

SD501013

# Deploy Better Plug-ins Faster Through CI/CD and Unit Testing in Azure DevOps

Andrea Tassera  
Woods Bagot

## Learning Objectives

- Discover the importance and benefits of a DevOps culture.
- Learn about collaborating in a safer space, empowering your teams, and giving space for creativity.
- Learn about applying delivery processes during development, allowing faster development and release.
- Learn about building better, more reliable tools and learn how to assess quality sooner.

## Description

At Woods Bagot, we managed to cut the deployment time of our tools from days to minutes, while increasing quality and reliability, by repurposing common concepts and tools from the software development world. By implementing a DevOps philosophy in your workflow, not only can you deliver automation tools quicker, but you can also empower your teams from the start, improving communication and collaboration and promoting professional growth. You can increase the quality and reliability of the deployed software product thanks to automated testing and a faster feedback cycle. Moreover, with the ability of more frequent and faster releases, backed up by live monitoring systems, developers can deploy tools that are tailored to the users and that really respond to their needs. The session will showcase how Woods Bagot has developed a full workflow for its Revit toolbar through the use of an Azure DevOps pipeline—but you can apply the principles to any platform that benefits from automation tools.

## Speaker(s)

With nearly 10 years of experience in the Design Technology / Computation field, Andrea now manages the Automation efforts for Woods Bagot. Leading a dedicated agile team, Andrea oversees that the company is taking advantage of all the benefits of internally developed tools, coming up with new ways of doing things better.

## Introduction

In the world of software development, DevOps is a pretty established and common concept. Its benefits are largely understood in the community and most companies apply them in their day-to-day work, from large to small companies.

This has only occasionally permeated into the world of Design Technology and the AEC world. Or perhaps it's more common than I think, but not many people talk about it. Anyway, that's the reason why I decided to share what we're doing at Woods Bagot with a wider audience.

After years of experience in developing plugins for Revit we started noticing that the development of the tool was the "easy bit", but the journey wasn't finishing there. Once the tool was developed, first, we had to find a way to get the right people to access the code we had written. Initially we thought that sending a library file (dll) to someone via Teams/Slack was a good enough approach. We quickly realized this approach wasn't scalable or efficient at all. We had other approaches, but none of them was quick or flexible or structured enough to have the dev team running smoothly. Also, if software development is the practice that transforms coffee into bugs, how could we ensure that the number of bugs in the code wasn't so high for the plugin to be unusable? Constantly running after users to test the plugin for us was neither easy nor pleasant.

That's how we got curious of "how does Google or Facebook do it?" and we became fascinated by the concept of DevOps. This is how the Design Technology team at Woods Bagot managed to cut down time and useless frustration when deploying Revit plugins to 1000+ people in 17 offices around the globe. Not an easy task.

## Assumptions

Before jumping deeper into what this means, let's remember that this AU session shouldn't be taken as a DevOps manual for the software world. Keep in mind that this is "made by architects for architects". With all the simplifications and shortcuts that can be expected. Moreover, we'll be talking about Revit plugins, more similar to what you'd do with desktop applications (but not entirely) and definitely not with web applications.

This process is something that happens at the end of your development process. That part (the development of the tool) stays pretty much unaltered. The only difference that I believe is important to point out, is that at Woods Bagot we use [pyRevit](#) to deploy our tools to our community of designers. We really love it. Kudos to Ehsan and contributors! ❤️

## What is DevOps and what are the benefits

[DevOps](#) is best described as a philosophy rather than something technical. It is a set of cultural shifts and practices that, together with some tools to make things practical, increases a team's efficiency, speed and ability to collaborate, while removing siloes.

The term *DevOps* is the combination of *Dev*, the software development team and *Ops*, or the IT operations team. This is because the practice intends to merge the two teams into one cooperating entity and it empowers the developers to deploy to the users and have a more direct relationship with them.

DevOps aims at removing the barriers between teams, that usually slow down the whole process, empowering people to be more efficient.

Thanks to the automation of processes that were previously manual, slow and prone to human error, it's now easier and more reliable to have frequent commits and releases to the users, accelerating to a quicker feedback cycle. This obviously means a happier customer (be it internal to the organization, or external) that is using a tool that works better for them, having them influenced its creation more.

Following the direction of DevOps, not only we're ensuring better quality and reliability of the tools thanks to advantages given by deploying rapidly, but we're also creating a healthy, highly collaborative environment where to work. The developers team works better internally thanks to a reliable process in place and have a more fulfilling experience working closer to their users. Moreover, thanks to automated testing they feel more confident that many bugs can be fixed before the tool is shipped to the end user.

Ultimately this enables us to create better tools, more reliable, with less bugs, that are more fitting to the users' needs and this allows us to win credibility with the teams and the company.

### **The armed wing of DevOps: CI/CD**

The technical backbone to this philosophy, that allows speed, frequency and rapidity of delivery are the practices of *Continuous Integration* and *Continuous Delivery*.

*Continuous Integration* refers to the practice where software developers merge (*integrate*) their code changes into a central repository. This triggers a pipeline of tasks, including automated builds, tests and other actions, that will, eventually, culminate in the deployment of the application developed.

The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

This not only makes it easier to merge code from different developers and reduces the number of legacy bugs accumulating, but also allows the devs to respond to users' needs faster, creating more fitting tools.

*Continuous Delivery* expands and extends upon the idea of CI, automatically preparing the code changes for the release. After the build stage (and any other optional steps) the code is deployed to a testing environment and it will require the action from a human that will consciously deploy to production to the "customers".

There is a second interpretation that translates CD to *Continuous Deployment*. In this case the deployment process is fully automated and doesn't require human interaction at all. If everything went fine, the code changes will be shipped to production. This is not a common practice and it's not going to be covered here.

## Case study

What you will see in the handout is what the Design Technology team at Woods Bagot created in order to cut down time and useless frustration when deploying Revit plugins to 1000+ people in 17 offices around the globe.

The pipeline we employ for most of our software projects is one specific example of CI/CD pipelines and it's closely tailored around Revit (even though we employ similar pipelines in other applications, including Rhino) and, as an overview, it is composed of a few steps:

1. Pipeline creation and setup;
2. Packages (NuGet) installation;
3. Solution building;
4. Code versioning;
5. Code signing;
6. Automated testing;
7. Deployment.

### Before we start

To replicate what you'll see following, you'll need to have a remote online version-controlled repository with your project. GitHub, Bitbucket and the likes will all work.

An Azure DevOps account is also necessary in order to run your pipelines there. Associated to this Azure account you'll need a [self-hosted](#) Windows Virtual Machine where you pre-install Revit. This will be necessary in order to run automated tests on your code.

Optionally, you will need a code certificate to sign your code. This is not mandatory, but you will see the benefits in the dedicated paragraph.

Finally, as we mentioned above, we'll be using pyRevit for the deployment.

### Note

The following code is in screenshot form instead of text. That is intentional as it is very unlikely that you'll be able to copy and paste our code into your project and it will just work. The exercise now is understanding how the process works, so then you can replicate for yourself.

Azure helps you with the creation of the pipeline and the setup of all the variables and moving parts. That's much better than copy-pasting the code!

## 1. Pipeline setup

The first thing to do would be to log into the Azure DevOps portal and create a new DevOps project, by clicking the “+ *New Project*” button.

Once created, it will be possible to navigate inside of the project, then to the pipelines section. Click the *New pipeline* button to create a new pipeline.

At this point you’ll be asked to connect to an online repo. Just follow the prompts.

You can now access the pipeline’s YAML file that has been automatically created by Azure DevOps and it will allow you to “drive” the pipeline itself and define the actions that will be run. To modify and write the YAML file, you can either update the local repository and open it from any text editor, or you can modify it from the browser directly. Doing it from the browser on the DevOps project’s page will enhance the experience as it will guide you through the creation of new steps and populating their parameters.

When you’re in the pipeline, the first thing to do is defining the variables. First, you need to define the *trigger*, which is what will start running your pipeline automatically. Then you define the *pool*, which represents the machine (virtual or not) that will run the pipeline. This is when we need to use the self-hosted Windows VM with Revit installed that we were previously mentioning. Do not use Microsoft-hosted VMs here.

Finally, some variables need to be defined, including *GitVersion.SemVer* which we’ll see in a few paragraphs.

```
1 # .NET Desktop
2 # Build and run tests for .NET Desktop or Windows classic desktop solutions.
3 # Add steps that publish symbols, save build artifacts, and more:
4 # https://docs.microsoft.com/azure/devops/pipelines/apps/windows/dot-net
5
6 trigger:
7   - master
8
9   pool: 'Default'
10
11 variables:
12   solution: '**/*.sln'
13   buildPlatform: 'Any CPU'
14   buildConfiguration: 'Debug'
15   GitVersion.SemVer: ''
16
17 steps:
18
19   - task: gitversion/setup@0
20     displayName: SetupGitVersion
21     inputs:
22       versionSpec: '5.3.2'
23
24   - task: gitversion/execute@0
25     displayName: GitVersion
26     inputs:
27       updateAssemblyInfo: true
```

## 2. NuGet tasks

A Visual Studio solution wouldn't build if it cannot find the referenced packages. Therefore we need the following NuGet tasks that follow.

First, we need to install the actual NuGet tool. When NuGet is ready, we can use it to install the packages for us.

To do that you use a *restore* command that will install all the external packages that are referenced by the projects in the solution.

In case you have any, you can conveniently install internal packages too, like for example class libraries you created for your company. In that case you need to provide the GUID that references your package in the *vstsFeed* parameter.

```
14  <!-- Configuration -->
15  <!-- Version -->
16
17  <steps>
18
19    <task
20      <name>GetVersionSetup
21      <displayname>SetupGetVersion
22      <inputs>
23        <input name="versionSpec" value="1.0.0" />
24      </inputs>
25    </task>
26
27    <task
28      <name>GetVersion
29      <displayname>GetVersion
30      <inputs>
31        <input name="updateAssemblyInfo" value="true" />
32        <input name="updateAssemblyInfoFilename" value="$(Build.SourcesDirectory)\obj\$(SolutionName)\ProjectName\Properties\AssemblyInfo.cs" />
33      </inputs>
34    </task>
35
36    <task name="NuGetToolInstaller@1" />
37
38    <task name="NuGetCommand@2" />
39      <inputs>
40        <input name="command" value="restore" />
41        <input name="restoreSolution" value="**/*.sln" />
42        <input name="feedsToUse" value="select" />
43        <input name="vstsFeed" value="<GUID-for-private-nuget>" />
44      </inputs>
45
46    <task name="VSTSSign" />
47
48  </steps>
```

### 3. Build solution

Building the solution is the most straight forward step if all previous tasks have been set up correctly.

Building a solution on Azure is just the same as building locally on your machine and all the setup happens in the *csproj* file.

The only real difference is with referencing the Revit APIs. When developing and building locally you'd normally reference the API library files that have been installed on your C drive when you installed Revit.

Because we're not building locally in this case though, this might cause some errors. For that reason, we use an external, publicly available NuGet package with those APIs.

```
21 - inputs:
22   versionSpec: '5.3.2'
23
24 - task: gitversion/execute@0
25   displayName: GitVersion
26 - inputs:
27   updateAssemblyInfo: true
28   updateAssemblyInfoName: '$(Build.SourcesDirectory)\obj\$(BuildConfiguration)\Properties\AssemblyInfo.cs'
29
30 - task: NuGetFeedInstall@0
31
32 - task: NuGetCommand@0
33 - inputs:
34   command: 'restore'
35   restoreSolution: '**\*.sln'
36   feedsToUse: 'select'
37   vstsFeed: '4000-for-private-nuget'
38
39 - task: VSBuild@1
40   displayName: Build
41 - inputs:
42   solution: '$(solution)'
43   platform: '$(buildPlatform)'
44   configuration: '$(buildConfiguration)'
45   versioningScheme: byEnvVar
46   versionEnvVar: 'GitVersion.SemVer'
47
```

### 4. Semantic versioning

This step is not mandatory to get the pipeline working, but still it gives important benefits and it's generally considered good practice.

It is possible to automatically version your code from the pipeline directly, following the [semantic version](#) format convention, based on the git history of your project. Therefore, for this step to succeed, your repository will need to have at least a *main/master* branch and a *develop* branch.

At the top of the pipeline we create a *GitVersion.SemVer* that will store the version number to be used later in the pipeline.

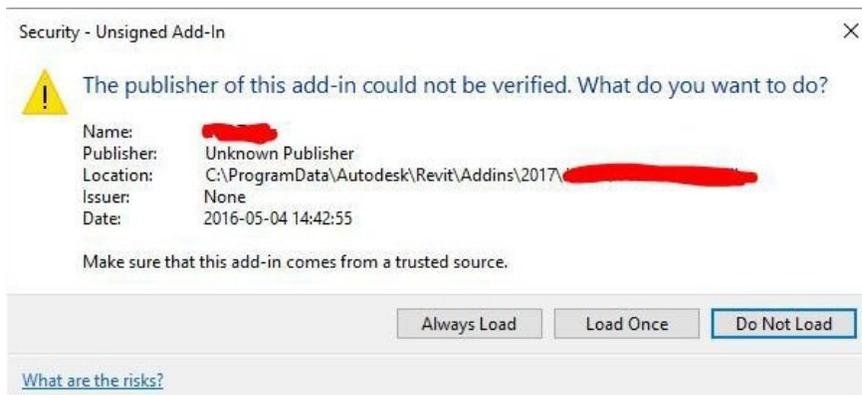
The following steps are quite straight forward as you need to install the *GitVersion* tool and run it, so it calculates the version for you.

That's the version number that is stored in your variable and that's going to be fed to the build step in the *versionEnvVar* variable. When the pipeline builds it will also assign that version number to the files.

```
3 # See https://aka.ms/azure-devops-ci-how-to for details on pipeline tasks and triggers.
4 # https://docs.microsoft.com/azure/devops/pipelines/apps/windows/dot-net
5
6 - triggers:
7   - master
8
9   pool: 'Default'
10
11  variables:
12    solution: '**/*.sln'
13    buildPlatform: 'Any CPU'
14    buildConfiguration: 'Debug'
15    GitVersion.SemVer: ''
16
17  steps:
18
19  - task: gitversion/setup@0
20    displayName: SetupGitVersion
21    inputs:
22      versionSpec: '5.3.2'
23
24  - task: gitversion/execute@0
25    displayName: GitVersion
26    inputs:
27      updateAssemblyInfo: true
28      updateAssemblyInfoFilename: '$(Build.SourcesDirectory)\Wb.SolutionName\ProjectName\Properties\AssemblyInfo.cs'
29
30  - task: NuGetToolInstaller@1
31
32  - task: NuGetCommand@2
33    inputs:
34      command: 'restore'
35      restoreSolution: '**/*.sln'
36      feedsToUse: 'select'
37      vstsFeed: '<GUID-for-private-nuget>'
38
39  - task: VSBuild@1
40    displayName: Build
41    inputs:
42      solution: '$(solution)'
43      platform: '$(buildPlatform)'
44      configuration: '$(buildConfiguration)'
45      versioningScheme: byEnvVar
46      versionEnvVar: 'GitVersion.SemVer'
```

## 5. Code signing

This step is also not mandatory, but it gives a huge benefit: it avoids popping up that annoying “Unsigned Add-In” message to your users.



Also, and more importantly, this step imprints a digital signature to the outgoing dll files, in order to confirm the author of the software and guarantee that the code has not been corrupted.

You have two (or probably more, but two that we considered) options to sign your code: the first one would be using a certificate that has been stored in the pipeline itself, and that would look as in the screenshot below. That’s how we were initially doing it at Woods Bagot.

```
42 | solution: '$(solution)'
43 | platform: '$(buildPlatform)'
44 | configuration: '$(buildConfiguration)'
45 | versioningScheme: hybrid
46 | versionEnvVar: '$(Version_Suffix)'
47 |
48 | # Sign Revit dll
49 | - task: codesigning@2
50 |   displayName: CodeSigning
51 |   inputs:
52 |     secureFileId: 'NameOfCertificate'
53 |     signCertPassword: 'CertificatePassword'
54 |     files: '$(Build.SourcesDirectory)\Path\To\Dlls.dll'
55 |     timeServer: 'http://timestamp.digicert.com'
56 |     hashingAlgorithm: 'SHA256'
57 |
58 | - task: PowerShell@1
59 |   displayName: PowerShellRunRevitTests
60 |   inputs:
61 |     targetType: 'inline'
62 |     scripts: |
63 |       cd ..\..\..\
64 |       cd C:\ProgramData\RevitTestFramework\bin\AnyCPU\Debug
65 |       .\RevitTestFrameworkConsole.exe -dir: PathToFramework -a: PathToAssembly\RevitTests.dll -r: PathTo
```

The other option, which is how we do things now, is by storing the certificate on Azure KeyVault. This second option is more efficient because in that way you don’t have to store the certificate in every single pipeline you create. And you don’t even have to upload the new certificate to all the

pipelines every time the certificate expires. Or even worse, you don't have to spend hours trying to understand why the pipeline isn't working and then find out that's just because you are using an expired certificate.

We're not going into the details of how to do this as it would take us off topic, but there's plenty of documentation on how to use [KeyVault](#).

## 6. Automated testing

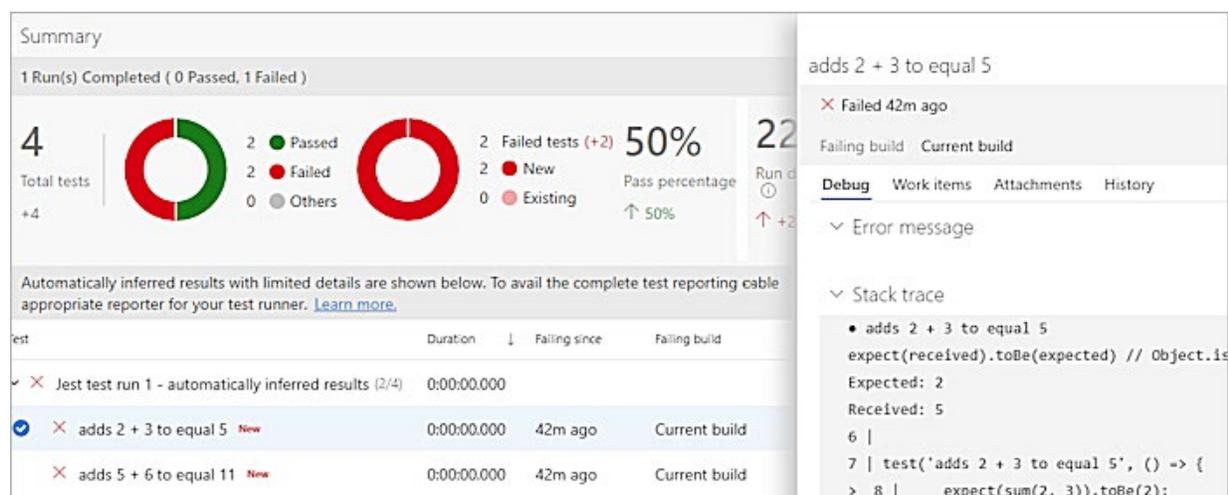
This step is also not mandatory, but it is the biggest time saver in the whole pipeline. This means that the pipeline is able to run successfully until the end even if you don't have a test step, but you're missing out on a great opportunity to save some serious time.

Without going into too much depth about testing (it is an extremely broad topic and would bring us way too far off topic, I'd suggest you read some specific literature about it) we'll just say that tests are simply other pieces of code, that live under a dedicated project in your solution and will perform some sort of sanity checks on the methods that you just wrote.

Let's consider an oversimplified example: you're creating a Revit plugin that will create an arbitrary number of sheets. To do so, at some point, you'll probably write a method that creates one sheet, right? In a test-oriented mindset, at the same time, you'll also write, in the tests project, a test to check if the sheet has actually been created. This is unit testing. More or less.

Be aware that writing tests for your code is time-consuming and not always the most fun among all activities, but, especially in the long run, it will pay back the initial time investment, as they will be run over and over every time you commit to the specified branch.

The pipeline will run the tests and, if everything went fine and all the tests have passed, it will keep running and go to the next step. On the contrary, if even just one of the tests failed, for any reason, the pipeline will stop and notify the error in an informative view like the one in the screenshot below.



The screenshot displays a test results summary and a detailed view of a failed test. The summary shows 4 total tests, with 2 passed and 2 failed, resulting in a 50% pass percentage. The failed test is 'adds 2 + 3 to equal 5', which failed 42m ago in the current build. The stack trace shows the test function expecting 2 but receiving 5.

test	Duration	Failing since	Failing build
✓ Jest test run 1 - automatically inferred results (2/4)	0:00:00.000		
✗ adds 2 + 3 to equal 5 <span style="color: red;">New</span>	0:00:00.000	42m ago	Current build
✗ adds 5 + 6 to equal 11 <span style="color: red;">New</span>	0:00:00.000	42m ago	Current build

```
adds 2 + 3 to equal 5
✗ Failed 42m ago
Failing build Current build
Debug Work items Attachments History
Error message
Stack trace
• adds 2 + 3 to equal 5
  expect(received).toBe(expected) // Object.is
  Expected: 2
  Received: 5
  6 |
  7 | test('adds 2 + 3 to equal 5', () => {
    > 8 |     expect(sum(2, 3)).toBe(2);
```

This, we understand, saves a lot of time as, if planned and developed correctly, the tests will spot bugs before they are shipped to our users (hence before they freak out and send us

thousands of angry emails) and allows the developers to fix them before they become a real bug in production.

The tool in question will still need to be tested by some users, but at this point the testers will mostly focus on usability, user experience and other similar factors, instead of more trivial bugs.

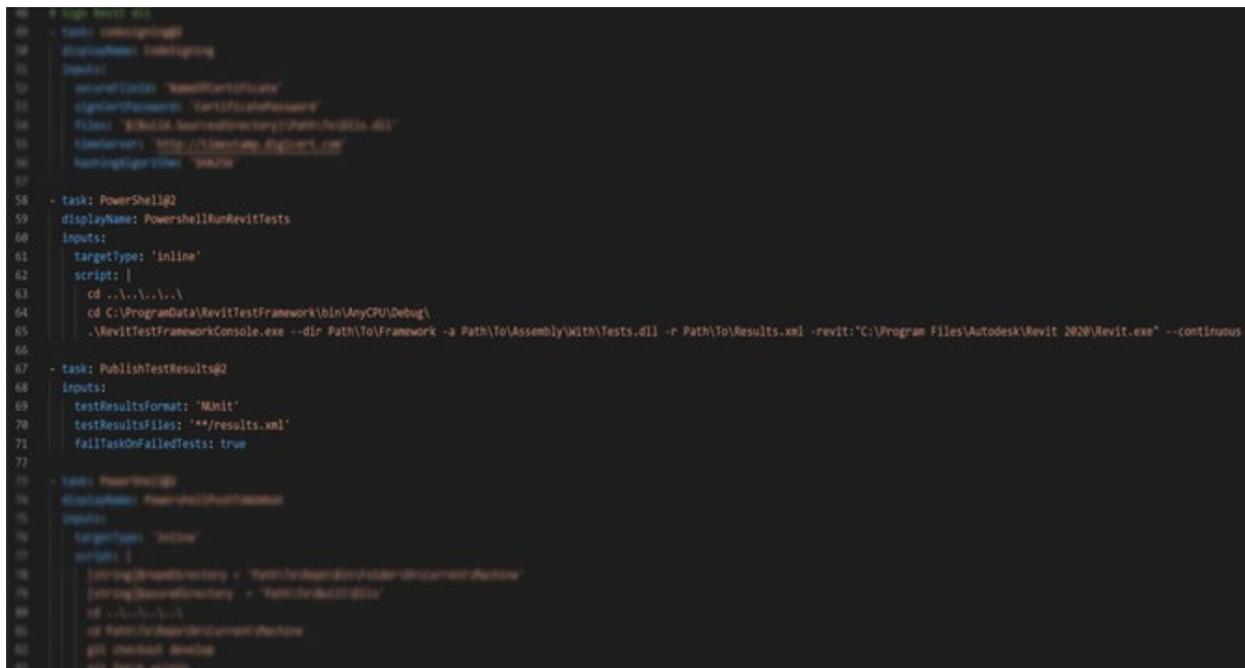
The situation at hand is understandably different from what canonically happens in the software development world, at we need to test code that will run on Revit. Luckily enough some nice people out there have developed systems to do so.

At Woods Bagot we run our Revit tests using [RevitTestFramework](#), developed by the Dynamo team, but you have a few options, as the Speckle team developed their own framework called [xUnitRevit](#). Big kudos to the teams ❤️.

Running the tests consists in running a Powershell script. All the information necessary is available from the user manual in the link above.

If you access the VM while the pipeline is running this step, you will see Revit opening and doing something in the background. This is the test framework taking control and running the tests.

Once all the tests have been run, the results of the test will be “communicated” to the pipeline (through an xml file that you’ve set up following the instructions) so that the pipeline will know if it should continue running or it should stop.



```
48 - task: PublishBuildArtifacts@1
49   displayName: PublishBuildArtifacts
50   inputs:
51     ArtifactPath: 'Build\bin\Debug'
52     PublishLocation: 'ContainerRegistry'
53     PublishProvider: 'AzureArtifactStashProvider'
54     RetentionStrategy: 'Default'
55     ArtifactLocation: 'ContainerRegistry'
56     ArtifactType: 'ContainerRegistry'
57   - task: PowerShell@2
58     displayName: PowerShellRunRevitTests
59     inputs:
60       targetType: 'inline'
61       script: |
62         cd ..\..\..\
63         cd C:\ProgramData\RevitTestFramework\bin\AnyCPU\Debug\
64         .\RevitTestFrameworkConsole.exe --dir PathToFramework -a PathToAssembly\WithTests.dll -r PathToResults.xml -revit:"C:\Program Files\Autodesk\Revit 2020\Revit.exe" --continuous
65     - task: PublishTestResults@2
66     inputs:
67       testResultsFormat: 'JUnit'
68       testResultsFiles: '**\results.xml'
69       failTaskOnFailedTests: true
70     - task: PowerShell@2
71     displayName: PowerShellPublishTestResults
72     inputs:
73       targetType: 'inline'
74       script: |
75         [string]$publishLocation = "PathToContainerRegistry\PathToContainerRegistry"
76         [string]$publishLocation = "PathToContainerRegistry"
77         cd ..\..\..\
78         cd PathToContainerRegistry\PathToContainerRegistry
79         git checkout master
80         git push origin
```

## 7. Deployment

This is the final step in this pipeline. If everything went fine, there was no error in building, versioning, signing, etc. and none of the tests failed, then Azure DevOps is ready to perform the task to deploy the code to the test environment we were mentioning at the beginning.

In our case, this step is deferred to pyRevit. Our internal Revit toolbar at Woods Bagot, called Wombat, is built upon pyRevit because it makes deployment extremely convenient, being it based on git.

If we look more closely at this step, you realize that, similarly to the test step, this is also a Powershell task and, this specifically, is simply running some git commands.

We first *checkout* the correct branch we intend to deploy to and then *pull* from origin to make sure the branch is up to date.

It's then time to take all the files that have been built by the pipeline and that we want to deploy so we can *add* them to the commit. Once that's done, we can actually *commit* appending a meaningful message (that will be repeated every time by the pipeline) and finally *push* to the Wombat repository.

Let's not get confused here: we're talking about two different repositories. The first repository we spoke about at the beginning of the process is the plugin that we're developing at this time. The second pipeline we just spoke about is the container of the plugins. In our Woods Bagot case, for example, the second repository would be the Wombat repository.

At this point pyRevit will work its magic and deploy to the beta testers through the *develop* branch. Awesome!

```
66
67 - task: PublishTestResults@2
68   inputs:
69     testResultsFormat: 'WixTest'
70     testResultsFiles: '**/results.xml'
71     failTaskOnFailedTests: true
72
73 - task: PowerShell@2
74   displayName: PowershellPushToWombat
75   inputs:
76     targetType: 'inline'
77     script: |
78       [string]$repoDirectory = 'Path\To\Repo\Bin\Folder\On\Current\Machine'
79       [string]$azureDirectory = 'Path\To\Built\Dlls'
80       cd ../../../../
81       cd Path\To\Repo\On\Current\Machine
82       git checkout develop
83       git fetch origin
84       git pull origin
85       Get-ChildItem -Path $repoDirectory -Include *.* -File -Recurse | foreach { $_.Delete()}
86       Copy-Item -Force -Recurse $azureDirectory -Destination $repoDirectory
87       git add .
88       git commit -m "commit message"
89       git push https://link/to/remote/repo.git HEAD:develop -q
90
```

## **Conclusion**

At this point I really hope the benefits of implementing a DevOps cultural model in your company are evident.

DevOps and its applications will improve the processes through the help of standardized, repetitive and reliable actions, reducing bugs and therefore saving time and money.

Not only that, but it will establish a better environment for collaboration, where the mentioned processes boost the developers' confidence, emphasizing feelings of ownership and accountability in a safe environment where bugs are spotted before production and where they collaborate more closely while they work on smaller code changes.

Lastly, you're about to deliver better results to your company, not only in wasting less time for development, but by releasing more reliable tools that are more fitting to the users' needs, thanks to a faster feedback cycle.