**Integrate with Azure Pipelines**

Advanced

Administrator

Developer

DevOps Engineer

Security Engineer

Security Operations Analyst

Service Adoption Specialist

Solution Architect

Technology Manager

Azure

Azure Artifacts

Azure Boards

Azure Cloud Services

Azure DevOps

Azure Pipelines

Azure Repos

Azure Test Plans

GitHub

This module details Azure Pipelines anatomy and structure, templates, YAML resources, and how to use multiple repositories in your pipeline.

**Learning objectives**

By the end of this module, you're able to:

* Describe advanced Azure Pipelines anatomy and structure
* Detail templates and YAML resources
* Implement and use multiple repositories

[**Start**](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/1-introduction/)Add

**Prerequisites**

None

**This module is part of these learning paths**

* [AZ-400: Implement CI with Azure Pipelines and GitHub Actions](https://learn.microsoft.com/training/paths/az-400-implement-ci-azure-pipelines-github-actions/)

**Module assessment**

Assess your understanding of this module. Sign in and answer all questions correctly to earn a pass designation on your profile.

[**Take the module assessment**](https://learn.microsoft.com/training/modules/integrate-azure-pipelines/8-knowledge-check/)

* [Introduction](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/1-introduction)1 min
* [Describe the anatomy of a pipeline](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/2-describe-anatomy-of-pipeline)6 min
* [Understand the pipeline structure](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/3-understand-pipeline-structure)4 min
* [Detail templates](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/4-detail-templates)3 min
* [Explore YAML resources](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/5-explore-yaml-resources)2 min
* [Use multiple repositories in your pipeline](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/6-use-multiple-repositories-your-pipeline)3 min
* [Migrate a pipeline from classic to YAML in Azure Pipelines](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/7-migrate-pipeline-classic-yaml-azure-pipelines)3 min
* [Knowledge check](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/8-knowledge-check)4 min
* [Summary](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/9-summary)1 min

**Introduction**

Completed100 XP

* 1 minute

This module details Azure Pipelines anatomy and structure, templates, YAML resources, and how to use multiple repositories in your pipeline.

Also, it explores communication to deploy using Azure Pipelines to target servers.

**Learning objectives**

After completing this module, students and professionals can:

* Describe advanced Azure Pipelines anatomy and structure.
* Detail templates and YAML resources.
* Implement and use multiple repositories.
* Explore communication to deploy using Azure Pipelines.

**Prerequisites**

* Understanding of what DevOps is and its concepts.
* Familiarity with version control principles is helpful but isn't necessary.
* Understanding of Azure Pipelines.
* Beneficial to have experience in an organization that delivers software.

**Next unit: Describe the anatomy of a pipeline**

**Describe the anatomy of a pipeline**

Completed100 XP

* 6 minutes

Azure Pipelines can automatically build and validate every pull request and commit to your Azure Repos Git repository.

Azure Pipelines can be used with Azure DevOps public projects and Azure DevOps private projects.

In future training sections, we'll also learn how to use Azure Repos with external code repositories such as GitHub.

Let's start by creating a hello world YAML Pipeline.

**Hello world**

Start slowly and create a pipeline that echoes "Hello world!" to the console. No technical course is complete without a hello world example.

YAMLCopy

name: 1.0$(Rev:.r)

# simplified trigger (implied branch)

trigger:

- main

# equivalents trigger

# trigger:

# branches:

# include:

# - main

variables:

name: John

pool:

vmImage: ubuntu-latest

jobs:

- job: helloworld

steps:

- checkout: self

- script: echo "Hello, $(name)"

Most pipelines will have these components:

* Name – though often it's skipped (if it's skipped, a date-based name is generated automatically).
* Trigger – more on triggers later, but without an explicit trigger. There's an implicit "trigger on every commit to any path from any branch in this repo."
* Variables – "Inline" variables (more on other types of variables later).
* Job – every pipeline must have at least one job.
* Pool – you configure which pool (queue) the job must run on.
* Checkout – the "checkout: self" tells the job which repository (or repositories if there are multiple checkouts) to check out for this job.
* Steps – the actual tasks that need to be executed: in this case, a "script" task (the script is an alias) that can run inline scripts.

**Name**

The variable name is a bit misleading since the name is in the build number format. You'll get an integer number if you don't explicitly set a name format. A monotonically increasing number of runs triggered off this pipeline, starting at 1. This number is stored in Azure DevOps. You can make use of this number by referencing $(Rev).

To make a date-based number, you can use the format $(Date:yyyyMMdd) to get a build number like 20221003.

To get a semantic number like 1.0.x, you can use something like 1.0.$(Rev:.r).

**Triggers**

If there's no explicit triggers section, then it's implied that any commit to any path in any branch will trigger this pipeline to run.

However, you can be more precise by using filters such as branches or paths.

Let's consider this trigger:

YAMLCopy

trigger:

branches:

include:

- main

This trigger is configured to queue the pipeline only when a commit to the main branch exists. What about triggering for any branch except the main? You guessed it: use exclude instead of include:

YAMLCopy

trigger:

branches:

exclude:

- main

**Tip**

You can get the name of the branch from the variables Build.SourceBranch (for the full name like refs/heads/main) or Build.SourceBranchName (for the short name like main).

What about a trigger for any branch with a name that starts with topic/ and only if the change is in the webapp folder?

Copy

trigger:

branches:

include:

- feature/\*

paths:

include:

- webapp/\*\*

You can mix includes and excludes if you need to. You can also filter on tags.

**Tip**

Don't forget one overlooked trigger: none. You can use none if you never want your pipeline to trigger automatically. It's helpful if you're going to create a pipeline that is only manually triggered.

There are other triggers for other events, such as:

* Pull Requests (PRs) can also filter branches and paths.
* Schedules allow you to specify cron expressions for scheduling pipeline runs.
* Pipelines will enable you to trigger pipelines when other pipelines are complete, allowing pipeline chaining.

You can find all the documentation on triggers [here](https://learn.microsoft.com/en-us/azure/devops/pipelines/build/triggers).

**Jobs**

A job is a set of steps an agent executes in a queue (or pool). Jobs are atomic – they're performed wholly on a single agent. You can configure the same job to run on multiple agents simultaneously, but even in this case, the entire set of steps in the job is run on every agent. You'll need two jobs if you need some steps to run on one agent and some on another.

A job has the following attributes besides its name:

* displayName – a friendly name.
* dependsOn - a way to specify dependencies and ordering of multiple jobs.
* condition – a binary expression: if it evaluates to true, the job runs; if false, the job is skipped.
* strategy - used to control how jobs are parallelized.
* continueOnError - specify if the rest of the pipeline should continue if this job fails.
* pool – the pool name (queue) to run this job on.
* workspace - managing the source workspace.
* container - for specifying a container image to execute the job later.
* variables – variables scoped to this job.
* steps – the set of steps to execute.
* timeoutInMinutes and cancelTimeoutInMinutes for controlling timeouts.
* services - sidecar services that you can spin up.

**Dependencies**

When you define multiple stages in a pipeline, by default, they run sequentially in the order in which you define them in the YAML file. The exception to this is when you add dependencies. With dependencies, stages run in the order of the dependsOn requirements.

Pipelines must contain at least one stage with no dependencies.

Let's look at a few examples. Consider this pipeline:

YAMLCopy

jobs:

- job: A

steps:

# steps omitted for brevity

- job: B

steps:

# steps omitted for brevity

Because no dependsOn was specified, the jobs will run sequentially: first A and then B.

To have both jobs run in parallel, we add dependsOn: [] to job B:

YAMLCopy

jobs:

- job: A

steps:

# steps omitted for brevity

- job: B

dependsOn: [] # This removes the implicit dependency on the previous stage and causes this to run in parallel.

steps:

# steps omitted for brevity

If we want to fan out and fan in, we can do that too:

YAMLCopy

jobs:

- job: A

steps:

- script: echo' job A.'

- job: B

dependsOn: A

steps:

- script: echo' job B.'

- job: C

dependsOn: A

steps:

- script: echo' job C.'

- job: D

dependsOn:

- B

- C

steps:

- script: echo' job D.'

- job: E

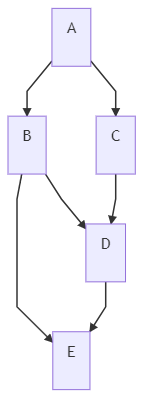
dependsOn:

- B

- D

steps:

- script: echo' job E.'



**Checkout**

Classic builds implicitly checkout any repository artifacts, but pipelines require you to be more explicit using the checkout keyword:

* Jobs check out the repo they're contained in automatically unless you specify checkout: none.
* Deployment jobs don't automatically check out the repo, so you'll need to specify checkout: self for deployment jobs if you want access to the YAML file's repo.

**Download**

Downloading artifacts requires you to use the download keyword. Downloads also work the opposite way for jobs and deployment jobs:

* Jobs don't download anything unless you explicitly define a download.
* Deployment jobs implicitly do a download: current, which downloads any pipeline artifacts created in the existing pipeline. To prevent it, you must specify download: none.

**Resources**

What if your job requires source code in another repository? You'll need to use resources. Resources let you reference:

* other repositories
* pipelines
* builds (classic builds)
* containers (for container jobs)
* packages

To reference code in another repo, specify that repo in the resources section and then reference it via its alias in the checkout step:

YAMLCopy

resources:

repositories:

- repository: appcode

type: git

name: otherRepo

steps:

- checkout: appcode

**Steps are Tasks**

Steps are the actual "things" that execute in the order specified in the job.

Each step is a task: out-of-the-box (OOB) tasks come with Azure DevOps. Many have aliases and tasks installed on your Azure DevOps organization via the marketplace.

Creating custom tasks is beyond the scope of this chapter, but you can see how to make your custom tasks [here](https://learn.microsoft.com/en-us/azure/devops/extend/develop/add-build-task).

**Variables**

It would be tough to achieve any sophistication in your pipelines without variables. Though this classification is partly mine, several types of variables exist, and pipelines don't distinguish between these types. However, I've found it helpful to categorize pipeline variables to help teams understand nuances when dealing with them.

Every variable is a key: value pair. The key is the variable's name, and it has a value.

To dereference a variable, wrap the key in $(). Let's consider this example:

YAMLCopy

variables:

name: John

steps:

- script: echo "Hello, $(name)!"

It will write Hello, John! To the log.

**Next unit: Understand the pipeline structure**

**Understand the pipeline structure**

Completed100 XP

* 4 minutes

A pipeline is one or more stages that describe a CI/CD process.

Stages are the primary divisions in a pipeline. The stages "Build this app," "Run these tests," and "Deploy to preproduction" are good examples.

A stage is one or more jobs, units of work assignable to the same machine.

You can arrange both stages and jobs into dependency graphs. Examples include "Run this stage before that one" and "This job depends on the output of that job."

A job is a linear series of steps. Steps can be tasks, scripts, or references to external templates.

This hierarchy is reflected in the structure of a YAML file like:

* Pipeline
  + Stage A
    - Job 1
      * Step 1.1
      * Step 1.2
      * ...
    - Job 2
      * Step 2.1
      * Step 2.2
      * ...
  + Stage B
    - ...

Simple pipelines don't require all these levels. For example, you can omit the containers for stages and jobs in a single job build because there are only steps.

Because many options shown in this article aren't required and have reasonable defaults, your YAML definitions are unlikely to include all of them.

**Pipeline**

The schema for a pipeline:

YAMLCopy

name: string # build numbering format

resources:

pipelines: [ pipelineResource ]

containers: [ containerResource ]

repositories: [ repositoryResource ]

variables: # several syntaxes

trigger: trigger

pr: pr

stages: [ stage | templateReference ]

If you have a single-stage, you can omit the stages keyword and directly specify the jobs keyword:

YAMLCopy

# ... other pipeline-level keywords

jobs: [ job | templateReference ]

If you've a single-stage and a single job, you can omit the stages and jobs keywords and directly specify the steps keyword:

YAMLCopy

# ... other pipeline-level keywords

steps: [ script | bash | pwsh | powershell | checkout | task | templateReference ]

**Stage**

A stage is a collection of related jobs. By default, stages run sequentially. Each stage starts only after the preceding stage is complete.

Use approval checks to control when a stage should run manually. These checks are commonly used to control deployments to production environments.

Checks are a mechanism available to the resource owner. They control when a stage in a pipeline consumes a resource.

As an owner of a resource like an environment, you can define checks required before a stage that consumes the resource can start.

This example runs three stages, one after another. The middle stage runs two jobs in parallel.

YAMLCopy

stages:

- stage: Build

jobs:

- job: BuildJob

steps:

- script: echo Building!

- stage: Test

dependsOn: Build

jobs:

- job: TestOnWindows

steps:

- script: echo Testing on Windows!

- job: TestOnLinux

steps:

- script: echo Testing on Linux!

- stage: Deploy

dependsOn: Test

jobs:

- job: Deploy

steps:

- script: echo Deploying the code!

**Job**

A job is a collection of steps run by an agent or on a server. Jobs can run conditionally and might depend on previous jobs.

YAMLCopy

jobs:

- job: MyJob

displayName: My First Job

continueOnError: true

workspace:

clean: outputs

steps:

- script: echo My first job

**Deployment strategies**

Deployment strategies allow you to use specific techniques to deliver updates when deploying your application.

Techniques examples:

* Enable initialization.
* Deploy the update.
* Route traffic to the updated version.
* Test the updated version after routing traffic.
* If there's a failure, run steps to restore to the last known good version.

**RunOnce**

runOnce is the most straightforward deployment strategy in all the presented lifecycle hooks.

YAMLCopy

strategy:

runOnce:

preDeploy:

pool: [ server | pool ] # See pool schema.

steps:

- script: [ script | bash | pwsh | powershell | checkout | task | templateReference ]

deploy:

pool: [ server | pool ] # See pool schema.

steps: ...

routeTraffic:

pool: [ server | pool ]

steps:

...

postRouteTraffic:

pool: [ server | pool ]

steps:

...

on:

failure:

pool: [ server | pool ]

steps:

...

success:

pool: [ server | pool ]

steps:

...

*For details and examples, see*[*Deployment jobs*](https://learn.microsoft.com/en-us/azure/devops/pipelines/process/deployment-jobs)*.*

**Rolling**

A rolling deployment replaces instances of the previous version of an application with instances of the new version. It can be configured by specifying the keyword rolling: under the strategy: node.

YAMLCopy

strategy:

rolling:

maxParallel: [ number or percentage as x% ]

preDeploy:

steps:

- script: [ script | bash | pwsh | powershell | checkout | task | templateReference ]

deploy:

steps:

...

routeTraffic:

steps:

...

postRouteTraffic:

steps:

...

on:

failure:

steps:

...

success:

steps:

...

*For details and examples, see*[*Deployment jobs*](https://learn.microsoft.com/en-us/azure/devops/pipelines/process/deployment-jobs)*.*

**Canary**

Using this strategy, you can first roll out the changes to a small subset of servers. The canary deployment strategy is an advanced deployment strategy that helps mitigate the risk of rolling out new versions of applications.

As you gain more confidence in the new version, you can release it to more servers in your infrastructure and route more traffic to it.

YAMLCopy

strategy:

canary:

increments: [ number ]

preDeploy:

pool: [ server | pool ] # See pool schema.

steps:

- script: [ script | bash | pwsh | powershell | checkout | task | templateReference ]

deploy:

pool: [ server | pool ] # See pool schema.

steps:

...

routeTraffic:

pool: [ server | pool ]

steps:

...

postRouteTraffic:

pool: [ server | pool ]

steps:

...

on:

failure:

pool: [ server | pool ]

steps:

...

success:

pool: [ server | pool ]

steps:

...

*For details and examples, see*[*Deployment jobs*](https://learn.microsoft.com/en-us/azure/devops/pipelines/process/deployment-jobs)*.*

**Lifecycle hooks**

You can achieve the deployment strategies technique by using lifecycle hooks. Depending on the pool attribute, each resolves into an agent or [server job](https://learn.microsoft.com/en-us/azure/devops/pipelines/process/phases).

Lifecycle hooks inherit the pool specified by the deployment job. Deployment jobs use the $(Pipeline.Workspace) system variable.

Available lifecycle hooks:

* **preDeploy:** Used to run steps that initialize resources before application deployment starts.
* **deploy:** Used to run steps that deploy your application. Download artifact task will be auto-injected only in the deploy hook for deployment jobs. To stop downloading artifacts, use - download: none or choose specific artifacts to download by specifying [Download Pipeline Artifact task](https://learn.microsoft.com/en-us/azure/devops/pipelines/yaml-schema/steps-download).
* **routeTraffic:** Used to run steps that serve the traffic to the updated version.
* **postRouteTraffic:** Used to run the steps after the traffic is routed. Typically, these tasks monitor the health of the updated version for a defined interval.
* **on: failure** or **on: success:** Used to run steps for rollback actions or clean-up.

**Steps**

A step is a linear sequence of operations that make up a job. Each step runs its process on an agent and accesses the pipeline workspace on a local hard drive.

This behavior means environment variables aren't preserved between steps, but file system changes are.

YAMLCopy

steps:

- script: echo This run in the default shell on any machine

- bash: |

echo This multiline script always runs in Bash.

echo Even on Windows machines!

- pwsh: |

Write-Host "This multiline script always runs in PowerShell Core."

Write-Host "Even on non-Windows machines!"

**Tasks**

Tasks are the building blocks of a pipeline. There's a catalog of tasks available to choose from.

YAMLCopy

steps:

- task: VSBuild@1

displayName: Build

timeoutInMinutes: 120

inputs:

solution: '\*\*\\*.sln'

**Next unit: Detail templates**

**Detail templates**

Completed100 XP

* 3 minutes

**Template references**

You can export reusable sections of your pipeline to a separate file. These individual files are known as templates.

Azure Pipelines supports four types of templates:

* Stage
* Job
* Step
* Variable

You can also use templates to control what is allowed in a pipeline and define how parameters can be used.

* Parameter

Templates themselves can include other templates. Azure Pipelines supports 50 individual template files in a single pipeline.

**Stage templates**

You can define a set of stages in one file and use it multiple times in other files.

In this example, a stage is repeated twice for two testing regimes. The stage itself is specified only once.

YAMLCopy

# File: stages/test.yml

parameters:

name: ''

testFile: ''

stages:

- stage: Test\_${{ parameters.name }}

jobs:

- job: ${{ parameters.name }}\_Windows

pool:

vmImage: windows-latest

steps:

- script: npm install

- script: npm test -- --file=${{ parameters.testFile }}

- job: ${{ parameters.name }}\_Mac

pool:

vmImage: macOS-latest

steps:

- script: npm install

- script: npm test -- --file=${{ parameters.testFile }}

Templated pipeline

YAMLCopy

# File: azure-pipelines.yml

stages:

- template: stages/test.yml # Template reference

parameters:

name: Mini

testFile: tests/miniSuite.js

- template: stages/test.yml # Template reference

parameters:

name: Full

testFile: tests/fullSuite.js

**Job templates**

You can define a set of jobs in one file and use it multiple times in other files.

In this example, a single job is repeated on three platforms. The job itself is specified only once.

YAMLCopy

# File: jobs/build.yml

parameters:

name: ''

pool: ''

sign: false

jobs:

- job: ${{ parameters.name }}

pool: ${{ parameters.pool }}

steps:

- script: npm install

- script: npm test

- ${{ if eq(parameters.sign, 'true') }}:

- script: sign

YAMLCopy

# File: azure-pipelines.yml

jobs:

- template: jobs/build.yml # Template reference

parameters:

name: macOS

pool:

vmImage: 'macOS-latest'

- template: jobs/build.yml # Template reference

parameters:

name: Linux

pool:

vmImage: 'ubuntu-latest'

- template: jobs/build.yml # Template reference

parameters:

name: Windows

pool:

vmImage: 'windows-latest'

sign: true # Extra step on Windows only

**Step templates**

You can define a set of steps in one file and use it multiple times in another.

YAMLCopy

# File: steps/build.yml

steps:

- script: npm install

- script: npm test

YAMLCopy

# File: azure-pipelines.yml

jobs:

- job: macOS

pool:

vmImage: 'macOS-latest'

steps:

- template: steps/build.yml # Template reference

- job: Linux

pool:

vmImage: 'ubuntu-latest'

steps:

- template: steps/build.yml # Template reference

- job: Windows

pool:

vmImage: 'windows-latest'

steps:

- template: steps/build.yml # Template reference

- script: sign # Extra step on Windows only

**Variable templates**

You can define a set of variables in one file and use it multiple times in other files.

In this example, a set of variables is repeated across multiple pipelines. The variables are specified only once.

YAMLCopy

# File: variables/build.yml

variables:

- name: vmImage

value: windows-latest

- name: arch

value: x64

- name: config

value: debug

YAMLCopy

# File: component-x-pipeline.yml

variables:

- template: variables/build.yml # Template reference

pool:

vmImage: ${{ variables.vmImage }}

steps:

- script: build x ${{ variables.arch }} ${{ variables.config }}

YAMLCopy

# File: component-y-pipeline.yml

variables:

- template: variables/build.yml # Template reference

pool:

vmImage: ${{ variables.vmImage }}

steps:

- script: build y ${{ variables.arch }} ${{ variables.config }}

**Next unit: Explore YAML resources**

**Explore YAML resources**

Completed100 XP

* 2 minutes

Resources in YAML represent sources of pipelines, repositories, and containers. For more information on Resources, [see here](https://learn.microsoft.com/en-us/azure/devops/pipelines/process/resources).

**General schema**

YAMLCopy

resources:

pipelines: [ pipeline ]

repositories: [ repository ]

containers: [ container ]

**Pipeline resource**

If you have an Azure pipeline that produces artifacts, your pipeline can consume the artifacts by using the pipeline keyword to define a pipeline resource.

YAMLCopy

resources:

pipelines:

- pipeline: MyAppA

source: MyCIPipelineA

- pipeline: MyAppB

source: MyCIPipelineB

trigger: true

- pipeline: MyAppC

project: DevOpsProject

source: MyCIPipelineC

branch: releases/M159

version: 20190718.2

trigger:

branches:

include:

- master

- releases/\*

exclude:

- users/\*

**Container resource**

Container jobs let you isolate your tools and dependencies inside a container. The agent launches an instance of your specified container then runs steps inside it. The container keyword lets you specify your container images.

Service containers run alongside a job to provide various dependencies like databases.

YAMLCopy

resources:

containers:

- container: linux

image: ubuntu:16.04

- container: windows

image: myprivate.azurecr.io/windowsservercore:1803

endpoint: my\_acr\_connection

- container: my\_service

image: my\_service:tag

ports:

- 8080:80 # bind container port 80 to 8080 on the host machine

- 6379 # bind container port 6379 to a random available port on the host machine

volumes:

- /src/dir:/dst/dir # mount /src/dir on the host into /dst/dir in the container

**Repository resource**

Let the system know about the repository if:

* If your pipeline has templates in another repository.
* If you want to use multi-repo checkout with a repository that requires a service connection.

The repository keyword lets you specify an external repository.

YAMLCopy

resources:

repositories:

- repository: common

type: github

name: Contoso/CommonTools

endpoint: MyContosoServiceConnection

**Next unit: Use multiple repositories in your pipeline**

**Use multiple repositories in your pipeline**

Completed100 XP

* 3 minutes

You might have micro git repositories providing utilities used in multiple pipelines within your project. Pipelines often rely on various repositories.

You can have different repositories with sources, tools, scripts, or other items that you need to build your code. By using multiple checkout steps in your pipeline, you can fetch and check out other repositories to the one you use to store your YAML pipeline.

Previously Azure Pipelines hasn't offered support for using multiple code repositories in a single pipeline. Using artifacts or directly cloning other repositories via script within a pipeline, you can work around it. It leaves access management and security down to you.

Repositories are now first-class citizens within Azure Pipelines. It enables some exciting use cases, such as checking out specific repository parts and checking multiple repositories.

There's also a use case for not checking out any repository in the pipeline. It can be helpful in cases where you're setting up a pipeline to do a job that has no dependency on any repository.

**Specify multiple repositories**

Repositories can be specified as a repository resource or in line with the checkout step. Supported repositories are Azure Repos Git, GitHub, and BitBucket Cloud.

The following combinations of checkout steps are supported.

* If there are no **checkout** steps, the default behavior is checkout: self is the first step.
* If there's a single **checkout: none** step, no repositories are synced or checked out.
* If there's a single **checkout: self** step, the current repository is checked out.
* If there's a single **checkout** step that isn't **self** or **none**, that repository is checked out instead of self.
* If there are multiple **checkout** steps, each named repository is checked out to a folder named after the repository. Unless a different path is specified in the checkout step, use **checkout: self** as one of the **checkout** steps.

**Repository resource - How to do it?**

If your repository type requires a service connection or other extended resources field, you must use a repository resource.

Even if your repository type doesn't require a service connection, you may use a repository resource.

For example, you have a repository resource defined already for templates in a different repository.

In the following example, three repositories are declared as repository resources. The repositories and the current self-repository containing the pipeline YAML are checked out.

YAMLCopy

resources:

repositories:

- repository: MyGitHubRepo # The name used to reference this repository in the checkout step.

type: github

endpoint: MyGitHubServiceConnection

name: MyGitHubOrgOrUser/MyGitHubRepo

- repository: MyBitBucketRepo

type: bitbucket

endpoint: MyBitBucketServiceConnection

name: MyBitBucketOrgOrUser/MyBitBucketRepo

- repository: MyAzureReposGitRepository

type: git

name: MyProject/MyAzureReposGitRepo

trigger:

- main

pool:

vmImage: 'ubuntu-latest'

steps:

- checkout: self

- checkout: MyGitHubRepo

- checkout: MyBitBucketRepo

- checkout: MyAzureReposGitRepository

- script: dir $(Build.SourcesDirectory)

If the self-repository is named CurrentRepo, the script command produces the following output: CurrentRepo MyAzureReposGitRepo MyBitBucketRepo MyGitHubRepo.

In this example, the repositories' names are used for the folders because no path is specified in the checkout step.

**Inline - How to do it?**

If your repository doesn't require a service connection, you can declare it according to your checkout step.

YAMLCopy

steps:

- checkout: git://MyProject/MyRepo # Azure Repos Git repository in the same organization

The default branch is checked out unless you choose a specific ref.

If you're using inline syntax, choose the ref by appending @ref. For example:

YAMLCopy

- checkout: git://MyProject/MyRepo@features/tools # checks out the features/tools branch

- checkout: git://MyProject/MyRepo@refs/heads/features/tools # also checks out the features/tools branch.

- checkout: git://MyProject/MyRepo@refs/tags/MyTag # checks out the commit referenced by MyTag.

**GitHub repository**

Azure Pipelines can automatically build and validate every pull request and commit to your GitHub repository.

When creating your new pipeline, you can select a GitHub repository and then a YAML file in that repository (self repository). By default, this is the repository that your pipeline builds.

Azure Pipelines must be granted access to your repositories to trigger their builds and fetch their code during builds.

There are three authentication types for granting Azure Pipelines access to your GitHub repositories while creating a pipeline.

* GitHub App.
* OAuth.
* Personal access token (PAT).

You can create a continuous integration (CI) trigger to run a pipeline whenever you push an update to the specified branches or push selected tags.

YAML pipelines are configured by default with a CI trigger on all branches.

YAMLCopy

trigger:

- main

- releases/\*

You can configure complex triggers that use **exclude** or **batch**.

YAMLCopy

# specific branches build

trigger:

branches:

include:

- master

- releases/\*

exclude:

- releases/old\*

Also, it's possible to configure pull request (PR) triggers to run whenever a pull request is opened with one of the specified target branches or when updates are made to such a pull request.

You can specify the target branches when validating your pull requests.

To validate pull requests that target main and releases/\* and start a new run the first time a new pull request is created, and after every update made to the pull request:

YAMLCopy

pr:

- main

- releases/\*

You can specify the full name of the branch or a wildcard.

For more information and guidance about GitHub integration, see:

* [Build GitHub repositories](https://learn.microsoft.com/en-us/azure/devops/pipelines/repos/github).

**Next unit: Migrate a pipeline from classic to YAML in Azure Pipelines**

[**Previous**](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/5-explore-yaml-resources/)

**Migrate a pipeline from classic to YAML in Azure Pipelines**

Completed100 XP

* 3 minutes

Migrating classic Azure Pipelines to YAML-based configurations offers a more efficient approach to defining and managing CI/CD pipelines. YAML enables version-controlled, code-like representations of pipelines, enhancing maintainability and flexibility compared to the traditional UI-based approach. By transitioning to YAML, development teams can leverage code review processes, reuse pipeline configurations across repositories, and easily track changes over time.

**Pipeline conversion**

The Azure DevOps portal includes the feature that facilitates migration from classic build pipelines to their YAML equivalents. You can access it via the Export to YAML context sensitive menu option of individual build pipelines. Selecting this option automatically generates a file containing the YAML representation of the converted content and downloads it to your computer. The classic pipeline is left intact.

The conversion process provides support for the following features:

* All tasks and all inputs
* Event and schedule-based triggers
* Individual and multiple jobs
* Job and step conditions
* Checkout options
* Execution plan parallelism
* Pool selection

There are, however, two features which aren't converted:

* Variables: Variables in a classic pipeline are excluded from the conversion process. Instead, they're mentioned by name in the comments of the autogenerated YAML pipeline as a reminder that you need to configure them manually.
* Time zone translation: Classic pipeline schedules are in the organization's time zone, while cron schedules in YAML are in UTC. To avoid any potential time zone conversion issues, the schedule is exported in its original format. This is also captured in the comments included in the new pipeline.

For release pipelines, you can leverage the View YAML link available for individual pipeline tasks. Selecting the link generates the YAML code corresponding to the configuration of the task. This helps to identify the underlying YAML syntax used to define the task. Note that this code doesn't include values of the individual task properties but instead substitutes them with variables. To set the values, refer to the graphical interface of the task and compare it against its YAML representation.

There's no equivalent option to view the YAML at the job or stage level. Effectively, this part of the conversion relies on your familiarity with the syntax of YAML pipelines.

**Pipeline authoring**

For those who are relatively new to YAML pipelines, Azure DevOps offers Task Assistant. Implemented as an extra pane displayed on the right side of the YAML pipeline editor page, the assistant helps with creating and modifying YAML steps. Unlike the View YAML feature, it works directly in the YAML pipeline. It simplifies the process of authoring a pipeline by providing a listing of tasks, which you can individually select, configure via a graphical interface, and add to the pipeline in the YAML format.

In addition, the Task Assistant provides intelligent autocomplete suggestions based on the context of the pipeline, inline documentation for task parameters to aid in understanding their usage, real-time validation to catch syntax errors or missing parameters, and the ability to search for specific tasks or task groups available in the Azure DevOps marketplace.

**Next unit: Knowledge check**

**Knowledge check**

Completed200 XP

* **Module assessment**
* 4 minutes

 Answer 100% of questions correctly in order to pass. [**Retake**](https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/8-knowledge-check)

Dismiss alert

Choose the best response for each question.

**Check your knowledge**

Top of Form

**1.**

**Which of the following choices is the YAML property responsible for creating a dependency between jobs?**

steps.

dependsOn.

**Correct. You can define dependencies between jobs using the dependensOn.**

condition.

**2.**

**Which of the following choices is responsible for always starting the communication between Azure Pipelines and its agent?**

Service Hook.

Azure Pipelines.

**Incorrect. The Agent always initiates the communication with Azure Pipelines.**

Agent.

**Correct. The Agent always initiates the communication with Azure Pipelines.**

**3.**

**Which of the following choices describes a reason for installing an agent using interactive mode?**

To run UI Tests.

**Correct. In some cases, you might need to run the agent interactively for production use - such as to run UI tests.**

To run Java pipelines.

**Incorrect. In some cases, you might need to run the agent interactively for production use - such as to run UI tests.**

To communicate with cloud environments from on-premises agents (self-hosted).

Bottom of Form

**Next unit: Summary**

**Summary**

Completed100 XP

* 1 minute

This module detailed Azure Pipelines anatomy and structure, templates, YAML resources, and how to use multiple repositories in your pipeline. Also, it explored communication to deploy using Azure Pipelines to target servers.

You learned how to describe the benefits and usage of:

* Describe advanced Azure Pipelines anatomy and structure.
* Detail templates and YAML resources.
* Implement and use multiple repositories.
* Explore communication to deploy using Azure Pipelines.

**Learn more**

* [Azure Pipelines New User Guide - Key concepts - Azure Pipelines | Microsoft Learn](https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/key-pipelines-concepts).
* [Azure Pipelines YAML pipeline editor guide - Azure Pipelines | Microsoft Learn](https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/yaml-pipeline-editor).
* [Check out multiple repositories in your pipeline - Azure Pipelines | Microsoft Learn](https://learn.microsoft.com/en-us/azure/devops/pipelines/repos/multi-repo-checkout).
* [Azure Pipelines Agents - Azure Pipelines | Microsoft Learn](https://learn.microsoft.com/en-us/azure/devops/pipelines/agents/agents).

“” <https://learn.microsoft.com/en-us/training/modules/integrate-azure-pipelines/9-summary#completion> “”