Lab 3 - Source Control Management Tool

Overview  
The objective of this lab is to describe the use of a continuous integration tool in conjuction with automated pipelines and supporting software. It will describe the steps involved including building, testing, packaging and/or deploying a project stored in a repository.

Aims/Description

* Research the role of CI/CD tools in a DevOps pipeline.
* Investigate various CI/CD tools.
* Based on the research: pick two suitable CI/CD tools.
* Generate an automated pipeline that can build, test, package and deploy a project.
* The automated pipeline should run when code changes are committed.
* Built output and test output should be retrievable after the pipeline runs.
* Explore where testing is applicable in a CI/CD tool and pipelines.
* Understand why a CI/CD tool is used as part of a DevOps pipeline.
* Contrast the two chosen CI/CD tools.

Method

* Researched various CI/CD tools and their benefits.
* Applied branch policies to the repository to prevent code being pushed into the main branch directly.
* Created a repository and commited the projects from the previous lab.
* Added a branch policy to the main branch on GitHub.
* Accessed the actions page on the github repository and created a new workflow for the project.
* Edited the workflow YAML file to contain triggers on pull request.
* Added a job with steps to install the .NET requirements, build the project, and test the project.
* Created a steps to always upload the test results as an artifact and
* Added a
* Noted the pipeline result, build output, and the result of the tests that ran.
* Made an additional job to demonstrate further possible steps in the pipeline such as deployment.
* Repeated the same workflow/pipeline steps using Azure DevOps for comparison.
* Added stakeholders as collaborators for the GitHub repository.

Results

* Research showed there are several different CI/CD tools to choose from. Some examples include: Jenkins, Circle CI, Azure DevOps, GitHub Actions (Katz, 2022).
* Learned in-depth about CI/CD tools and their usage as part of a DevOps pipeline. Each tool has it's advantages and disadvantages but most are extensible through plugins.
* A repository with two projects one unit tested and another as an example API to be deployed. These projects were from the previous lab.
* The repository on GitHub has a branch policy preventing direct pushes into main. Code changes can only be committed to main via a pull request.
* GitHub Actions is included with GitHub projects. It was available on my project repository through the actions tab.
* The project now has a GitHub Actions workflow to build, test, package and deploy the project.
* Upon merging my code into the main branch from a pull request, the workflow/pipeline was automatically kicked off by GitHub actions.
* Created a small python integration test project in the same repository. This was made to test the resulting executable from the build phase.
* The build pipeline in Azure DevOps was very similar to GitHub actions. The project now has an Azure DevOps pipeline that can be used to build, test, package and deploy the project.
* I now have two pipelines written in different CI/CD tools to contrast.

Conclusion  
The purpose of this lab was to describe the use of a continuous integration tool with automated pipelines and supporting software as part of a devops pipeline.

To begin this lab I needed to understand CI/CD in DevOps and agile development. CI or continuous integration is the practice of development teams making small and frequent code changes that are checked into a version control system, while CD or continuous delivery refers to the automation of code changes made during CI being pushed to different environments (Sacolick, 2022). From the previous lab I learned that continuous testing also plays a key role in CI/CD and the DevOps pipeline. Sacolick (2022) additionally mentions that another step called continuous deployment is possible for robust or mature pipelines. This is the process of automatically deploying builds that have passed automated testing into production environments. Next I needed to understand the role of CI/CD tools and how they automate this process. From my own experience working on legacy systems and bespoke internal build systems, these homebrewed systems tend to be inflexible and made with a specific set of products in mind. In my research on different CI/CD tools one of the commonalities with the popular tools is the extensibility of the tools through the use of plugins (Katz, 2022). For this lab I choose two CI/CD tools to contrast against: GitHub Actions and Azure DevOps (specifically Azure Pipelines).

To speed up my ability to test this CI/CD tool I used the previous labs C# projects. These projects contained unit tests and a set of JSON API requests that could be used in integration testing or system testing. In addition I made a change to the repository to protect the main branch using branch policies requiring a pull request. I did this to prevent accidental pushes into the main branch, so that the pull requests need to be done that can trigger builds upon merging into main, and it's a best practice for repositories where multiple teams will be working on the code as discovered in the first lab.

GitHub Actions is a CI/CD tool built into each GitHub repository. I was surprised to see how simple it was to use and immediately available to every repository without any setup. Doing a simple build and execution of unit tests can be done without writing any workflow yaml as GitHub supplies a set of basic workflows depending on the language used in your project. I wanted to get a better understanding of the yaml syntax required for building and testing so I followed Githubs official documentation for .NET projects (*Building and testing .NET,* no date). One advantage I noticed using yaml is that it's easily readable. With my pipeline building and testing the code, the next steps was for me to add deployment step (in my implementation this step was simply deploying to the development environment for dev testing). I didn't deploy to a real environment (there's a price and security cost to setting up environments), I simply called a script to emulate the using CLI arguments to deploy the application to the cloud. For the next step of running our test suite I had a similar test script that would call out integration or system tests against the deployed software depending on the environment we've deployed to. Finally to wrap it all up I added a trigger to kick off the work flow when code is pushed into main (in this case only pull requests due to branch policies). All in all my setup follows the principles of DevOps. Continuous integration as code was added through branches, pull requested into main, and finally a build was triggered. Continuous delivery (or deployment) when the code built and the unit tests had passed (or any additional tests required could be added). And continuous testing as the testing is done early (shift-left principle) and frequently as in the unit tests and system testing after deployment. I had never use GitHub actions before this lab so overall I found it to be a great tool to use. If you have a GitHub project it's readily available for use. Having base templates for getting a project up and running quickly are a nice touch.

Azure DevOps is an eco-system of features that can be used in a DevOps pipeline. For this lab I focused on one specific feature: Azure Pipelines. This tool works in a very similar to to GitHub actions. Pipelines (the equivalent to workflows in GitHub Actions) can be created in yaml format to build, test, package and deploy code projects. In comparison to GitHub Actions the repository is not immediately accessible, I had to authorize a connection to my GitHub repository. Once that was done I worked on the pipeline. Azure pipelines are also written in yaml and have strikingly similar syntax. Most of the keywords were synonyms of GitHub Actions keywords. The biggest difference I noticed was that Azure Pipelines natively supports .NET tasks compared to GitHub Actions requiring calls to the dotnet CLI. For the pipeline I kept the same flow as the one setup in GitHub Actions, this can be seen in Figures 1 and 2. From the line count Azure Pipelines is a little more verbose than GitHub Actions. With this comparison in mind I had the question of: *Why choose Azure DevOps over GitHub?*. Azure DevOps itself comes with many other features that makes up the full product. Some of those include: Azure Boards, Azure Repos (their version of GitHub), Azure Test Plans, Azure Artifacts and of course Azure pipelines (*What is Azure DevOps?, no date).* Azure DevOps as a whole appears to be a more complete platform for implementing a DevOps process/approach or culture into a project or organisation. In my opinion I prefer Azure DevOps over GitHub actions. I thought the UI was easier to read and see results. It wass easier to kick off manual builds (GitHub Actions required an extra trigger). On top of this Azure also has the Azure portal for spinning up cloud resources

Both of these tools have made it apparent to me how important a CI/CD tool is to the DevOps pipeline. The resulting workflow/pipeline I created shows that building, testing, release and deployment can all be automated by a CI/CD tool. With the ability to extend both platforms using plugins or just manually creating scripts to do the functionality needed, these tools can fit a large number of use-cases. Triggers on our workflows and pipelines allow automated reactions to events happing within our codebase or even accross the organisation. When picking a CI/CD tool to use Katz (2022) desribes a few things to look for: Ease of installation and configuration, integration with other issue tracking tools and version control tools, security and compliance, and price. The Jenkin CI/CD tool is a good example of not being entirely easy to setup. It requires you to have your own build machine to install it and possibly several extra machines to be treated as agents (*Hardware Recommendations - Jenkins,* no date). On integration, GitHub Actions is built into the repository in use while Azure requires some secure connection setup. But Azure has the added benefit of support different kinds of version control systems as shown in Figure 3. For Security and Compliance, GitHub and Azure has project level permissions available to prevent malicious actors from interacting with builds and other parts of the codebase (examples shown in Figure 4 and 5). Lastly is price, many of the platforms have a freemium model allowing up to a certain amount of compute time while others such as jenkins may have an initial hardware cost. Picking a CI/CD tool looks to be difficult, the requirements could be project or organisation dependent. Many of the open source projects I've seen on GitHub use GitHub Actions or the free tier of Azure DevOps.

References & Bibliography  
Katz, E. (2022) *Top 10 CI/CD Automation Tools.* Available at: <https://spectralops.io/blog/top-10-ci-cd-automation-tools/>  
(Accessed 17 November 2022).

*Building and testing .NET - GitHub* (n.d.) Available at: <https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-net>  
(Accessed 17 November 2022).

Sacolick, I. *What is CI/CD? Continuous integration and continuous delivery explained* (2022) Available at: <https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html>  
(Accessed 18 November 2022).

*What is Azure DevOps? - Microsoft Learn* (n.d.) Available at: <https://learn.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?view=azure-devops>  
(Accessed 18 November 2022).

*Hardware Recommendations - Jenkins* (n.d.) Available at: <https://www.jenkins.io/doc/book/scaling/hardware-recommendations/>  
(Accessed 18 November 2022).

Appendices  
  
Figure 1 - GitHub Actions workflow trigger after pull request.

Figure 2 - Azure DevOps pipelines triggered after pull request.