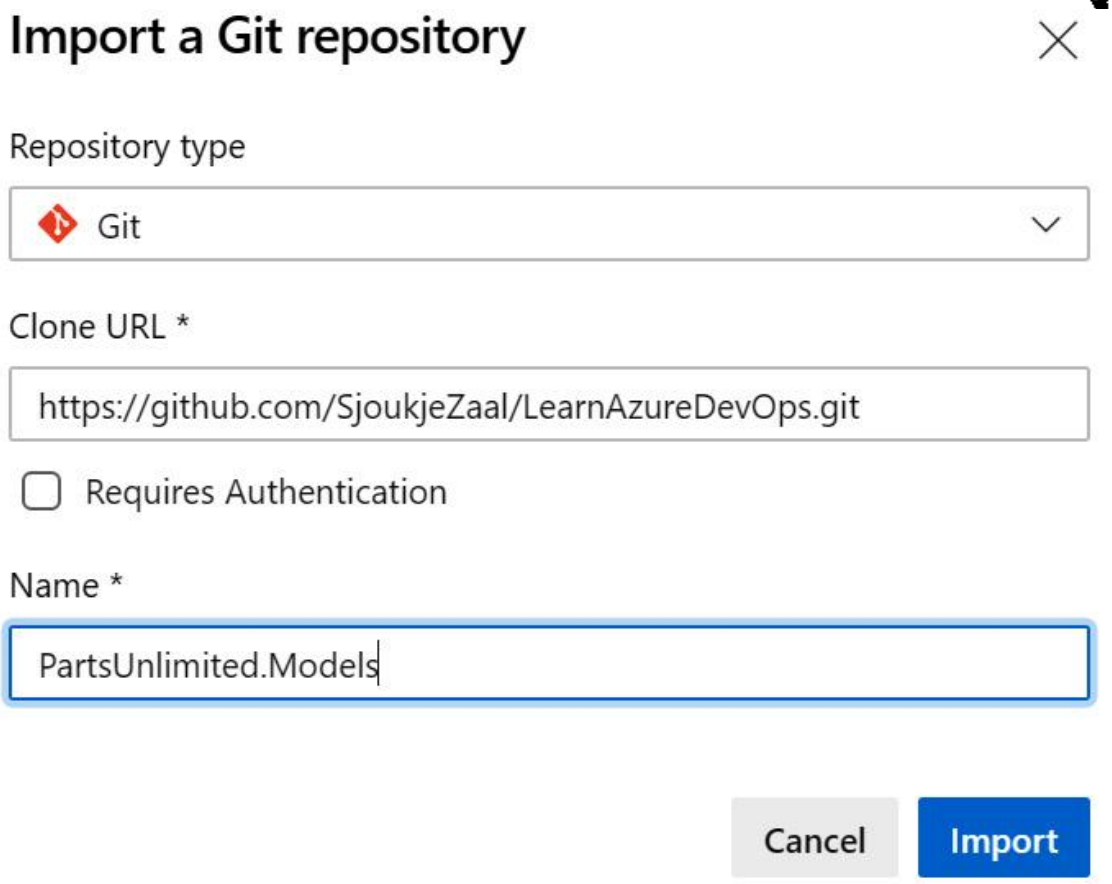
Adding the sample project to the PartsUnlimited repository

To add the sample models project to the PartsUnlimited repository, perform the following steps:

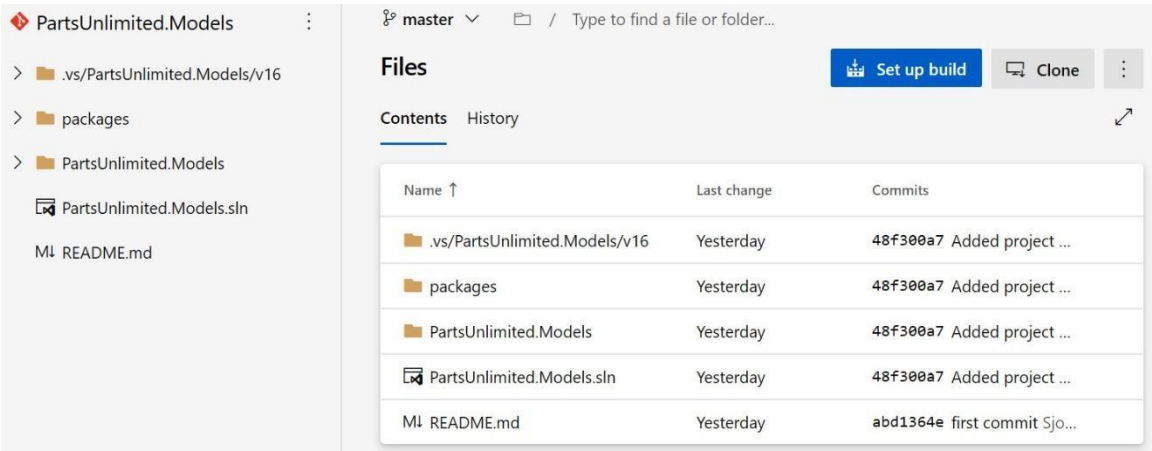
- Navigate to the **PartsUnlimited** project in Azure DevOps and go to **Repos > Files**.
- Select **Import repository** from the **PartsUnlimited** dropdown:



- Enter the URL of the source repository into the **Clone URL** box and add a name for your new GitHub repository:



- Click **Import**.
- Once the project has been imported, the repository will look as follows:

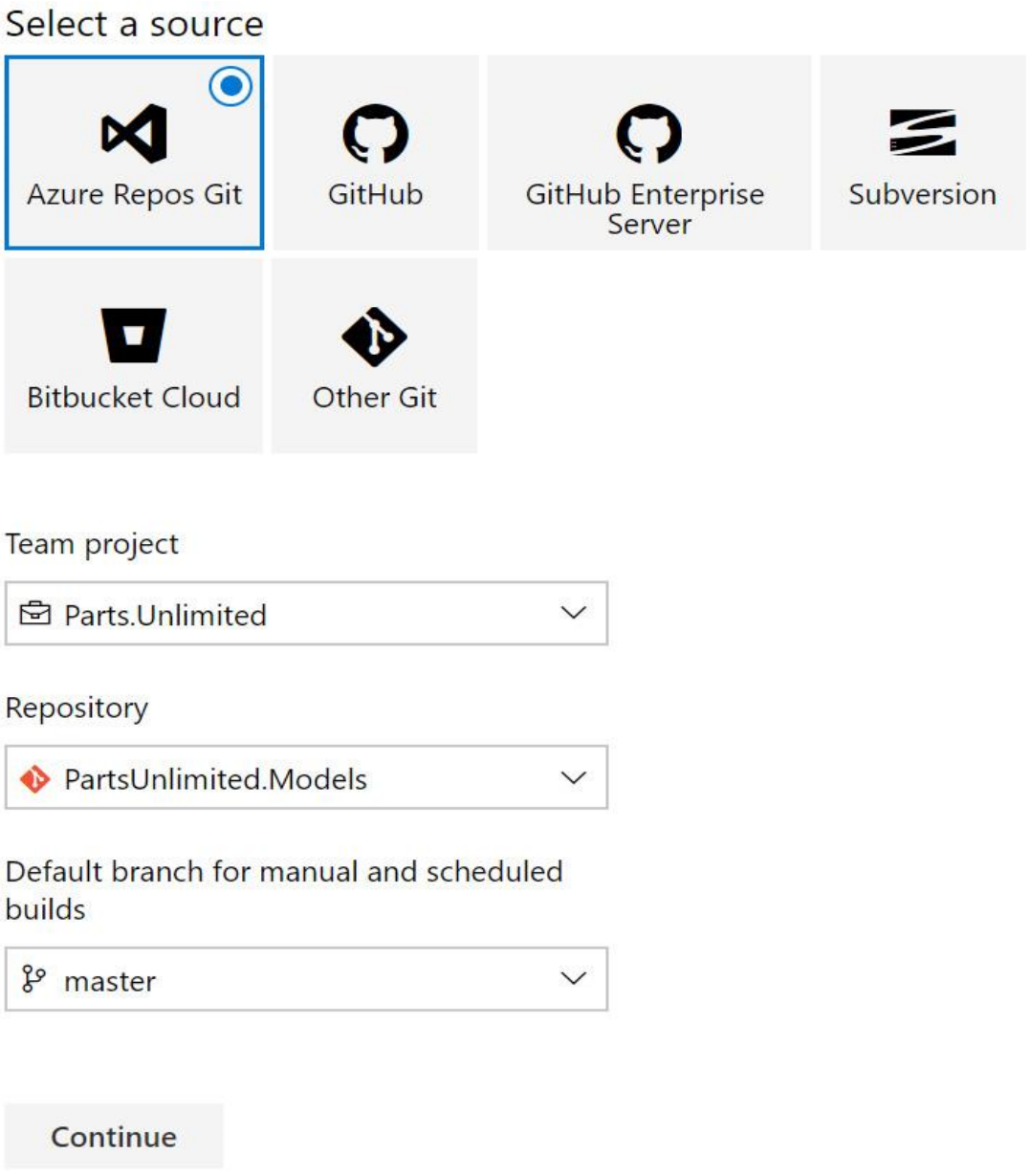


- Now that we have imported the **PartsUnlimited.Models** project into an Azure DevOps repository, we can use in a build pipeline to create a NuGet package of it.
- In the next section, we are going to create a build pipeline that will automatically package our project into an Artifact package.

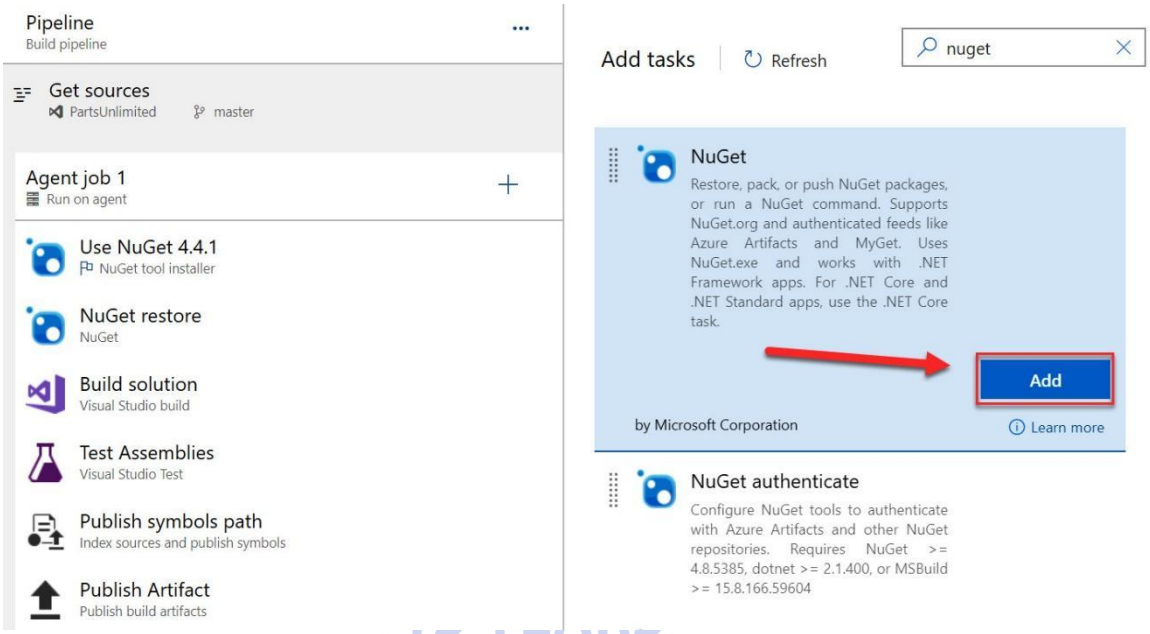
Creating the build pipeline

Now that the project has been added to the repository, we can create the build pipeline. To do this, perform the following steps:

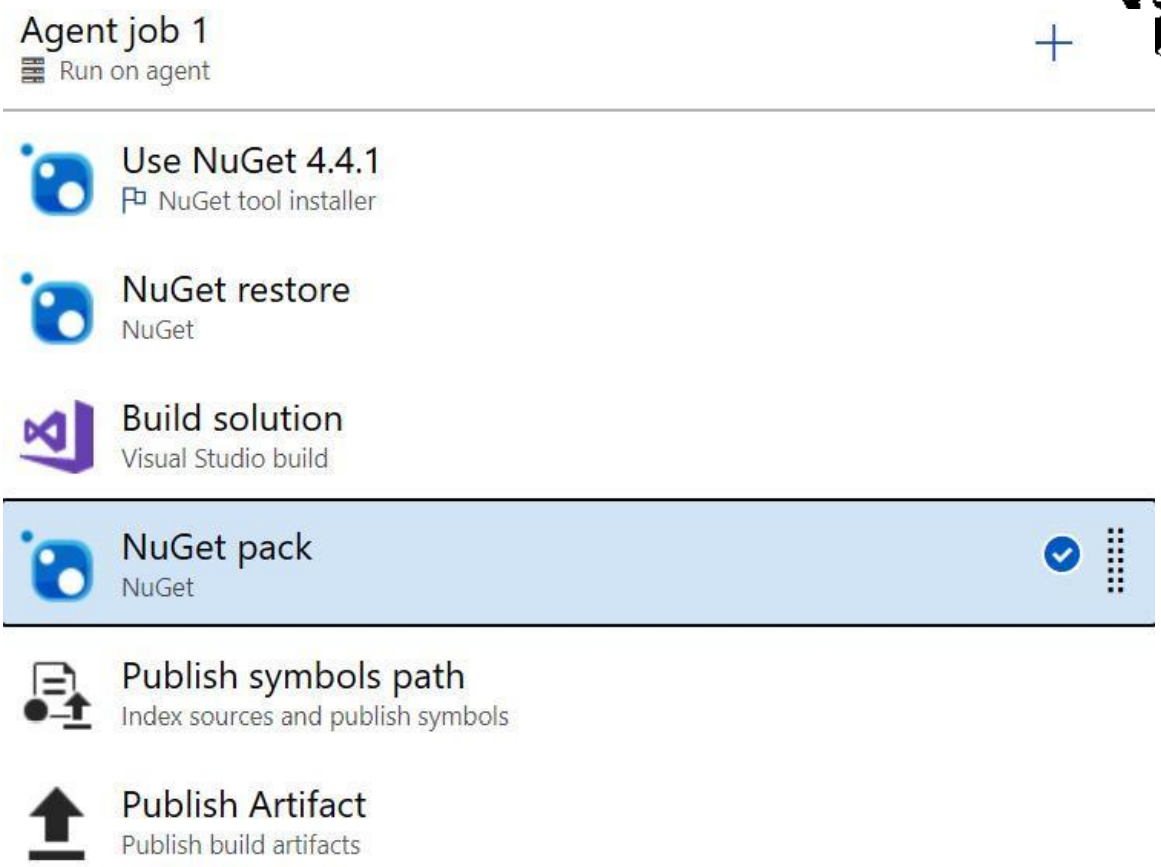
- Navigate to Azure DevOps and open the **PartsUnlimited.Models** project once more. From the left menu, click on **Pipelines**.
- Click on **New pipeline** from the top-right menu and select **Use the classic editor** on the first screen of the wizard.
- On the second screen, set the properties shown in the following screenshot and click **Continue**:



- Select **ASP.NET** on the next screen of the wizard and click **Apply**. With that, the Build pipeline will be created. Click on the **+** sign on the right-hand side of **Agent job 1** and search for **NuGet**.
- Add the NuGet task to the pipeline:



- Reorder the tasks and drag the NuGet task so that it's after the **Build Solution** task. Delete the **Test Assemblies** method since we don't have any tests in this project:



- Make the following changes to the settings of the newly added task:
  - **Display name**: **NuGet pack**
  - **Command**: **pack**
  - **Path to csproj or nuspec file(s) to pack**: ****/*.csproj**
- After making these changes, the task will look as follows:

- Next, let's set the versioning of the package. A recommended approach to versioning packages is to use **Semantic Versioning**. Expand the **Pack Options** section and add the following values to set up versioning:
  - **Automatic package versioning**: Use the date and time
  - **Major**: **1**
  - **Minor**: **0**
  - **Patch**: **0**
  - **Time zone**: **UTC**
- From the top menu, select **Save & queue** and then **Save and run.**
- The build pipeline will now run successfully. In the next section, we are going to publish the **PartsUnlimited.Models** NuGet package that we created in the first demo to our feed.

PUBLISHING THE PACKAGE TO THE FEED FROM A BUILD PIPELINE

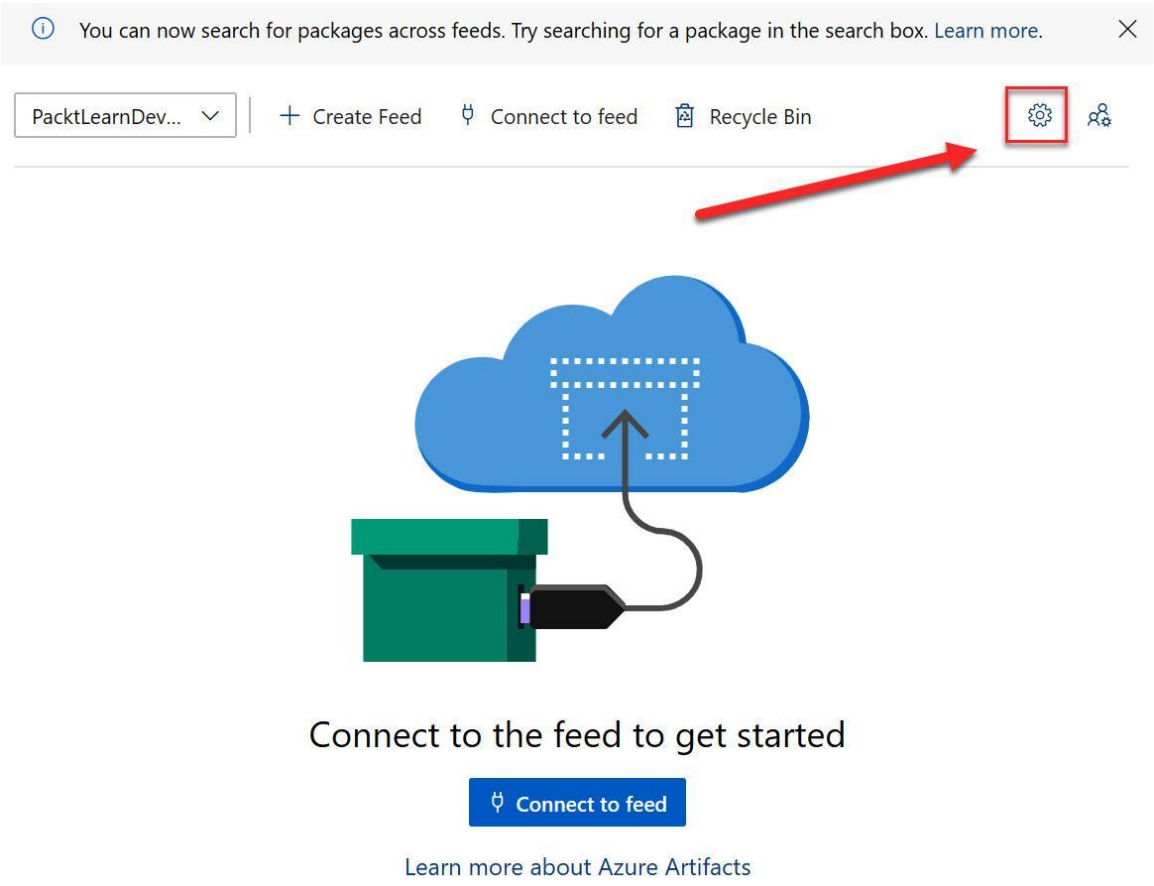
Now that we've built the application and the package from our build pipeline, . we can publish the package to the feed that we created in our first demo.

For this, we need to set the required permissions on the feed. The identity that the build will run under needs to have **Contributor** permissions on the feed. Once these permissions have been set, we can extend our pipeline to push the package to the feed.

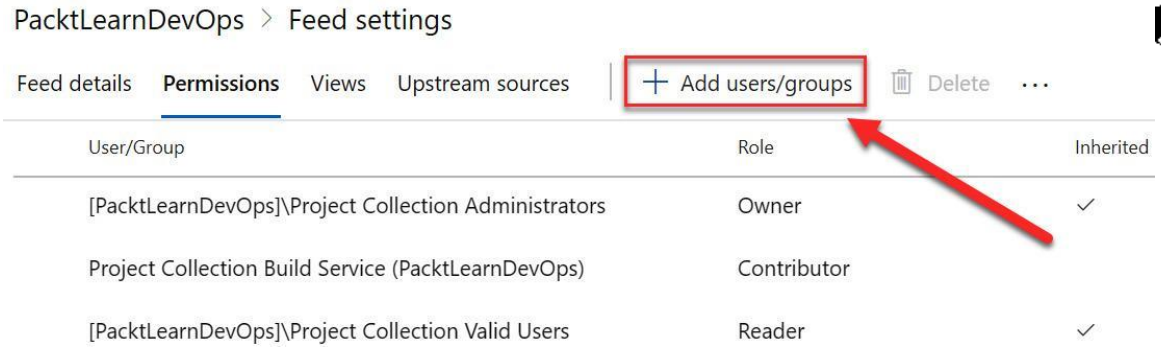
Setting the required permissions on the feed

To set the required permissions, we need to go to the settings of our feed:

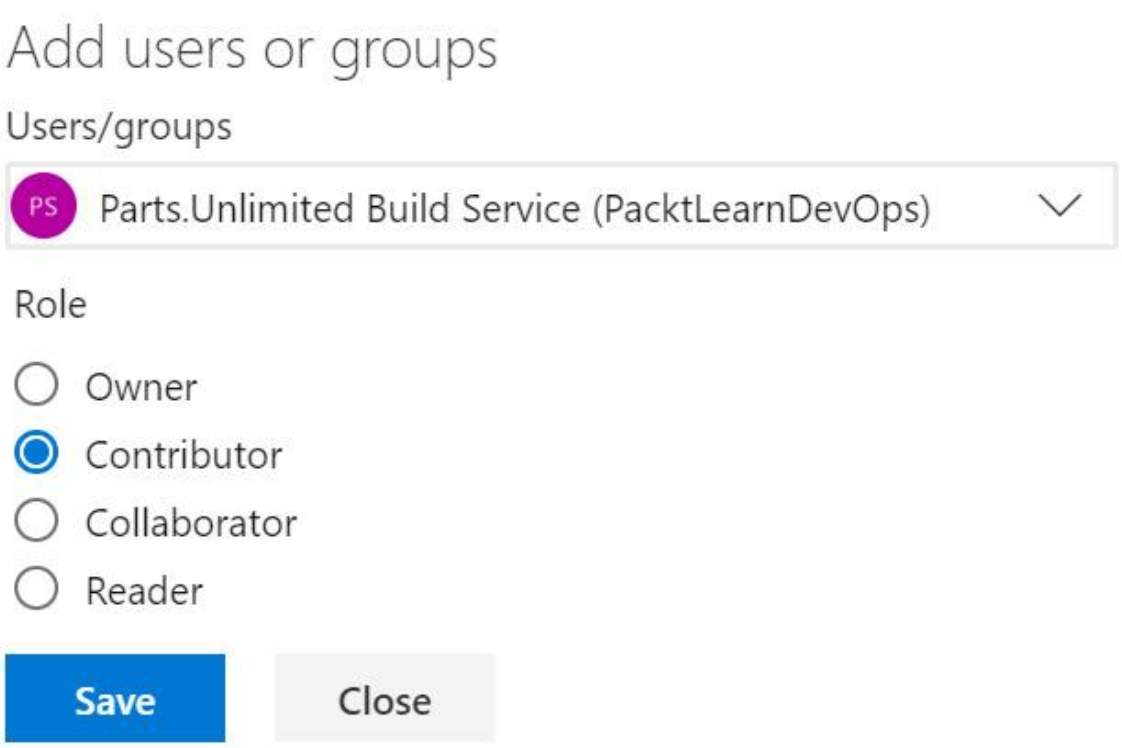
- Log in with your Microsoft account and from the left menu, select **Artifacts**.
- Go to the settings of the feed by selecting the **Settings** button from the top-right menu:



- Then, click on **Permissions** from the top menu and click on **+ Add users/groups**:



- Add the build that has the same name as the project, which in my case is the **Parts.Unlimited Build Service** identity:



- Click **Save**.

Now that the identity of the build pipeline has the required permissions on the feed, we can push the package to it during while it's being built.

Publishing the package

We are now ready to extend our build pipeline and push the package from it to the feed. To do this, we need to perform the following steps:

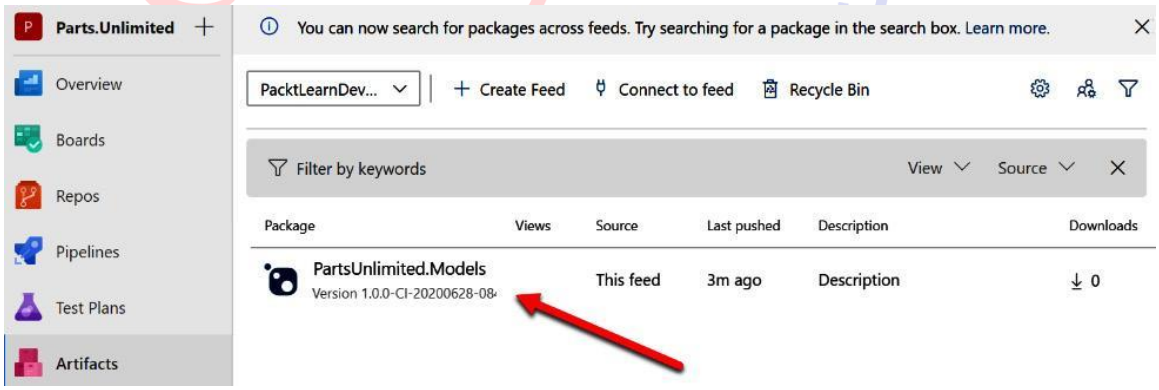
- Navigate to Azure DevOps and open the **PartsUnlimited.Models** project. Click on **Pipelines** in the left menu.
- Select the build pipeline that we created in the previous step and click on the **Edit** button, which can be found in the top-right menu.
- Click on the **+** button again next to **Agent job 1** and search for NuGet. Add the task to the pipeline.
- Drag the newly added task below the NuGet task that we created in the previous step. Make the following changes to the settings of the task:

--Display name: NuGet push
--Command: push
--Path to NuGet package(s) to publish:
$(Build.ArtifactStagingDirectory)/**/*.nupkg;!$(Build.ArtifactStagingDirectory)/**/*.symbols.nupkg
--Target feed location: This organization/collection
--Target feed: LearnDevOps

After making these changes, the task will look as follows:

From the top menu, select **Save & queue** and then **Save and run.** Wait until the build pipeline has finished successfully.

Finally, let's check whether the package has been successfully published. Click on **Artifacts** from the left menu. You will see that the package has been pushed to the feed:



Deploying Applications with Azure DevOps

we saw how you can automate your development processes by using build pipelines for your code. But an important part of the software life cycle is also the release phase. In this chapter, we will cover an overview of release pipelines; we'll

see how to create a release pipeline with Azure DevOps and how you can automate and improve the deployment of your solutions by using release approvals and multi-stage pipelines.

AN OVERVIEW OF RELEASE PIPELINES

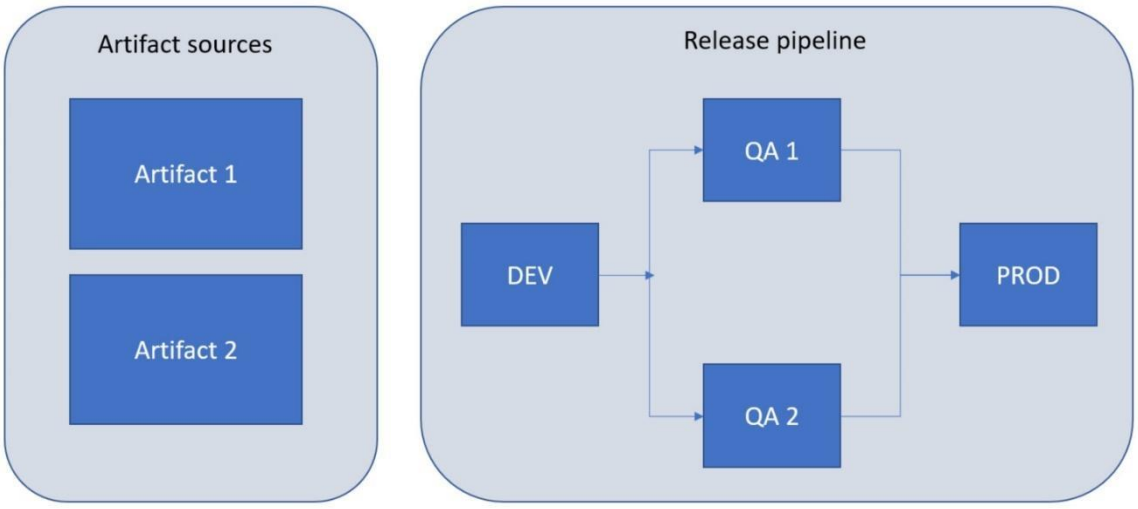
Release pipelines permit you to implement the continuous delivery phase of a software life cycle. With a release pipeline, you can automate the process of testing and deliver your solutions (committed code) to the final environments or directly to the customer's site (continuous delivery and continuous deployment).

With **continuous delivery**, you deliver code to a certain environment for testing or quality control, while **continuous deployment** is the phase where you release code to a final production environment.

A release pipeline can be triggered manually (you decide when you want to deploy your code) or it can be triggered according to events such as a code commit on the master branch, after the completion of a stage (for example, the production testing stage), or according to a schedule.

A release pipeline is normally connected to an **artifact store** (a deployable component for an application and output of a build). An artifact store contains a set of artifacts for a build (distinct artifact versions), and a release pipeline takes these artifacts and provisions the needed infrastructure and steps for deploying the artifacts.

A schema of a release pipeline is as follows:



As you can see in the preceding diagram, a release pipeline starts from artifacts (the output of a successfully completed build) and then moves between stages, executing jobs and tasks.

In Azure DevOps, a release pipeline is executed according to the following steps:

- When a deployment request is triggered, Azure Pipelines checks whether a pre-deployment approval phase is required and eventually sends approval notifications to the involved people in a team.
- When approved, the deployment job is queued and waits for an agent.
- An agent that is able to run this deployment job picks up the job.
- The agent downloads the artifacts as specified in the release pipeline definition.
- The agent runs the tasks defined in the deployment job and creates a log for each step.
- When the deployment for a stage is completed, Azure Pipelines executes a post-deployment approval (if present).
- The deployment then goes to the next stage.

In a release pipeline, an artifact is deployed to an **environment** (where your final application will run), and these environments can be the following:

- A machine on your corporate network
- A virtual machine in the cloud
- A containerized environment, such as Docker or Kubernetes
- A managed service, such as Azure App Service
- A serverless environment, such as Azure Functions

A way to define an Azure Pipelines environment is with a YAML file, where you can include an environment section that specifies the Azure Pipelines environment where you'll deploy your artifact, or by using the classic UI-based editor

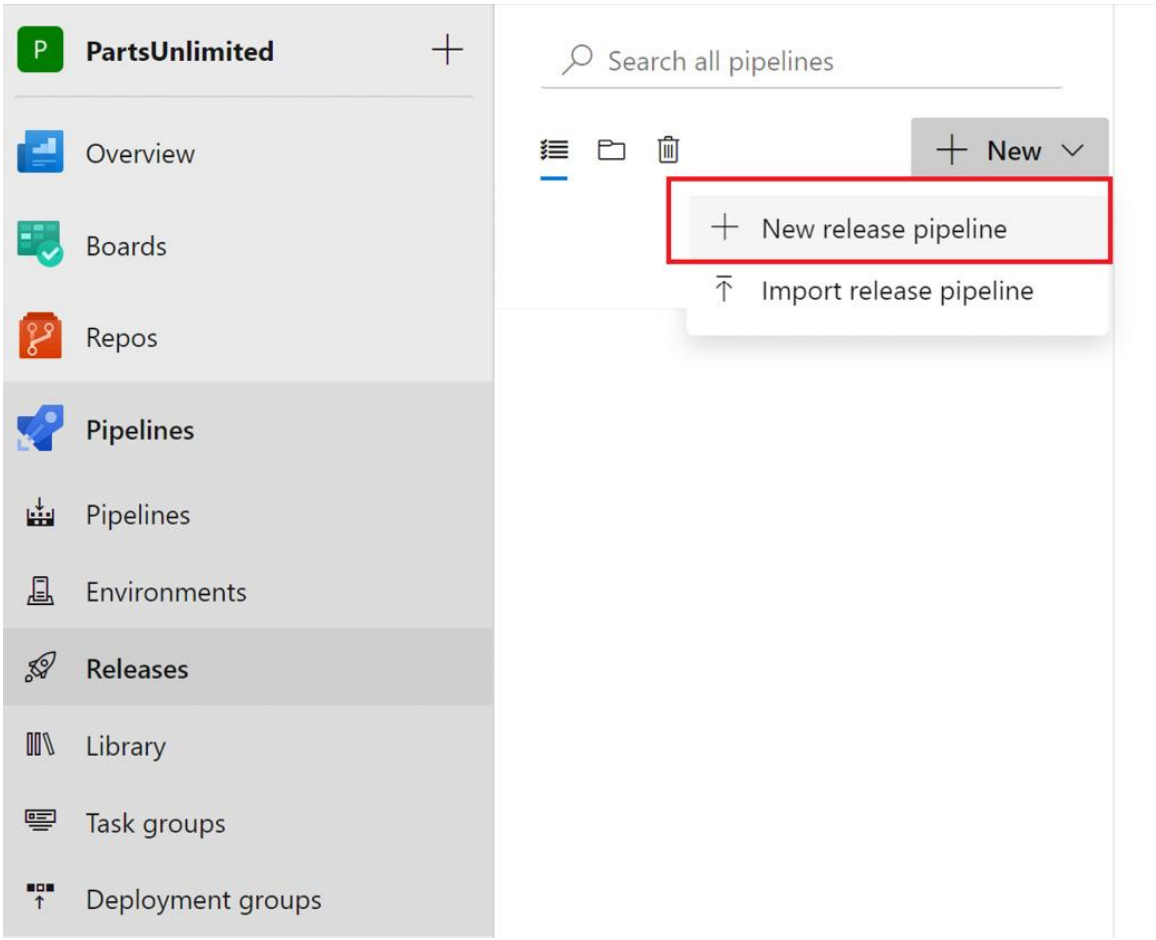
CREATING A RELEASE PIPELINE WITH AZURE DEVOPS

The final goal for implementing a complete CI/CD process with DevOps is to automate the deployment of your software to a final environment (for example, the final customer), and to achieve this goal, you need to create a **release pipeline**.

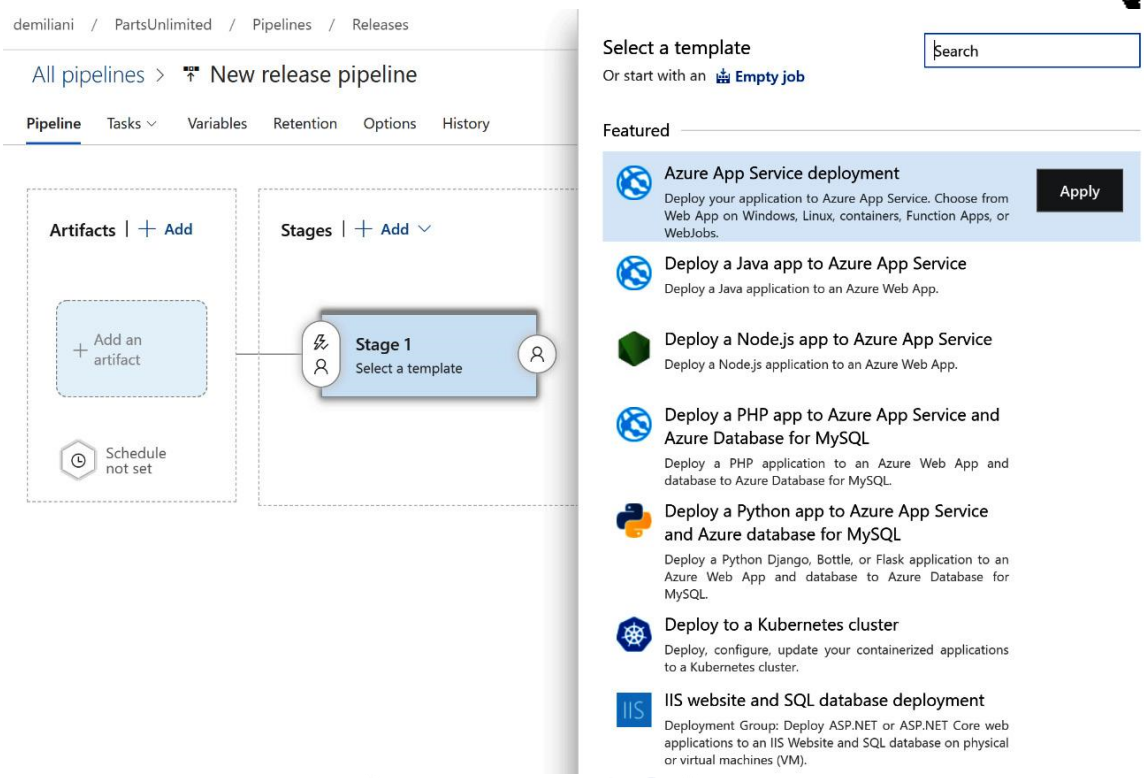
A release pipeline takes the build artifacts (the result of your build process) and deploys those artifacts to one or more final environments.

To create our first release pipeline, we'll use the **PartsUnlimited** web application project previously deployed on Azure DevOps:

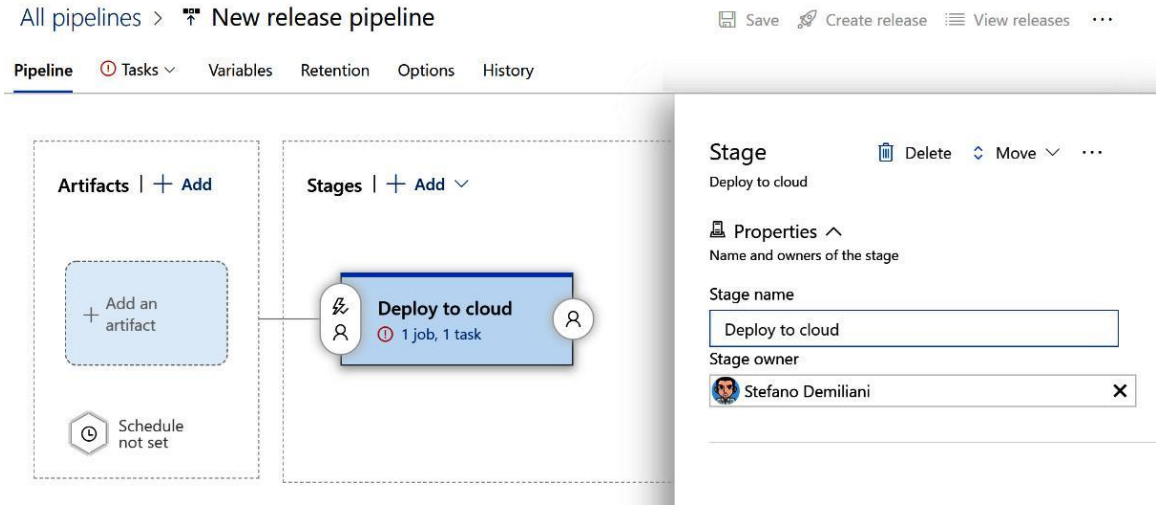
- To create a release pipeline with Azure DevOps, click on **Pipelines** on the left menu, select **Releases**, and then click on **New release pipeline**:



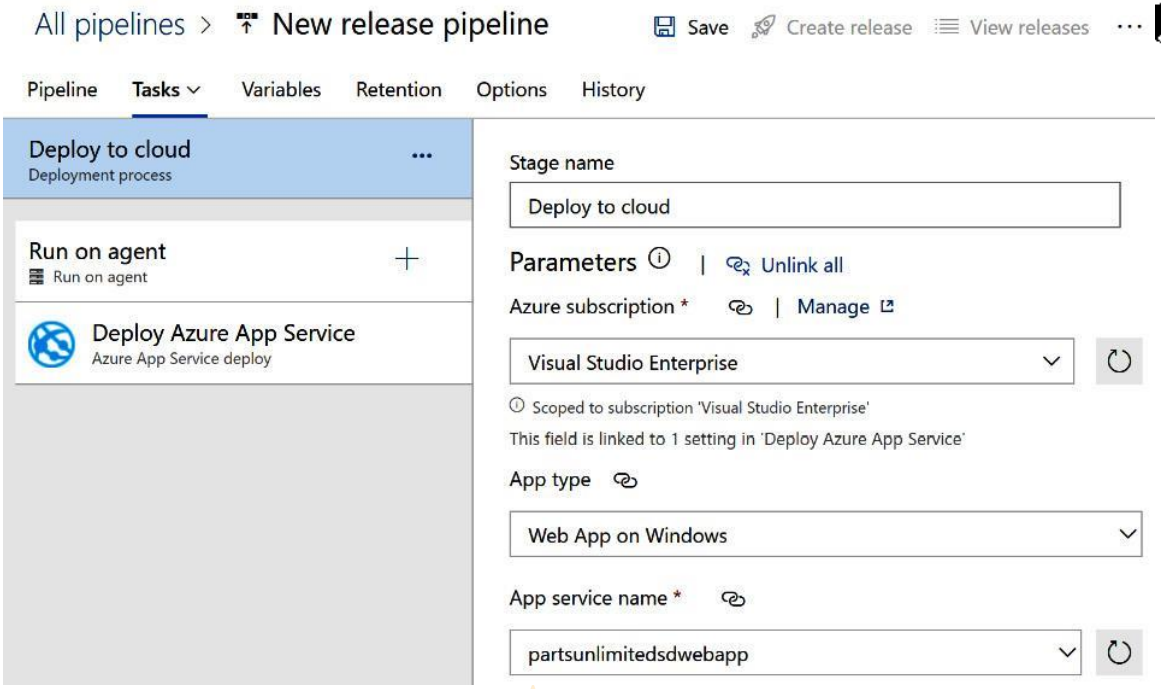
- In the **Select a template** list that appears on the right, you have a set of available templates for creating releases for different types of applications and platforms. For our application, select **Azure App Service deployment** and click **Apply**:



- Now, provide a name for the stage that will contain the release tasks. Here, I'm calling it **Deploy to cloud**:



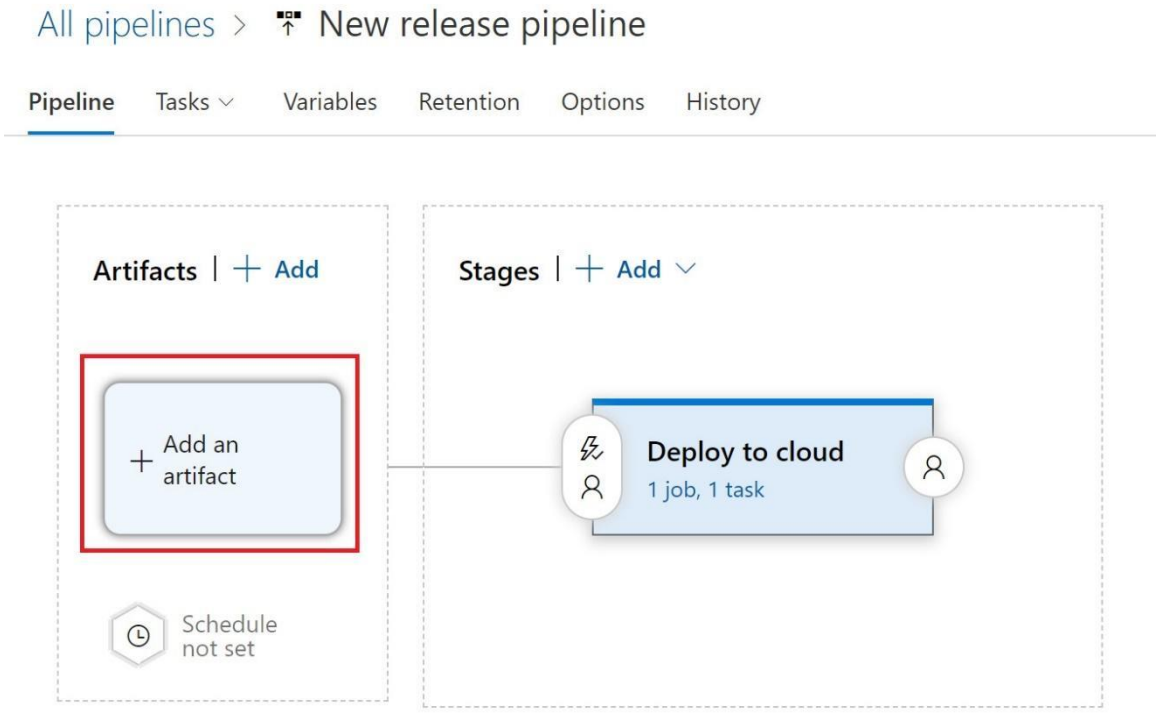
- In the **Stages** section, click on the **1 job, 1 task** link. Here, you need to provide the settings of the Azure web app environment where your application will be deployed, such as your Azure subscription and the App Service instance (web app) where the code will be deployed:



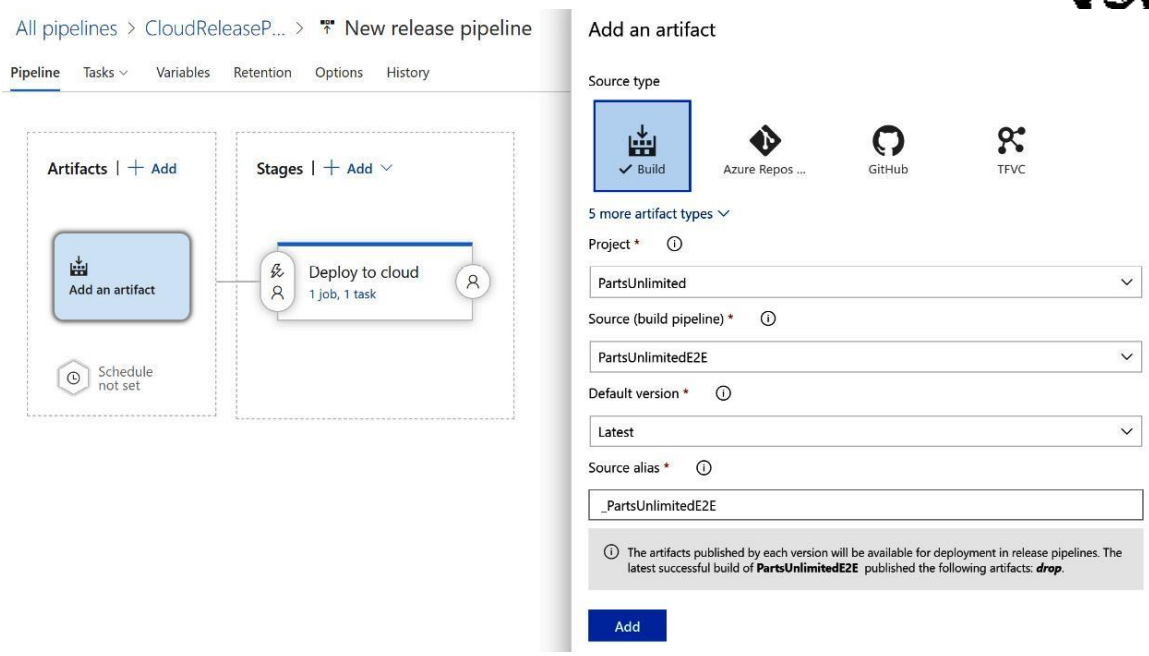
- You have now defined the stage of your release pipeline (single-stage). In the next section, we'll see how to specify the artifacts for your release pipeline.
- Defining artifacts for a release pipeline

Artifacts are all the items (output of a build) that must be deployed in your final environment, and Azure Pipelines can deploy artifacts that come from different artifact sources:

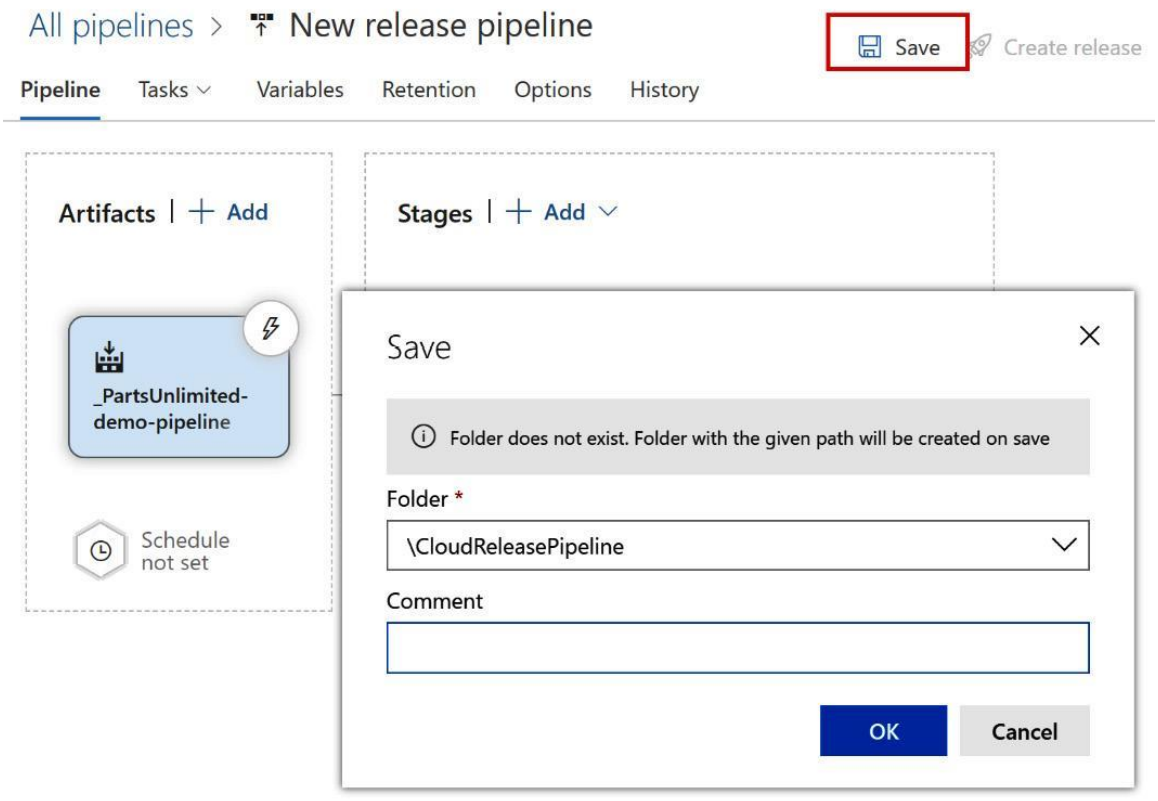
- To select artifacts, on the main release pipeline screen, click on **Add an artifact**:



- In the **Add an artifact** panel, you have **Source type** automatically set to **Build** (this means that you're deploying the output of a build pipeline). Here, you need to select the build pipeline that you want to use as the source (the name or ID of the build pipeline that publishes the artifact; here, I'm using the **PartsUnlimitedE2E** build pipeline) and the default version (the default version will be deployed when new releases are created. The version can be changed for manually created releases at the time of release creation):



- Click on the **Add** button to save the artifact configuration, and then click on the **Save** button in the top-right corner to save your release pipeline:

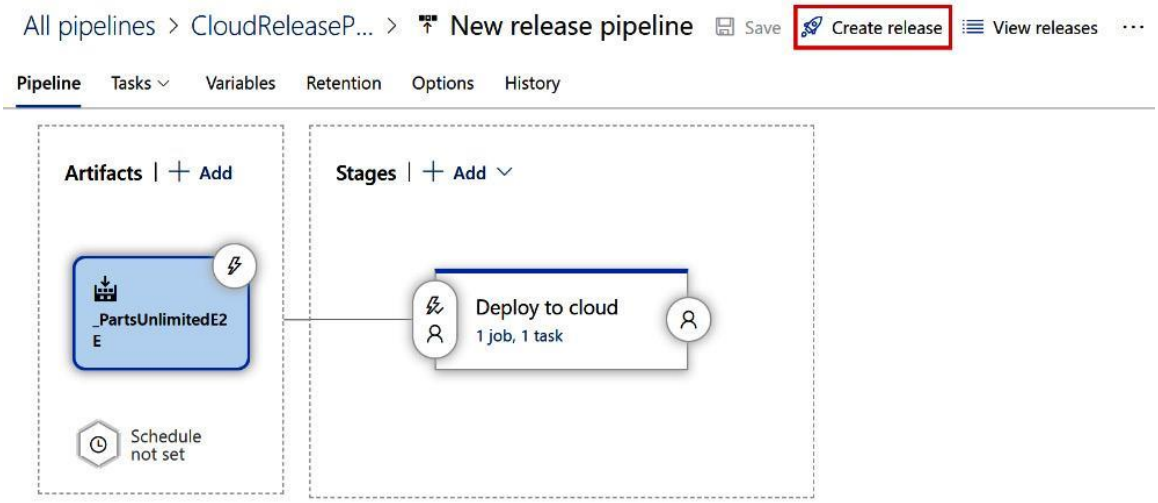


- Your release pipeline is now ready. In the next section, we'll see how to create the Azure DevOps release process.

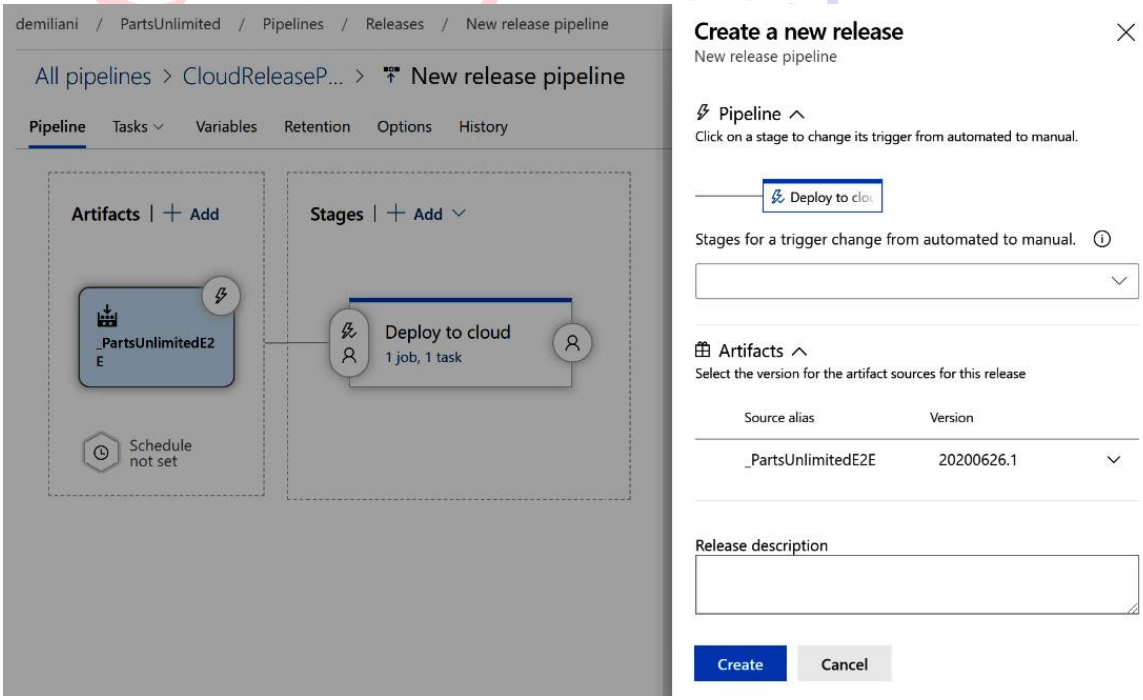
Creating the Azure DevOps release

After defining our release pipeline (stages and artifacts), we need to create a . **release**. A release is simply a run of your release pipeline:

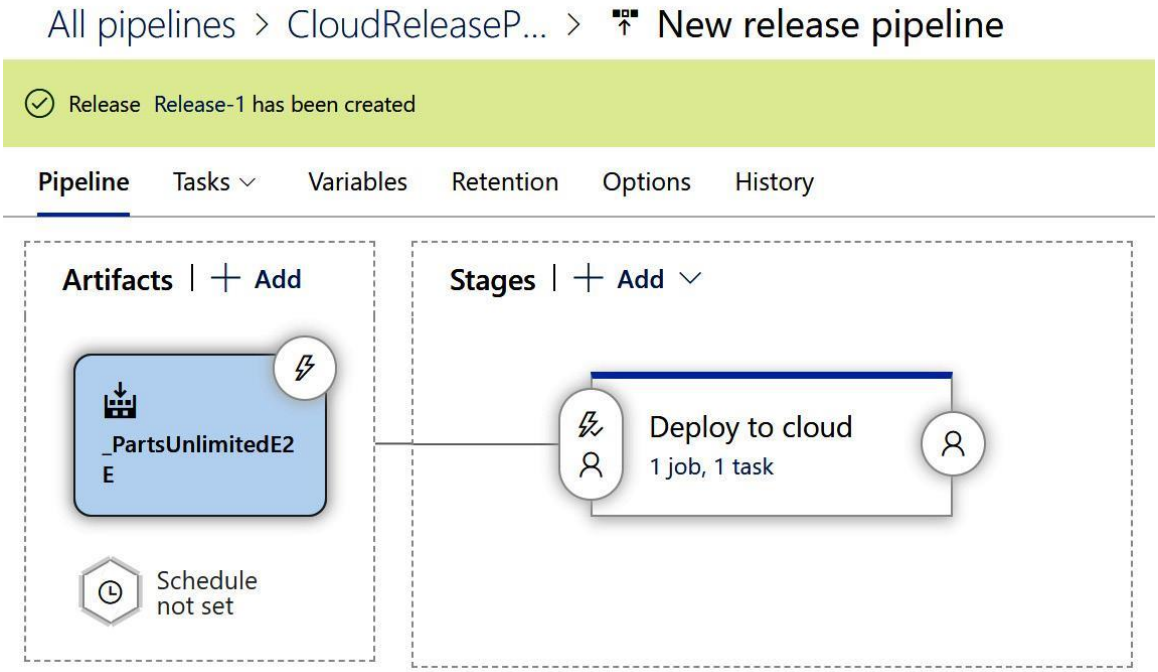
- To create a release, on the release pipeline definition page, click on the **Create release** button in the top-right corner:



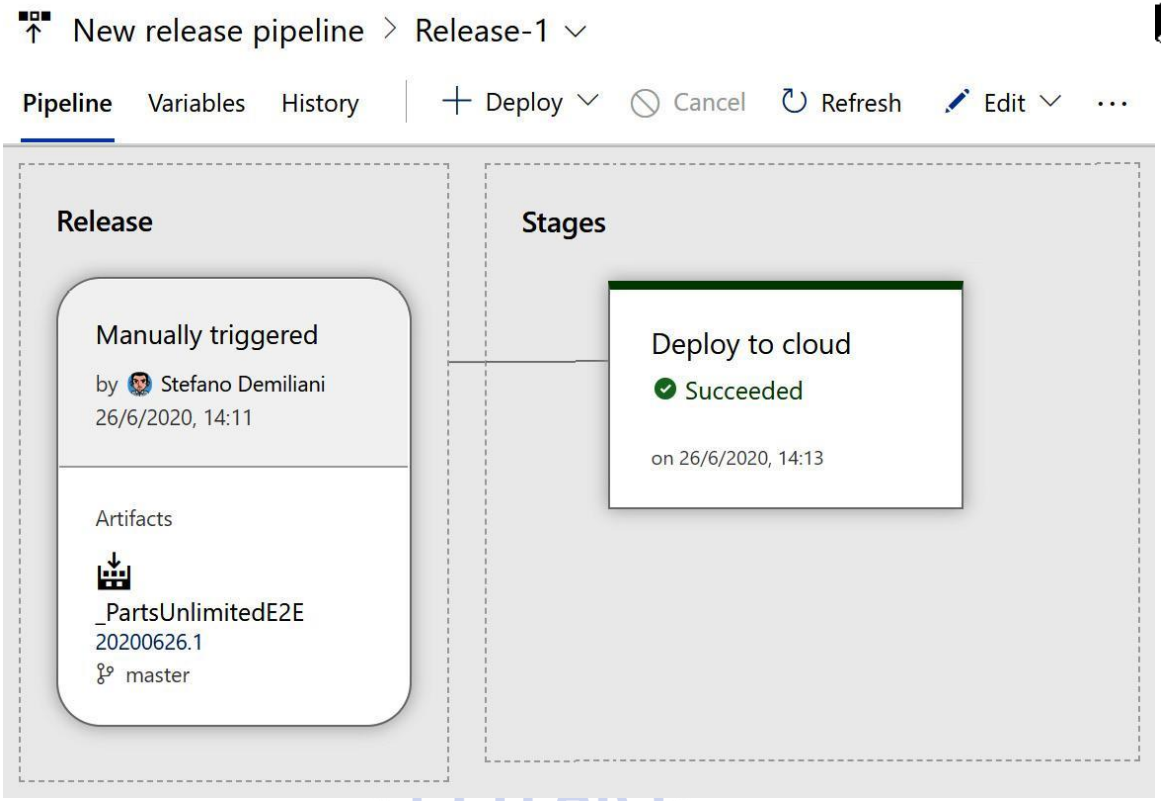
- On the **Create a new release** page, accept all the default values (you need to have a successfully completed build with artifacts created), and then click on **Create**:



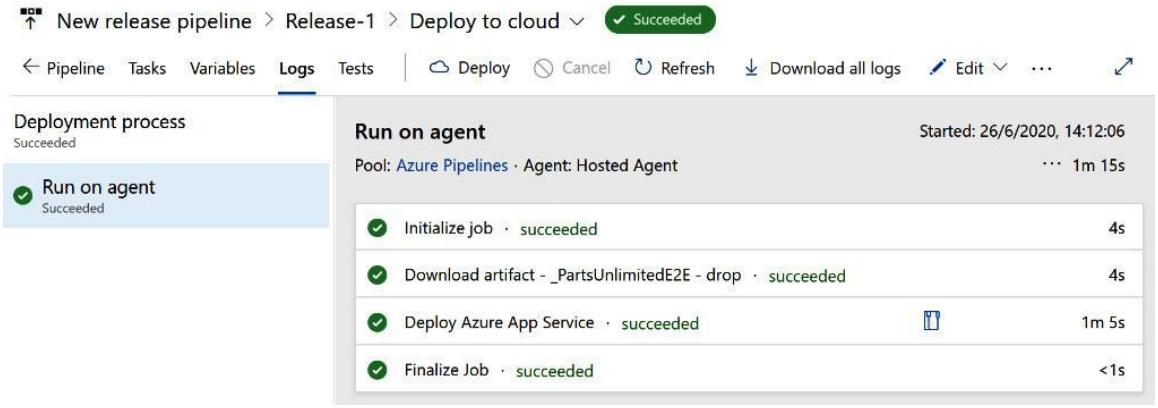
- A new release is now created, and you will see a green bar indicating that:



- Now, you can click on the release name (here, it is **Release-1**) and you will be redirected to the details of the release process:



- If you click on the stage, you can see the details of each step:

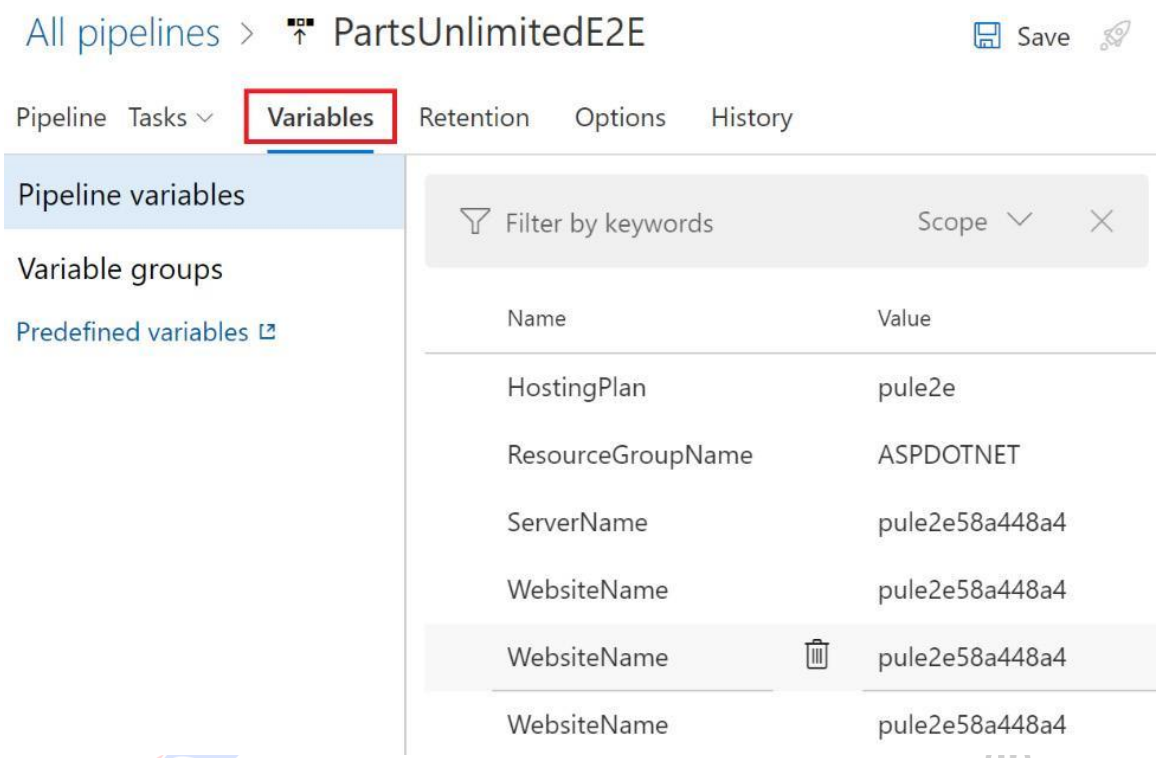


- You have completed your first release pipeline. Here, we have triggered it manually. In the next section, we'll see how to use variables in your pipeline.

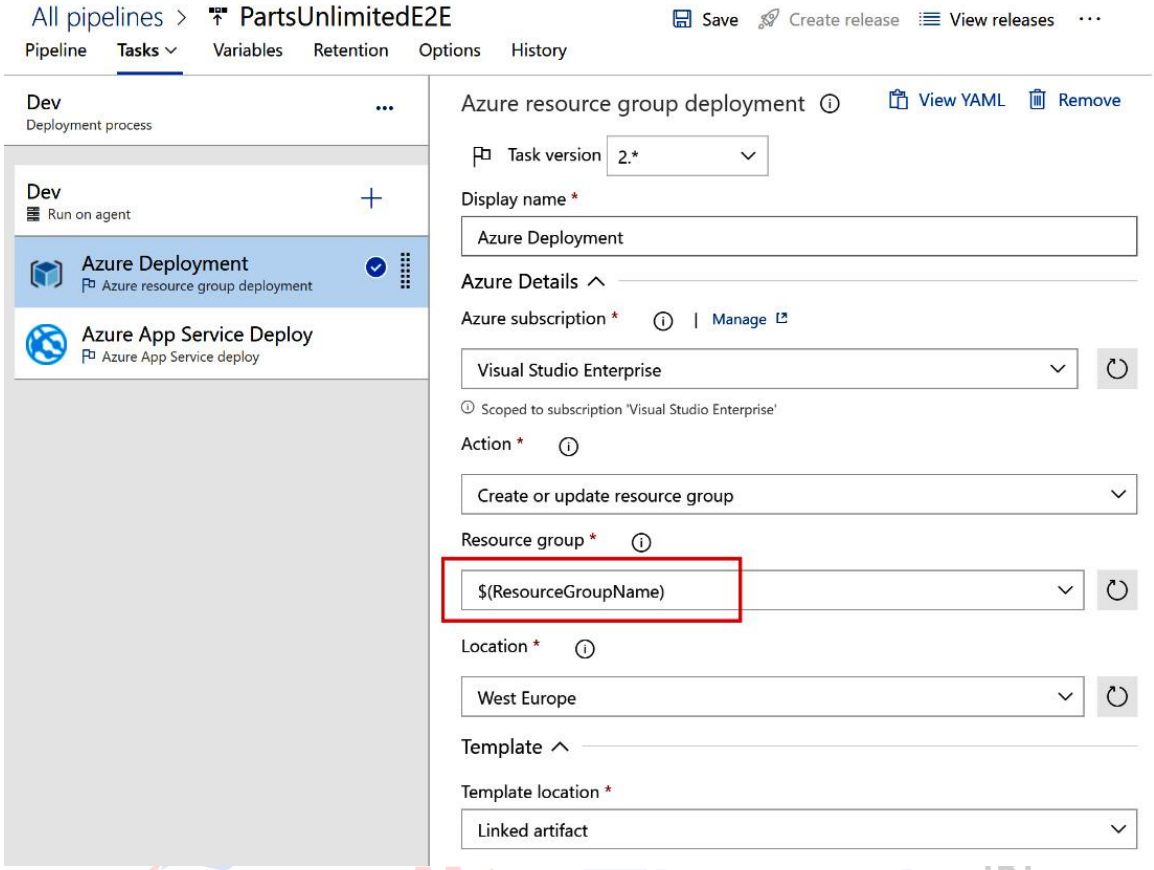
Using variables in a release pipeline

In a release pipeline, you can also use variables and variable groups to specify variable parameters that can be used in your pipeline tasks. To specify a variable

for your release pipeline, select the **Variables** tab and specify the name and value of your variable:



You can then use the variables in your pipeline's tasks by using the **$(VariableName)** notation, as in the following screenshot:

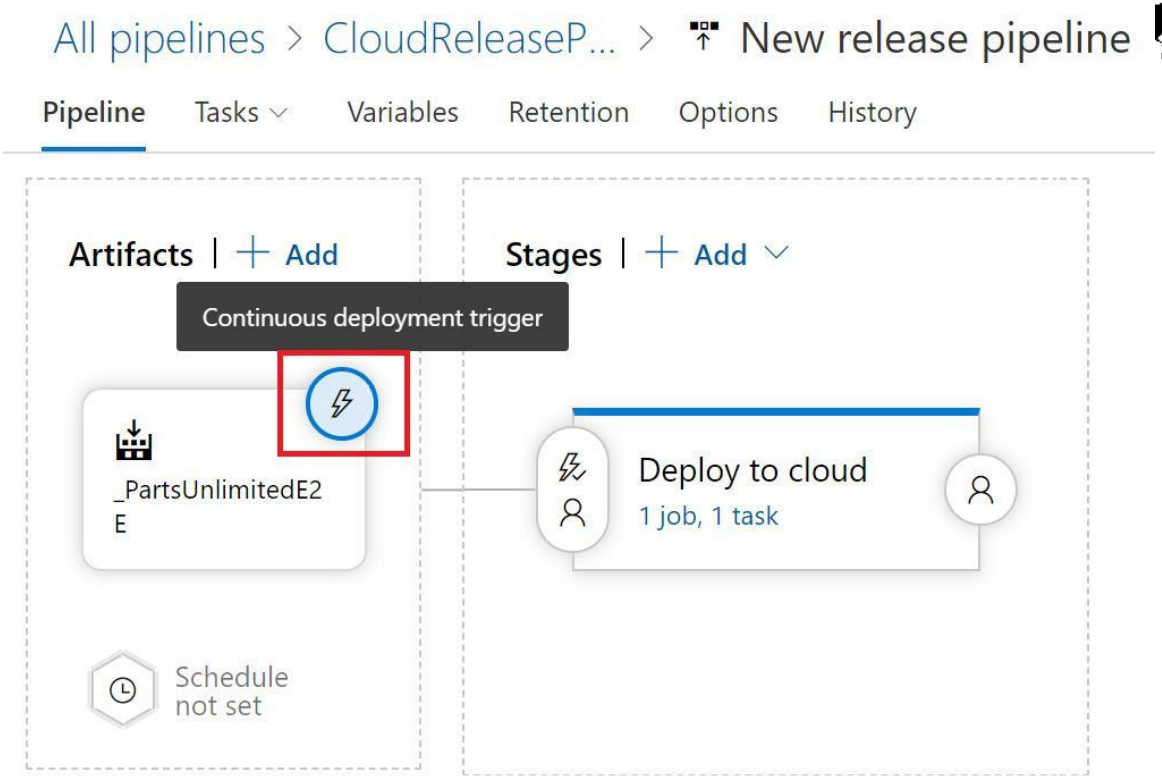


Using variables is recommended if you have parameters that change on your pipeline. In the next section, we'll see how to configure triggers for continuous deployment.

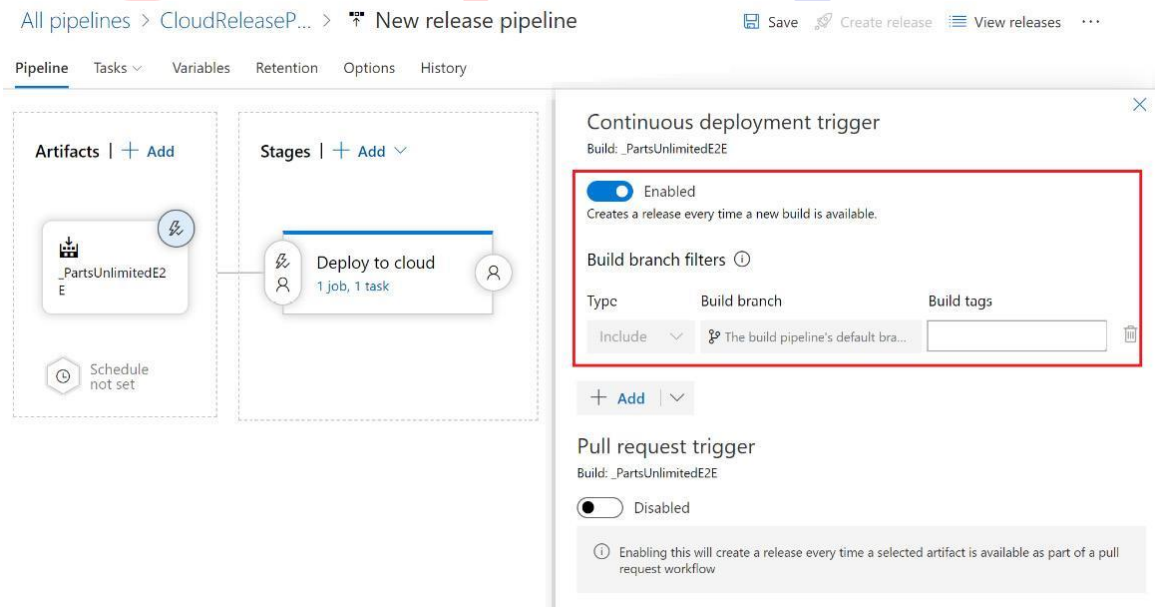
Configuring the release pipeline triggers for continuous deployment

To automate the continuous deployment of your application, you need to configure triggers in your release pipeline definition:

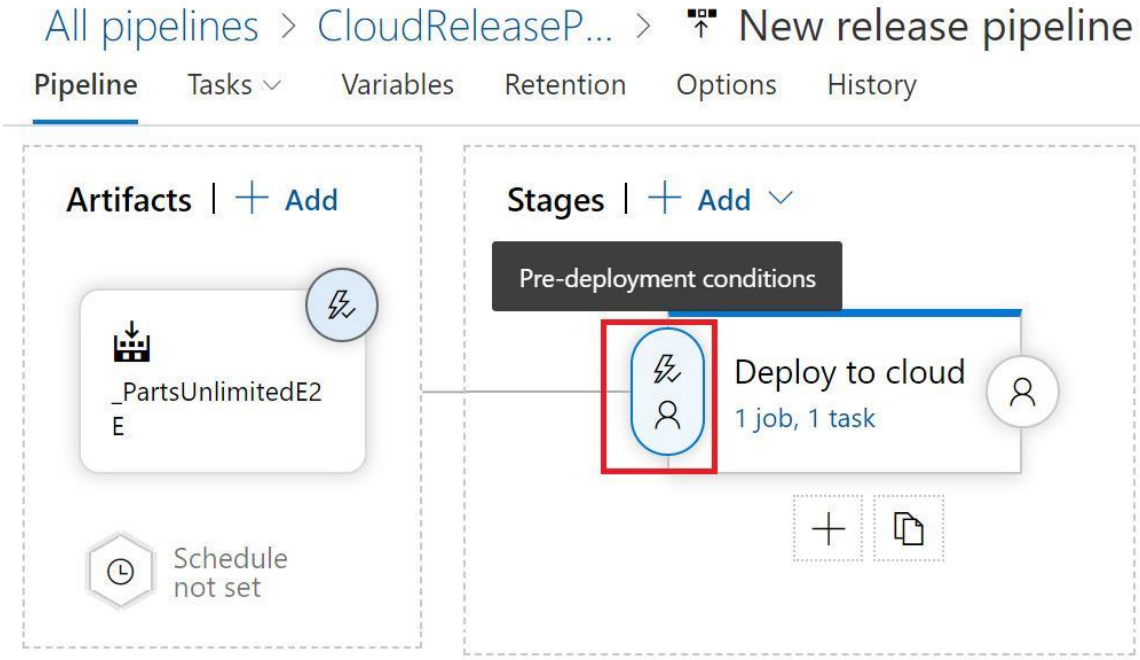
- To do that, click on the **Continuous deployment trigger** icon in the pipeline's **Artifacts** section:



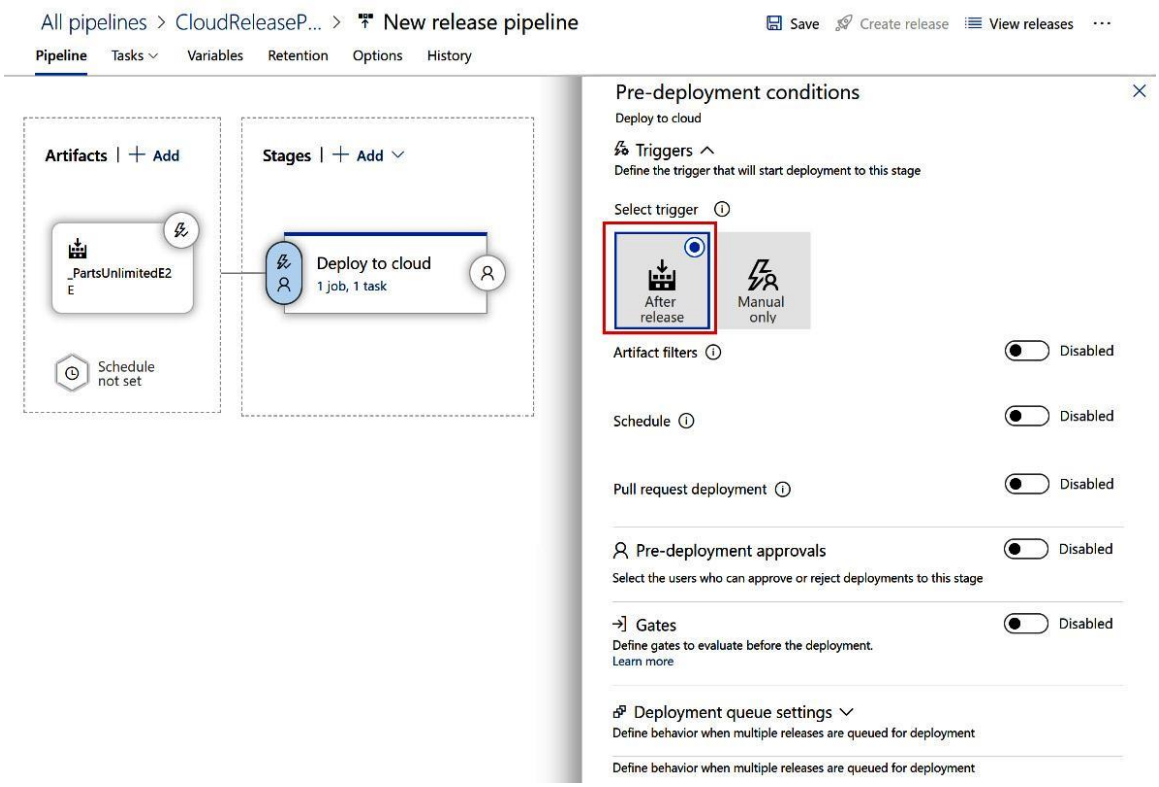
- In the **Continuous deployment trigger** panel, enable it to automatically create a new release after every successfully completed build and select a branch filter (for example, the build pipeline's default branch):



- Now, in the **Stages** section, select the **Pre-deployment conditions** icon:



- In the **Pre-deployment conditions** pane, check that the trigger for this stage is set to **After release** (this means that the deployment stage will start automatically when a new release is created from this pipeline):



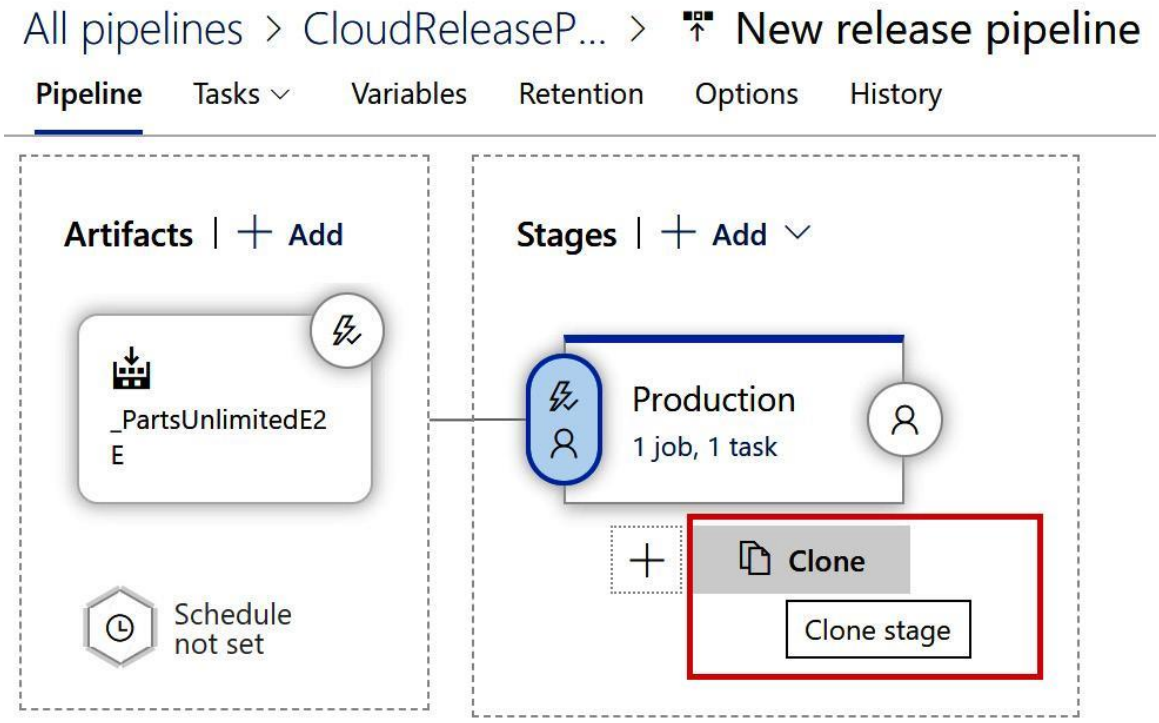
- In this pane, you can also define other parameters, such as selecting artifact condition(s) to trigger a new deployment (a release will be deployed to this stage only if all the artifact conditions match), setting up a schedule for the deployment, allowing pull request-based releases to be deployed to this stage, selecting the users who can approve or reject deployments to this stage (pre-deployment approvals), defining gates to evaluate before deployment, and defining behavior when multiple releases are queued for deployment.
- You have now created a release pipeline that takes your artifacts and deploys them to the cloud by using Azure DevOps and also by applying continuous deployment triggers and pre-deployment conditions checks.

Creating a multi-stage release pipeline

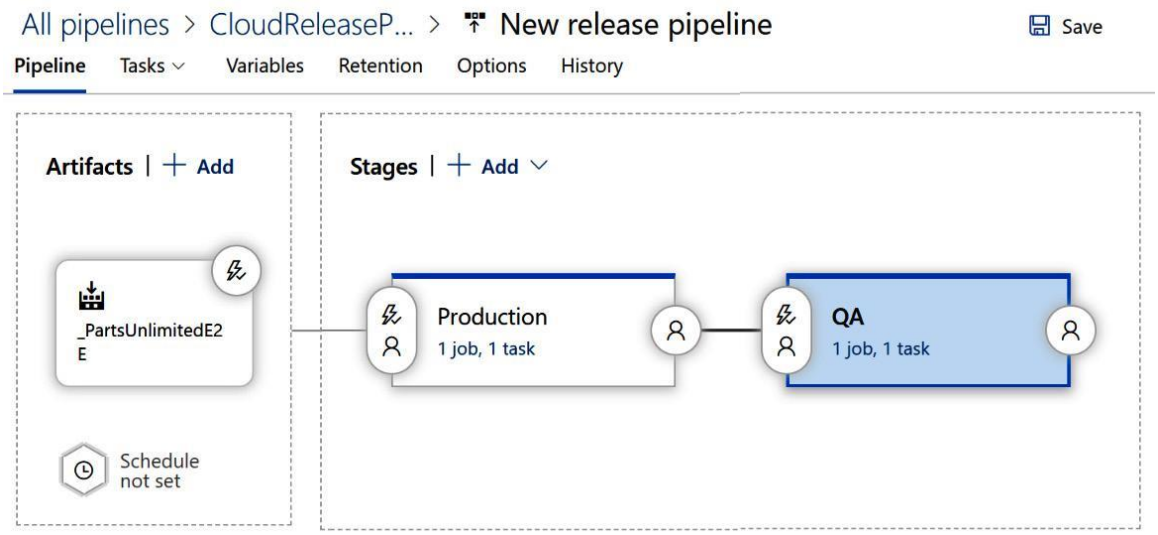
A multi-stage release pipeline is useful when you want to release your applications with multiple steps (staging), such as, for example, development, staging, and production. A quite common scenario in the real world is, for example, deploying an application initially to a testing environment. When tests are finished, the application is moved to a quality acceptance stage, and then, if the customer accepts the release, the application is moved to a production environment.

Here, we'll do the same: starting from the previously created single-stage pipeline, we'll create a new release pipeline with three stages, called **DEV**, **QA**, and **Production**. Each stage is a deployment target for our pipeline:

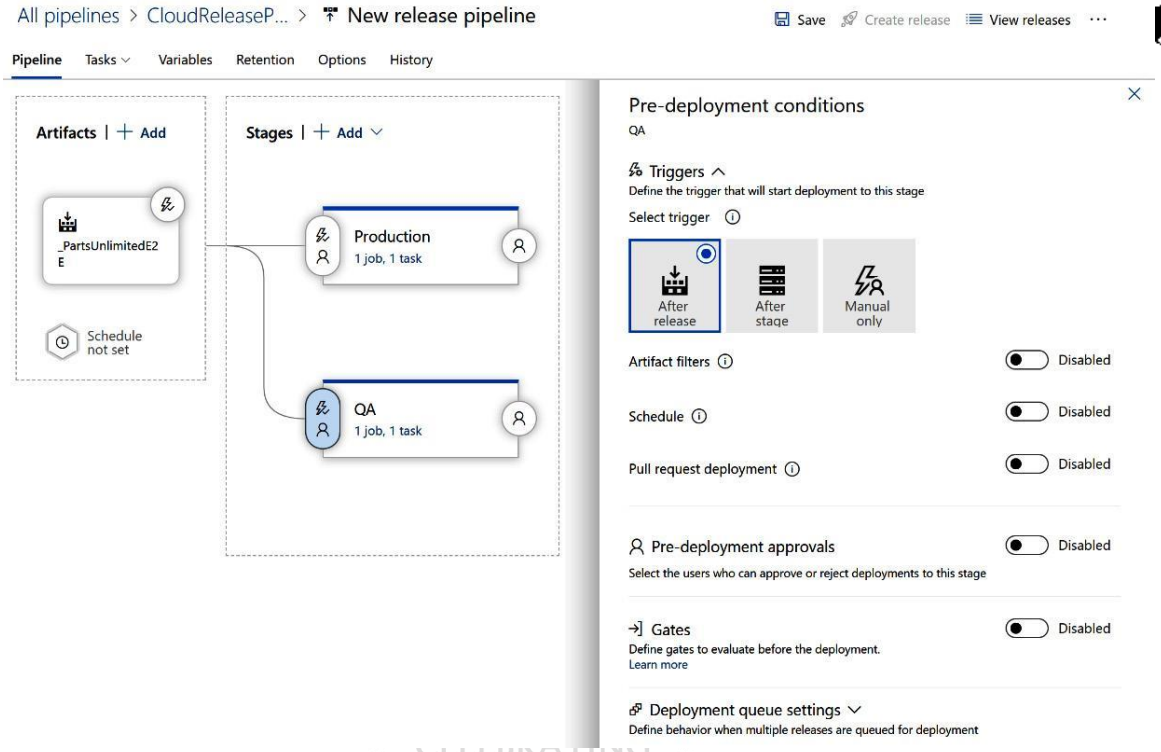
- In the previously defined pipeline, as a first step, I renamed the **Deploy to cloud** stage to **Production**. This will be the final stage of the release pipeline.
- Now, click on the **Clone** action to clone the defined stage into a new stage:



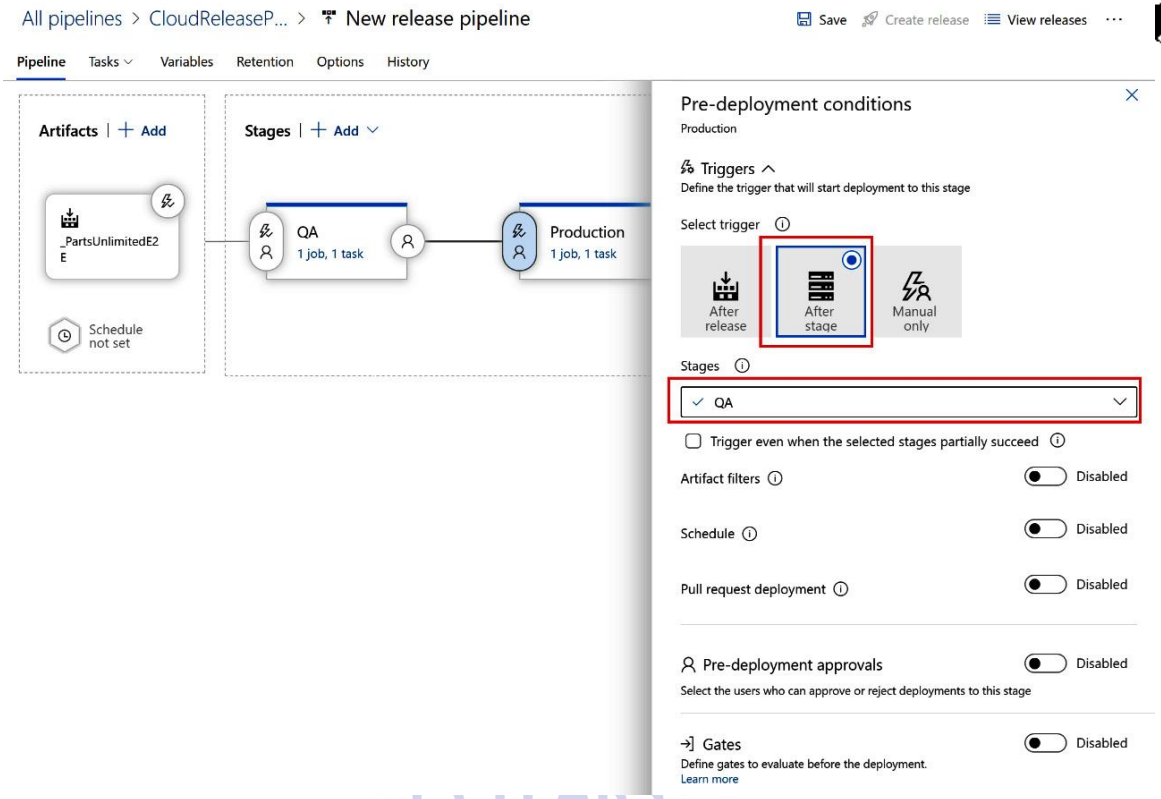
- A new cloned stage appears after the previously created stage. Change the name of this stage to **QA**:



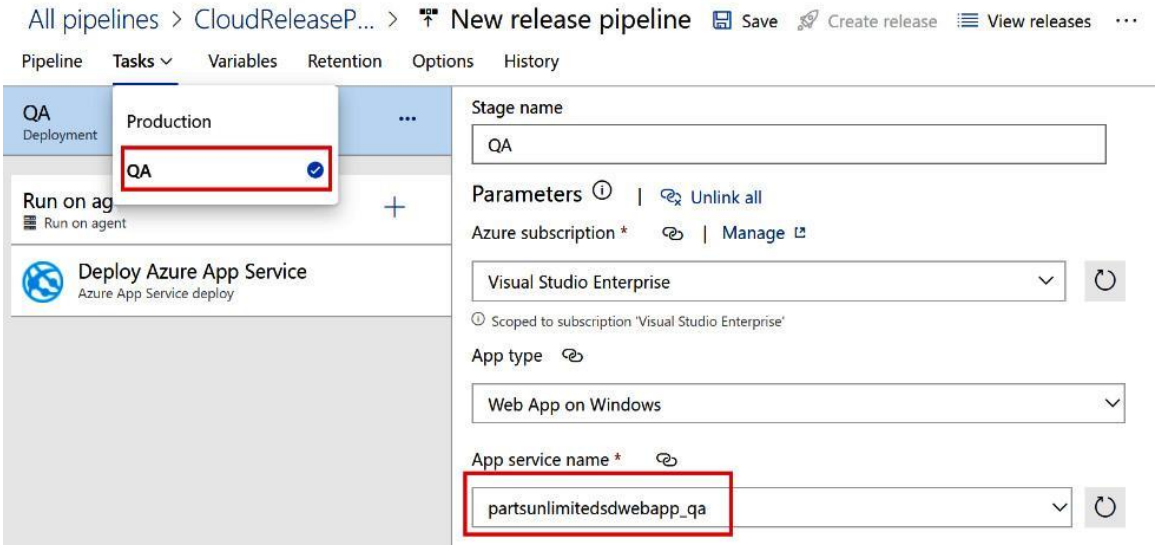
- Now, we need to reorganize the stages because the **QA** stage must occur before the **Production** stage. To reorganize these stages, select the **QA** stage and choose the pre-deployment conditions. In the **Pre-deployment conditions** pane, select **After release** as the trigger (instead of **After stage**):



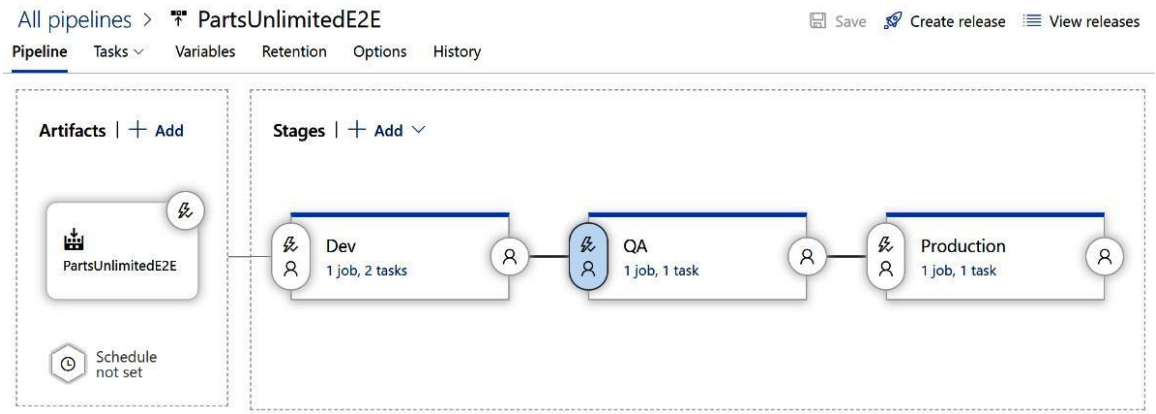
- As you can see, the pipeline diagram has now changed (you have the **QA** and **Production** stages executed in parallel). Now, select the **Pre-deployment conditions** properties for the **Production** stage; set the trigger to **After stage** and select **QA** as the stage:



- The stages are now ordered as we want (**QA** occurs before **Production**).
- At this point, we have two stages that deploy the application to the same environment (**QA** was created as a clone of **Production**). Select the **QA** stage from the **Tasks** drop-down list and change **App service name** to a new instance:



- Now, we need to repeat the same steps for creating the **DEV** stage. Clone it from **QA**, set its **Pre-deployment conditions** properties with the trigger set to **After Release**, and change the **QA** trigger to **After stage**, with **DEV** as the selected stage. Your pipeline will now look as follows:



- You have now created a release pipeline with different stages (**Dev**, **QA**, and **Production**) for controlling the deployment steps of your code.

USING APPROVALS AND GATES FOR MANAGING DEPLOYMENTS

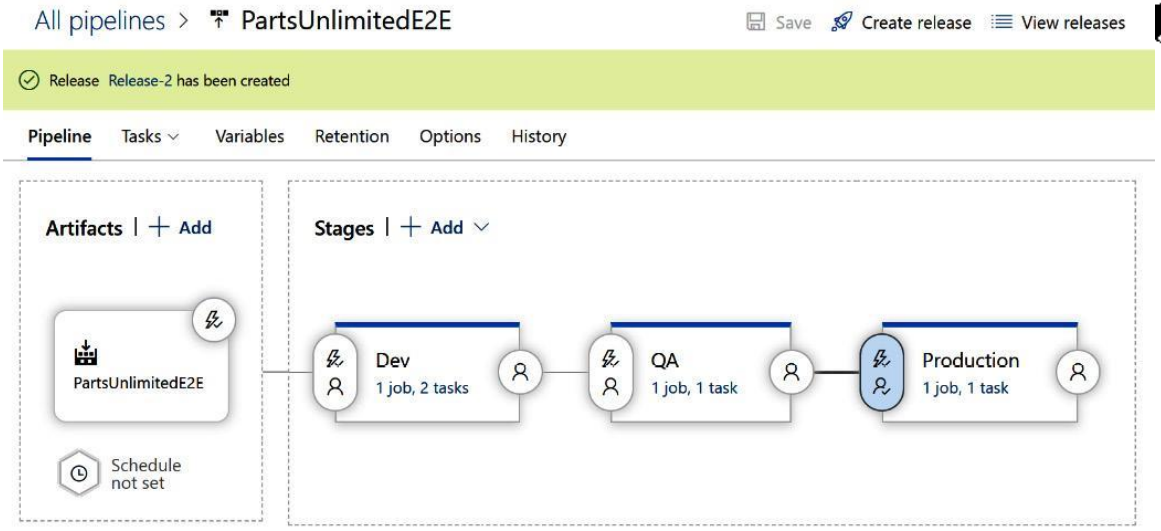
As previously configured, our release pipeline will move between stages only if the previous stage is completed successfully. This is okay for moving from **DEV** to **QA** because on this transition, our application is deployed to a testing environment, but the transition from **QA** to **Production** should usually be controlled because the release of an application into a production environment normally occurs after an approval.

Creating approvals

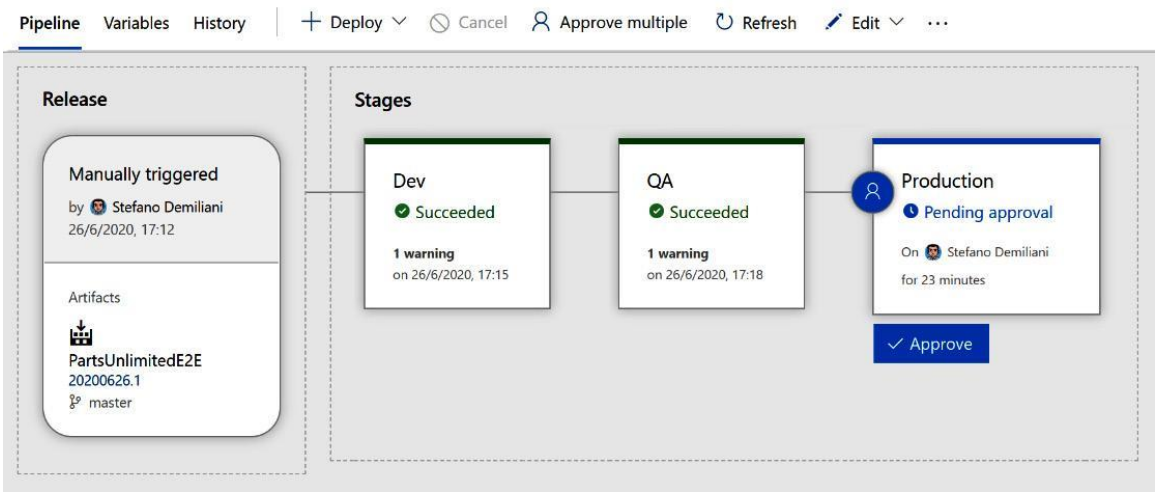
Let's follow these steps to create approvals:

- To create an approval step, from our pipeline definition, select the **Pre-deployment conditions** properties of the **Production** stage. Here, go to the **Pre-deployment approvals** section and enable it. Then, in the **Approvers** section, select the users that will be responsible for approving. Please also check that the **The user requesting a release or deployment should not approve it** option is not ticked:

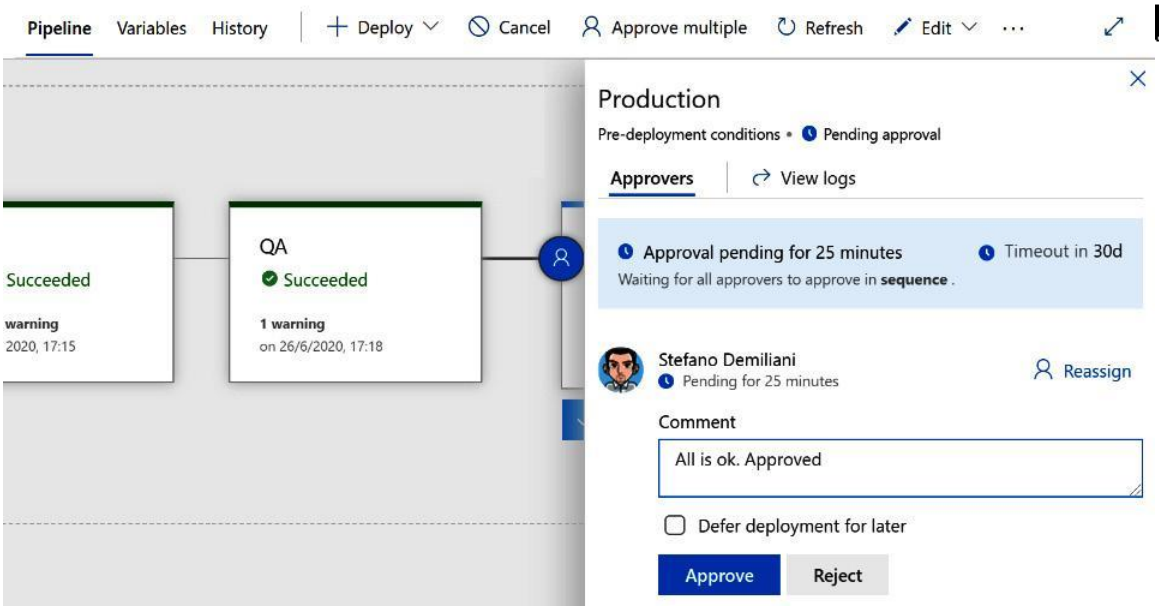
- Click on **Save** to save your pipeline definition.
- Now, create a new release to start our pipeline and click on the name of the created release (here, it is called **Release-2**):



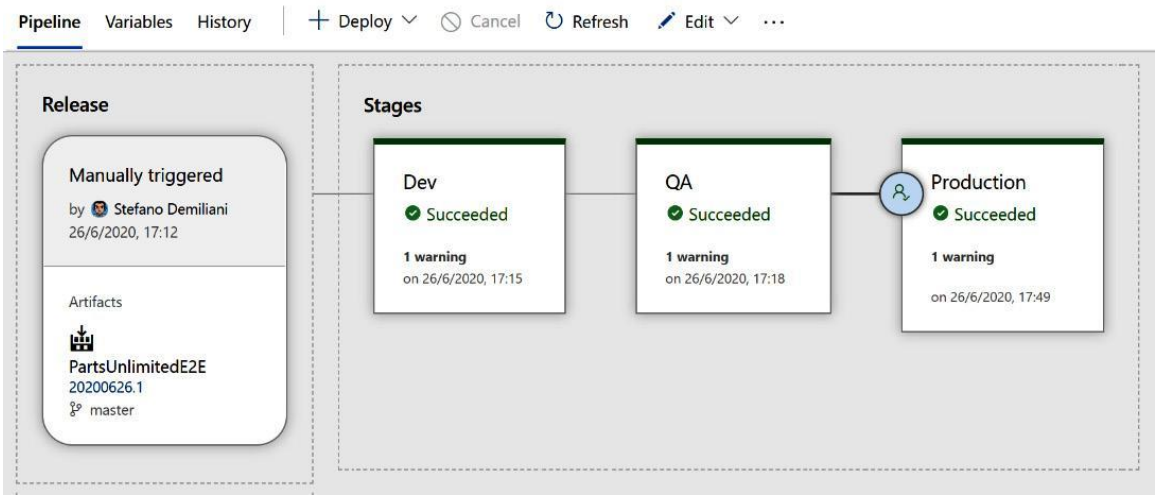
- The release pipeline starts. The **DEV** and **QA** stages are completed, while on the **Production** stage, a **Pending approval** status appears:



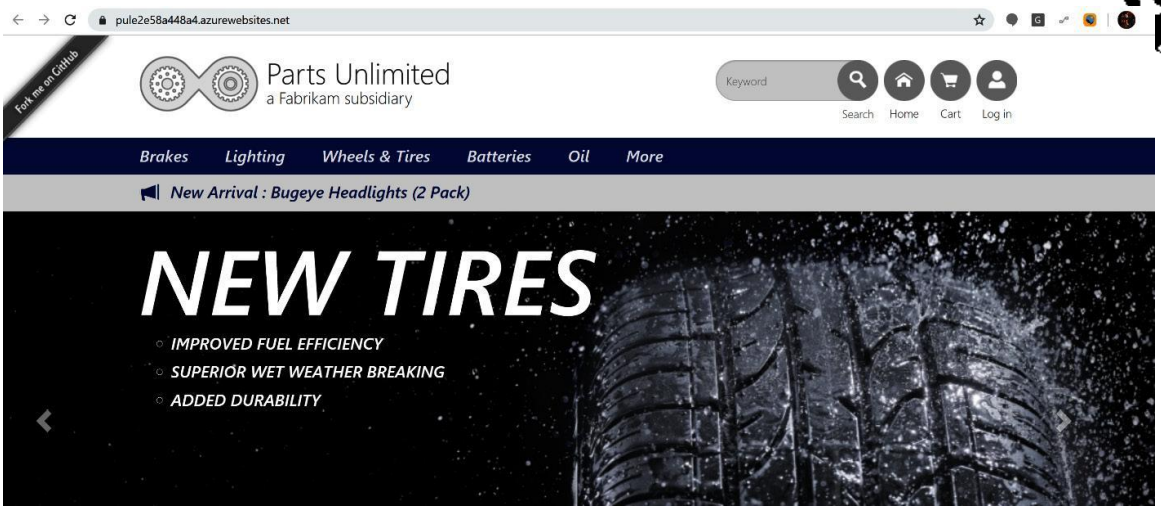
- The release pipeline is waiting for approval. You can click on the **Pending approval** icon and the approval dialog is opened. Here, you can insert a comment and then approve or reject the release:



- You can also defer the stage to a specific date if needed or reassign the approval to another user.
- If you click on **Approve**, the stage is approved and the release pipeline is completed:



- If you now click on the Azure App Service instance deployed by your pipeline, you can see that the final code (the **PartsUnlimited** website) is deployed in the cloud:



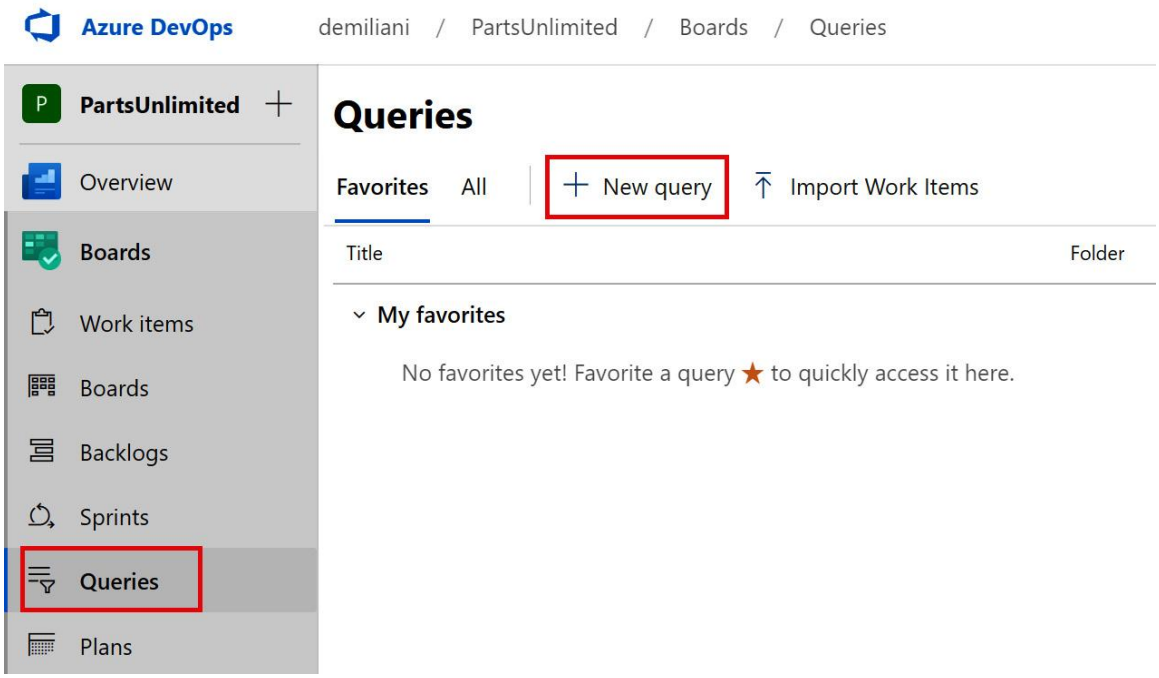
Using gates to check conditions

In the previously explained scenario, we saw how to configure a manual approval process for a release pipeline. Sometimes, you need to avoid the manual process and instead have a policy in place that permits your pipeline to go ahead only if some checks are successfully performed. This is where **gates** come in action.

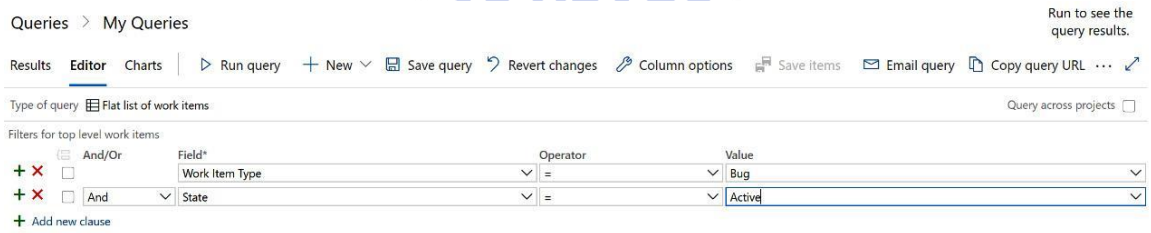
In Azure Pipelines, a gate allows you to automatically check for specific conditions from Azure DevOps from external services and then enable the release process only when the conditions are met. You can use gates to check the status of work items and issues of a project and enable the release only if you have no open bugs. You can also query test results, check whether security scans on artifacts are performed before releasing, monitor the infrastructure health before releasing, and so on.

As an example, here we want to configure a gate for our previously created release pipeline where we check for open bugs on Azure Boards. We will see how to do this with the help of the following steps:

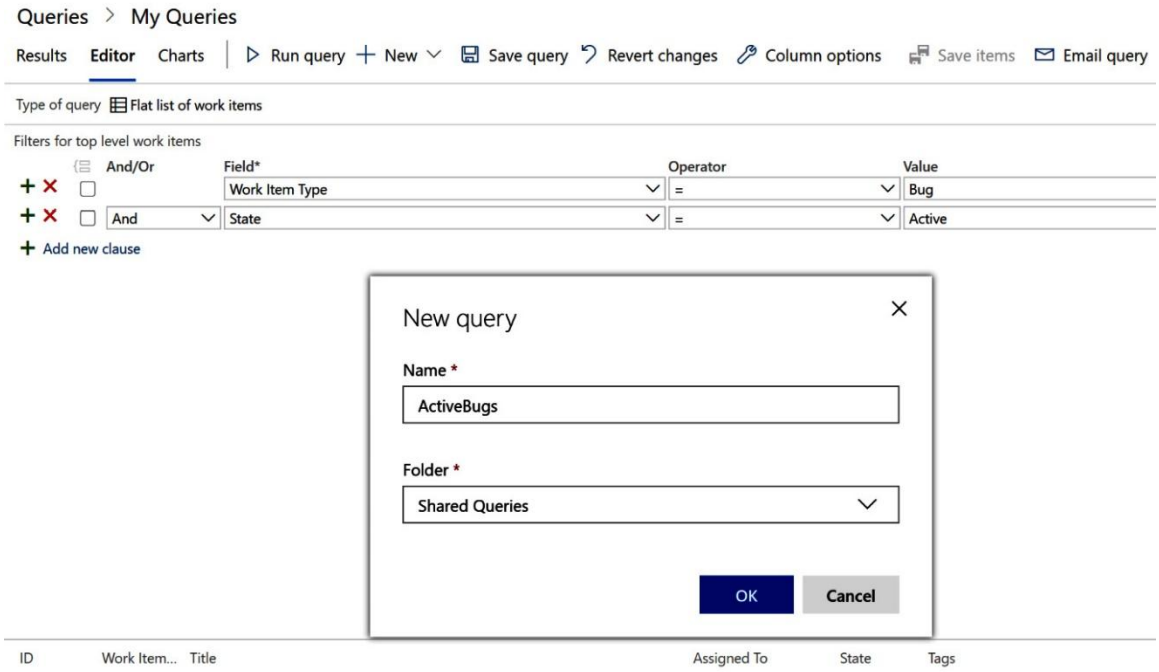
To check for open bugs in our project, we need to define a query for work items. From our Azure DevOps project, select **Boards**, click on **Queries**, and then select **New query**:



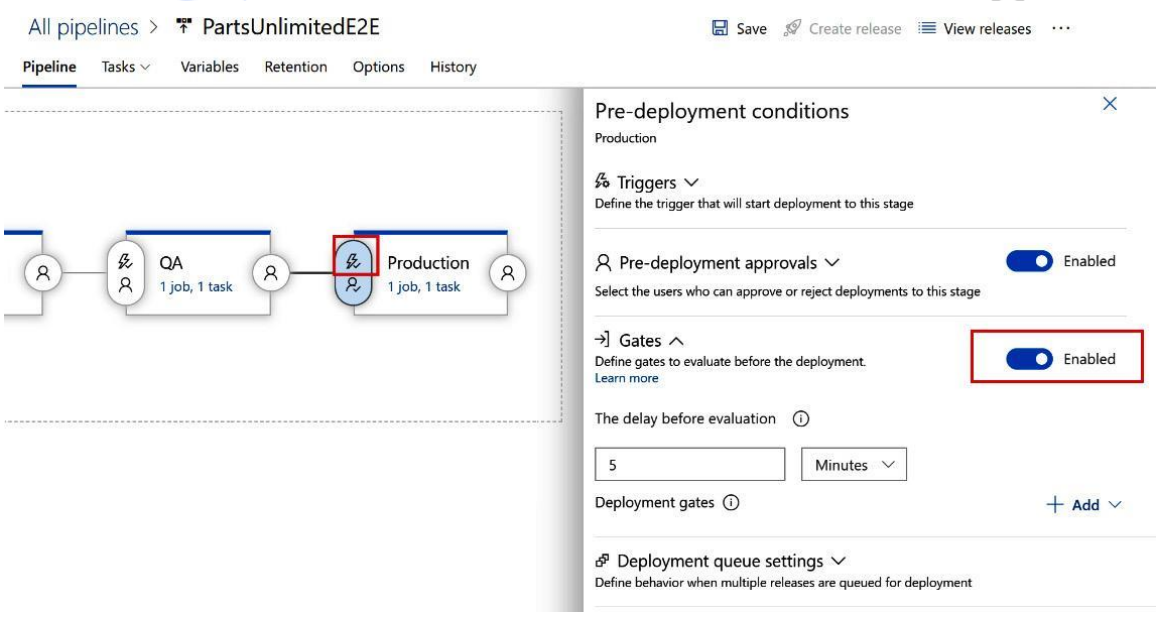
Here, I've defined a query as follows:



Save the query by giving it a name (for example, **ActiveBugs**) and specifying a folder (here, I've selected the **Shared Queries** folder):



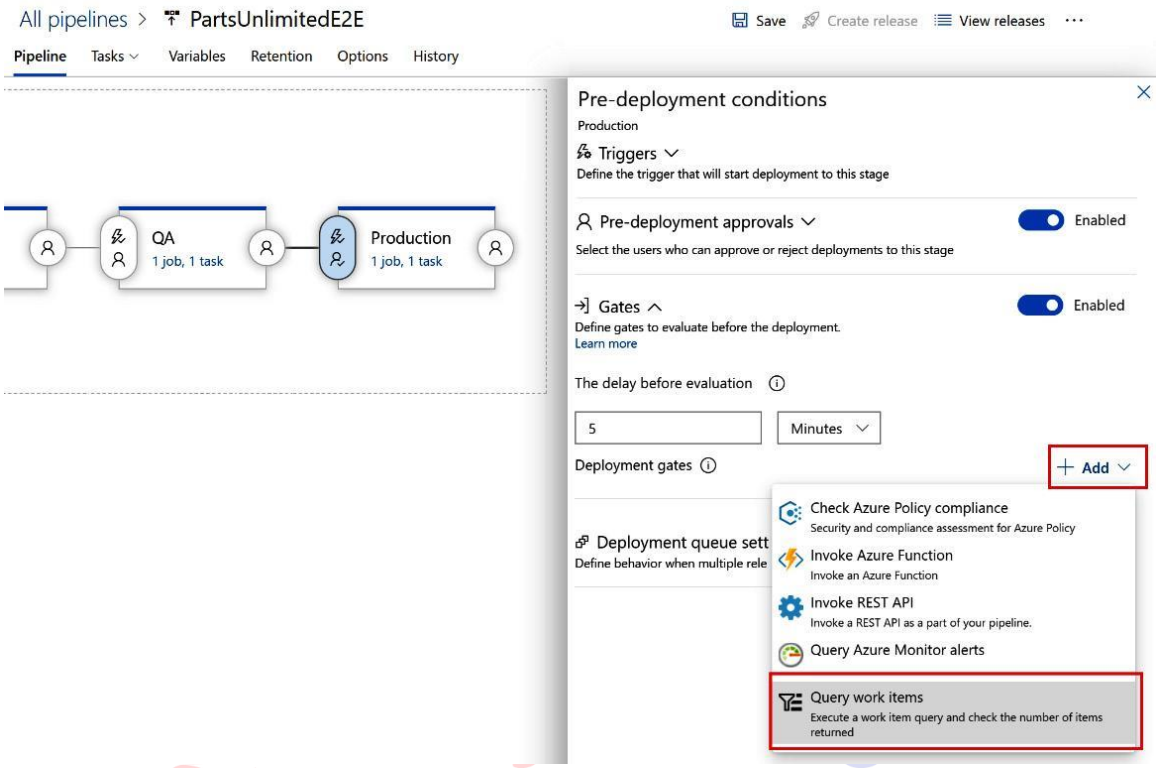
Now we're ready to define our gate. From the multi-stage release pipeline we previously created, select the **Production** stage, click on the bolt icon, and then enable gates, as shown in the following screenshot:



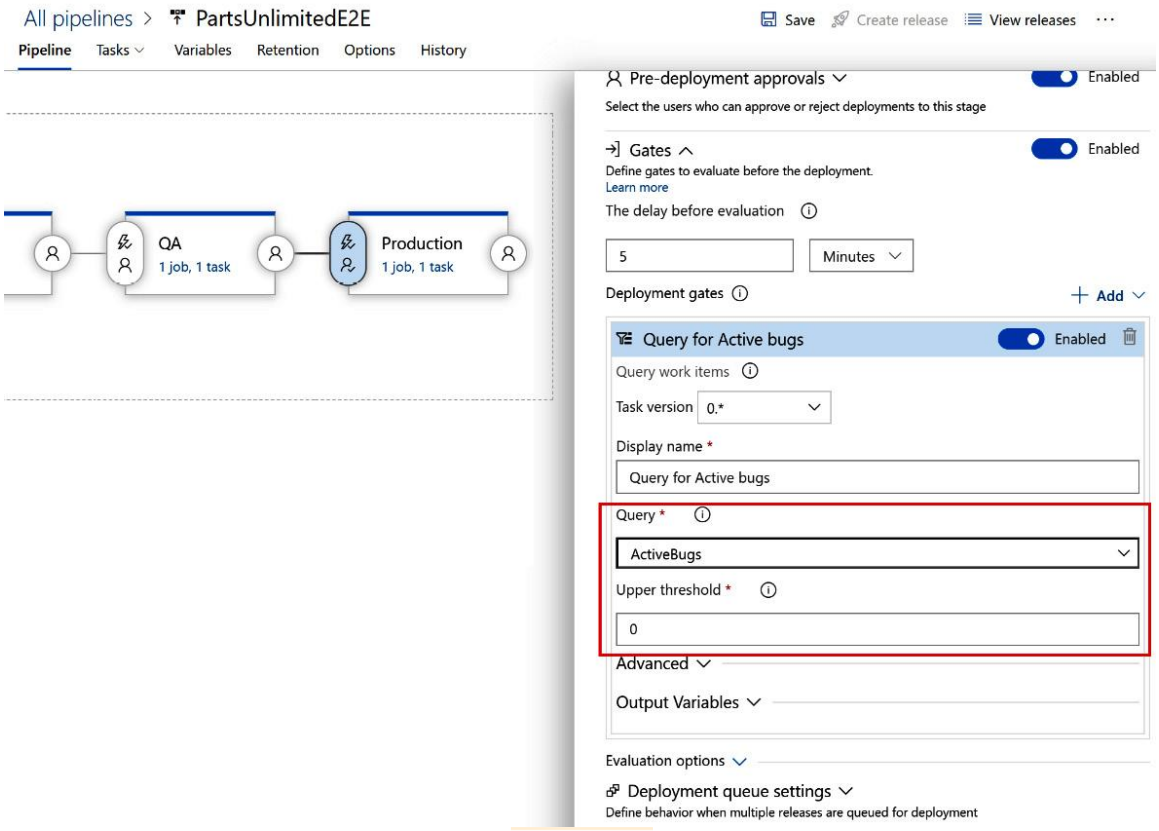
Here, you can also specify the delay before the evaluation of gates (the time before the added gates are evaluated for the first time. If no gates are added, then the deployment will wait for the specified duration before proceeding), and we can specify the deployment gates (adding gates that evaluate health

parameters). These gates are periodically evaluated in parallel and if the gates succeed, the deployment will proceed; otherwise, the deployment is rejected.

To specify our gate, click on **Add** and then select **Query work items** (this will execute a work item query and check the results):



Now, select the **ActiveBugs** query from the folder where you previously saved it (the **Shared folder**, in my case) and specify **Upper threshold** as **0** (the maximum number of matching work items from the query) because we want the release pipeline to only be completed if we have 0 active bugs:

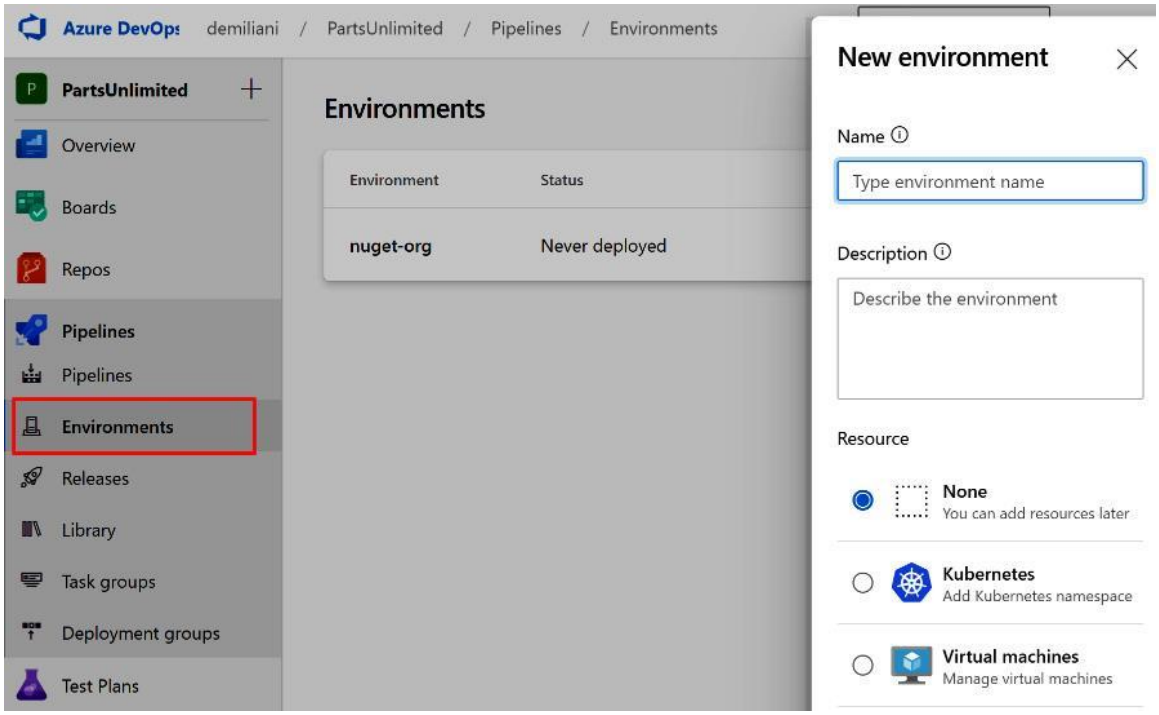


Here, you can also define evaluations options such as **time between re-evaluation of gates** (the duration after which the gates are re-evaluated; this must be greater than the longest typical response time of the configured gates to allow all responses to be received in each evaluation), **Minimum duration for steady results after a successful gates evaluation** (all gates must continuously be successful for this duration; **0** means deployment will proceed when all gates succeed in the same evaluation cycle), **timeout after which gates fail** (the maximum evaluation period for gates; the deployment will be rejected if the timeout is reached before gates succeed).

Our gate is now defined and active. You can also define other types of gates and you can also have gates that call Azure Functions to evaluate a release condition (which is useful if you want to integrate your release check with specific conditions on an external system).

YAML RELEASE PIPELINES WITH AZURE DEVOPS

Environments are a group of resources targeted by a pipeline – for example, . Azure Web Apps, virtual machines, or Kubernetes clusters. You can use environments to group resources by scope – for example, you can create an environment called **development** with your development resources and an environment called **production** with the production resources. Environments can be created by going to the **Environments** section under **Pipelines**:



The following is an example of a multi-stage release pipeline for deploying a .NET Core application on Azure Web Apps:

```
stages:
- stage: Build_Source_# Build Source Code for Dotnet Core Web App
  jobs:
  - job: Build
    pool: 'Hosted VS2017'
    variables:
      buildConfiguration: 'Release'
    continueOnError: false
    steps:
    - task: DotNetCoreCLI@2
      inputs:
```

```
        command: build
        arguments: '--configuration $(buildConfiguration)'
    - task: DotNetCoreCLI@2
      inputs:
        command: publish
        arguments: '--configuration $(buildConfiguration) --output
$(Build.ArtifactStagingDirectory)'
        modifyOutputPath: true
        zipAfterPublish: true
    - task: PublishBuildArtifacts@1
      inputs:
        path: $(Build.ArtifactStagingDirectory)
        artifact: drop
- stage: Deploy_In_Dev # Deploy artifacts to the dev environment
  jobs:
  - deployment: azure_web_app_dev
    pool: 'Hosted VS2017'
    variables:
      WebAppName: 'PartsUnlimited-dev'
    environment: 'dev-environment'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: AzureRMWebAppDeployment@4
            displayName: Azure App Service Deploy
            inputs:
              WebAppKind: webApp
              ConnectedServiceName: 'pay-as-you-go'
              WebAppName: $(WebAppName)
              Package: $(System.WorkFolder)/**/*.zip
- stage: Deploy_In_QA # Deploy artifacts to the qa environment
  jobs:
  - deployment: azure_web_app_qa
    pool: 'Hosted VS2017'
    variables:
      WebAppName: 'PartsUnlimited-qa'
    environment: 'qa-environment'
    strategy:
      runOnce:
        deploy:
          steps:
```

```
      - task: AzureRMWebAppDeployment@4
        displayName: Azure App Service Deploy
        inputs:
          WebAppKind: webApp
          ConnectedServiceName: 'pay-as-you-go'
          WebAppName: $(WebAppName)
          Package: $(System.WorkFolder)/**/*.zip
- stage: Deploy_In_Production # Deploy artifacts to the production environment
  jobs:
  - deployment: azure_web_app_prod
    pool: 'Hosted VS2017'
    variables:
      WebAppName: 'PartsUnlimited'
    environment: 'prod-environment'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: AzureRMWebAppDeployment@4
            displayName: Azure App Service Deploy
            inputs:
              WebAppKind: webApp
              ConnectedServiceName: 'pay-as-you-go'
              WebAppName: $(WebAppName)
              Package: $(System.WorkFolder)/**/*.zip
```

As you can see in the preceding YAML file, the pipeline defines four stages: **Build Source**, **Deploy in Dev**, **Deploy in QA**, and **Deploy in Production**. At each of these stages, the application is deployed on the specified environment.