# Introduction to Creating Multi-Stage Pipelines

In Create a release pipeline with Azure Pipelines , you built a basic release pipeline. That pipeline has a Build stage that builds the artifact and a Deploy stage that installs the web app on Azure App Service. Mara and Andy built this pipeline as a proof of concept that they showed to the rest of the team.

Of course, an actual release pipeline has more stages. Each stage has its own set of tasks that can potentially take an artifact all the way to production.

In this module, you'll join the Tailspin Toys web team as they design a realistic release pipeline that contains multiple stages. You'll also learn different ways to control how an artifact is promoted from one stage to the next.

A good release-management workflow enables you to release more frequently and more consistently. In practice, you'll want to define a process that maps to your team's needs. Here you'll create a basic workflow. That means first designing the environments. The environments define the runtimes of each stage in the pipeline. You'll then deploy the Space Game web application to these stages: Dev, Test, and Staging. Each stage deploys the application to its own App Service instance.

## Learning objectives

In this module, you will:

* Identify the stages, or major divisions of the pipeline, that you need to implement a multistage pipeline.
* Understand when to use conditions, triggers, and approvals to promote changes from one stage to the next.
* Promote a build through these stages: Build, Dev, Test, and Staging.

## Prerequisites

The modules in this learning path form a progression. To follow the progression from the beginning, be sure to first complete these learning paths:

* Evolve your DevOps practices
* Build applications with Azure DevOps

We also recommend you start at the beginning of this learning path: Deploy applications with Azure DevOps .

If you want to go through just this module, you need to set up a development environment on your Windows, macOS, or Linux system. You need:

* An Azure DevOps organization .
* An Azure subscription .
* A GitHub  account.
* Visual Studio Code .
* .NET Core 3.1 SDK .
* Git .

You can get started with Azure and Azure DevOps for free. You don't need an Azure subscription to work with Azure DevOps, but here you'll use Azure DevOps to deploy to Azure resources that exist in your Azure subscription.

This environment lets you complete the exercises in this and future modules. You can also use it to apply your new skills to your own projects.

**Note**

Keep in mind that you can use Azure DevOps to build and deploy almost any kind of application written in any language. In this module, you'll be working with a .NET Core application written in C#.

You don't need to be an expert in .NET or C# to complete this module. You can apply the patterns you learn here to your own projects that use your favorite programming languages and frameworks.

## Meet the team

You met the Space Game web team at Tailspin Toys in previous modules. As a refresher, here's who you'll work with in this module.



Andy is the development lead.



Amita is in QA.



Tim is in operations.



Mara just joined as a developer and reports to Andy.

Mara has prior experience with DevOps. She's helping the team adopt a more automated process that uses Azure DevOps.

# Design the pipeline

* 15 minutes

In this part, you follow the Tailspin web team as they define their release pipeline for the Space Game website.

When you plan a release pipeline, you usually begin by identifying the stages, or major divisions, of that pipeline. Each stage typically maps to an environment. For example, in the previous module, Andy and Mara's basic pipeline had a Deploy stage that mapped to an Azure App Service instance.

In this module, you promote changes from one stage to the next. Within each stage, you deploy the Space Game website to the environment associated with that stage.

After you define which stages you need, consider how changes are promoted from one stage to the next. Each stage can define the success criteria that must be met before the build can move to the next stage. Azure Pipelines provides several ways to help you control how and when changes move through the pipeline. As a whole, these approaches are used for release management.

In this section, you'll:

* Learn the differences between common pipeline stages, such as Build, Dev, Test, and Staging.
* Understand how manual, scheduled, and continuous deployment triggers enable you to control when an artifact moves to the next stage in the pipeline.
* See how a release approval pauses the pipeline until an approver accepts or rejects the release.

## The meeting

The entire Tailspin web team is gathered together. In Create a release pipeline with Azure Pipelines , the team planned their tasks for the current sprint. Each task relates to building their release pipeline for the Space Game website.

Recall that the team decided on these five tasks for their sprint:

* Create a multistage pipeline
* Connect the web app to a database
* Automate quality tests
* Automate performance tests
* Improve release cadence

The team meets to talk about the first task, Create a multistage pipeline. After defining the pipeline, the team can move from their basic proof of concept to a release pipeline that includes additional stages, quality checks, and approvals.

Amita and Tim are watching Andy and Mara demonstrate the release pipeline a second time. They see that the artifact is built and installed on App Service.

## What pipeline stages do you need?

When you want to implement a release pipeline, it's important to first identify which stages you need. The stages you choose depend on your requirements. Let's follow along with the team as they decide on their stages.

**Tim:** OK, I understand the idea of an automated pipeline. I like how easy it is to deploy to Azure. But where do we go from this demo? We need something we can actually use for our releases.

**Amita:** Right. We need to add some other stages. We currently have no place for a testing stage, for example.

**Tim:** Plus, we need a stage where we can show new features to management. I can't send anything to production without management approval.

**Andy:** Absolutely. Now that we're up to speed on what a release pipeline does, how do we make this pipeline do what we need?

**Mara:** Let's sketch out our requirements to help us plan our next steps. Let's start with what we have.

Mara moves to the whiteboard and sketches the existing pipeline.



**Mara:** The Build stage builds the source code and produces a package. In our case, that package is a .zip file. The Deploy stage installs the .zip file, which is the Space Game website, on an App Service instance. What's missing from our release pipeline?

## Add the Dev stage

**Andy:** I might be biased, but I think we need a Dev stage. This stage should be the first stop for the artifact after it's built. Developers can't always run the entire service from their local development environment. For example, an e-commerce system might require the website, the products database, a payment system, and so on. We need a stage that includes everything the app needs.

In our case, the Space Game website's leaderboard feature reads high scores from an external source. Right now, it reads fictitious scores from a file. Setting up a Dev stage would give us an environment where we can integrate the web app with a real database. That database might still hold fictitious scores, but it brings us one step closer to our final app.

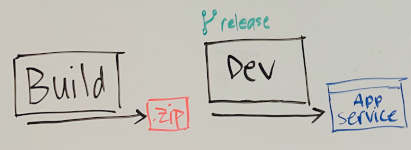
**Mara:** I like it. We won't integrate with a real database yet. But in a Dev stage we can deploy to an environment where we can add a database.

Mara updates her drawing on the whiteboard. She replaces "Deploy" with "Dev" to show the Dev stage.



**Andy:** You bring up an interesting point. We build the app each time we push a change to GitHub. Does that mean each build is promoted to the Dev stage after it finishes?

**Mara:** Building continuously gives us important feedback about our build and test health. But we want to promote to the Dev stage only when we merge code into some central branch: either master or some other release branch. I'll update the drawing to show that requirement.



**Mara:** I think this promotion will be easy to accomplish. We can define a condition that promotes to the Dev stage only when changes happen on a release branch.

## What are conditions?

In Azure Pipelines, a condition enables you to run a task or job based on the state of the pipeline. You worked with conditions in previous modules.

Remember, some of the conditions that you can specify are:

* Only when all previous dependent tasks have succeeded.
* Even if a previous dependency has failed, unless the run was canceled.
* Even if a previous dependency has failed, even if the run was canceled.
* Only when a previous dependency has failed.
* Some custom condition.

Here's a basic example:

ymlCopy

steps:

- script: echo Hello!

condition: always()

The always() condition causes this task to print "Hello!" unconditionally, even if previous tasks failed.

This condition is used if you don't specify a condition:

ymlCopy

condition: succeeded()

The succeeded() built-in function checks whether the previous task succeeded. If the previous task failed, this task and later tasks that use the same condition are skipped.

Here you want to build a condition that specifies:

* The previous task succeeded.
* The name of the current Git branch is "release".

To build this condition, you use the built-in and() function. This function checks whether each of its conditions is true. If any condition isn't true, the overall condition fails.

To get the name of the current branch, you use the built-in Build.SourceBranchName variable. You can access variables within a condition in a few ways. Here you use the variables[] syntax.

To test a variable's value, you can use the built-in eq() function. This function checks whether its arguments are equal.

With that in mind, you apply this condition to run the Dev stage only when the current branch name is "release":

ymlCopy

condition: |

and

(

succeeded(),

eq(variables['Build.SourceBranchName'], 'release')

)

The first condition in the and() function checks whether the previous task succeeded. The second condition checks whether the current branch name equals "release".

In YAML, you use the pipe (|) syntax to define a string that spans multiple lines. You could define the condition on a single line, but we write it this way to make it more readable.

**Note**

In this module, we use the release branch as an example. You can combine conditions to define the behavior that you need. For example, you could build a condition that runs the stage only when the build is triggered by a pull request against the master branch.

You'll work with a more complete example shortly when you set up the Dev stage.

A more complete description of conditions in Azure Pipelines can be found in the expressions documentation .

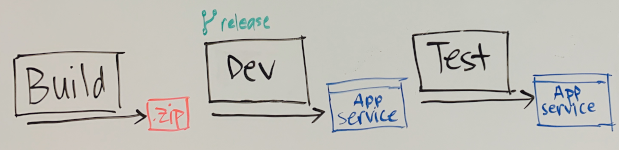
**Mara:** Conditions let us control which changes are promoted to which stages. We can produce a build artifact for any change to validate our build and confirm that it's healthy. When we're ready, we can merge those changes into a release branch and promote that build to the Dev stage.

## Add the Test stage

**Mara:** So far, we have the Build and Dev stages. What comes next?

**Amita:** Can we add the Test stage next? That seems like the right place for me to test out the latest changes.

Mara adds the Test stage to her drawing on the whiteboard.



**Amita:** One concern I have is how often I need to test the app. An email notifies me whenever Mara or Andy makes a change. Changes happen throughout the day, and I never know when to jump in. I think I'd like to see a build once a day, maybe when I get in to the office. Can we do that?

**Andy:** Sure. Why don't we deploy to Test during nonworking hours? Let's say we send you a build every day at 3 A.M.

**Mara:** That sounds good. We can always manually trigger the process as well if we need to. For example, we can trigger it if we need you to verify an important bug fix right away.

Mara updates her drawing to show that the build moves from the Dev stage to the Test stage at 3 A.M. each morning.



### What are triggers?

**Amita:** I'm feeling better now that we know how one stage moves to another. But how do we control when a stage runs?

**Mara:** In Azure Pipelines, we can use triggers. A trigger defines when a stage runs. Azure Pipelines provides a few types of triggers. Here are our choices:

* A continuous integration (CI) trigger
* A pull request (PR) trigger
* A scheduled trigger
* A build completion trigger

CI and PR triggers let us control which branches participate in the overall process. For example, we want to build the project when a change is made on any branch. A scheduled trigger starts a deployment at a specific time. A build completion trigger runs a build when another build, such as one for a dependent component, finishes successfully. It seems like we want a scheduled trigger.

### What are scheduled triggers?

A scheduled trigger uses cron syntax  to cause a build to run on a defined schedule.

On Unix and Linux systems, cron is a popular way to schedule jobs to run on a set time interval or at a specific time. In Azure Pipelines, scheduled triggers use the cron syntax to define when a stage runs.

A cron expression includes fields that match certain time parameters. Here are the fields:

Copy

mm HH DD MM DW

\ \ \ \ \\_\_ Days of week

\ \ \ \\_\_\_\_ Months

\ \ \\_\_\_\_\_\_ Days

\ \\_\_\_\_\_\_\_\_ Hours

\\_\_\_\_\_\_\_\_\_\_ Minutes

For example, this cron expression describes "3 A.M. every day": 0 3 \* \* \*

A cron expression can include special characters to specify a list of values or a range of values. In this example, the asterisk (\*) matches all values for the **Days**, **Months**, and **Days of week** fields.

Put another way, this cron expression reads as:

* At minute 0,
* At the third hour,
* On any day of the month,
* On any month,
* On any day of the week,
* Run the job

To specify 3 A.M. only on days Monday through Friday, you would use this expression: 0 3 \* \* 1-5

**Note**

The time zone for cron schedules is Coordinated Universal Time (UTC), so in this example, 3 A.M. refers to 3 A.M. in UTC. In practice, you might want to adjust the time in your cron schedule relative to UTC so that the pipeline runs at the expected time for you and your team.

To set up a scheduled trigger in Azure Pipelines, you need a schedules section in your YAML file. Here's an example:

ymlCopy

schedules:

- cron: '0 3 \* \* \*'

displayName: Deploy every day at 3 A.M.

branches:

include:

- release

always: false

In this schedules section:

* cron specifies the cron expression.
* branches specifies to deploy only from the release branch.
* always specifies whether to run the deployment unconditionally (true) or only when the release branch has changed since the last run (false). Here we specify false because we need to deploy only when the release branch has changed since the last run.

The entire pipeline runs when Azure Pipelines executes a scheduled trigger. The pipeline also runs under other conditions, such as when you push a change to GitHub. To run a stage only in response to a scheduled trigger, you can use a condition that checks whether the reason for the build is a scheduled run.

Here's an example:

ymlCopy

- stage: 'Test'

displayName: 'Deploy to the Test environment'

condition: and(succeeded(), eq(variables['Build.Reason'], 'Schedule'))

This stage, Test, runs only when the previous stage succeeds and the built-in Build.Reason pipeline variable equals Schedule.

You'll see a more complete example later in this module.

**Amita:** I like this. I don't even have to pick up the release manually and install it. It will be ready for me.

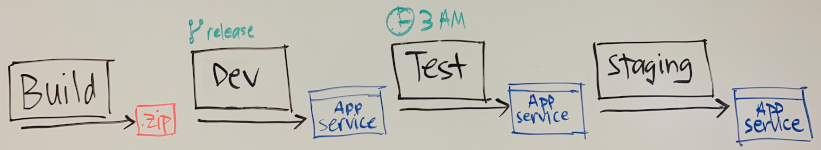
**Andy:** And remember, if we want to automate more later, we can. Nothing's written in stone. The pipeline evolves as we improve and learn.

## Add the Staging stage

**Tim:** It's my turn. I need a stage to run additional stress tests. We also need a stage where we can demo to management to get their approval. For now, we can combine those two needs into a stage that we can call Staging.

**Andy:** Well said, Tim. Having a staging, or preproduction, environment is important. This staging environment is often the last stop before a feature or bug fix reaches our users.

Mara adds Staging to her drawing on the whiteboard.

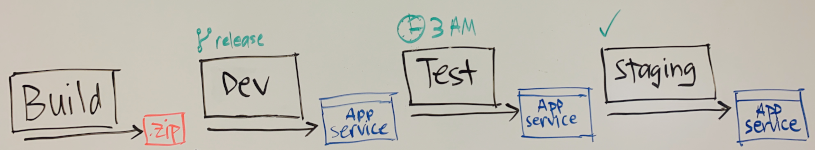


**Amita:** We use a scheduled trigger to promote changes from the Dev stage to the Test stage. But how will we promote changes from Test to Staging? Does that promotion also have to happen on a schedule?

**Mara:** I think the best way to handle that would be a release approval. A release approval lets you manually promote a change from one stage to the next.

**Amita:** That sounds like exactly what I need! A release approval would give me the time to test out the latest changes before we present the build to management. I can promote the build when I'm ready.

Mara updates her drawing to show that the build moves from Test to Staging only when Amita approves it.



**Tim:** I could also imagine us using release approvals to promote from Staging to Production after management signs off. I can never predict how long that takes. After they sign off, I can approve the release and promote it to production manually. But how do release approvals work?

### What are release approvals?

A release approval is a way to pause the pipeline until an approver accepts or rejects the release. You can combine approvals, conditions, and triggers to define your release workflow.

Recall that in Create a release pipeline with Azure Pipelines , you defined an environment in your pipeline configuration to represent your deployment environment. Here's an example from your existing pipeline:

ymlCopy

- stage: 'Deploy'

displayName: 'Deploy the web application'

dependsOn: Build

jobs:

- deployment: Deploy

pool:

vmImage: 'ubuntu-18.04'

environment: dev

variables:

- group: Release

In this example, Azure Pipelines creates the **dev** environment for you if it doesn't already exist. You can also define an environment that includes specific criteria for your release. The criteria can specify which pipelines can deploy to that environment and what human approvals are needed to promote the release from one stage to the next.

Later in this module, you'll define the **staging** environment and assign yourself as an approver to promote the Space Game web application from the Test stage to Staging.

## Automate as little or as much as you need

Azure Pipelines gives you the flexibility to automate some stages while manually controlling stages that aren't ready for automation.

**Tim:** I like how we can define the criteria that promote changes from one stage to the next. But we've defined some manual criteria in our pipeline. I thought DevOps was about automating everything.

**Mara:** You raise a good point. DevOps is really about automating repetitive and error-prone tasks. Sometimes human intervention is necessary. For example, we get approval from management before we release new features. As we get more experience with our automated deployments, we can automate more of our manual steps to speed up the process. We can automate additional quality checks in the Test stage so Amita doesn't have to approve each build, for example.

**Tim:** Sounds great. Let's go with this plan for now and see how we can speed up the system later.

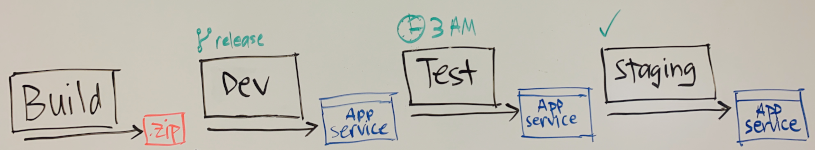
**Amita:** Speaking of our plan, can we summarize our next steps?

## The plan

Let's review the Tailspin team's plan as they move toward next steps.

**Mara:** Here's the release pipeline we want to build.

Mara points to the whiteboard.



**Mara:** To summarize, our steps are to:

1. Produce a build artifact each time we push a change to GitHub. This step happens in the Build stage.
2. Promote the build artifact to the Dev stage. This step happens automatically when the build stage succeeds and the change is on the release branch.
3. Promote the build artifact to the Test stage each morning at 3 A.M. We use a scheduled trigger to promote the build artifact automatically.
4. Promote the build artifact to Staging after Amita tests and approves the build. We use a release approval to promote the build artifact.

After management approves the build, we can deploy the build artifact to a production environment.

**Amita:** Is this going to be hard to do? It seems like a lot of work.

**Mara:** I don't think it will be too bad. Every stage is separate from every other stage. Stages are discrete. Each stage has its own set of tasks. What happens in the Test stage, for example, stays in the Test stage.

Every deployment stage in our pipeline also has its own environment. For example, when we deploy the app to Dev or Test, the environment is an App Service instance.

Finally, we test only one release at a time. We never change releases in the middle of the pipeline. We use the same release in the Dev stage as in the Staging stage, and every release has its own version number. If the release breaks in one of the stages, we fix it and build it again with a new version number. That new release then goes through the pipeline from the very beginning.

## A few words about quality

You've just seen the team design a pipeline that takes their app all the way from build to staging. The whole point of this pipeline isn't just to make their lives easier. It's to ensure the quality of the software they're delivering to their customers.

How do you measure the quality of your release process? The quality of your release process can't be measured directly. What you can measure is how well your process works. If you're constantly changing the process, this might be an indication that there's something wrong. Releases that fail consistently at a particular point in the pipeline might also indicate that there's a problem with the release process.

Do the releases always fail on a particular day or time? Do they always fail after you deploy to a particular environment? Look for these and other patterns to see if some aspects of the release process are dependent or related.

A good way to keep track of your release process quality is to create visualizations of the quality of the releases. For example, add a dashboard widget that shows you the status of every release.

When you want to measure the quality of a release itself, you can perform all kinds of checks within the pipeline. For example, you can execute different types of tests, such as load tests and UI tests while running your pipeline.

Using a quality gate is also a great way to check the quality of your release. There are many different quality gates. For example, work item gates can verify the quality of your requirements process. You can also add additional security and compliance checks. For example, do we comply with the 4-eyes principle, or do we have the proper traceability?

As you progress through this learning path, you'll see a number of these techniques put into practice.

Finally, when you design a quality release process, think about what kind of documentation or release notes that you'll need to provide to the user. Keeping your documentation current can be difficult. You might want to consider using a tool, such as the Azure DevOps Release Notes Generator . The generator is a function app that contains a HTTP triggered function. It creates a Markdown file whenever a new release is created in Azure DevOps, using Azure Blob Storage.