Paper AD04

Testing Times:
Implementing CI/CD Test Automation Methodology for SAS

Paul Shannon, Amadeus Software, Oxfordshire, UK

ABSTRACT

The benefits of DevOps practices including Continuous Integration and Continuous Delivery (CI/CD) are widely documented throughout the software sector. Significant savings in resource requirements can be achieved while simultaneously improving the quality of the end product. Cloud deployments with SAS Viya means CI/CD can be implemented for SAS Program code and SAS Platforms including such as a validated Statistical Computing Environment (SCE). Test automation can work to support validated SCEs, particularly with applying updates and subsequently the re-validation process.

Using real-world examples, this paper demonstrates how test automation can be applied in a SAS programming environment. Consideration is given to differences for SAS 9.4 and Viya, mock data sources, third-party testing utilities and ultimately how much time can be saved using CI/CD.

INTRODUCTION

Since its inception in 2005, Git has rapidly become the de facto standard solution for source control. The drive for automation of software development and release has seen many tools built on Git source control to provide solutions following the DevOps process and agile software development methods. While such techniques were originally formed from the application development sector, SAS users are now implementing DevOps practices particularly with a view to automation of testing and release. Test automation brings more frequent but smaller changes helps detect breaking changes quickly and simplify the resulting resolution thus ultimately increases code quality whilst reducing delivery time. All this makes planning and risk management easier since progress is incremental, with added improvement in system stability.

Implementing CI/CD for testing SAS programs is much simpler than for other software programming disciplines, as there is no requirement to compile or build code into an executable. YAML syntax allows scripting of deployment, which together with cloud technology creates opportunities for an entire SAS environment (hardware, software installation, and SAS programs) to be version controlled automating deployment and testing. This offers optimal change management. Validated SAS environments will further benefit from automation of validation and revalidation scripts.

THE DEVOPS LIFECYCLE

CI/CD primarily covers the Build and Test (CI) and Release and Deploy (CD) steps of the DevOps lifecycle, as shown in Figure 1. To utilize these automated methods, SAS programs must be managed in a recognized source control system. A Git repository is by far the most common, and easiest source control solution to integrate with, although support for others (e.g.: Subversion) is available, but this varies based on the DevOps solution used. In this paper, Microsoft® Azure DevOps is used for demonstrations, but the methods described in this paper will translate to almost any DevOps solution. Common DevOps solutions include GitHub, Jira and BitBucket.

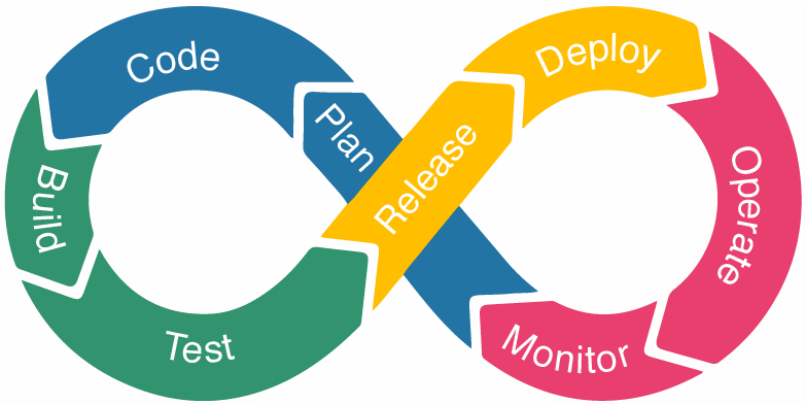

Figure 1: Diagram of the DevOps lifecycle [1]

GIT SOURCE CONTROL

Git source control provides the foundation for most DevOps processes including CI/CD. However, it is possible to use other source control systems such as Subversion as your source repository. All the major DevOps software use Git by default. There is little value in build and deployments being triggered with *every* Git commit by *every* user. In this example, consider a git repository with three types of branch:

- Main: The default for a repository.
- Integration: Used to merge reviewed changes. This is the branch that CI/CD pipelines will target.
- Feature/Bugfix: Branches for code in development. There will be multiple branches of this type.

Using Git branches allows targeted CI/CD on changes that are subject to a pull request thus are peer-reviewed. This leaves the integration branch, for only reviewed code.

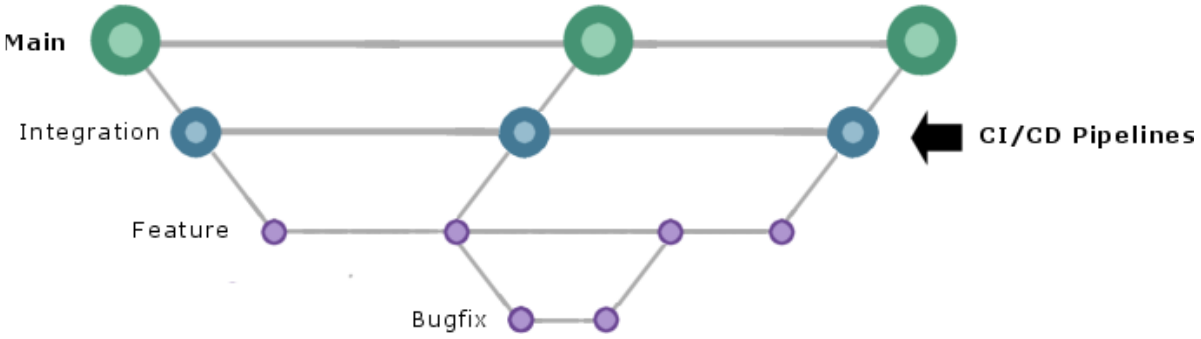

Figure 2: Git Branch Best Practice

FOLDER STRUCTURE

Using Git and CI/CD practices requires new files and file types to be added to your SAS programs. Frequent Git users will be familiar with configuration files such as `.gitignore`. Implementing CI/CD brings the addition of YAML files. It is recommended you ensure clear separation of your files within your Git repository, as this will pay dividends later, particularly for the CD Pipeline.

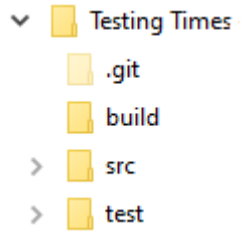

Figure 3: Example folder structure

.git: This is your standard Git folder, containing the local Git repository. You will not edit these files.
Build: Store anything related to CI/CD in this folder, such as your YAML files.
Src: Source files, anything in this folder will be published to your production environment.
Test: Stores everything for your unit tests, programs, reports, logs, and mock data.

UNIT TESTING

This author has seen many different methods for implementing automated unit testing for SAS programs, with some inevitably better than others. Unit testing is a pre-requisite of implementing CI/CD but is not the focus of this paper. There are many papers already written on the topic of unit testing in SAS, some presented previously at PhUSE [2].
Whatever method you use to create unit tests, ensure your testing system meets the following rules:

- Readable, simple tests using the Arrange, Aact, Aaasert structure[3].
- Tests must be deterministic.
- Mock datasets must be realistic and comprehensive.
- Tests must be automated.
- Test one scenario per test.

- Isolate tests, avoid interdependence.
- Ensure tests are repeatable and scalable.

There are various unit testing libraries for SAS. This example uses SASUnit [4]; a mature unit testing library written as a suite of SAS macro programs. SASUnit supports SAS 9.4 and requires minimal modification for SAS Viya. Both Windows and Linux operating systems are supported too. Nonetheless, this is not an endorsement of SASUnit, nor is it the only suitable utility for unit testing. The requirement is to implement a testing method that meets the rules listed above.

BUILDING A PIPELINE FOR SAS PROGRAMS

The method of implementing CI/CD in is known as a DevOps Pipeline. As DevOps tools are commonly built on Git repositories, they are commonly referred to as Git Pipelines. Briefly this comprises a set of processes and tools to build, test and deploy software applications. We will create two separate pipelines: one for CI, one for CD. A Git Pipeline requires a YAML file which described briefly is a language containing the steps for your pipeline.

YAML SYNTAX

Yet Another Markup Language? This may be something you are asking yourself, and this is the full name of YAML. Quirky name aside, the syntax will be familiar to anyone with Python knowledge and is described as a human-readable data-serialization language. A simple example taken from a Git Pipeline is shown below:

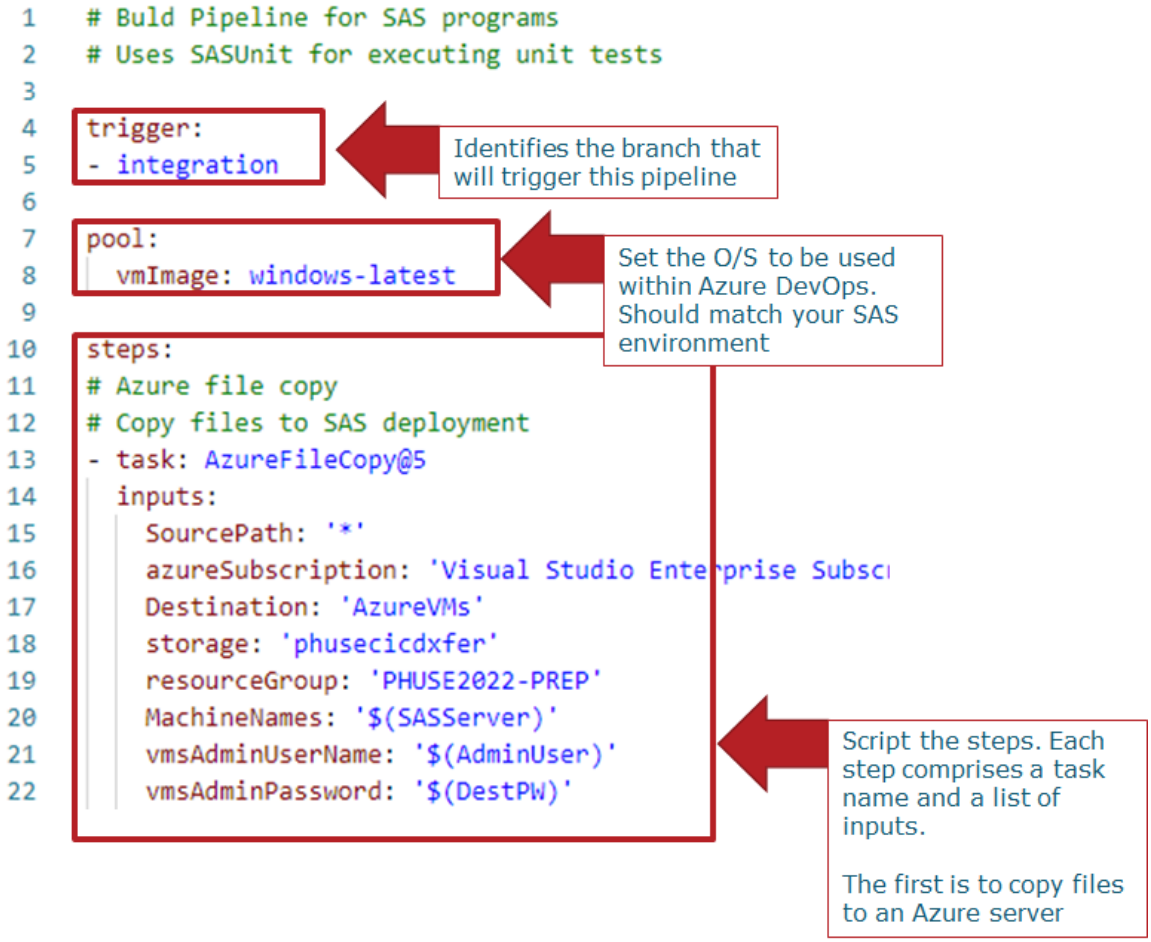

Figure 4: Example of YAML syntax

Note the main highlights of YAML syntax:

- Indentation is critical, used to identify nesting elements. Tabs are not allowed.
- Key-value pairs are denoted by the colon (:) character.
- List items are indicated with a hyphen (-) symbol at the start of the line.
- A comment is denoted by the hash (#) symbol at the start of the line.

CREATING A CONTINUOUS INTEGRATION PIPELINE

We don't need to write the script for our Git Pipeline, Azure DevOps will create it via a wizard. Firstly, let's consider the steps required to be implemented. As mentioned earlier, SAS programs have the benefit of not requiring any compilation, therefore our pipeline will comprise the following steps:

- Trigger the pipeline on change to the Integration branch.
- Copy files to a SAS server, using the Azure File Copy command.
- Batch submit the SASUnit tests.

Figure 5 shows the browser interface for creating a Git Pipeline in Azure DevOps. There is the option to write YAML code directly or use the assistant to aid in creating pipeline steps.

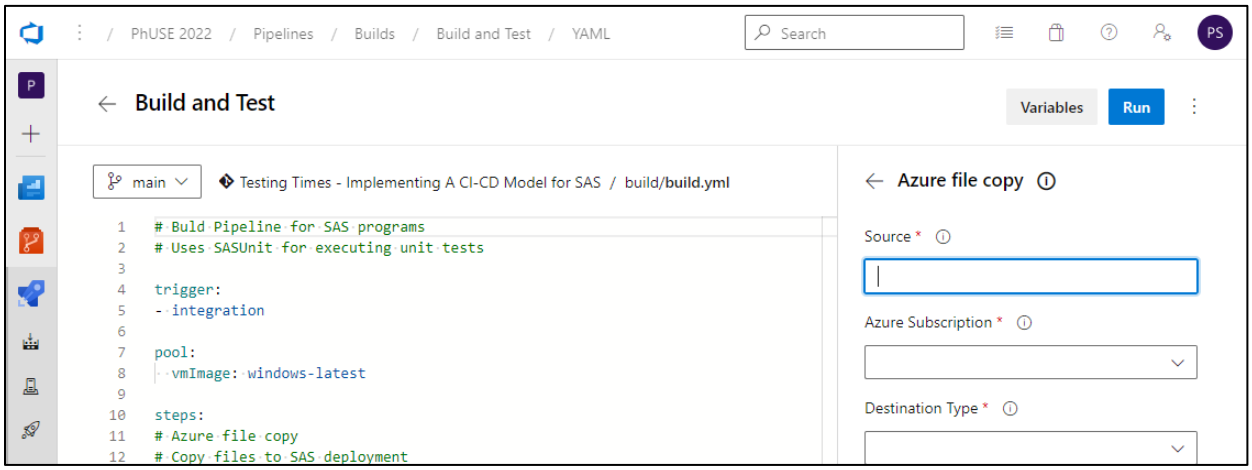

Figure 5: Editing a Git Pipeline in Azure DevOps

PIPELINE STEPS

The Assistant in Azure DevOps provides a user interface to write YAML code for common tasks. CI/CD pipelines for SAS programs will use the following steps:

- Azure File Copy
- PowerShell on Target Machines (Windows SAS deployments)
- SSH (Linux SAS deployments)

PIPELINE VARIABLES

Variables allow parameters for CI/CD steps to be pre-defined and referenced in code. Variables are referenced in YAML syntax by using `$(variable)` notation. This is useful for supporting best practice programming in YAML. Examples of values stored in variables are:

- Target server name(s)
- Publish/test folder location
- Connection credentials

Remember, your YAML code is committed and stored in your Git repository. It is a security necessity to avoid storing hard-coded credentials such as passwords in your repository. You can create variables with a "secret" option which prevents them from being revealed once set.

Figure 6 shows examples of variables configured. Notice the `DestPW` (destination password) variable is masked with asterisks. No user in Azure DevOps can reveal the password, although it can be replaced/updated. Similarly, the `PublishFolder` and `TestFolder` variables reference other variables in their value using the `$(variable)` notation.

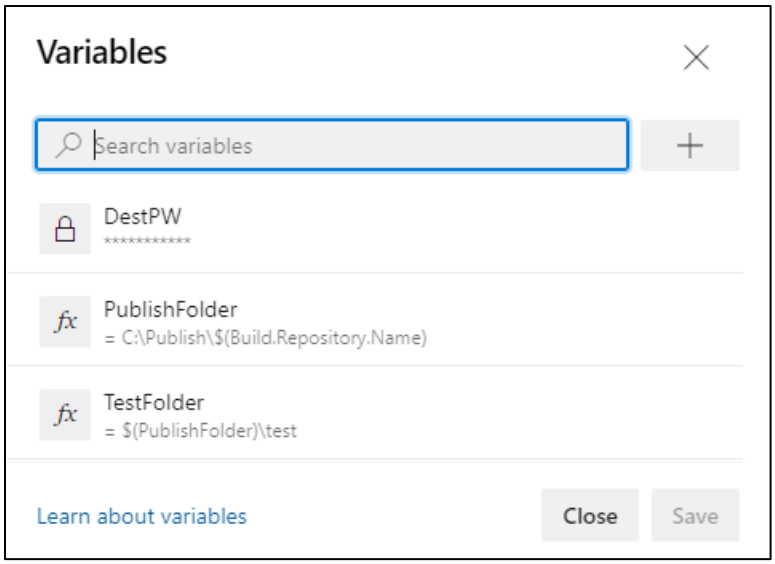

Figure 6: Configuring Pipeline Variables

The full YAML file for the CI Pipeline is shown below:

```
# Build Pipeline for SAS programs
# Uses SASUnit for executing unit tests
trigger:
- integration
pool:
  vmImage: windows-latest
steps:
# Azure file copy
# Copy files to SAS deployment
- task: AzureFileCopy@5
  inputs:
    SourcePath: '*'
    azureSubscription: 'Visual Studio Enterprise Subscription – MPN(d49d9…0564365)'
    Destination: 'AzureVMs'
    storage: 'phusecicdxfer'
    resourceGroup: 'PHUSE2022-PREP'
    vmsAdminUserName: 'svc_CICD_User'
    vmsAdminPassword: '$(DestPW)'
    TargetPath: '$(PublishFolder)'
    enableCopyPrerequisites: true
    skipCACheck: true
    BlobPrefix: 'phuse'
    CleanTargetBeforeCopy: true
# Run PowerShell script on SAS deployment
# Runs SASUnit test suite
- task: PowerShellOnTargetMachines@3
  inputs:
    Machines: '$(ServerWinRM)'
    UserName: 'svc_CICD_User'
    UserPassword: '$(DestPW)'
    InlineScript: '.\sasunit.ps1'
    ErrorActionPreference: 'silentlyContinue'
    WorkingDirectory: '$(TestFolder)'
    RunPowershellInParallel: false
```

CONTINUOUS DELIVERY PIPELINE

To implement Continuous Delivery; the next step in the DevOps lifecycle, another Git Pipeline is used. The syntax of this YAML file is very similar to the CI pipeline. In-fact it is shorter as we only need to copy files, rather than execute

tests, although depending on the type of project you may have integration or system tests to run. This example takes only the `/src` folder from our repository and deploys to a version-specific release folder.

CONDITIONAL TRIGGERS

As mentioned previously, the CD pipeline should only be executed when the CI pipeline runs successfully. This is not a condition we set in the YAML file. Instead, we configure this in Azure DevOps. Figure 7 shows the user interface in Azure DevOps for configuring a conditional trigger. The pre-requisite pipeline is selected with the target branch.

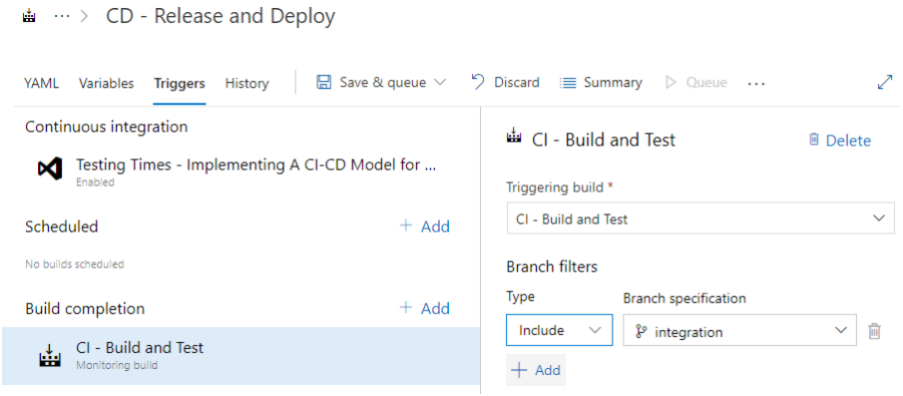

Figure 7: Setting A Conditional Trigger

CI/CD BEYOND SAS PROGRAMS

There are many areas of SAS where CI/CD and Git Pipelines can be used. SAS Viya has more comprehensive support for Git across its architecture and enables Git to be used for other areas of SAS, such as SAS Studio Tasks and Jobs.

Further, cloud technologies of Containers and Kubernetes enable the configuration of an entire SAS environment to be programmed in YAML. This is a separate topic and warrants its own paper. Nonetheless, with YAML, we can programmatically set parameters such as server sizing, storage capacity, installation media, etc. CI/CD Pipelines enable automated deployment and testing of our SAS environment on each change, e.g.: installing updates, configuration changes. This means even validated SAS environments can be updated with all validation tests and regression testing executed automatically. Further, this technique essentially versions the environment so rolling back to a previous “known-good” state can be achieved even years after change.

CONCLUSION

In conclusion, implementing the DevOps lifecycle for managing SAS programs creates the possibility of implementing CI/CD to automate unit tests and rollout of programs. Git source control offers the best baseline for implementing CI/CD. Many software solutions are available, for on-premise implementation or in the cloud via software-as-a-service. Sensitive information such as server credentials can be stored securely. Valid DevOps users can use a password legitimately but cannot reveal its value. CI/CD is expanding into statistical environments, a validated computing environment can be deployed and (re)validated entirely autonomously after a change which will drastically reduce timelines when applying updates.

REFERENCES

1. The DevOps lifecycle: A quick and easy walkthrough. https://nulab.com/learn/software-development/devops-lifecycle-quick-easy-walkthrough/, visited 02 Sep 2022.
2. Unit Testing and Code Coverage Assessment with SASUnit, https://www.lexjansen.com/phuse/2010/ts/TS05.pdf visited 02 Sep 2022.
3. SASUnit download, https://sourceforge.net/projects/sasunit/ visited 02 Sep 2022.
4. Unit testing fundamentals https://learn.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2022 visited 02 Sep 2022

RECOMMENDED READING

1. Azure DevOps documentation | Microsoft Learn, https://learn.microsoft.com/en-us/azure/devops/?view=azure-devops.
2. YAML schema reference | Microsoft Learn, https://learn.microsoft.com/en-us/azure/devops/pipelines/yaml-schema/?view=azure-pipelines.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Paul Shannon
Amadeus Software Ltd.
11 Wesley Walk
Witney
Oxfordshire, OX28 6ZJ
Work Phone: +44 (0) 1993 848010
Email: paul.shannon@amadeus.co.uk
Web: https://amadeus.co.uk