AZ-400: Implement CI with Azure Pipelines and GitHub Actions

Explore Azure Pipelines

Introduction

Completed

- 1 minute

Azure Pipelines is a fully featured service used to create cross-platform CI (Continuous Integration) and CD (Continuous Deployment). It works with your preferred Git provider and can deploy to most major cloud services, including Azure. Azure DevOps offers a comprehensive Pipelines offering.

This module introduces Azure Pipelines concepts and explains key terms and components of the tool, helping you decide your pipeline strategy and responsibilities.

Learning objectives

After completing this module, students and professionals can:

- Describe Azure Pipelines.
- Explain the role of Azure Pipelines and its components.
- Decide Pipeline automation responsibility.
- Understand Azure Pipeline key terms.

Prerequisites

- Understanding of what DevOps is and its concepts.
- Familiarity with version control principles is helpful but is not necessary.
- Beneficial to have experience in an organization that delivers software.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore the concept of pipelines in DevOps

Completed

- 4 minutes

Business demands continuous delivery of value. Value is created only when a product is delivered to a satisfied customer.

It's not created when one silo in the process is completed.

It demands that you reset focus from silos to an end-to-end flow of value.

The core idea is to create a repeatable, reliable, and incrementally-improving process for taking software from concept to customer.

The goal is to enable a constant flow of changes into production via an automated software production line.

Think of it as a pipeline. The pipeline breaks down the software delivery process into stages.

Each stage aims to verify the quality of new features from a different angle to validate the new functionality and prevent errors from affecting your users.

The pipeline should provide feedback to the team. Also, visibility into the changes flows to everyone involved in delivering the new feature(s).

A delivery pipeline enables the flow of more minor changes more frequently, with a focus on flow.

Your teams can concentrate on optimizing the delivery of changes that bring quantifiable value to the business.

This approach leads teams to continuously monitor and learn where they're finding obstacles, resolve those issues, and gradually improve the pipeline's flow.

As the process continues, the feedback loop provides new insights into new issues and barriers to be resolved.

The pipeline is the focus of your continuous improvement loop.

A typical pipeline will include the following stages: build automation and continuous integration, test automation, and deployment automation.

Build automation and continuous integration

The pipeline starts by building the binaries to create the deliverables passed to the following stages. New features implemented by the developers are integrated into the central code base, built, and unit tested. It's the most direct feedback cycle that informs the development team about the health of their application code.

Test automation

The new version of an application is rigorously tested throughout this stage to ensure that it meets all wished system qualities. It's crucial that all relevant aspects—whether functionality, security, performance, or compliance—are verified by the pipeline. The stage may involve different types of automated or (initially, at least) manual activities.

Deployment automation

A deployment is required every time the application is installed in an environment for testing, but the most critical moment for deployment automation is rollout time.

Since the preceding stages have verified the overall quality of the system, It's a low-risk step.

The deployment can be staged, with the new version being initially released to a subset of the production environment and monitored before being rolled out.

The deployment is automated, allowing for the reliable delivery of new functionality to users within minutes if needed.

Your pipeline needs platform provisioning and configuration management

The deployment pipeline is supported by platform provisioning and system configuration management. It allows teams to create, maintain, and tear down complete environments automatically or at the push of a button.

Automated platform provisioning ensures that your candidate applications are deployed to, and tests carried out against correctly configured and reproducible environments.

It also helps horizontal scalability and allows the business to try out new products in a sandbox environment at any time.

Orchestrating it all: release and pipeline orchestration

The multiple stages in a deployment pipeline involve different groups of people collaborating and supervising the release of the new version of your application.

Release and pipeline orchestration provide a top-level view of the entire pipeline, allowing you to define and control the stages and gain insight into the overall software delivery process.

By carrying out value stream mappings on your releases, you can highlight any remaining inefficiencies and hot spots and pinpoint opportunities to improve your pipeline.

These automated pipelines need infrastructure to run on. The efficiency of this infrastructure will have a direct impact on the effectiveness of the pipeline.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe Azure Pipelines

Completed

- 3 minutes

Azure Pipelines is a cloud service that automatically builds and tests your code project and makes it available to other users. It works with just about any language or project type.

Azure Pipelines combines continuous integration (CI) and continuous delivery (CD) to test and build your code and ship it to any target constantly and consistently.

Does Azure Pipelines work with my language and tools?

Azure Pipelines is a fully featured cross-platform CI and CD service. It works with your preferred Git provider and can deploy to most major cloud services, including Azure services.

Languages

You can use many languages with Azure Pipelines, such as Python, Java, PHP, Ruby, C#, and Go.

Version control systems

Before you use continuous integration and continuous delivery practices for your applications, you must have your source code in a version control system. Azure Pipelines integrates with GitHub, GitLab, Azure Repos, Bitbucket, and Subversion.

Application types

You can use Azure Pipelines with most application types, such as Java, JavaScript, Python, .NET, PHP, Go, XCode, and C++.

Deployment targets

Use Azure Pipelines to deploy your code to multiple targets. Targets including:

- Container registries.
- Virtual machines.
- Azure services, or any on-premises or cloud target such:
    - Microsoft Azure.
    - Google Cloud.
    - Amazon Web Services (AWS).

Package formats

To produce packages that others can consume, you can publish NuGet, npm, or Maven packages to the built-in package management repository in Azure Pipelines.

You also can use any other package management repository of your choice.

Why should I use CI and CD, and Azure Pipelines?

Implementing CI and CD pipelines help to ensure consistent and quality code that's readily available to users.

Azure Pipelines is a quick, easy, and safe way to automate building your projects and making them available to users.

Use CI and CD for your project

Continuous integration is used to automate tests and builds for your project. CI helps to catch bugs or issues early in the development cycle when they're easier and faster to fix. Items known as artifacts are produced from CI systems. The continuous delivery release pipelines use them to drive automatic deployments.

Continuous delivery is used to automatically deploy and test code in multiple stages to help drive quality. Continuous integration systems produce deployable artifacts, which include infrastructure and apps. Automated release pipelines consume these artifacts to release new versions and fixes to the target of your choice.

Continuous integration (CI)

Continuous delivery (CD)

Increase code coverage.

Automatically deploy code to production.

Build faster by splitting test and build runs.

Ensure deployment targets have the latest code.

Automatically ensure you don't ship broken code.

Use tested code from the CI process.

Run tests continually.

Use Azure Pipelines for CI and CD

There are several reasons to use Azure Pipelines for your CI and CD solution. You can use it to:

- Work with any language or platform.
- Deploy to different types of targets at the same time.
- Integrate with Azure deployments.
- Build on Windows, Linux, or macOS machines.
- Integrate with GitHub.
- Work with open-source projects.

[Continue](#)

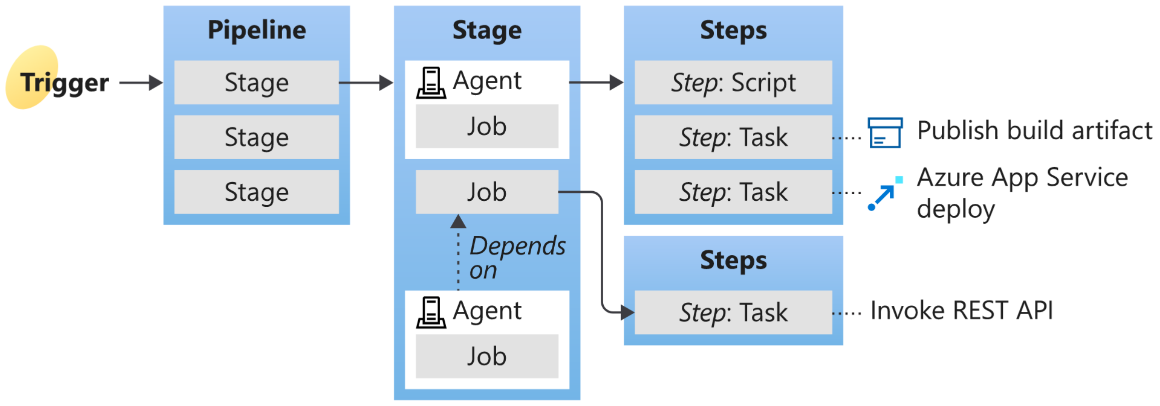
Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Understand Azure Pipelines key terms](#)

Completed

- 4 minutes

Understanding the basic terms and parts of Azure Pipelines helps you further explore how it can help you deliver better code more efficiently and reliably.



Agent

When your build or deployment runs, the system begins one or more jobs. An agent is installable software that runs a build or deployment job.

Artifact

An artifact is a collection of files or packages published by a build. Artifacts are made available for the tasks, such as distribution or deployment.

Build

A build represents one execution of a pipeline. It collects the logs associated with running the steps and the test results.

Continuous delivery

Continuous delivery (CD) (also known as Continuous Deployment) is a process by which code is built, tested, and deployed to one or more test and production stages. Deploying and testing in multiple stages helps drive quality. Continuous integration systems produce deployable artifacts, which include infrastructure and apps. Automated release pipelines consume these artifacts to release new versions and fix existing systems. Monitoring and alerting systems constantly run to drive visibility into the entire CD process. This process ensures that errors are caught often and early.

Continuous integration

Continuous integration (CI) is the practice used by development teams to simplify the testing and building of code. CI helps to catch bugs or problems early in the development cycle, making them more accessible and faster to fix. Automated tests and builds are run as part of the CI process. The process can run on a schedule, whenever code is pushed, or both. Items known as artifacts are produced from CI systems. The continuous delivery release pipelines use them to drive automatic deployments.

Deployment target

A deployment target is a virtual machine, container, web app, or any service used to host the developed application. A pipeline might deploy the app to one or more deployment targets after the build is completed and tests are run.

Job

A build contains one or more jobs. Most jobs run on an agent. A job represents an execution boundary of a set of steps. All the steps run together on the same agent.

For example, you might build two configurations - x86 and x64. In this case, you have one build and two jobs.

Pipeline

A pipeline defines the continuous integration and deployment process for your app. It's made up of steps called tasks.

It can be thought of as a script that describes how your test, build, and deployment steps are run.

Release

When you use the visual designer, you can create a release or a build pipeline. A release is a term used to describe one execution of a release pipeline. It's made up of deployments to multiple stages.

Stage

Stages are the primary divisions in a pipeline: "build the app," "run integration tests," and "deploy to user acceptance testing" are good examples of stages.

Task

A task is the building block of a pipeline. For example, a build pipeline might consist of build and test tasks. A release pipeline consists of deployment tasks. Each task runs a specific job in the pipeline.

Trigger

A trigger is set up to tell the pipeline when to run. You can configure a pipeline to run upon a push to a repository at scheduled times or upon completing another build. All these actions are known as triggers.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices aren't characteristic of a pipeline?

○

Reliable.

○

Repeatable.

○

Immutable.

2.

Which of the following is a correct statement about Azure Pipelines?

○

Azure Pipelines works exclusively with Azure and Microsoft Technology. For open-source, GitHub is recommended.

○

Azure Pipelines work with any language or platform.

○

Azure Pipelines only deploys in cloud environments.

3.

Which of the following choices is a Continuous Integration (CI) benefit?

○

Automatically ensure you don't ship broken code.

○

Automatically deploy code to production.

○

Ensure deployment targets have the latest code.

Check your answers

You must answer all questions before checking your work.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

This module introduced Azure Pipelines concepts and explained key terms and components of the tool. It helped you decide your pipeline strategy and responsibilities.

You learned how to describe the benefits and usage of:

- Describe Azure Pipelines.
- Explain the role of Azure Pipelines and its components.

- Decide Pipeline automation responsibility.
- Understand Azure Pipeline key terms.

Learn more

- What is Azure Pipelines? - Azure Pipelines | Microsoft Docs .
- Use Azure Pipelines - Azure Pipelines | Microsoft Docs .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Manage Azure Pipeline agents and pools

Introduction

Completed

- 1 minute

This module explores the differences between Microsoft-hosted and self-hosted agents, detail job types, and introduces agent pool configuration. Understand typical situations to use agent pools and how to manage their security.

Learning objectives

After completing this module, students and professionals can:

- Choose between Microsoft-hosted and self-hosted agents.
- Install and configure Azure Pipelines Agents.
- Configure agent pools.
- Make the agents and pools secure.

Prerequisites

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

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Choose between Microsoft-hosted versus self-hosted agents

Completed

- 3 minutes

To build your code or deploy your software, you generally need at least one agent.

As you add more code and people, you'll eventually need more.

When your build or deployment runs, the system begins one or more jobs.

An agent is an installable software that runs one build or deployment job at a time.

Microsoft-hosted agent

If your pipelines are in Azure Pipelines, then you've got a convenient option to build and deploy using a Microsoft-hosted agent.

With a Microsoft-hosted agent, maintenance and upgrades are automatically done.

Each time a pipeline is run, a new virtual machine (instance) is provided. The virtual machine is discarded after one use.

For many teams, this is the simplest way to build and deploy.

You can try it first and see if it works for your build or deployment. If not, you can use a self-hosted agent.

A Microsoft-hosted agent has job time limits.

Self-hosted agent

An agent that you set up and manage on your own to run build and deployment jobs is a self-hosted agent.

You can use a self-hosted agent in Azure Pipelines. A self-hosted agent gives you more control to install dependent software needed for your builds and deployments.

You can install the agent on:

- Linux.
- macOS.
- Windows.
- Linux Docker containers.

After you've installed the agent on a machine, you can install any other software on that machine as required by your build or deployment jobs.

A self-hosted agent doesn't have job time limits.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore job types

Completed

- 1 minute

In Azure DevOps, there are four types of jobs available:

- Agent pool jobs.
- Container jobs.
- Deployment group jobs.
- Agentless jobs.

Agent pool jobs

The most common types of jobs. The jobs run on an agent that is part of an agent pool.

Container jobs

Similar jobs to Agent Pool Jobs run in a container on an agent part of an agent pool.

Deployment group jobs

Jobs that run on systems in a deployment group.

Agentless jobs

Jobs that run directly on the Azure DevOps. They don't require an agent for execution. It's also-often-called Server Jobs.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Introduction to agent pools

Completed

- 1 minute

Instead of managing each agent individually, you organize agents into agent pools. An agent pool defines the sharing boundary for all agents in that pool.

In Azure Pipelines, pools are scoped to the entire organization so that you can share the agent machines across projects.

If you create an Agent pool for a specific project, only that project can use the pool until you add the project pool into another project.

When creating a build or release pipeline, you can specify which pool it uses, organization, or project scope.

Pools scoped to a project can only use them across build and release pipelines within a project.

To share an agent pool with multiple projects, use an organization scope agent pool and add them in each of those projects, add an existing agent pool, and choose the organization agent pool. If you create a new agent pool, you can automatically grant access permission to all pipelines.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore predefined agent pool

Completed

- 1 minute

Azure Pipelines provides a pre-defined agent pool-named **Azure Pipelines** with Microsoft-hosted agents.

It will often be an easy way to run jobs without configuring build infrastructure.

The following virtual machine images are provided by default:

- Windows Server 2022 with Visual Studio 2022.
- Windows Server 2019 with Visual Studio 2019.
- Ubuntu 20.04.
- Ubuntu 18.04.
- macOS 11 Big Sur.
- macOS X Catalina 10.15.

By default, all contributors in a project are members of the User role on each hosted pool.

It allows every contributor to the author and runs build and release pipelines using a Microsoft-hosted pool.

Pools are used to run jobs. Learn about specifying pools for jobs .

Note

See Microsoft-hosted agents for the most up-to-date list of Agent Pool Images. Also, the complete list of software installed on these machines.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Understand typical situations for agent pools

Completed

- 2 minutes

If you've got many agents intended for different teams or purposes, you might want to create more pools, as explained below.

Create agent pools

Here are some typical situations when you might want to create agent pools:

- You're a project member, and you want to use a set of machines your team owns for running build and deployment jobs.
    - First, make sure you're a member of a group in All Pools with the Administrator role.
    - Next, create a New project agent pool in your project settings and select the option to Create a new organization agent pool. As a result, both an organization and project-level agent pool will be created.
    - Finally, install and configure agents to be part of that agent pool.
- You're a member of the infrastructure team and would like to set up a pool of agents for use in all projects.
    - First, make sure you're a member of a group in All Pools with the Administrator role.
    - Next, create a New organization agent pool in your admin settings and select Autoprovision corresponding project agent pools in all projects while creating the pool. This setting ensures all projects have a pool pointing to the organization agent pool. The system creates a pool for existing projects, and in the future, it will do so whenever a new project is created.
    - Finally, install and configure agents to be part of that agent pool.
- You want to share a set of agent machines with multiple projects, but not all of them.
    - First, create a project agent pool in one of the projects and select the option to Create a new organization agent pool while creating that pool.
    - Next, go to each of the other projects, and create a pool in each of them while selecting the option to Use an existing organization agent pool.
    - Finally, install and configure agents to be part of the shared agent pool.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Communicate with Azure Pipelines

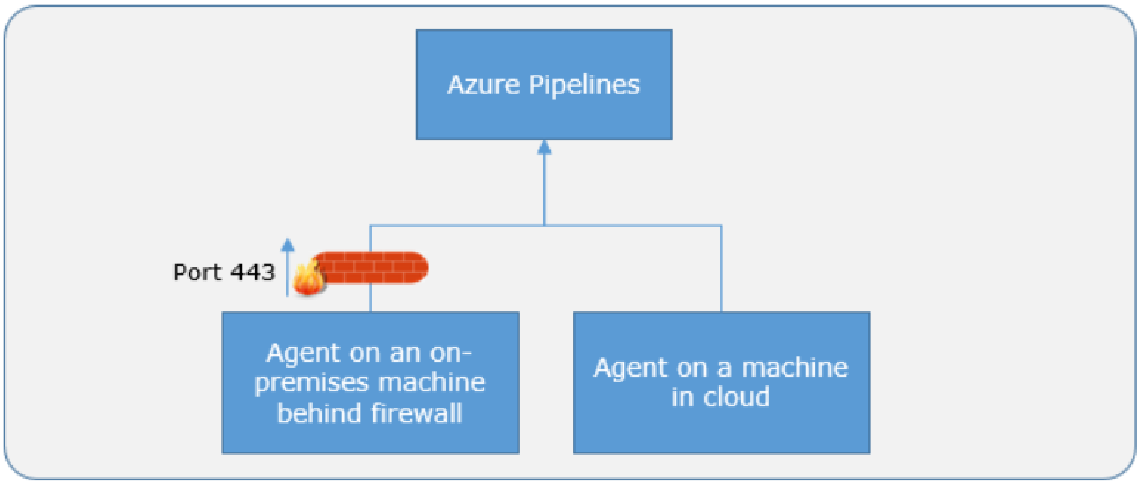
Completed

- 2 minutes

The agent communicates with Azure Pipelines to determine which job to run and reports the logs and job status.

The agent always starts this communication. All the messages from the agent to Azure Pipelines over HTTPS depend on how you configure the agent.

This pull model allows the agent to be configured in different topologies, as shown below.


Azure Pipelines
Port 443
Agent on an on-premises machine behind firewall
Agent on a machine in cloud

Here's a standard communication pattern between the agent and Azure Pipelines.

The user registers an agent with Azure Pipelines by adding it to an agent pool. You must be an agent pool administrator to register an agent. The identity of the agent pool administrator is needed only at the time of registration. It isn't persisted on the agent or used to communicate further between the agent and Azure Pipelines.

Once the registration is complete, the agent downloads a listener OAuth token and uses it to listen to the job queue.

Periodically, the agent checks to see if a new job request has been posted in the job queue in Azure Pipelines. The agent downloads the job and a job-specific OAuth token when a job is available. Azure Pipelines generate this token for the scoped identity specified in the pipeline. That token is short-lived and is used by the agent to access resources (for example, source code) or modify resources (for example, upload test results) on Azure Pipelines within that job.

Once the job is completed, the agent discards the job-specific OAuth token and checks if there's a new job request using the listener OAuth token.

The payload of the messages exchanged between the agent and Azure Pipelines are secured using asymmetric encryption. Each agent has a public-private key pair, and the public key is exchanged with the server during registration.

The server uses the public key to encrypt the job's payload before sending it to the agent. The agent decrypts the job content using its private key. Secrets stored in build pipelines, release pipelines, or variable groups are secured when exchanged with the agent.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Communicate to deploy to target servers

Completed

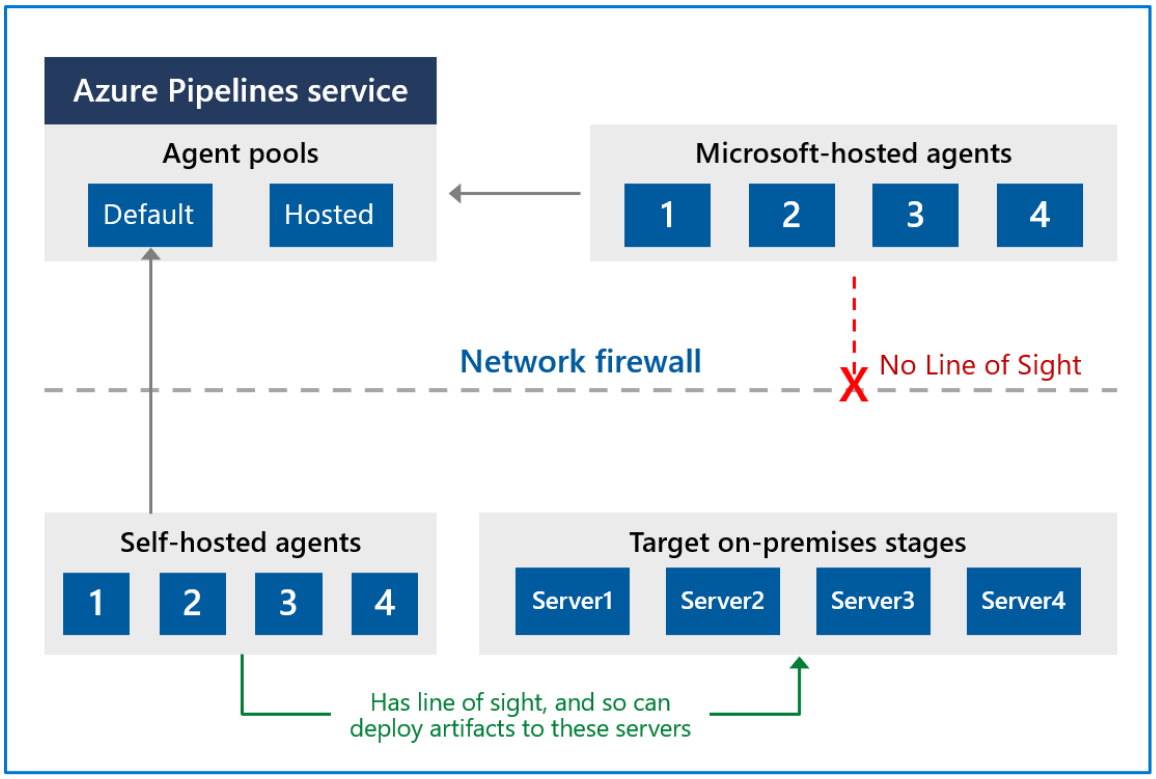
- 1 minute

When you use the agent to deploy artifacts to a set of servers, it must-have "line of sight" connectivity to those servers.

The Microsoft-hosted agent pools, by default, have connectivity to Azure websites and servers running in Azure.

Suppose your on-premises environments don't have connectivity to a Microsoft-hosted agent pool (because of intermediate firewalls). In that case, you'll need to manually configure a self-hosted agent on the on-premises computer(s).

The agents must have connectivity to the target on-premises environments and access to the Internet to connect to Azure Pipelines or Azure DevOps Server, as shown in the following diagram.



Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Examine other considerations

Completed

- 4 minutes

Authentication

To register an agent, you need to be a member of the administrator role in the agent pool.

The identity of the agent pool administrator is only required at the time of registration. It's not persisted on the agent and isn't used in any following communication between the agent and Azure Pipelines.

Also, you must be a local administrator on the server to configure the agent.

Your agent can authenticate to Azure DevOps using one of the following methods:

Personal access token (PAT)

Generate and use a PAT to connect an agent with Azure Pipelines. PAT is the only scheme that works with Azure Pipelines. Also, as explained above, this PAT is used only when registering the agent and not for succeeding communication.

Interactive versus service

You can run your agent as either a service or an interactive process. Whether you run an agent as a service or interactively, you can choose which account you use to run the agent.

It's different from your credentials when registering the agent with Azure Pipelines. The choice of agent account depends solely on the needs of the tasks running in your build and deployment jobs.

For example, to run tasks that use Windows authentication to access an external service, you must run the agent using an account with access to that service.

However, if you're running UI tests such as Selenium or Coded UI tests that require a browser, the browser is launched in the context of the agent account.

After configuring the agent, we recommend you first try it in interactive mode to ensure it works. Then, we recommend running the agent in one of the following modes so that it reliably remains to run for production use. These modes also ensure that the agent starts automatically if the machine is restarted.

You can use the service manager of the operating system to manage the lifecycle of the agent. Also, the experience for auto-upgrading the agent is better when it's run as a service.

As an interactive process with autologon enabled. In some cases, you might need to run the agent interactively for production use, such as UI tests.

When the agent is configured to run in this mode, the screen saver is also disabled.

Some domain policies may prevent you from enabling autologon or disabling the screen saver.

In such cases, you may need to seek an exemption from the domain policy or run the agent on a workgroup computer where the domain policies don't apply.

Note

There are security risks when you enable automatic login or disable the screen saver. You allow other users to walk up to the computer and use the account that automatically logs on. If you configure the agent to run in this way, you must ensure the computer is physically protected; for example, located in a secure facility. If you use Remote Desktop to access the computer on which an agent is running with autologon, simply closing the Remote Desktop causes the computer to be locked, and any UI tests that run on this agent may fail. To avoid this, use the tscon command to disconnect from Remote Desktop.

Agent version and upgrades

Microsoft updates the agent software every few weeks in Azure Pipelines.

The agent version is indicated in the format {major}.{minor}. For instance, if the agent version is 2.1, the major version is 2, and the minor version is 1.

When a newer version of the agent is only different in minor versions, it's automatically upgraded by Azure Pipelines.

This upgrade happens when one of the tasks requires a newer version of the agent.

If you run the agent interactively or a newer major version of the agent is available, you must manually upgrade the agents. Also, you can do it from the agent pools tab under your project collection or organization.

You can view the version of an agent by navigating to Agent pools and selecting the Capabilities tab for the wanted agent.

```
Azure Pipelines: [https://dev.azure.com/{your_organization}/_admin/_AgentPool]
(https://dev.azure.com/{your_organization}/_admin/_AgentPool)
```

Question and Answer

Do self-hosted agents have any performance advantages over Microsoft-hosted agents?

In many cases, yes. Specifically:

- If you use a self-hosted agent, you can run incremental builds. For example, you define a CI build pipeline that doesn't clean the repo or do a clean build. Your builds will typically run faster.
    - You don't get these benefits when using a Microsoft-hosted agent. The agent is destroyed after the build or release pipeline is completed.
- A Microsoft-hosted agent can take longer to start your build. While it often takes just a few seconds for your job to be assigned to a Microsoft-hosted agent, it can sometimes take several minutes for an agent to be allocated, depending on the load on our system.

Can I install multiple self-hosted agents on the same machine?

Yes. This approach can work well for agents who run jobs that don't consume many shared resources. For example, you could try it for agents that run releases that mostly orchestrate deployments and don't do much work on the agent itself.

In other cases, you might find that you don't gain much efficiency by running multiple agents on the same machine.

For example, it might not be worthwhile for agents that run builds that consume many disks and I/O resources.

You might also have problems if parallel build jobs use the same singleton tool deployment, such as npm packages.

For example, one build might update a dependency while another build is in the middle of using it, which could cause unreliable results and errors.

Further instructions on how to set up self-hosted agents can be found at:

- Self-hosted Windows agents .
- Run a self-hosted agent behind a web proxy.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe security of agent pools

Completed

- 4 minutes

Understanding how security works for agent pools helps you control sharing and use of agents.

Azure Pipelines

In Azure Pipelines, roles are defined on each agent pool. Membership in these roles governs what operations you can do on an agent pool.

Note

There are differences between **Organization** and **Project** agent pools.

Role on an organization agent pool	Purpose
Reader	Members of this role can view the organization's agent pool and agents. You typically use it to add operators that are responsible for monitoring the agents and their health.
Service Account	Members of this role can use the organization agent pool to create a project agent pool in a project. If you follow the guidelines above for creating new project agent pools, you typically don't have to add any members here.

Role on an organization agent pool	Purpose
Administrator	Also, with all the above permissions, members of this role can register or unregister agents from the organization's agent pool. They can also refer to the organization agent pool when creating a project agent pool in a project. Finally, they can also manage membership for all roles of the organization agent pool. The user that made the organization agent pool is automatically added to the Administrator role for that pool.

The All agent pools node in the Agent Pools tab is used to control the security of all **organization** agent pools.

Role memberships for individual **organization** agent pools are automatically inherited from the 'All agent pools' node.

Roles are also defined on each organization's agent pool. Memberships in these roles govern what operations you can do on an agent pool.

Role on a project agent pool	Purpose
Reader	Members of this role can view the project agent pool. You typically use it to add operators responsible for monitoring the build and deployment jobs in that project agent pool.
User	Members of this role can use the project agent pool when authoring build or release pipelines.
Administrator	Also, to all the above operations, members of this role can manage membership for all roles of the project agent pool. The user that created the pool is automatically added to the Administrator role for that pool.

The All agent pools node in the Agent pools tab controls the security of all **project** agent pools in a project.

Role memberships for individual **project** agent pools are automatically inherited from the 'All agent pools' node.

By default, the following groups are added to the Administrator role of 'All agent pools': Build Administrators, Release Administrators, Project Administrators.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Configure agent pools and understanding pipeline styles

Completed

- 45 minutes

Estimated time: 45 minutes.

Lab files: none.

Scenario

YAML-based pipelines allow you to fully implement CI/CD as code, in which pipeline definitions reside in the same repository as the code that is part of your Azure DevOps project. YAML-based pipelines support a wide range of features that are part of the classic pipelines, such as pull requests, code reviews, history, branching, and templates.

Regardless of the choice of the pipeline style, to build your code or deploy your solution by using Azure Pipelines, you need an agent. An agent hosts compute resources that run one job at a time. Jobs can be run directly on the host machine of the agent or in a container. You have an option to run your jobs using Microsoft-hosted agents, which are managed for you, or implementing a self-hosted agent that you set up and manage on your own.

In this lab, you will learn how to implement and use self-hosted agents with YAML pipelines.

Objectives

After completing this lab, you'll be able to:

- Implement YAML-based pipelines.
- Implement self-hosted agents.

Requirements

- This lab requires **Microsoft Edge** or an Azure DevOps-supported browser .
- **Set up an Azure DevOps organization:** If you don't already have an Azure DevOps organization that you can use for this lab, create one by following the instructions available at Create an organization or project collection .
- Git for Windows download page. This will be installed as part of the prerequisites for this lab.
- Visual Studio Code . This will be installed as part of the prerequisites for this lab.

Exercises

During this lab, you'll complete the following exercises:

- Exercise 0: Configure the lab prerequisites.
- Exercise 1: Author YAML-based Azure Pipelines.
- Exercise 2: Manage Azure DevOps agent pools.

Launch Exercise

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices are the two types of agents being used in pipelines?

○

Client-Hosted and Server-Hosted.

○

Self-Hosted and Microsoft-Hosted.

○

Windows-Hosted and Linux-Hosted.

2.

Which of the following is a correct statement about Agent Pools?

○

Agent pools are scoped to the entire project and can be shared across pipelines.

○

Agent pools are scoped to the entire repository and can be shared across pipelines.

○

Agent pools are scoped to the entire organization and can be shared across projects.

3.

Which of the following choices is the role that can manage membership for all roles of the project agent pool?

○

Administrator.

○

User.

○

Contributor.

Check your answers

You must answer all questions before checking your work.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

This module explored differences between Microsoft-hosted and self-hosted agents, detailed job types, and introduced agent pools configuration.

You learned how to describe the benefits and usage of:

- Choose between Microsoft-hosted and self-hosted agents.
- Install and configure Azure Pipelines Agents.
- Configure agent pools.
- Make the agents and pools secure.

Learn more

- Microsoft-hosted agents for Azure Pipelines - Azure Pipelines | Microsoft Docs .
- Deploy an Azure Pipelines agent on Linux - Azure Pipelines | Microsoft Docs .
- Deploy a build and release agent on macOS - Azure Pipelines | Microsoft Docs .
- Deploy an Azure Pipelines agent on Windows - Azure Pipelines | Microsoft Docs .
- Agents pools - Azure Pipelines | Microsoft Docs .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe pipelines and concurrency

Introduction

Completed

- 1 minute

This module describes parallel jobs and how to estimate their usage. Also, it presents Azure DevOps for open-source projects, explores Visual Designer and YAML pipelines.

Learning objectives

After completing this module, students and professionals can:

- Use and estimate parallel jobs.
- Use Azure Pipelines for open-source or private projects.
- Use Visual Designer.
- Work with Azure Pipelines and YAML.

Prerequisites

- Understanding of what DevOps is and its concepts.
- Familiarity with version control principles is helpful but is not necessary.
- Beneficial to have experience in an organization that delivers software.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Understand parallel jobs

Completed

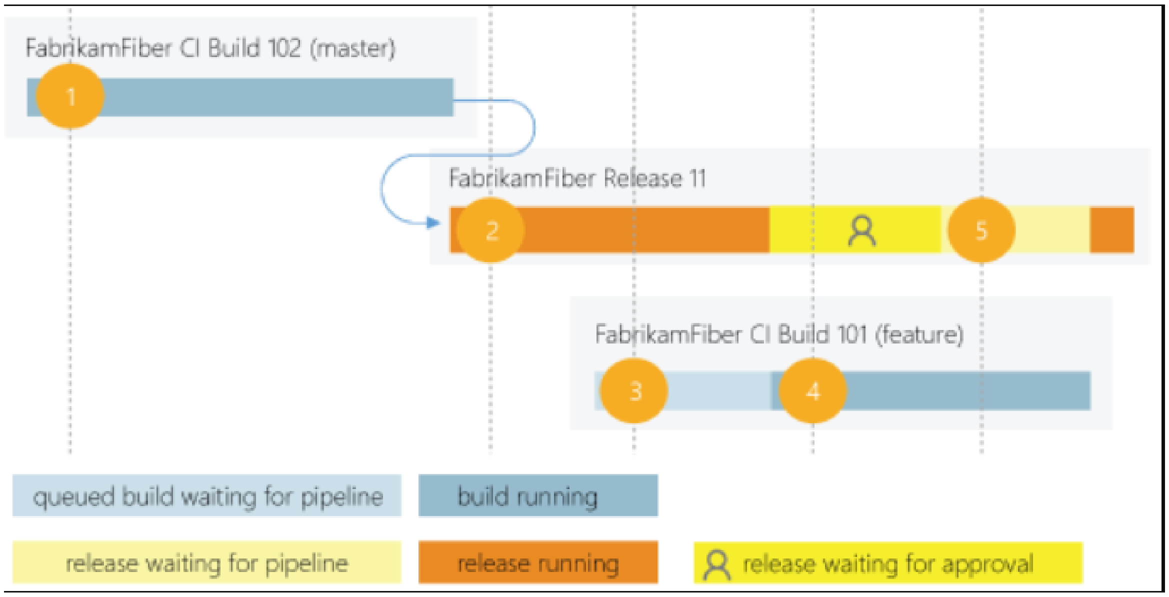
- 2 minutes

How a parallel job is consumed by a build or release

Consider an organization that has only one Microsoft-hosted parallel job.

This job allows users in that organization to collectively run only one build or release job at a time.

When more jobs are triggered, they're queued and will wait for the previous job to finish.


FabrikamFiber CI Build 102 (master)
1
FabrikamFiber Release 11
2
5
FabrikamFiber CI Build 101 (feature)
3
4
queued build waiting for pipeline
build running
release waiting for pipeline
release running
release waiting for approval

A release consumes a parallel job only when it's being actively deployed to a stage.

While the release is waiting for approval or manual intervention, it doesn't consume a parallel job.

A simple example of parallel jobs

- FabrikamFiber CI Build 102 (main branch) starts first.
- Deployment of FabrikamFiber Release 11 is triggered by the completion of FabrikamFiber CI Build 102.
- FabrikamFiber CI Build 101 (feature branch) is triggered. The build can't start yet because Release 11's deployment is active. So, the build stays queued.
- Release 11 waits for approvals. Fabrikam CI Build 101 starts because a release waiting for approvals doesn't consume a parallel job.
- Release 11 is approved. It resumes only after Fabrikam CI Build 101 is completed.

Relationship between jobs and parallel jobs

The term job can refer to multiple concepts, and its meaning depends on the context:

- When you define a build or release, you can define it as a collection of jobs. When a build or release runs, you can run multiple jobs as part of that build or release.
- Each job consumes a parallel job that runs on an agent. When there aren't enough parallel jobs available for your organization, then the jobs are queued up and run one after the other.

You don't consume any parallel jobs when you run a server job or deploy to a deployment group.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Estimate parallel jobs

Completed

- 3 minutes

Determine how many parallel jobs you need

You could begin by seeing if the free tier offered in your organization is enough for your teams.

When you've reached the 1,800 minutes per month limit for the free tier of Microsoft-hosted parallel jobs, you can start by buying one parallel job to remove this monthly time limit before deciding to purchase more.

As the number of queued builds and releases exceeds the number of parallel jobs you have, your build and release queues will grow longer.

When you find the queue delays are too long, you can purchase extra parallel jobs as needed.

Simple estimate

A simple rule of thumb: Estimate that you'll need one parallel job for every four to five users in your organization.

Detailed estimate

In the following scenarios, you might need multiple parallel jobs:

- If you have multiple teams, and if each of them requires a CI build, you'll likely need a parallel job for each team.
- If your CI build trigger applies to multiple branches, you'll likely need a parallel job for each active branch.
- If you develop multiple applications by using one organization or server, you'll likely need more parallel jobs: one to deploy each application simultaneously.

View available parallel jobs

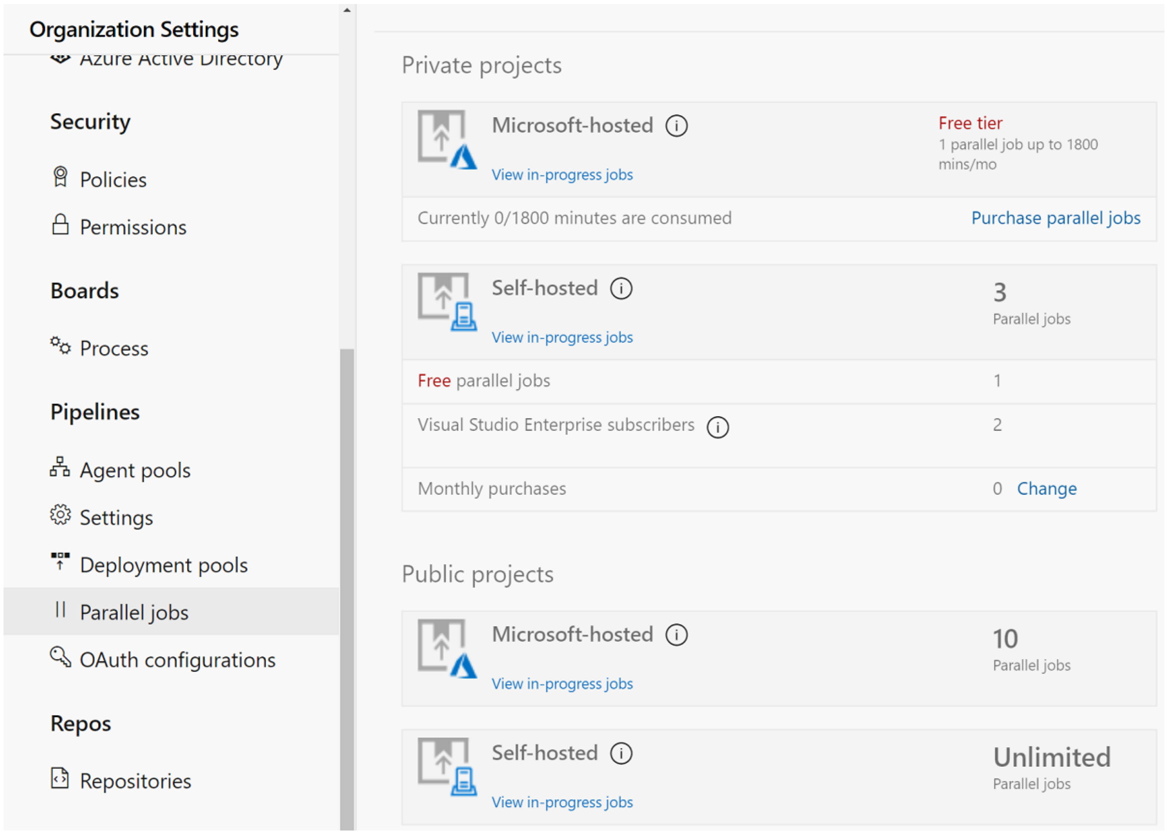
Browse to `Organization settings > Pipelines > Parallel jobs.`

Location of parallel jobs in organization settings

URL example: `https://{your_organization}/_settings/buildqueue?_a=concurrentJobs`

View the maximum number of parallel jobs that are available in your organization.

Select View in-progress jobs to display all the builds and releases that are actively consuming an available parallel job or queued waiting for a parallel job to be available.



Private projects

Microsoft-hosted — View in-progress jobs	Free tier — 1 parallel job up to 1800 mins/mo
Currently 0/1800 minutes are consumed	Purchase parallel jobs

Self-hosted — View in-progress jobs	3 Parallel jobs
Free parallel jobs	1
Visual Studio Enterprise subscribers	2
Monthly purchases	0 Change

Public projects

Microsoft-hosted — View in-progress jobs	10 Parallel jobs
Self-hosted — View in-progress jobs	Unlimited Parallel jobs

Sharing of parallel jobs across projects in a collection

Parallel jobs are purchased at the organization level, and they're shared by all projects in an organization.

Currently, there isn't a way to partition or dedicate parallel job capacity to a specific project or agent pool. For example:

- You purchase two parallel jobs in your organization.
- You queue two builds in the first project, and both the parallel jobs are consumed.
- You queue a build in the second project. That build won't start until one of the builds in your first project is completed.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe Azure Pipelines and open-source projects

Completed

- 4 minutes

Azure DevOps offers developers a suite of DevOps capabilities, including Source control, Agile planning, Build, Release, Test, and more.

But to use Azure DevOps features requires the user to first sign in using a Microsoft or GitHub Account.

However, this blocks many engaging scenarios where you want to publicly share your code and artifacts or provide a wiki library or build status page for unauthenticated users.

With public projects, users can mark an Azure DevOps Team Project as public.

This will enable anonymous users to view the contents of that project in a read-only state enabling collaboration with anonymous (unauthenticated) users that wasn't possible before.

Anonymous users will essentially see the same views as authenticated users, with non-public functionality such as settings or actions (such as queue build) hidden or disabled.

Public versus private projects

Projects in Azure DevOps provide a repository for source code and a place for a group of developers and teams to plan, track progress, and collaborate on building software solutions.

One or more projects can be defined within an organization in Azure DevOps.

Users that aren't signed into the service have read-only access to public projects on Azure DevOps.

Private projects require users to be granted access to the project and signed in to access the services.

Supported services

Non-members of a public project will have read-only access to a limited set of services, precisely:

- Browse the code base, download code, view commits, branches, and pull requests.
- View and filter work items.
- View a project page or dashboard.
- View the project Wiki.
- Do a semantic search of the code or work items.

For more information, see Differences and limitations for non-members of a public project
.

A practical example: .NET Core CLI

Supporting open-source development is one of the most compelling scenarios for public projects. A good example is the .NET Core CLI.

Their source is hosted on GitHub, and they use Azure DevOps for their CI builds.

However, if you click on the build badges in their readme, you'll not see the build results unless you were one of the project's maintainers.

Since this is an open-source project, everybody should view the full results to see why a build failed and maybe even send a pull request to help fix it.

Thanks to public projects capabilities, the team will enable just that experience. Everyone in the community will have access to the same build results, whether they are a maintainer on the project.

How do I qualify for the free tier of Azure Pipelines for public projects?

Microsoft will automatically apply the free tier limits for public projects if you meet both conditions:

- Your pipeline is part of an Azure Pipelines public project.
- Your pipeline builds a public repository from GitHub or the same public project in your Azure DevOps organization.

Are there limits on who can use Azure Pipelines?

You can have as many users as you want when you're using Azure Pipelines. There's no per-user charge for using Azure Pipelines.

Users with both basic and stakeholder access can author as many builds and releases as they want.

Are there any limits on the number of builds and release pipelines that I can create?

No. You can create hundreds or even thousands of definitions for no charge. You can register any number of self-hosted agents for no cost.

As a Visual Studio Enterprise subscriber, do I get more parallel jobs for Azure Pipelines?

Yes. Visual Studio Enterprise subscribers get one self-hosted parallel job in each Azure DevOps Services organization where they're a member.

When you're using the per-minute plan, you can run only one job at a time.

If you run builds for more than 14 paid hours in a month, the per-minute plan might be less cost-effective than the parallel jobs model.

See Azure DevOps Services Pricing | Microsoft Azure for current pricing.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore Azure Pipelines and Visual Designer

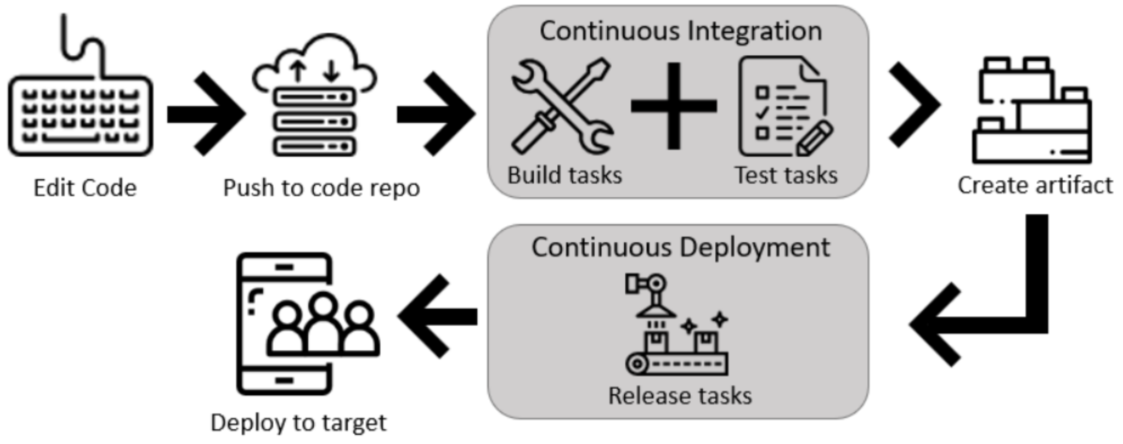
Completed

- 2 minutes

You can create and configure your build and release pipelines in the Azure DevOps web portal with the visual designer. (Often referred to as "Classic Pipelines").

Configure Azure Pipelines to use your Git repo.

1. Use the Azure Pipelines visual designer to create and configure your build and release pipelines.
2. Push your code to your version control repository. This action triggers your pipeline and runs tasks such as building or testing code.
3. The build creates an artifact used by the rest of your pipeline to run tasks such as deploying to staging or production.
4. Your code is now updated, built, tested, and packaged. It can be deployed to any target.



Benefits of using the Visual Designer

The visual designer is great for new users in continuous integration (CI) and continuous delivery (CD).

- The visual representation of the pipelines makes it easier to get started.
- The visual designer is in the same hub as the build results. This location makes it easier to switch back and forth and make changes.

If you think the designer workflow is best for you, create your first pipeline using the visual designer .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe Azure Pipelines and YAML

Completed

- 3 minutes

Mirroring the rise of interest in infrastructure as code, there has been considerable interest in defining pipelines as code. However, pipeline as code doesn't mean executing a script that's stored in source control.

Codified pipelines use their programming model to simplify the setup and maximize reuse.

A typical microservice architecture will require many deployment pipelines that are identical. It's tedious to craft these pipelines via a user interface or SDK.

The ability to define the pipeline and the code helps apply all principles of code sharing, reuse, templatization, and code reviews. Azure DevOps offers you both experiences. You can either use YAML to define your pipelines or use the visual designer to do the same. You will, however, find that more product-level investments are being made to enhance the YAML pipeline experience.

When you use YAML, you define your pipeline mostly in code (a YAML file) alongside the rest of the code for your app. When using the visual designer, you define a build pipeline to build and test your code and publish artifacts.

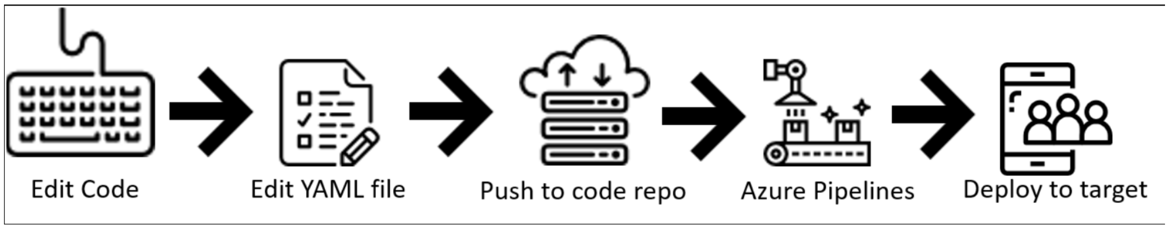
You also specify a release pipeline to consume and deploy those artifacts to deployment targets.

Use Azure Pipelines with YAML

You can configure your pipelines in a YAML file that exists alongside your code.

1. Configure Azure Pipelines to use your Git repo.

2. Edit your azure-pipelines.yml file to define your build.
3. Push your code to your version control repository. This action kicks off the default trigger to build and deploy and then monitor the results.
4. Your code is now updated, built, tested, and packaged. It can be deployed to any target.



Benefits of using YAML

- The pipeline is versioned with your code and follows the same branching structure. You get validation of your changes through code reviews in pull requests and branch build policies.
- Every branch you use can modify the build policy by adjusting the azure-pipelines.yml file.
- A change to the build process might cause a break or result in an unexpected outcome. Because the change is in version control with the rest of your codebase, you can more easily identify the issue.

If you think the YAML workflow is best for you, create your first pipeline by using YAML .

While there's a slightly higher learning curve and a higher degree of code orientation when defining pipelines with YAML, it's now the preferred method.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices isn't used to create and configure an Azure Pipeline?

○

XML.

○

YAML.

○

Visual Designer.

2.

Which of the following choices is a benefit of using the Visual Designer?

○

Every branch you use can modify the build policy by modifying the azure-pipelines.yml file.

○

The visual designer is in the same hub as the build results.

○

A change to the build process might cause a break or result in an unexpected outcome. Because the change is in version control with the rest of your codebase, you can more easily identify the issue.

3.

Which of the following choices is a benefit of using YAML?

○

The pipeline is versioned with your code and follows the same branching structure.

○

The YAML representation of the pipelines makes it easier to get started.

○

The YAML is in the same hub as the build results.

Check your answers

You must answer all questions before checking your work.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

This module described parallel jobs and how to estimate their usage. Also, it presented Azure DevOps for open-source projects, explored Visual Designer and YAML pipelines.

You learned how to describe the benefits and usage of:

- Use and estimate parallel jobs.
- Use Azure Pipelines for open-source or private projects.
- Use Visual Designer.
- Work with Azure Pipelines and YAML.

Learn more

- Configure and pay for parallel jobs .
- Azure DevOps Services Pricing | Microsoft Azure .
- What is a public project? .
- Create your first pipeline .
- Customize your pipeline .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore continuous integration

Introduction

Completed

- 2 minutes

Continuous Integration is one of the key pillars of DevOps.

Once you have your code in a version control system, you need an automated way of integrating the code on an ongoing basis.

Azure Pipelines can be used to create a fully featured cross-platform CI and CD service.

It works with your preferred Git provider and can deploy to most major cloud services, including Azure.

This module details continuous integration practice and the pillars for implementing it in the development lifecycle, its benefits, and properties.

Learning objectives

After completing this module, students and professionals can:

- Explain why Continuous Integration matters.
- Implement Continuous Integration using Azure Pipelines.
- Explain the benefits of Continuous Integration.
- Describe build properties.

Prerequisites

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

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Introduction to continuous integration

Completed

- 3 minutes

Continuous integration (CI) is the process of automating the build and testing of code every time a team member commits changes to version control.

CI encourages developers to share their code and unit tests by merging their changes into a shared version control repository after every small task completion.

Committing code triggers an automated build system to grab the latest code from the shared repository and build, test, and validate the entire main branch (also known as the trunk or main).

The idea is to minimize the cost of integration by making it an early consideration.

Developers can discover conflicts at the boundaries between new and existing code early, while conflicts are still relatively easy to reconcile.

Once the conflict is resolved, work can continue with confidence that the new code honors the requirements of the existing codebase.

Integrating code frequently doesn't offer any guarantees about the quality of the new code or functionality.

In many organizations, integration is costly because manual processes ensure that the code meets standards, introduces bugs, and breaks existing functionality.

Frequent integration can create friction when the level of automation doesn't match the amount of quality assurance measures in place.

In practice, continuous integration relies on robust test suites and an automated system to run those tests to address this friction within the integration process.

When a developer merges code into the main repository, automated processes kick off a build of the new code.

Afterward, test suites are run against the new build to check whether any integration problems were introduced.

If either the build or the test phase fails, the team is alerted to work to fix the build.

The end goal of continuous integration is to make integration a simple, repeatable process part of the everyday development workflow to reduce integration costs and respond to early defects.

Working to make sure the system is robust, automated, and fast while cultivating a team culture that encourages frequent iteration and responsiveness to build issues is fundamental to the strategy's success.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Learn the four pillars of continuous integration

Completed

- 4 minutes

Continuous integration relies on four key elements for successful implementation: a Version Control System, Package Management System, Continuous Integration System, and an Automated Build Process.

A **version control system** manages changes to your source code over time.

- Git
- Apache Subversion
- Team Foundation Version Control

A **package management system** install uninstalls, and manages software packages.

- NuGet
- Node Package Manager (npm)
- Chocolatey
- HomeBrew
- RPM

A **continuous integration system** merges all developer working copies into a shared mainline several times daily.

- Azure DevOps

- TeamCity
- Jenkins

An **automated build process** creates a software build, including compiling, packaging, and running automated tests.

- Apache Ant
- NAnt2
- Gradle

Note

Your team needs to select the specific platforms and tools they'll use for each element. You need to ensure that you've established each pillar before proceeding.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore benefits of continuous integration

Completed

- 3 minutes

Continuous integration (CI) provides many benefits to the development process, including:

- Improving code quality based on rapid feedback
- Triggering automated testing for every code change
- Reducing build times for quick feedback and early detection of problems (risk reduction)
- Better managing technical debt and conducting code analysis
- Reducing long, complex, and bug-inducing merges
- Increasing confidence in codebase health long before production deployment

Key Benefit: Rapid Feedback for Code Quality

Possibly the most essential benefit of continuous integration is rapid feedback to the developer.

If the developer commits something and breaks the code, they'll know that immediately from the build, unit tests, and other metrics.

Suppose successful integration is happening across the team.

In that case, the developer will also know if their code change breaks something that another team member did to a different part of the codebase.

This process removes long, complex, and drawn-out bug-inducing merges, allowing organizations to deliver swiftly.

Continuous integration also enables tracking metrics to assess code quality over time. For example, unit test passing rates, code that breaks frequently, code coverage trends, and code analysis.

It can provide information on what has been changed between builds for traceability benefits. Also, introduce evidence of what teams do to have a global view of build results.

For more information, you can see: What is Continuous Integration?

CI implementation challenges

- Have you tried to implement continuous integration in your organization?
- Were you successful?
- If you did successfully, what lessons did you learn?
- If you didn't get successful, what were the challenges?

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe build properties

Completed

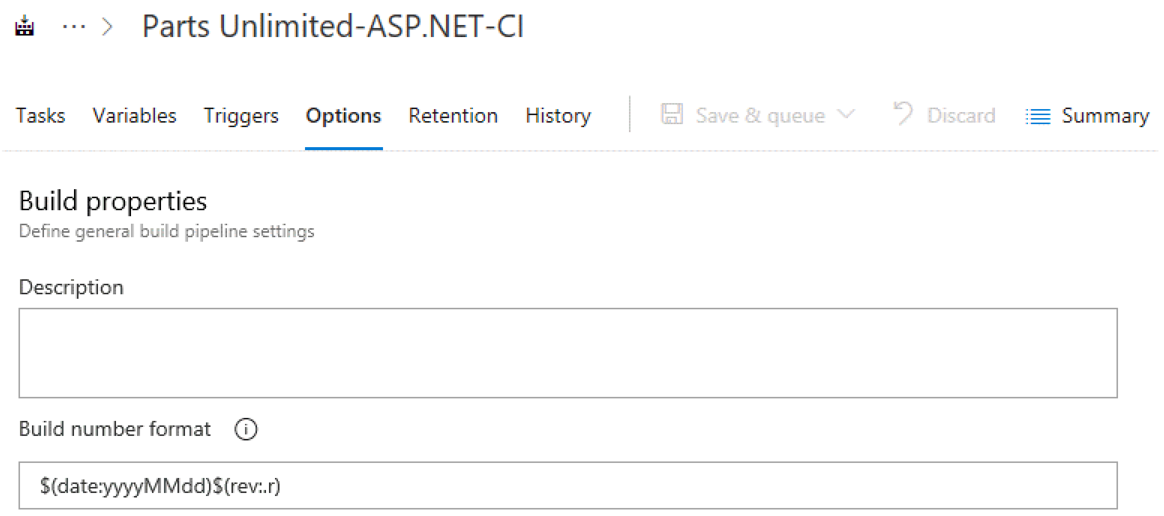
- 4 minutes

You may have noticed that in some demos, the build number was just an integer, yet in other demos, there's a formatted value that was based upon the date.

It is one of the items that can be set in the **Build Options** .

Build number formatting

The example shown below is from the build options that were configured by the ASP.NET Web Application build template:


Parts Unlimited-ASP.NET-CI

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

Build properties

Define general build pipeline settings

Description

Build number format

$(date:yyyyMMdd)$(rev:.r)

In this case, the date has been retrieved as a system variable, then formatted via yyyyMMdd, and the revision is then appended.

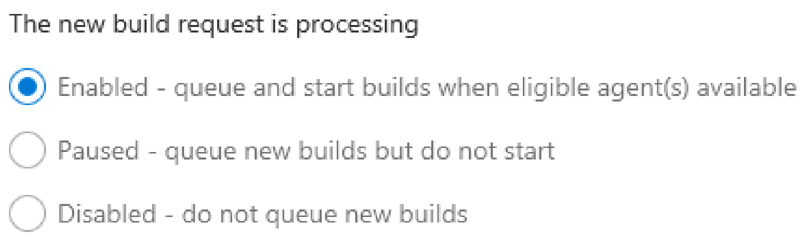
Build status

While we have been manually queuing each build, we'll soon see that builds can be automatically triggered.

It's a key capability required for continuous integration.

But there are times that we might not want the build to run, even if it's triggered.

It can be controlled with these settings:


The new build request is processing

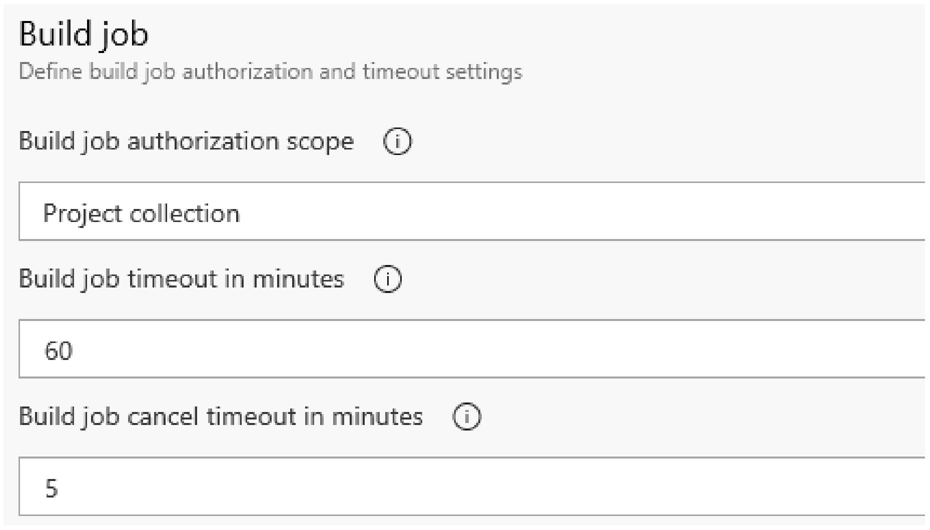
- [✓ Enabled - queue and start builds when eligible agent(s) available]
- Paused - queue new builds but do not start
- Disabled - do not queue new builds

Note

You can use the **Paused** setting to allow new builds to queue but to hold off then starting them.

Authorization and timeouts

You can configure properties for the build job as shown here:


Build job

Define build job authorization and timeout settings

Build job authorization scope: Project collection

Build job timeout in minutes: 60

Build job cancel timeout in minutes: 5

The authorization scope determines whether the build job is limited to accessing resources in the current project. Or accessing resources in other projects in the project collection.

The build job timeout determines how long the job can execute before being automatically canceled.

A value of zero (or leaving the text box empty) specifies that there's no limit.

The build job cancel timeout determines how long the server will wait for a build job to respond to a cancellation request.

Badges

Some development teams like to show the state of the build on an external monitor or website.

These settings provide a link to the image to use for it. Here's an example Azure Pipelines badge that has Succeeded:

Azure Pipelines
succeeded

For more information, see Build Pipeline Options .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Enable Continuous Integration with Azure Pipelines

Completed

- 45 minutes

Estimated time: 45 minutes.

Lab files: none.

Scenario

In this lab, you will learn how to define build pipelines in Azure DevOps using YAML. The pipelines will be used in two scenarios:

- As part of Pull Request validation process.
- As part of the Continuous Integration implementation.

Objectives

After completing this lab, you'll be able to:

- Include build validation as part of a Pull Request.
- Configure CI pipeline as code with YAML.

Requirements

- This lab requires **Microsoft Edge** or an Azure DevOps-supported browser .
- **Set up an Azure DevOps organization:** If you don't already have an Azure DevOps organization that you can use for this lab, create one by following the instructions available at Create an organization or project collection .

Exercises

During this lab, you'll complete the following exercises:

- Exercise 0: Configure the lab prerequisites.
- Exercise 1: Include build validation as part of a Pull Request.
- Exercise 2: Configure CI Pipeline as Code with YAML.

Launch Exercise

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 5 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices isn't a pillar of Continuous Integration?

○

Version Control System.

○

Automated Build Process.

○

Automated Deploy Process.

2.

Which of the following choices describe the status of a build pipeline that will queue new build requests and not start them?

○

Disabled.

○

Paused.

○

On Hold.

3.

Which of the following choices is where you change the build number format?

○

Build properties.

○

Library.

○

Project Settings.

Check your answers

You must answer all questions before checking your work.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

This module detailed the Continuous Integration practice—also the pillars for implementing it in the development lifecycle, its benefits, and its properties.

You learned how to describe the benefits and usage of:

- Explain why Continuous Integration matters.
- Implement Continuous Integration using Azure Pipelines.
- Explain the benefits of Continuous Integration.
- Describe build properties.

Learn more

- Design a CI/CD pipeline using Azure DevOps - Azure Example Scenarios | Microsoft Docs .
- Build options - Azure Pipelines | Microsoft Docs .
- Create your first pipeline - Azure Pipelines | Microsoft Docs .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Implement a pipeline strategy

Introduction

Completed

- 1 minute

This module describes pipeline strategies, configuration, implementation of multi-agent builds, and what source controls Azure Pipelines supports.

Learning objectives

After completing this module, students and professionals can:

- Define a build strategy.
- Explain and configure demands.
- Implement multi-agent builds.
- Use different source control types available in Azure Pipelines.

Prerequisites

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

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Configure agent demands

Completed

- 2 minutes

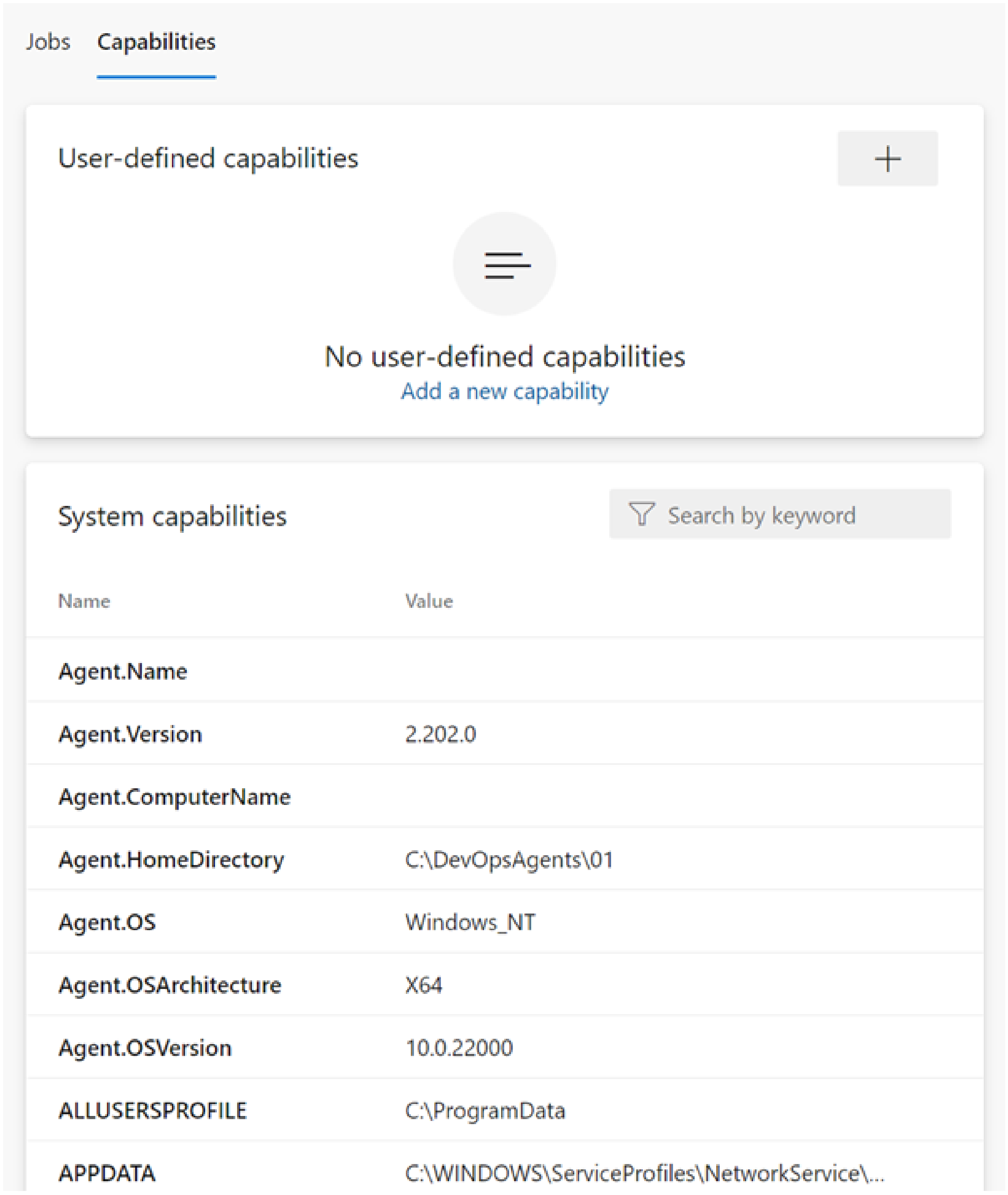
Not all agents are the same. We've seen that they can be based on different operating systems, but they can also install different dependencies.

To describe it, every agent has a set of capabilities configured as name-value pairs. The capabilities such as machine name and operating system type that are automatically discovered are referred to as **System capabilities** . The ones that you define are called **User-defined capabilities** .

There's a tab for Capabilities on the Agent Pools page (at the Organization level) **when you select an agent** .

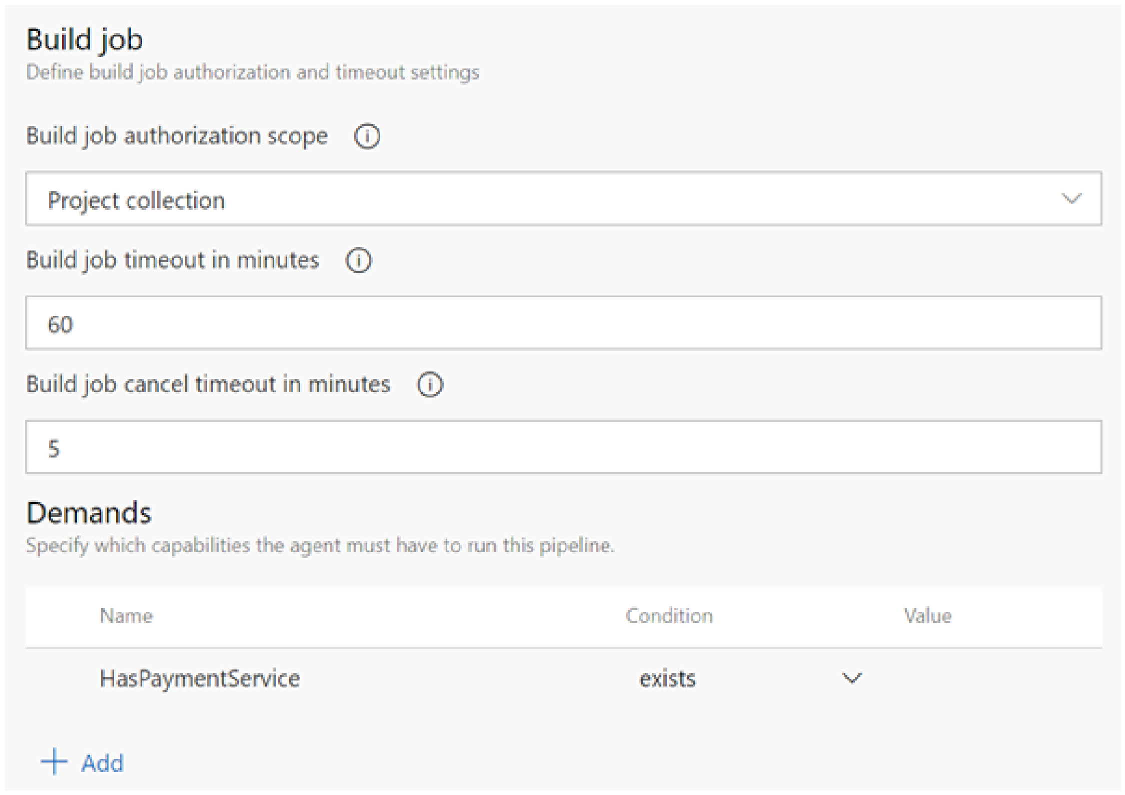
You can use it to see the available capabilities for an agent and to configure user capabilities.

Opening a configured self-hosted agent, you can see the capabilities on that tab:



Name	Value
Agent.Name	
Agent.Version	2.202.0
Agent.ComputerName	
Agent.HomeDirectory	C:\DevOpsAgents\01
Agent.OS	Windows_NT
Agent.OSArchitecture	X64
Agent.OSVersion	10.0.22000
ALLUSERSPROFILE	C:\ProgramData
APPDATA	C:\WINDOWS\ServiceProfiles\NetworkService\...

When you configure a build pipeline and the agent pool to use, you can specify specific demands that the agent must meet on the Options tab.


Build job

Define build job authorization and timeout settings

Build job authorization scope: Project collection

Build job timeout in minutes: 60

Build job cancel timeout in minutes: 5

Demands

Specify which capabilities the agent must have to run this pipeline.

Name	Condition	Value
HasPaymentService	exists	

In the build job image, the HasPaymentService is required in the collection of capabilities. And an **exists** condition, you can choose that a capability **equals** a specific value.

For more information, see Capabilities .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Implement multi-agent builds

Completed

- 3 minutes

You can use multiple build agents to support multiple build machines. Either distribute the load, run builds in parallel, or use different agent capabilities.

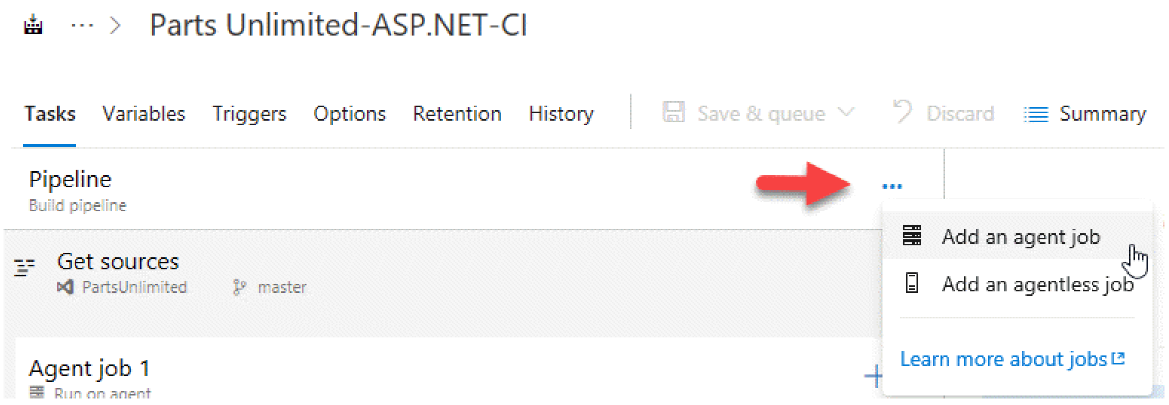
As an example, components of an application might require different incompatible versions of a library or dependency.

Multiple jobs in a pipeline

Adding multiple jobs to a pipeline lets you:

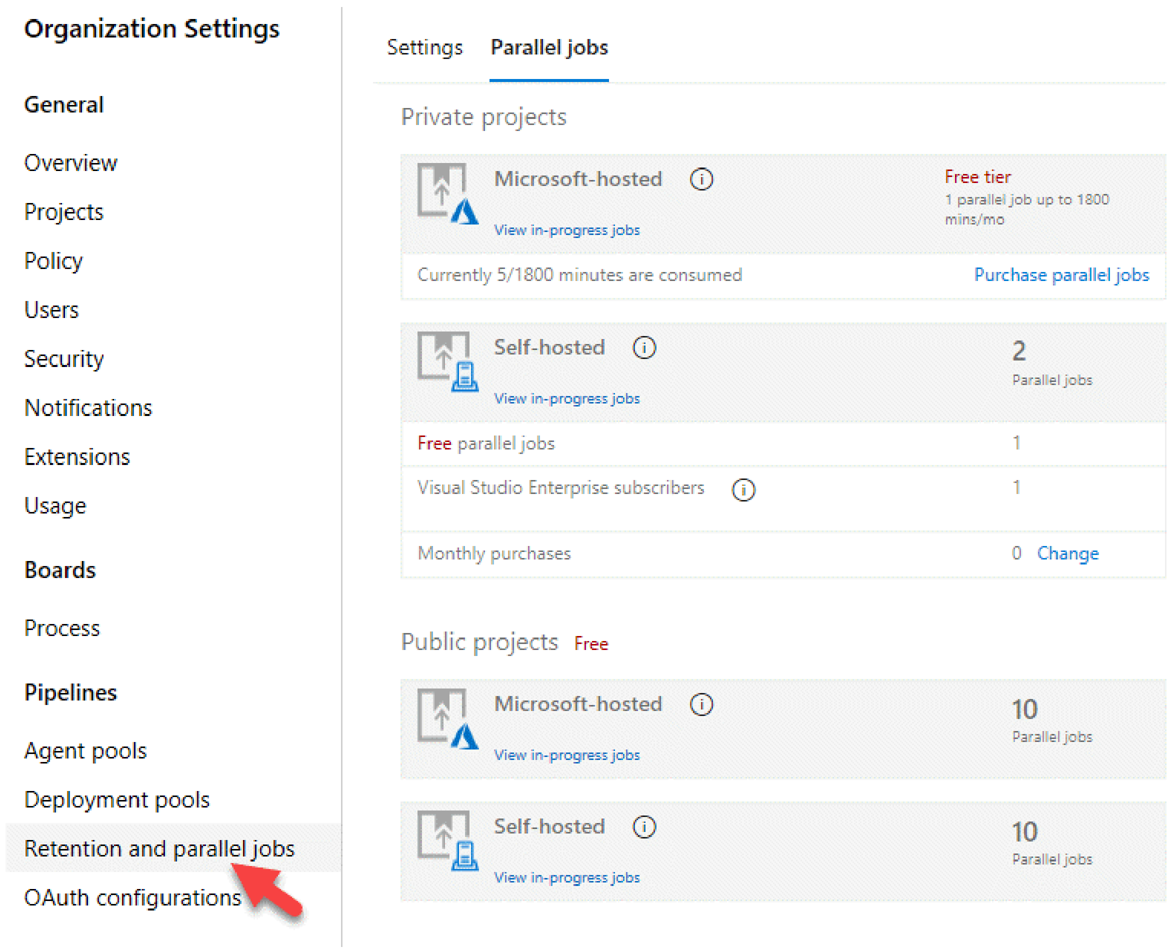
- Break your pipeline into sections that need different agent pools or self-hosted agents.
- Publish artifacts in one job and consume them in one or more subsequent jobs.
- Build faster by running multiple jobs in parallel.
- Enable conditional execution of tasks.

To add another agent job to an existing pipeline, click on the ellipsis and choose as shown in this image:



Parallel jobs

At the organization level, you can configure the number of parallel jobs that are made available.



Private projects

Microsoft-hosted
View in-progress jobs
Free tier
1 parallel job up to 1800 mins/mo
Currently 5/1800 minutes are consumed
Purchase parallel jobs

Self-hosted
View in-progress jobs
2 Parallel jobs

Free parallel jobs	1
Visual Studio Enterprise subscribers	1
Monthly purchases	0 Change

Public projects Free

Microsoft-hosted
View in-progress jobs
10 Parallel jobs

Self-hosted
View in-progress jobs
10 Parallel jobs

The free tier allows for one parallel job of up to 1800 minutes per month. The self-hosted agents have higher levels.

Note

You can define a build as a collection of jobs rather than as a single job. Each job consumes one of these parallel jobs that run on an agent. If there aren't enough parallel jobs available for your organization, the jobs will be queued and run sequentially.

Build Related Tooling

Azure DevOps can be integrated with a wide range of existing tooling used for builds or associated with builds.

Which build-related tools do you currently use?

What do you like or don't like about the tools?

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore source control types supported by Azure Pipelines

Completed

- 1 minute

Azure Pipelines offers both YAML-based pipelines and the classic editor.

The table shows the repository types supported by both.

Repository type	**Azure Pipelines (YAML)**	**Azure Pipelines (classic editor)**
Azure Repos Git	Yes	Yes
Azure Repos TFVC	No	Yes
Bitbucket Cloud	Yes	Yes
Other Git (generic)	No	Yes
GitHub	Yes	Yes
GitHub Enterprise Server		

Yes

Yes

Subversion

No

Yes

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices could be a user capability?

○

Agent.ComputerName.

○

Agent.OS.

○

ContosoApplication.Path.

2.

Which of the following choices is the scope of where parallel jobs are defined?

○

Project scope.

○

Organization scope.

○

Pipeline scope.

3.

Which of the following choices of repository types does a YAML pipeline supports?

○

Azure Repos TFVC.

○

Azure Repos Git.

○

Other Git (generic).

Check your answers

You must answer all questions before checking your work.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

This module described pipeline strategies, configuration, implementation of multi-agent builds, and what source controls Azure Pipelines supports.

You learned how to describe the benefits and usage of:

- Define a build strategy.
- Explain and configure demands.
- Implement multi-agent builds.
- Use different source control types available in Azure Pipelines.

Learn more

- Create a multi-platform pipeline .
- Specify demands .
- Supported source repositories .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Integrate with Azure Pipelines

Introduction

Completed

- 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.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe the anatomy of a pipeline

Completed

- 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.

```
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:

```
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:

```
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?

```
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 .

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:

```
jobs:

- job: A
  steps:
  # steps omitted for brevity


- job: B
  steps:
  # steps omitted for brevity
```

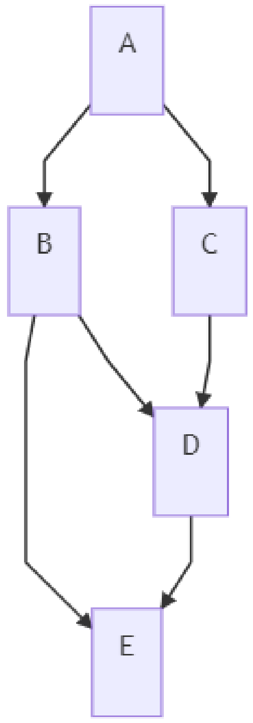
Because no dependsOn was specified, the jobs will run sequentially: first A and then B.

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

```
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:

```
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:

```
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 .

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:

```
variables:
  name: John
steps:
```

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

It will write Hello, John! To the log.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Understand the pipeline structure

Completed

- 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:

```
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:

```
# ... 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:

```
# ... 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.

```
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.

```
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.

```
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 .

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.

```
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 .

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.

```
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 .

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 .

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 .
- **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.

```
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.

```
steps:

- task: VSBuild@1
  displayName: Build
  timeoutInMinutes: 120
  inputs:
    solution: '**\*.sln'
```

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Detail templates

Completed

- 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.

```
# 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

```
# 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.

```
# 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


# 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.

```
# File: steps/build.yml

steps:

- script: npm install
- script: npm test


# 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.

```
# File: variables/build.yml
variables:

- name: vmImage
  value: windows-latest

- name: arch
  value: x64

- name: config
```

```
  value: debug


# File: component-x-pipeline.yml
variables:

- template: variables/build.yml  # Template reference
pool:
  vmImage: ${{ variables.vmImage }}
steps:

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


# File: component-y-pipeline.yml
variables:

- template: variables/build.yml  # Template reference
pool:
  vmImage: ${{ variables.vmImage }}
steps:

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

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore YAML resources

Completed

- 2 minutes

Resources in YAML represent sources of pipelines, repositories, and containers. For more information on Resources, see here .

General schema

```
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.

```
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.

```
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.

```
resources:
  repositories:

  - repository: common
    type: github
    name: Contoso/CommonTools
    endpoint: MyContosoServiceConnection
```

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Use multiple repositories in your pipeline

Completed

- 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.

```
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.

```
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:

```
- 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.

```
trigger:
    - main
    - releases/*
```

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

```
# 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:

```
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 .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

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

○

steps.

○

dependsOn.

○

condition.

2.

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

○

Service Hook.

○

Azure Pipelines.

○

Agent.

3.

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

○

To run UI Tests.

○

To run Java pipelines.

○

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

Check your answers

You must answer all questions before checking your work.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 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 Docs .
- Azure Pipelines YAML pipeline editor guide - Azure Pipelines | Microsoft Docs .
- Check out multiple repositories in your pipeline - Azure Pipelines | Microsoft Docs .
- Azure Pipelines Agents - Azure Pipelines | Microsoft Docs .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Introduction to GitHub Actions

Introduction

Completed

- 1 minute

GitHub Actions are the primary mechanism for automation within GitHub.

They can be used for a wide variety of purposes, but one of the most common is to implement Continuous Integration.

In this module, you will learn what GitHub Actions, action flow, and its elements are. Understand what events are, explore jobs and runners, and how to read console output from actions.

Learning objectives

After completing this module, students and professionals can:

- Explain GitHub Actions and workflows.
- Create and work with GitHub Actions and Workflows.
- Describe Events, Jobs, and Runners.
- Examine the output and release management for actions.

Prerequisites

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

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

What are Actions?

Completed

- 1 minute

Actions are the mechanism used to provide workflow automation within the GitHub environment.

They're often used to build continuous integration (CI) and continuous deployment (CD) solutions.

However, they can be used for a wide variety of tasks:

- Automated testing.
- Automatically responding to new issues, mentions.
- Triggering code reviews.
- Handling pull requests.
- Branch management.

They're defined in YAML and stay within GitHub repositories.

Actions are executed on "runners," either hosted by GitHub or self-hosted.

Contributed actions can be found in the GitHub Marketplace. See Marketplace Actions .

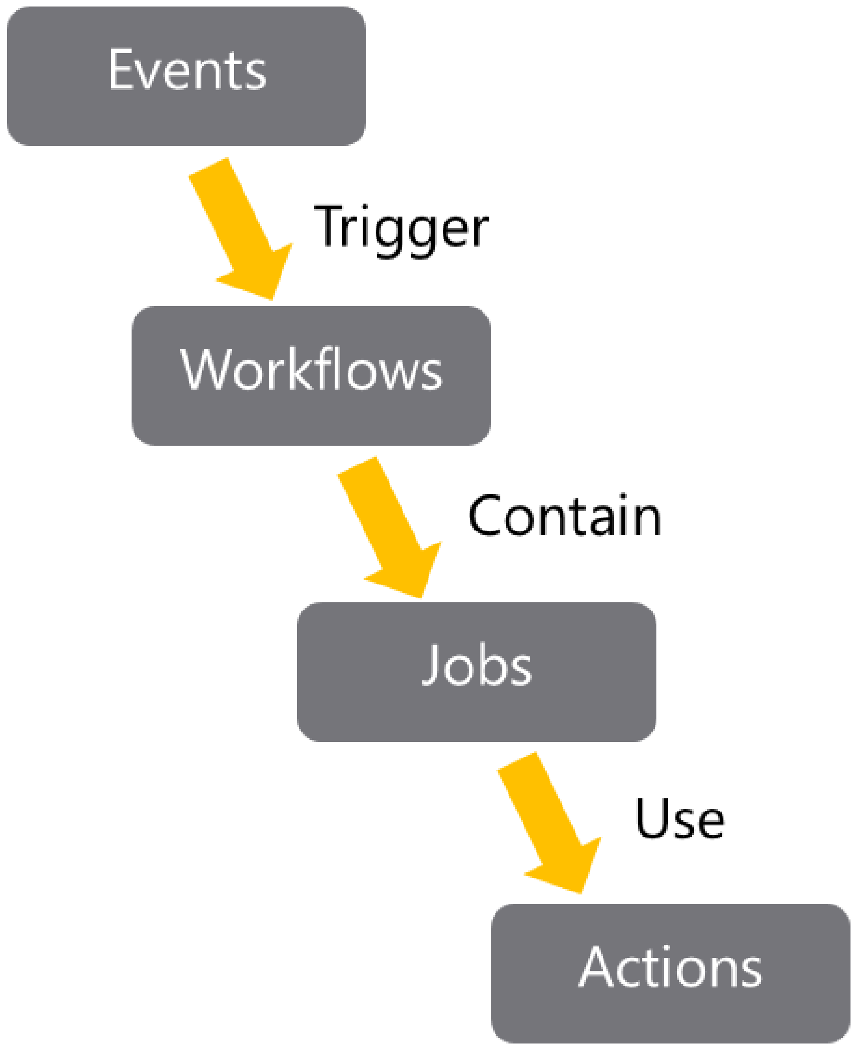
Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore Actions flow

Completed

- 1 minute



GitHub tracks events that occur. Events can trigger the start of workflows.

Workflows can also start on cron-based schedules and can be triggered by events outside of GitHub.

They can be manually triggered.

Workflows are the unit of automation. They contain Jobs.

Jobs use Actions to get work done.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Understand workflows

Completed

- 2 minutes

Workflows define the automation required. It details the events that should trigger the workflow.

Also, define the jobs that should run when the workflow is triggered.

The job defines the location in which the actions will run, like which runner to use.

Workflows are written in YAML and live within a GitHub repository at the place **.github/workflows.**

Example workflow:

```
# .github/workflows/build.yml
name: Node Build.

on: [push]

jobs:
    mainbuild:

        runs-on: ${{ matrix.os }}

    strategy:
        matrix:
            node-version: [12.x]
            os: [windows-latest]

    steps:

    - uses: actions/checkout@v1
    - name: Run node.js on latest Windows.
      uses: actions/setup-node@v1
      with:
        node-version: ${{ matrix.node-version }}

    - name: Install NPM and build.
      run: |
        npm ci
        npm run build
```

You can find a set of starter workflows here: [Starter Workflows](#) .

You can see the allowable syntax for workflows here: [Workflow syntax for GitHub Actions](#) .

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Describe standard workflow syntax elements](#)

Completed

- 1 minute

Workflows include several standard syntax elements.

- **Name:** is the name of the workflow. It's optional but is highly recommended. It appears in several places within the GitHub UI.
- **On:** is the event or list of events that will trigger the workflow.
- **Jobs:** is the list of jobs to be executed. Workflows can contain one or more jobs.
- **Runs-on:** tells Actions which runner to use.
- **Steps:** It's the list of steps for the job. Steps within a job execute on the same runner.
- **Uses:** tells Actions, which predefined action needs to be retrieved. For example, you might have an action that installs node.js.
- **Run:** tells the job to execute a command on the runner. For example, you might execute an NPM command.

You can see the allowable syntax for workflows here: Workflow syntax for GitHub Actions .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore events

Completed

- 2 minutes

Events are implemented by the **on** clause in a workflow definition.

There are several types of events that can trigger workflows.

Scheduled events

With this type of trigger, a cron schedule needs to be provided.

```
on:
    schedule:

        - cron: '0 8-17 * * 1-5'
```

Cron schedules are based on five values:

- Minute (0 - 59)
- Hour (0 - 23)
- Day of the month (1 - 31)
- Month (1 - 12)
- Day of the week (0 - 6)

Aliases for the months are JAN-DEC and for days of the week are SUN-SAT.

A wild card means any. (* is a special value in YAML, so the cron string will need to be quoted)

So, in the example above, the schedule would be 8 AM - 5 PM Monday to Friday.

Code events

Code events will trigger most actions. It occurs when an event of interest occurs in the repository.

```
on:
    pull_request
```

The above event would fire when a pull request occurs.

```
on:
    [push, pull_request]
```

The above event would fire when either a push or a pull request occurs.

```
on:
    pull_request:
        branches:

            - develop
```

The event shows how to be specific about the section of the code that is relevant.

In this case, it will fire when a pull request is made in the develop branch.

Manual events

There's a unique event that is used to trigger workflow runs manually. You should use the **workflow_dispatch** event.

Your workflow must be in the default branch for the repository.

Webhook events

Workflows can be executed when a GitHub webhook is called.

```
on:
    gollum
```

This event would fire when someone updates (or first creates) a Wiki page.

External events

Events can be on **repository_dispatch** . That allows events to fire from external systems.

For more information on events, see Events that trigger workflows .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore jobs

Completed

- 2 minutes

Workflows contain one or more jobs. A job is a set of steps that will be run in order on a runner.

Steps within a job execute on the same runner and share the same filesystem.

The logs produced by jobs are searchable, and artifacts produced can be saved.

Jobs with dependencies

By default, if a workflow contains multiple jobs, they run in parallel.

```
jobs:
  startup:
    runs-on: ubuntu-latest
    steps:

      - run: ./setup_server_configuration.sh
  build:
    steps:

      - run: ./build_new_server.sh
```

Sometimes you might need one job to wait for another job to complete.

You can do that by defining dependencies between the jobs.

```
jobs:
  startup:
    runs-on: ubuntu-latest
    steps:

      - run: ./setup_server_configuration.sh
  build:
    needs: startup
    steps:

      - run: ./build_new_server.sh
```

Note

If the startup job in the example above fails, the build job won't execute.

For more information on job dependencies, see the section **Creating Dependent Jobs** at Managing complex workflows .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore runners

Completed

- 2 minutes

When you execute jobs, the steps execute on a Runner.

The steps can be the execution of a shell script or the execution of a predefined Action.

GitHub provides several hosted runners to avoid you needing to spin up your infrastructure to run actions.

Now, the maximum duration of a job is 6 hours, and for a workflow is 72 hours.

For JavaScript code, you have implementations of node.js on:

- Windows
- macOS
- Linux

If you need to use other languages, a Docker container can be used. Now, the Docker container support is only Linux-based.

These options allow you to write in whatever language you prefer.

JavaScript actions will be faster (no container needs to be used) and more versatile runtime.

The GitHub UI is also better for working with JavaScript actions.

Self-hosted runners

If you need different configurations to the ones provided, you can create a self-hosted runner.

GitHub has published the source code for self-hosted runners as open-source, and you can find it here: https://github.com/actions/runner .

It allows you to customize the runner completely. However, you then need to maintain (patch, upgrade) the runner system.

Self-hosted runners can be added at different levels within an enterprise:

- Repository-level (single repository).
- Organizational-level (multiple repositories in an organization).
- Enterprise-level (multiple organizations across an enterprise).

GitHub strongly recommends that you don't use self-hosted runners in public repos.

Doing it would be a significant security risk, as you would allow someone (potentially) to run code on your runner within your network.

For more information on self-hosted runners, see: About self-hosted runners .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Examine release and test an action

Completed

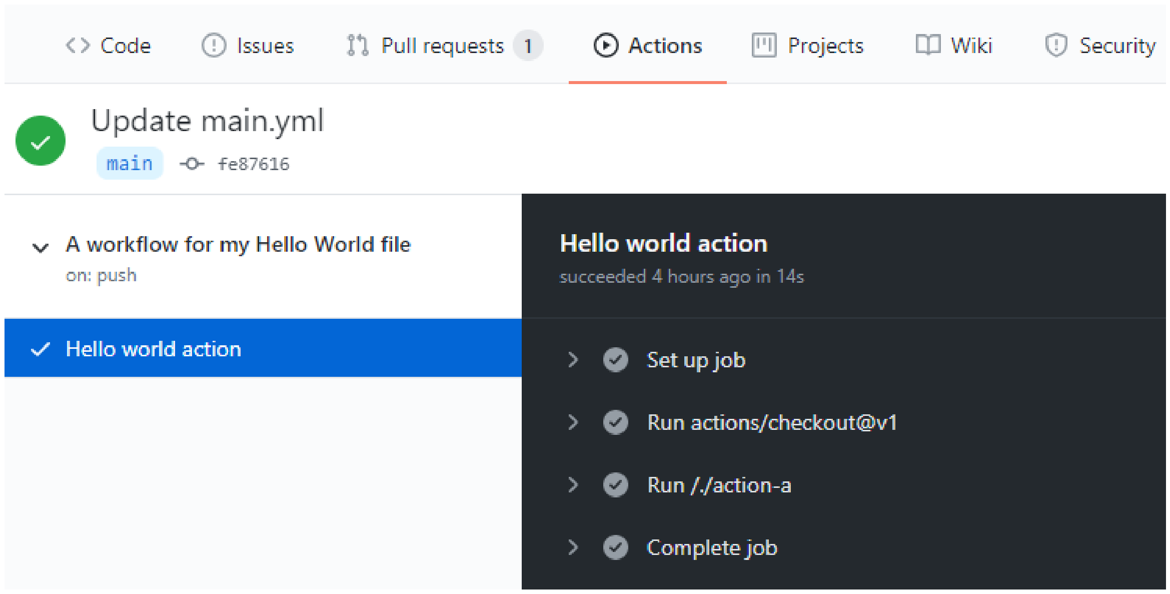
- 4 minutes

Actions will often produce console output. You don't need to connect directly to the runners to retrieve that output.

The console output from actions is available directly from within the GitHub UI.

Select **Actions** on the top repository menu to see a list of executed workflows to see the output.

Next, click on the job's name to see the steps' output.



Update main.yml
main fe87616

A workflow for my Hello World file
on: push

Hello world action

Hello world action
succeeded 4 hours ago in 14s

Set up job
Run actions/checkout@v1
Run ./action-a
Complete job

Console output can help debug. If it isn't sufficient, you can also enable more logging. See: Enabling debug logging .

Release Management for Actions

While you might be happy to retrieve the latest version of the action, there are many situations where you might want a specific version of the action.

You can request a specific release of action in several ways:

Tags

Tags allow you to specify the precise versions that you want to work.

```
steps:
    -uses: actions/install-timer@v2.0.1
```

SHA-based hashes

You can specify a requested SHA-based hash for an action. It ensures that the action hasn't changed. However, the downside to this is that you also won't receive updates to the action automatically either.

```
steps:
    -uses: actions/install-timer@327239021f7cc39fe7327647b213799853a9eb98
```

Branches

A common way to request actions is to refer to the branch you want to work with. You'll then get the latest version from that branch. That means you'll benefit from updates, but it also increases the chance of code-breaking.

```
steps:
    -uses: actions/install-timer@develop
```

Test an Action

GitHub offers several learning tools for actions.

GitHub Actions: hello-world

You'll see a basic example of how to:

- Organize and identify workflow files.
- Add executable scripts.
- Create workflow and action blocks.
- Trigger workflows.
- Discover workflow logs.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 4 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices is a keyword to make one job wait for another job to complete?

○

on.

○

needs.

○

uses.

2.

Which of the following choices isn't a level that self-hosted runners can be added within an enterprise?

○

Enterprise-level (multiple organizations across an enterprise).

○

Organizational-level (multiple repositories in an organization).

○

Project-level.

3.

Which of the following choices is the location where you can find workflows?

○

.github/workflows.

○

.github/build/workflows.

○

.github/actions/workflows.

Check your answers

You must answer all questions before checking your work.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

In this module, you learned what GitHub Actions, action flow, and its elements are. Also, understood what events are, explored jobs and runners, and how to read console output from actions.

You learned how to describe the benefits and usage of:

- Explain GitHub Actions and workflows.
- Create and work with GitHub Actions and Workflows.
- Describe Events, Jobs, and Runners.
- Examine the output and release management for actions.

Learn more

- Quickstart for GitHub Actions .
- Workflow syntax for GitHub Actions - GitHub Docs .
- Events that trigger workflows - GitHub Docs .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Learn continuous integration with GitHub Actions

Introduction

Completed

- 1 minute

This module details continuous integration using GitHub Actions and describes environment variables, artifacts, best practices, and how to secure your pipeline using encrypted variables and secrets.

Learning objectives

After completing this module, students and professionals can:

- Implement Continuous Integration with GitHub Actions.
- Use environment variables.
- Share artifacts between jobs and use Git tags.
- Create and manage secrets.

Prerequisites

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

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe continuous integration with actions

Completed

- 3 minutes

It's an example of a basic continuous integration workflow created by using actions:

```
name: dotnet Build

on: [push]

jobs:
    build:
        runs-on: ubuntu-latest
        strategy:
            matrix:
                node-version: [10.x]
        steps:

        - uses: actions/checkout@main
        - uses: actions/setup-dotnet@v1
            with:
                dotnet-version: '3.1.x'

        - run: dotnet build awesomeproject
```

- **On:** Specifies what will occur when code is pushed.
- **Jobs:** There's a single job called **build.**
- **Strategy:** It's being used to specify the Node.js version.
- **Steps:** Are doing a checkout of the code and setting up dotnet.
- **Run:** Is building the code.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Examine environment variables

Completed

- 3 minutes

When using Actions to create CI or CD workflows, you'll typically need to pass variable values to the actions. It's done by using Environment Variables.

Built-in environment variables

GitHub provides a series of built-in environment variables. It all has a GITHUB_ prefix.

Note

Setting that prefix for your variables will result in an error.

Examples of built-in environment variables are:

GITHUB_WORKFLOW is the name of the workflow.

GITHUB_ACTION is the unique identifier for the action.

GITHUB_REPOSITORY is the name of the repository (but also includes the name of the owner in owner/repo format)

Using variables in workflows

Variables are set in the YAML workflow files. They're passed to the actions that are in the step.

```
jobs:
    verify-connection:
        steps:
            - name: Verify Connection to SQL Server
            - run: node testconnection.js
        env:
            PROJECT_SERVER: PH202323V
            PROJECT_DATABASE: HAMaster
```

For more information on environment variables, including a list of built-in environment variables, see Environment variables .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Share artifacts between jobs

Completed

- 4 minutes

When using Actions to create CI or CD workflows, you'll often need to pass artifacts created by one job to another.

The most common ways to do it are by using the **upload-artifact** and **download-artifact** actions.

Upload-artifact

This action can upload one or more files from your workflow to be shared between jobs.

You can upload a specific file:

```
- uses: actions/upload-artifact
  with:
    name: harness-build-log
    path: bin/output/logs/harness.log
```

You can upload an entire folder:

```
- uses: actions/upload-artifact
  with:
    name: harness-build-logs
    path: bin/output/logs/
```

You can use wildcards:

```
- uses: actions/upload-artifact
  with:
    name: harness-build-logs
    path: bin/output/logs/harness[ab]?/*
```

You can specify multiple paths:

```
- uses: actions/upload-artifact
  with:
    name: harness-build-logs
    path: |
        bin/output/logs/harness.log
        bin/output/logs/harnessbuild.txt
```

For more information on this action, see upload-artifact.

Download-artifact

There's a corresponding action for downloading (or retrieving) artifacts.

```
- uses: actions/download-artifact
  with:
    name: harness-build-log
```

If no path is specified, it's downloaded to the current directory.

For more information on this action, see download-artifact.

Artifact retention

A default retention period can be set for the repository, organization, or enterprise.

You can set a custom retention period when uploading, but it can't exceed the defaults for the repository, organization, or enterprise.

```
- uses: actions/upload-artifact
  with:
    name: harness-build-log
    path: bin/output/logs/harness.log
    retention-days: 12
```

Deleting artifacts

You can delete artifacts directly in the GitHub UI.

For details, you can see: Removing workflow artifacts .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Examine Workflow badges

Completed

- 3 minutes

Badges can be used to show the status of a workflow within a repository.

They show if a workflow is currently passing or failing. While they can appear in several locations, they typically get added to the README.md file for the repository.

Badges are added by using URLs. The URLs are formed as follows:

https://github.com/
<OWNER>/<REPOSITORY>/actions/workflows/<WORKFLOW_FILE>/badge.svg

Where:

- AAAAA is the account name.
- RRRRR is the repository name.
- WWWWW is the workflow name.

timeclock
failing

They usually indicate the status of the default branch but can be branch-specific. You do this by adding a URL query parameter:

?branch=BBBBB

where:

- BBBBB is the branch name.

For more information, see: Adding a workflow status badge .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Describe best practices for creating actions

Completed

- 2 minutes

It's essential to follow best practices when creating actions:

- Create chainable actions. Don't create large monolithic actions. Instead, create smaller functional actions that can be chained together.
- Version your actions like other code. Others might take dependencies on various versions of your actions. Allow them to specify versions.
- Provide the **latest** label. If others are happy to use the latest version of your action, make sure you provide the **latest** label that they can specify to get it.
- Add appropriate documentation. As with other codes, documentation helps others use your actions and can help avoid surprises about how they function.
- Add details **action.yml** metadata. At the root of your action, you'll have an **action.yml** file. Ensure it has been populated with author, icon, expected inputs, and outputs.
- Consider contributing to the marketplace. It's easier for us to work with actions when we all contribute to the marketplace. Help to avoid people needing to relearn the same issues endlessly.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

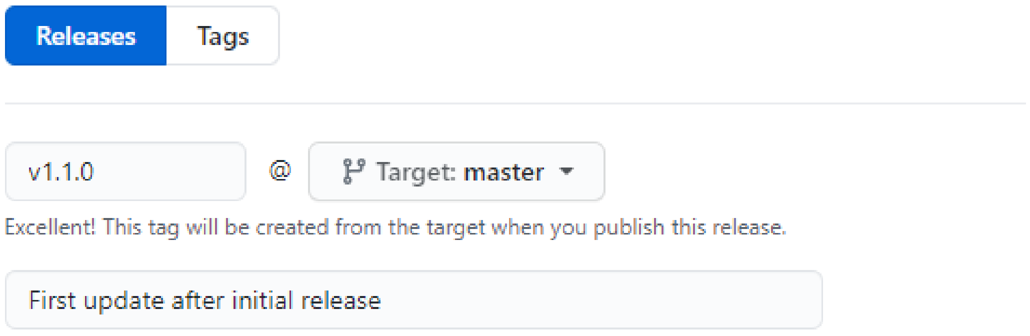
Mark releases with Git tags

Completed

- 2 minutes

Releases are software iterations that can be packed for release.

In Git, releases are based on Git tags. These tags mark a point in the history of the repository. Tags are commonly assigned as releases are created.



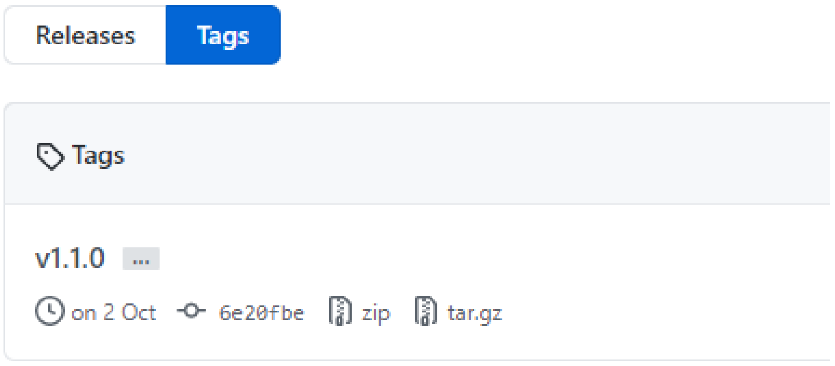
v1.1.0

Excellent! This tag will be created from the target when you publish this release.

First update after initial release

Often these tags will contain version numbers, but they can have other values.

Tags can then be viewed in the history of a repository.



For more information on tags and releases, see: [About releases](#) .

[Continue](#)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Create encrypted secrets](#)

Completed

- 3 minutes

Actions often can use secrets within pipelines. Common examples are passwords or keys.

In GitHub actions, It's called **Secrets** .

Secrets

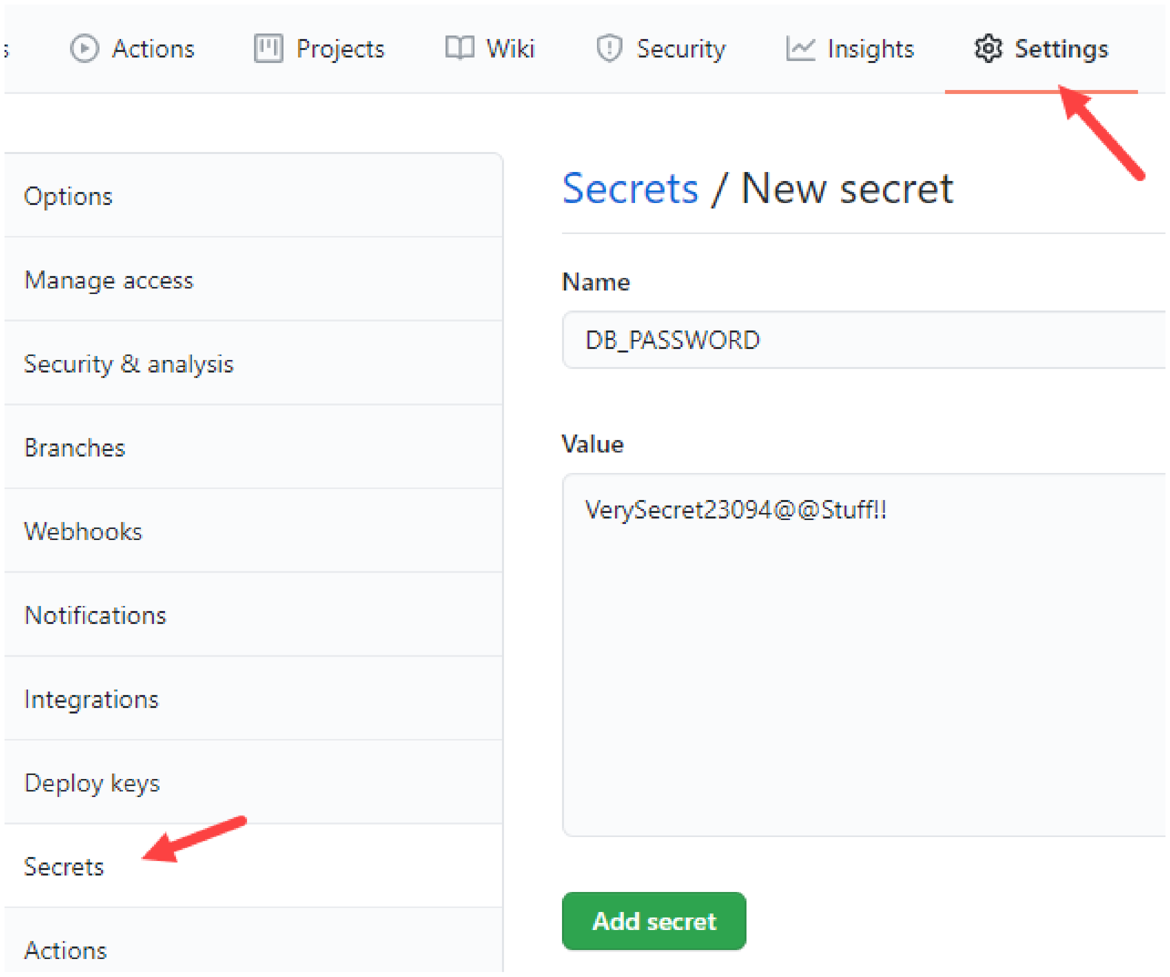
Secrets are similar to environment variables but encrypted. They can be created at two levels:

- Repository
- Organization

If secrets are created at the organization level, access policies can limit the repositories that can use them.

Creating secrets for a repository

To create secrets for a repository, you must be the repository's owner. From the repository **Settings** , choose **Secrets** , then **New Secret** .



DB_PASSWORD

VerySecret23094@@Stuff!!

For more information on creating secrets, see [Encrypted secrets](.) .

[Continue](.)

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

[Use secrets in a workflow](.)

Completed

- 3 minutes

Secrets aren't passed automatically to the runners when workflows are executed.

Instead, when you include an action that requires access to a secret, you use the **secrets** context to provide it.

```
steps:

  - name: Test Database Connectivity
```

```
    with:
      db_username: ${{ secrets.DBUserName }}
      db_password: ${{ secrets.DBPassword }}
```

Command-line secrets

Secrets shouldn't be passed directly as command-line arguments as they may be visible to others. Instead, treat them like environment variables:

```
steps:

  - shell: pwsh
    env:
      DB_PASSWORD: ${{ secrets.DBPassword }}
    run: |
      db_test "$env:DB_PASSWORD"
```

Limitations

Workflows can use up to 100 secrets, and they're limited to 64 KB in size.

For more information on creating secrets, see Encrypted secrets .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Implement GitHub Actions for CI/CD

Completed

- 40 minutes

Estimated time: 40 minutes.

Lab files: none.

Scenario

In this lab, you'll learn how to implement a GitHub Action workflow that deploys an Azure web app.

Objectives

After completing this lab, you'll be able to:

- Implement a GitHub Action workflow for CI/CD.
- Explain the basic characteristics of GitHub Action workflows.

Requirements

- This lab requires **Microsoft Edge** or an Azure DevOps-supported browser .
- Identify an existing Azure subscription or create a new one.
- Verify that you have a Microsoft or Microsoft Entra account with the Contributor or the Owner role in the Azure subscription. For details, refer to List Azure role assignments using the Azure portal .
- If you don't already have a GitHub account that you can use for this lab, follow the instructions available at Signing up for a new GitHub account to create one.

Exercises

During this lab, you'll complete the following exercises:

- Exercise 0: Import eShopOnWeb to your GitHub Repository.
- Exercise 1: Setup your GitHub Repository and Azure access.
- Exercise 2: Remove the Azure lab resources.

Launch Exercise

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 5 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following choices is where the database passwords that are needed in a CI pipeline should be stored?

○

Repo.

○

action.yml.

○

Encrypted Secrets.

2.

Which of the following files is the metadata for an action held?

○

workflow.yml.

○

action.yml.

○

meta.yml.

3.

Which of the following choices is how can the status of a workflow be shown in a repository?

○

Using Badges.

○

Status Files.

○

Conversation Tab.

Check your answers

You must answer all questions before checking your work.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

This module detailed continuous integration using GitHub Actions. It described environment variables, artifacts, best practices, and how to secure your pipeline using encrypted variables and secrets.

You learned how to describe the benefits and usage of:

- Implement Continuous Integration with GitHub Actions.
- Use environment variables.
- Share artifacts between jobs and use Git tags.
- Create and manage secrets.

Learn more

- About continuous integration - GitHub Docs .
- Environment variables - GitHub Docs .
- Storing workflow data as artifacts - GitHub Docs .
- Encrypted secrets - GitHub Docs .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Design a container build strategy

Introduction

Completed

- 4 minutes

Container

Containers are the third computing model, after bare metal and virtual machines – and containers are here to stay.

Unlike a VM, which provides hardware virtualization, a container provides operating-system-level virtualization by abstracting the "user space," not the entire operating system. The operating system level architecture is being shared across containers. It's what makes containers so lightweight.

Docker gives you a simple platform for running apps in containers. Either old or new apps on Windows and Linux, and that simplicity is a powerful enabler for all aspects of modern IT.

Containers aren't only faster and easier to use than VMs; they also make far more efficient use of computing hardware. Also, they have provided engineering teams with dramatically more flexibility for running cloud-native applications.

Containers package up the application services and make them portable across different computing environments for dev/test and production use.

With containers, it's easy to ramp application instances to match spikes in demand quickly. And because containers draw on resources of the host OS, they're much lighter weight than virtual machines. It means containers make highly efficient use of the underlying server infrastructure.

Though the container runtime APIs are well suited to managing individual containers, they're woefully inadequate for managing applications that might comprise hundreds of containers spread across multiple hosts.

You need to manage and connect containers to the outside world for scheduling, load balancing, and distribution. It's where a container orchestration tool like Azure Kubernetes Services (AKS) comes into its own.

AKS handles the work of scheduling containers onto a compute cluster and manages the workloads to ensure they run as the user intended.

AKS is an open-source system for deploying, scaling and managing containerized applications. Instead of bolting operations as an afterthought, AKS brings software development and operations together by design.

AKS enables an order-of-magnitude increase in the operability of modern software systems. With declarative, infrastructure-agnostic constructs to describe how applications are composed. Also how they interact and how they're managed.

This module helps you plan a container build strategy, explains containers and their structure, and introduces Docker and related services.

What other benefits do containers offer?

Containers are **portable** . A container will run wherever Docker is supported.

Containers allow you to have a **consistent** development environment. For example, a SQL Server 2019 CU2 container that one developer is working with will be identical to another developer.

Containers can be lightweight. A container may be only tens of megabytes in size, but a virtual machine with its entire operating system may be several gigabytes. Because of it, a single server can host far more containers than virtual machines.

Containers can be efficient: fast to deploy, fast to boot, fast to patch, and quick to update.

Learning objectives

After completing this module, students and professionals can:

- Design a container strategy.
- Work with Docker Containers.

Prerequisites

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

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Examine structure of containers

Completed

- 5 minutes

If you're a programmer or techie, you've at least heard of Docker: a helpful tool for packing, shipping, and running applications within "containers."

With all the attention it's getting these days, it would be hard not to, from developers and system admins alike.

There's a difference between containers and Docker. A container is a thing that runs a little program package, while Docker is the container runtime and orchestrator.

What are containers, and why do you need them?

Containers are a solution to the problem of how to get the software to run reliably when moved from one computing environment to another.

It could be from a developer's laptop to a test environment, from a staging environment to production. Also, from a physical machine in a data center to a VM in a private or public cloud.

Problems arise when the supporting software environment isn't identical.

For example, say you'll develop using Python 3, but when it gets deployed to production, it will run on Python 2.7. It's likely to cause several issues.

It's not limited to the software environment; you're likely to come across issues in production if there are differences in the networking stack between the two environments.

How do containers solve this problem?

A container consists of an entire runtime environment:

- An application, plus all its dependencies.
- Libraries and other binaries.
- Configuration files needed to run it, bundled into one package.

You can resolve it by containerizing the application platform and its dependencies. Also, differences in OS distributions and underlying infrastructure are abstracted.

What's the difference between containers and virtualization?

Containers and VMs are similar in their goals: to isolate an application and its dependencies into a self-contained unit that can run anywhere. They remove the need for physical hardware, allowing for:

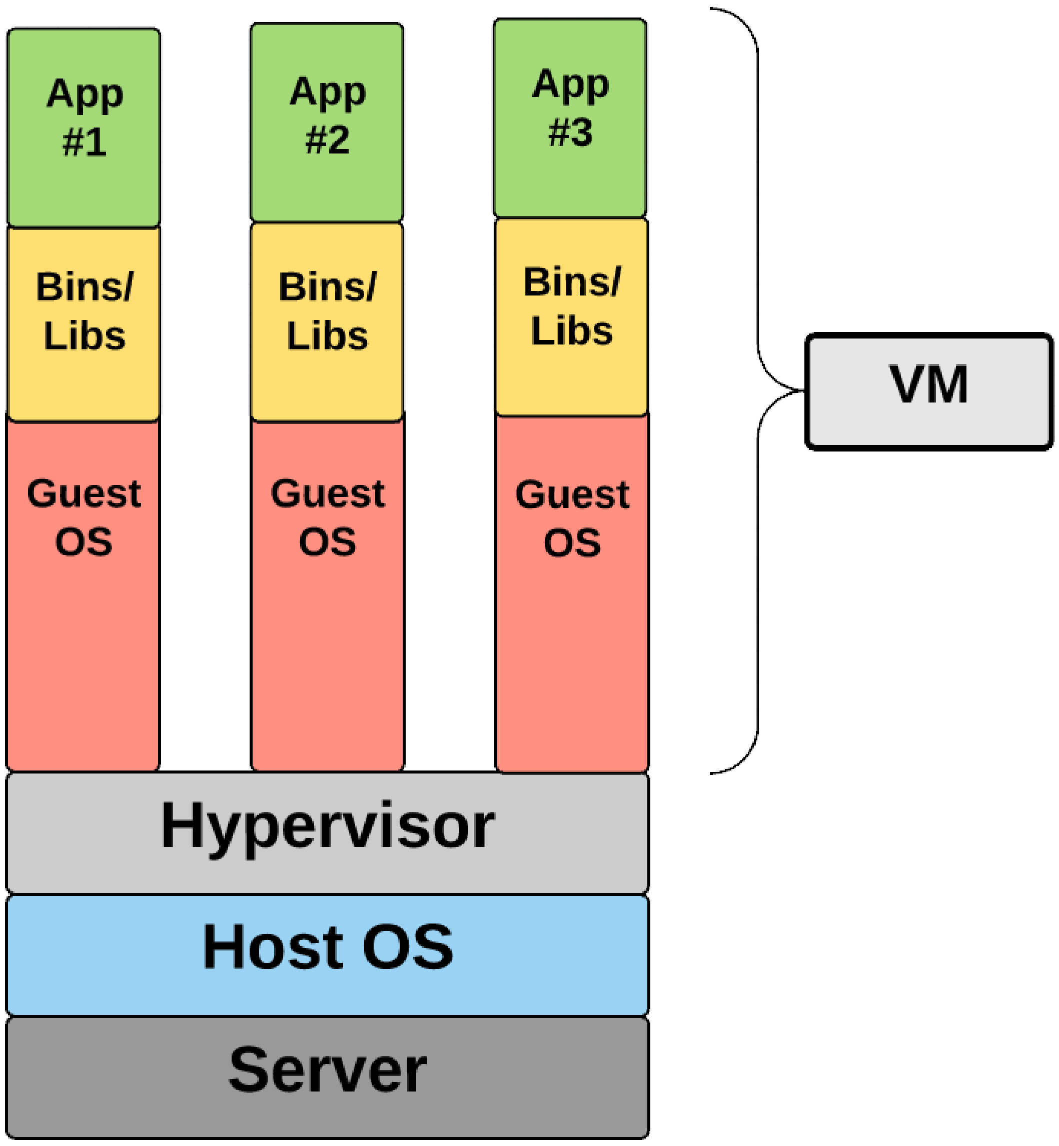
- More efficient use of computing resources.
- Energy consumption.
- Cost-effectiveness.

The main difference between containers and VMs is in their architectural approach. Let's take a closer look.

Virtual Machines

A VM is essentially an emulation of a real computer that executes programs like a real computer. VMs run on top of a physical machine using a "hypervisor."

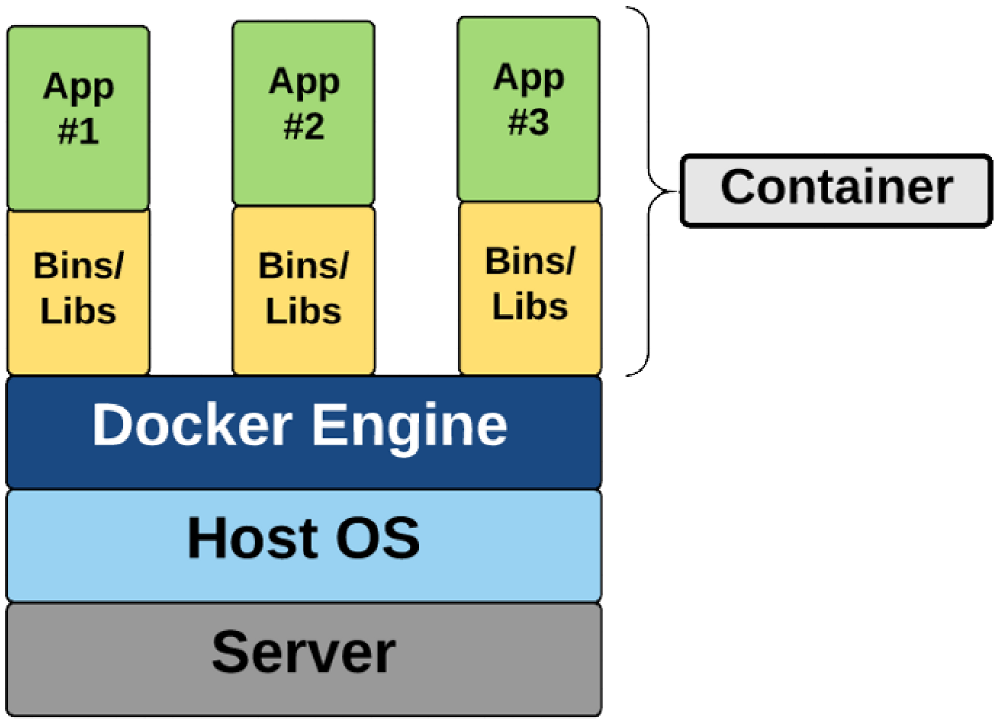
As you can see in the diagram, VMs package up the virtual hardware, a kernel (OS), and user space for each new VM.



Container

Unlike a VM, which provides hardware virtualization, a container provides operating-system-level virtualization by abstracting the "user space."

This diagram shows that containers package up just the user space, not the kernel or virtual hardware like a VM does. Each container gets its isolated user space to allow multiple containers to run on a single host machine. We can see that all the operating system-level architecture is being shared across containers. The only parts that are created from scratch are the bins and libs. It's what makes containers so lightweight.



Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Work with Docker containers

Completed

- 2 minutes

Container Lifecycle

The standard steps when working with containers are:

Docker build - You create an image by executing a Dockerfile.

Docker pull - You retrieve the image, likely from a container registry.

Docker run - You execute the container. An instance is created of the image.

You can often execute the docker run without needing first to do the docker pull.

In that case, Docker will pull the image and then run it. Next time, it won't need to pull it again.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Understand Dockerfile core concepts

Completed

- 4 minutes

Dockerfiles are text files that contain the commands needed by **docker build** to assemble an image.

Here's an example of a basic Dockerfile:

```
FROM ubuntu
LABEL maintainer="johndoe@contoso.com"
ADD appsetup /
RUN /bin/bash -c 'source $HOME/.bashrc; \
echo $HOME'
CMD ["echo", "Hello World from within the container"]
```

The first line refers to the parent image based on which this new image will be based.

Generally, all images will be based on another existing image. In this case, the Ubuntu image would be retrieved from either a local cache or from DockerHub.

An image that doesn't have a parent is called a **base** image. In that rare case, the FROM line can be omitted, or **FROM scratch** can be used instead.

The second line indicates the email address of the person who maintains this file. Previously, there was a MAINTAINER command, but that has been deprecated and replaced by a label.

The third line adds a file to the root folder of the image. It can also add an executable.

The fourth and fifth lines are part of a RUN command. Note the use of the backslash to continue the fourth line onto the fifth line for readability. It's equivalent to having written it instead:

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'
```

The RUN command is run when the docker build creates the image. It's used to configure items within the image.

By comparison, the last line represents a command that will be executed when a new container is created from the image; it's run after container creation.

For more information, you can see:

[Dockerfile reference]()

[Continue]()

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Examine multi-stage dockerfiles

Completed

- 4 minutes

What are multi-stage Dockerfiles? Multi-stage builds give the benefits of the builder pattern without the hassle of maintaining three separate files.

Let us look at a multi-stage Dockerfile.

```
FROM mcr.microsoft.com/dotnet/core/aspnetcore:3.1 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /src
COPY ["WebApplication1.csproj", ""]
RUN dotnet restore "./WebApplication1.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "WebApplication1.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "WebApplication1.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "WebApplication1.dll"]
```

At first, it simply looks like several dockerfiles stitched together. Multi-stage Dockerfiles can be layered or inherited.

When you look closer, there are a couple of key things to realize.

Notice the third stage.

```
FROM build AS publish
```

build isn't an image pulled from a registry. It's the image we defined in stage 2, where we named the result of our-build (SDK) image "builder." `Docker build` will create a named image we can later reference.

We can also copy the output from one image to another. It's the real power to compile our code with one base SDK image (`mcr.microsoft.com/dotnet/core/sdk:3.1`) while creating a production image based on an optimized runtime image (`mcr.microsoft.com/dotnet/core/aspnet:3.1`). Notice the line.

```
COPY --from=publish /app/publish .
```

It takes the /app/publish directory from the published image and copies it to the working directory of the production image.

Breakdown of stages

The first stage provides the base of our optimized runtime image. Notice it derives from `mcr.microsoft.com/dotnet/core/aspnet:3.1` .

We would specify extra production configurations, such as registry configurations, MSIexec of other components. You would hand off any of those environment configurations to your ops folks to prepare the VM.

The second stage is our build environment. `mcr.microsoft.com/dotnet/core/sdk:3.1` This includes everything we need to compile our code. From here, we have compiled binaries we can publish or test—more on testing in a moment.

The third stage derives from our build stage. It takes the compiled output and "publishes" them in .NET terms.

Publish means taking all the output required to deploy your "app/publish/service/component" and placing it in a single directory. It would include your compiled binaries, graphics (images), JavaScript, and so on.

The fourth stage takes the published output and places it in the optimized image we defined in the first stage.

Why is publish separate from the build?

You'll likely want to run unit tests to verify your compiled code. Or the aggregate of the compiled code from multiple developers being merged continues to function as expected.

You could place the following stage between builder and publish to run unit tests.

```
FROM build AS test
WORKDIR /src/Web.test
RUN dotnet test
```

If your tests fail, the build will stop to continue.

Why is base first?

You could argue it's simply the logical flow. We first define the base runtime image. Get the compiled output ready, and place it in the base image.

However, it's more practical. While debugging your applications under Visual Studio Container Tools, VS will debug your code directly in the base image.

When you hit F5, Visual Studio will compile the code on your dev machine. The first stage then volume mounts the output to the built runtime image.

You can test any configurations you have made to your production image, such as registry configurations or otherwise.

When the `docker build --target base` is executed, docker starts processing the dockerfile from the beginning through the stage (target) defined.

Since the base is the first stage, we take the shortest path, making the F5 experience as fast as possible.

If the base were after compilation (builder), you would have to wait for all the next steps to complete.

One of the perf optimizations we make with VS Container Tools is to take advantage of the Visual Studio compilations on your dev machine.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Examine considerations for multiple stage builds

Completed

- 3 minutes

Adopt container modularity

Try to avoid creating overly complex container images that couple together several applications.

Instead, use multiple containers and try to keep each container to a single purpose.

The website and the database for a web application should likely be in separate containers.

There are always exceptions to any rule but breaking up application components into separate containers increases the chances of reusing containers.

It also makes it more likely that you could scale the application.

For example, in the web application mentioned, you might want to add replicas of the website container but not for the database container.

Avoid unnecessary packages

To help minimize image sizes, it's also essential to avoid including packages that you suspect might be needed but aren't yet sure if they're required.

Only include them when they're required.

Choose an appropriate base

While optimizing the contents of your Dockerfiles is essential, it's also crucial to choose the appropriate parent (base) image. Start with an image that only contains packages that are required.

Avoid including application data

While application data can be stored in the container, it will make your images more prominent.

It would be best to consider using **docker volume** support to maintain the isolation of your application and its data. Volumes are persistent storage mechanisms that exist outside the lifespan of a container.

For more information, see Use multiple-stage builds .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Explore Azure container-related services

Completed

- 4 minutes

Azure provides a wide range of services that help you work with containers.

Here are the essential services that are involved:

Azure Container Instances (ACI)

Running your workloads in Azure Container Instances (ACI) allows you to create your applications rather than provisioning and managing the infrastructure that will run the applications.

ACIs are simple and fast to deploy, and when you're using them, you gain the security of hypervisor isolation for each container group. It ensures that your containers aren't sharing an operating system kernel with other containers.

Azure Kubernetes Service (AKS)

Kubernetes has quickly become the de-facto standard for container orchestration. This service lets you quickly deploy and manage Kubernetes, to scale and run applications while maintaining overall solid security.

This service started life as Azure Container Services (ACS) and supported Docker Swarm and Mesos/Mesosphere DC/OS at release to manage orchestrations. These original ACS workloads are still supported in Azure, but Kubernetes support was added.

It quickly became so popular that Microsoft changed the acronym for Azure Container Services to AKS and later changed the name of the service to Azure Kubernetes Service (also AKS).

Azure Container Registry (ACR)

This service lets you store and manage container images in a central registry. It provides you with a Docker private registry as a first-class Azure resource.

All container deployments, including DC/OS, Docker Swarm, and Kubernetes, are supported. The registry is integrated with other Azure services such as the App Service, Batch, Service Fabric, and others.

Importantly, it allows your DevOps team to manage the configuration of apps without being tied to the configuration of the target-hosting environment.

Azure Container Apps

Azure Container Apps allows you to build and deploy modern apps and microservices using serverless containers. It deploys containerized apps without managing complex infrastructure.

You can write code using your preferred programming language or framework and build microservices with full support for Distributed Application Runtime (Dapr) . Scale dynamically based on HTTP traffic or events powered by Kubernetes Event-Driven Autoscaling (KEDA) .

Azure App Service

Azure Web Apps provides a managed service for both Windows and Linux-based web applications and provides the ability to deploy and run containerized applications for both platforms. It provides autoscaling and load balancing options and is easy to integrate with Azure DevOps.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Deploy Docker containers to Azure App Service web apps

Completed

- 30 minutes

Estimated time: 30 minutes.

Lab files: none.

Scenario

In this lab, you will learn how to use an Azure DevOps CI/CD pipeline to build a custom Docker image, push it to Azure Container Registry, and deploy it as a container to Azure App Service.

Objectives

After completing this lab, you'll be able to:

- Build a custom Docker image by using a Microsoft hosted Linux agent.
- Push an image to Azure Container Registry.
- Deploy a Docker image as a container to Azure App Service by using Azure DevOps.

Requirements

- This lab requires **Microsoft Edge** or an Azure DevOps-supported browser .
- **Set up an Azure DevOps organization:** If you don't already have an Azure DevOps organization that you can use for this lab, create one by following the instructions available at Create an organization or project collection .
- Identify an existing Azure subscription or create a new one.
- Verify that you have a Microsoft or Microsoft Entra account with the Contributor or the Owner role in the Azure subscription. For details, refer to List Azure role assignments using the Azure portal and View and assign administrator roles in Microsoft Entra ID. .

Exercises

During this lab, you'll complete the following exercises:

- Exercise 0: Configure the lab prerequisites.
- Exercise 1: Manage the service connection.
- Exercise 2: Import and run the CI pipeline.
- Exercise 3: Import and run the CD pipeline.
- Exercise 4: Remove the Azure lab resources

Launch Exercise

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Knowledge check

Completed

- 5 minutes

Choose the best response for each question. Then select **Check your answers** .

Check your knowledge

1.

Which of the following commands do you use to retrieve an image from a container registry?

○

Docker run.

○

Docker pull.

○

Docker build.

2.

Which of the following choices isn't container-related Azure services?

○

Azure App Service.

○

Azure Container Instances.

○

Azure Virtual Machine Scale Sets.

3.

Which of the following choices creates an instance of the image?

○

Docker run.

○

Docker build.

○

Docker pull.

Check your answers

You must answer all questions before checking your work.

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .

Summary

Completed

- 1 minute

This module helped you plan a container build strategy, explained containers and their structure, introduced Docker and related services.

You learned how to describe the benefits and usage of:

- Design a container strategy.
- Work with Docker Containers.

Learn more

- Container Jobs in Azure Pipelines - Azure Pipelines .
- Build an image - Azure Pipelines .
- Service Containers - Azure Pipelines .
- Quickstart - Create registry in portal - Azure Container Registry .
- What are Microservices? .
- CI/CD for microservices .

Continue

Need help? See our troubleshooting guide or provide specific feedback by reporting an issue .