Continues Delivery in DevOps

DevOps as the union of people, process, and products to enable continuous delivery of value to our end users.

CD by itself is a set of processes, tools, and techniques that enable rapid, reliable, and continuous delivery of software. So CD isn't only about setting up a pipeline, although that part is important. CD is about setting up a working environment where:

* We have a reliable and repeatable process for releasing and deploying software.
* We automate as much as possible.
* We don't put off doing something that's difficult or painful. Instead, we do it more often so that we figure out how to make it routine.
* We keep everything in source control.
* We all agree that *done* means *released*.
* We build quality into the process. Quality is never an afterthought.
* We're all responsible for the release process. We no longer work in silos.
* We always try to improve.

Why do I need continuous delivery?

CD helps software teams deliver reliable software updates to their customers at a rapid cadence. CD also helps ensure that both customers and stakeholders have the latest features and fixes quickly

We need CD because, as we all know, the world has changed. New features are being released faster. Updates and bug fixes need to be available right away. It isn't just our management that wants to speed up our releases. Management is simply reacting to the demands of our customers. If customers can't get what they want from us, they'll go somewhere else.

The old way of doing things, where we all worked in our separate silos and didn't talk to each other, is too slow. And honestly, it's frustrating for all of us.

We need to work together if we're going to succeed. All of us have seen how automation gives us a reliable, repeatable, and faster process to build artifacts. CD will bring those benefits to an entire release pipeline.

## How does continuous delivery compare to right-click publishing?

Many development tools provide ways to publish your application directly to some target environment, such as Microsoft Internet Information Services (IIS) or Azure

Right-click publishing is a great way to quickly build a prototype. For example, you might right-click publish your application to Azure so that you can share a new idea with your team. But this technique has limitations.

Continuous delivery provides a consistent way for you and your team to continuously test, deploy, and monitor your application each time you check in your code. When you right-click publish your application, there's no guarantee that the code was properly tested or will behave as expected under real-world usage.

## What are the parts of a basic CD pipeline? : A basic CD pipeline contains a *trigger* to get the process going and at least one *stage*, or deployment phase. A stage is made up of jobs. A job is a series of steps that defines how to build, test, or deploy your software.

## 

## We already have the Callout 1 build artifact. It's the *.zip* file that our existing build pipeline creates. But how do we deploy it to a Callout 2 live environment?

## we defined the deployment process in stages, such as building the artifact and deploying the artifact to the various testing and production environments. Each stage breaks down into one or more jobs. Each job breaks down into tasks, just like the ones we use in our existing build pipeline.

## What is a pipeline stage? A *stage* is a part of the pipeline that can run independently and be triggered by different mechanisms. A mechanism might be the success of the previous stage, a schedule, or even a manual trigger. You'll learn more about these mechanisms in the next module.

## I think that deciding on our stages is a great start and pretty straightforward. Let's define a *stage* as a major division in a pipeline. Every stage is independent of every other stage. We could have a stage that builds the app and another stage that runs tests. There are many possibilities. Because we want to keep it simple, how about we start with two stages?

## 

We've already defined the tasks for the Callout 1 build stage in our pipeline. Our Callout 2 deployment stage can be similar, including tasks that deploy the build to an environment.

**The question is, where should we deploy the artifact?**

## What is an environment?

You've likely used the term *environment* to refer to where your application or service is running. For example, your *production environment* might be where your end users access your application.

Following this example, your production environment might be:

* A physical machine or virtual machine (VM).
* A containerized environment, such as Kubernetes.
* A managed service, such as Azure App Service.
* A serverless environment, such as Azure Functions.

An artifact is deployed to an environment. Azure Pipelines makes it easy to deploy to almost any kind of environment, whether it's on-premises or in the cloud.

In Azure Pipelines, the term *environment* has a second meaning. Here, an *environment* is an abstract representation of your deployment environment, such as a Kubernetes cluster, an App Service instance, or a virtual machine.

An Azure Pipelines environment records the deployment history to help you identify the source of changes. By using Azure Pipelines environments, you can also define security checks and ways to control how an artifact is promoted from one stage of a pipeline to another. What an environment includes depends on what you want to do with the artifact. An environment where you want to test the artifact will probably be defined differently than one where you want to deploy the artifact for your end users.

One way to define an Azure Pipelines environment is with a YAML/Classic file. Your YAML file includes an environment section, which specifies the Azure Pipelines environment where you'll deploy your artifact.

As you plan your release pipeline, you'll need to decide where your application or service will run.

Andy lists these options on the whiteboard:

* Virtual machines
* Containers
* Azure App Service
* Serverless computing

How does Azure Pipelines perform deployment steps?To deploy your software, Azure Pipelines first needs to authenticate with the target environment. Azure Pipelines provides different authentication mechanisms. The one you use depends on the target environment you're deploying to. You'll find more information about these mechanisms at the end of this module.

We have our build artifact, and we know we'll build and deploy in stages of the pipeline. We've also defined the target environment for our deployment. That's App Service. My question now is, how does Azure Pipelines authenticate with App Service? We need to ensure the process is secure.

1. Specify the target deployment environment in the pipeline configuration.
2. Provide a way for Azure Pipelines to authenticate access to that environment.
3. Use Azure Pipelines tasks to deploy the build artifact to that environment.

## we need to create a *service connection* to specify the target environment and authenticate access to it. After we define the service connection, it will be available for all of our tasks to use. Then we need to use the built-in tasks

### Deploy a web application to App Service : You use the AzureWebApp@1 task to deploy a web application to App Service. This task works with several programming languages and frameworks, including ASP.NET, ASP.NET Core, PHP, Java, Python, Go, and Node.js. We use this task to do the deployment. To use it, though, we have to have App Service running on our Azure subscription.

This task also requires a few inputs:

* **azureSubscription** is the service connection we talked about earlier. We need this input to authenticate with the target environment.
* **appName** specifies the name of our App Service instance.
* **package** specifies where on the build agent to find the package to deploy.

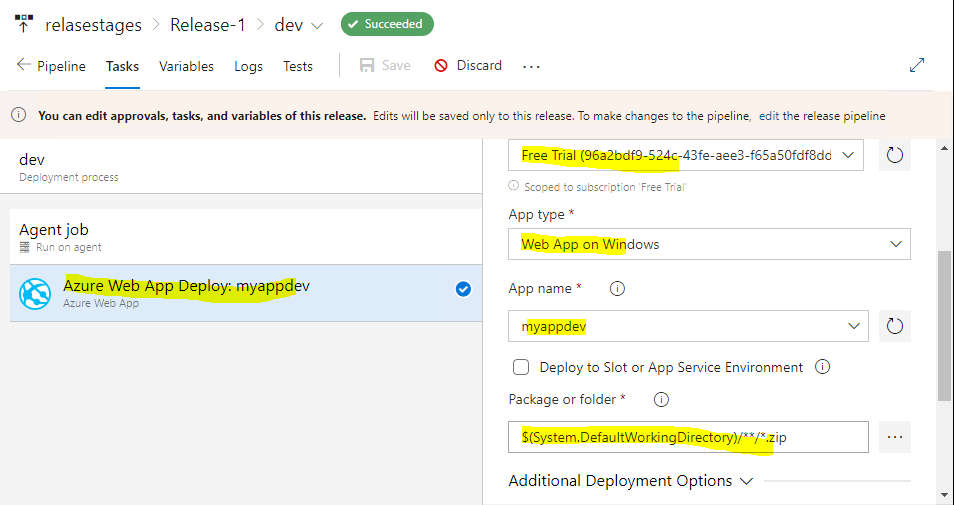
The azureSubscription part specifies the service connection that's named MyServiceConnection. You'll work with service connections shortly.

### What are jobs and strategies?

Your existing build pipeline defines a build agent, pipeline variables, and the tasks needed to build your software.

The deployment part of your pipeline contains these same elements. Your deployment configuration typically also defines one or more jobs, a pipeline environment, and strategies. You learned about pipeline environments earlier.

Example:



#### Jobs :

A *job* is a series of steps, or tasks, that run sequentially as a unit. Every pipeline stage has one job by default, even when that stage doesn't use the job keyword.

A job can run in an agent pool, on a container, or directly on the Azure DevOps server. The example job shown here runs on a Microsoft-hosted **default** agent.

You can specify the conditions under which each job runs. The example job shown here doesn't define any conditions. By default, a job runs if it doesn't depend on any other job or if all of the jobs that it does depend on have finished successfully.

You can also run jobs in parallel or sequentially. Using your existing build pipeline as an example, you can use parallel jobs to build your software on Windows, Linux, and macOS agents simultaneously.

A *deployment job* is a special type of job that plays an important role in your deployment stages. Deployment jobs record the status of your deployments in Azure Pipelines, providing you with an audit trail. Deployment jobs also help you define your deployment strategy, which we'll do shortly.

### How does Azure Pipelines connect to Azure?

To deploy your app to an Azure resource, such as a virtual machine or App Service, you need a *service connection*. A service connection provides secure access to your Azure subscription by using one of two methods:

* Service principal authentication
* Managed identities for Azure resources

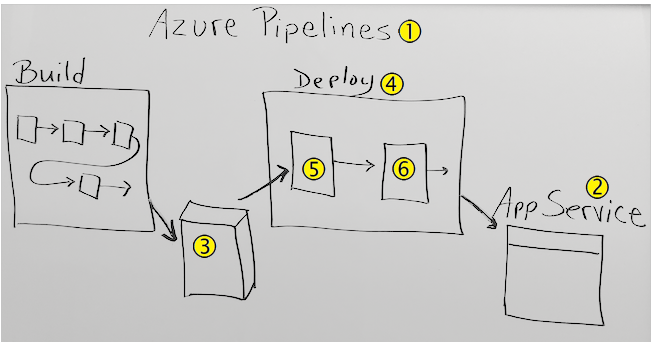
You can learn more about these security models at the end of this module, but in short:

* A *service principal* is an identity with a limited role that can access Azure resources. Think of a service principal as a service account that can do automated tasks on your behalf.
* *Managed identities* for Azure resources are a feature of Azure Active Directory (Azure AD). Managed identities simplify the process of working with service principals. Because managed identities exist on the Azure AD tenant, Azure infrastructure can automatically authenticate the service and manage the account for you.

Managed identities simplify the process of working with service principals. But in this module you use service principal authentication because a service connection can automatically discover your Azure resources and assign the appropriate service principal roles for you.

The plan :

* Build on their existing Azure Pipelines build configuration.
* Define a build stage that creates the artifact.
* Define a deployment stage that deploys the artifact to App Service.



Is this drawing correct? We use Callout **1** Azure Pipelines to deploy to Callout **2** App Service. To do that, we take the Callout **3** build artifact as the input to the Callout **4** deployment stage. The tasks in the deployment stage Callout **5** download the artifact and use a service connection to Callout **6** deploy the artifact to App Service.

Terms :

Agent:

* When your build or deployment runs, the system begins one/more jobs. Agent is installable s/w that runs one job at a time.
* To build your code or deploy your s/w using azure pipelines, you need at least one agent.
* Here we have two types of agent
  1. Microsoft-hosted agent
  2. Self-hosted agent

Artifacts:

* An artifact is a collection of files or packages which are created by build run.
* These artifacts are made available for the next task i.e deployment.

Environment: It is a place where we deploy our application and it’s a collection of resources (VM, app service …etc).

Release PipeLine ca deploy the application on one or more environments.

Continue Integration: CI is development practice that requires developers to integrate code into shared repository several times a day.

Each check-In is then verified by an automated build, allowing teams to detect problem early and deliver the software early.