



Cover image by Pixabay on [Pexels](#) under CC0 License

GitHub Actions: An Introduction to CI

Soumitra Shewale

Licensed under the [CC BY-SA 4.0 License](#)

CI? What's that?

CI stands for Continuous Integration, and no, this is not Calculus 1, Continuous Integration refers to the integration or merging of the work done by a team of developers into the main source code or mainline or trunk.^[1] Basically, it is a tool that allows a team of developers or contributors in the case of Open Source projects to simultaneously work and collaborate on the same code base. Some CI services — such as GitHub — even provide tools for the maintainers of the project to accept these changes, make suggestions to improve the code, or even completely reject the changes. It essentially allows a large team to collaborate on the same project.


Most CI services need the user to make a local copy of the base or mainline of the code, work on it independently, and submit their changes as suggestions to the mainline of the code. These suggestions are then reviewed by peers or supervisors, who either accept, reject, or suggest changes to the changed code. As this process of reviewing changes goes on, some changes are accepted into the mainline code and the local copy of the developer slowly ceases to reflect the actual mainline, and that requires the developer to refresh the local copy. Not updating the local copy often results in conflicts, which the maintainers and the developer are expected to iron out.

In this blog post, we will examine some advantages and disadvantages of CI/CD from a GitHub and GitHub Actions perspective, and ways to use CI/CD in Julia development.

What's CD?

CD is CI's best friend; The two are often paired together and are used in conjunction to refer to an incremental style of software development.

Using the acronym CD actually introduces a little ambiguity; CD can stand for both Continuous Delivery and Continuous Deployment. They may sound the same, but there is a subtle difference between them. First, we will discuss what they have in common. Continuous Deployment and Continuous Delivery both produce software in small and short cycles and make incremental changes to the code base such as adding features, bug fixes, and user experience improvements rather than releasing a major reworked product, which is completely different from its predecessor every year or so.



The two refer to rapid development (not to be confused with Agile), with testing and deploy cycles that typically last a few months. Some well-known projects such as Windows 10^[2], Linux distros (e.g. Arch Linux, Gentoo Linux etc.^[3]), and most Open Source Packages use this form of development.

The main difference between Continuous Deployment and Continuous Delivery, however, is that Continuous Deployment uses automated systems to build, deploy, and test the software before releasing an update, while the same is done manually in Continuous Delivery.

In this blog post, we will discuss only Continuous Deployment in the context of GitHub Actions, and ways to implement this on an existing repo. Hence, hereafter the term CD will stand for Continuous Deployment.

Why CI/CD?

The advantage of CI is clear: an entire team of developers can seamlessly work on the same code simultaneously. In this blog post, we will limit our discussion to GitHub; while there may be better CI systems out there that can handle merge conflicts better We will discuss GitHub as a mainstream service and how GitHub Actions can benefit the workflow of development teams.

However, CD also has the following constraints^[4]:

- **Customer preference:** If a customer prefers not to push small incremental updates that may be critical at times (such as security patches), and does not update their systems, the systems could be compromised, and bugs can affect the functionality of the system itself. An example of this could be a case where mission-critical software is running or a case where the workflow at that organization is not CI/CD friendly. Then, some systems may not be able to change system files or restart after updates.

Due to this, companies have adopted their own counters to this disadvantage: Ubuntu releases LTS (Long Term Support) releases every two years, which are debugged extensively before release, and the software is expected to run and be supported for years to come. Windows 10 offers different update channels for different types of customers. Enterprise users receive updates after 6 months of the release, during which numerous security patches, bug fixes, user experience improvements, and optimizations are pushed to the enterprise release. Additionally,

enterprises are advised to test the new updates on selected machines before pushing the updates to all the PCs in that organization.

- **Tests needing human-intervention:** In the medical research field, years of testing and certification need to be performed, and all that red-tape slows the process down, and such a development cycle isn't really applicable in such fields.

Julia and CI/CD

Of course, just like every other programming language, you can set up a code repository on GitHub, and you can find out about that [here](#). We will not delve too deep into this since this blog post assumes that you have previous experience with setting up GitHub repos, cloning repos, creating Issues, Pull Requests, forking a repo, and working with multiple branches.

Once you have a repo containing Julia code, it is very easy to set up CD workflows with GitHub Actions. GitHub actions are essentially scripts that perform given steps every time an event occurs. The event could be the passage of time, creation of an issue or pull request or creation of a comment on the same, a commit or any other event specified [here](#). The user can define the steps as required. It could be running a script, testing a build, creating a new release, labeling issues and pull requests, checking the feasibility of changes in pull requests, and many more.

For Julia development, there are 2 GitHub Actions at the time of writing:

- Setting up a Julia environment to run Julia scripts: [Setup Julia environment](#)
- Adding tags for releases of Julia packages: [Julia TagBot](#)

Unless you're maintaining and writing a Julia package, there is very little chance you need to know about the second one. Let us now look into how we can get started with working with GitHub Actions on Julia repos. Later, we will look into the process of setting up a CI workflow for packages.

Setting up a CI script for running Julia scripts on GitHub

After setting up a repo that contains a .jl file you wish to run, perform the following steps to automate the running of this script:

1. Create a ``.github`` folder in the root of your repo, and create a `workflows`` folder inside the ``.github`` folder you just created.
2. Create a `name.yml`` file in this workflows folder where the `name`` is the name you want to set for the script that automates the running.

- Go back to the root of the repo, and create a folder called `src`. Move all your Julia code into this folder. Now, your folders should have the following hierarchy :

```
Repo_name
|
|----> src
|      |----> mycode.jl
|      |----> morecode.jl
|----> .github
|      |----> workflows
|      |
|      |----> ...
|      |----> name.yml
```

Note: The ... stands for all the other files (README, LICENSE, and all other unimportant files)

- Now that our skeleton structure for our code is ready, let us actually start writing the name.yml script that will automate our script. Head back to name.yml, and we'll add some code there to automate it.

Head over to <https://github.com/marketplace/actions/setup-julia-environment>, and click on "Use latest version". Copy the code that appears in the dialog box. If you are copying any other text while writing your automation script, paste this in a notepad. Now, let us start writing our script. We will first need to define the name for our automation script. For this, you type the following at the top of the script:

```
name: 'Name of your script'
description: 'Description of what your script does'
author: 'Your name'
```

- Below this, we will set up the schedule for running our task.

```
on:
  schedule:
    - cron: '*/* * * * *
```

The above script runs every 5 minutes. To change this schedule or to set the script to run on a different event, see [this](#) article.

Now, let us actually make the script run our Julia code. To do this, we will create a `job`, and add `steps` to that `job`. Since running our code is essentially a build activity, we will enclose our steps in a `build`.

The `runs-on` is just a specification of which operating system the script runs on. This could be any OS mentioned [here](#). I have used `ubuntu-latest`. You will also need to set up Julia using the code we copied from the dialog box earlier.

We will build the package (our Julia code) using the Build step, and the Run step will execute our Julia script. Paste the code below the `on` block we pasted earlier.

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Setup Julia environment
        uses: julia-actions/setup-julia@v1.0.2
      - name: Build
        uses: julia-actions/julia-buildpkg@master
      - name: Run
        run: julia --project src/mycode.jl
```

Note that you will have to change the name of the script in the last line to make this work.

- Now, we need to set up some dependencies for our project (Julia code). Since the GitHub Actions Virtual Machine will not have the required Julia packages installed, we will have to install them. You can do this by simply changing

```
using Example, Example1
```

To

```
import Pkg
Pkg.add("Example"); using Example
Pkg.add("Example1"); using Example1
```

This will install the latest version of the Packages required every time you run your code.

With this, we have successfully automated our script!

Julia Packages and CI/CD

Julia packages are maintained as a registry of packages in one place that has entries in it with information about packages. Every time the registry is updated using the JuliaRegistrator app, new releases need to be generated on the actual repo of the Julia Package. TagBot automates this updating of the releases on the Package repo.

To set up TagBot, perform the following steps:

1. Create a `.github` folder in the root of your repo.
2. Create a `workflows` folder inside the `.github` folder
3. Create a file called `TagBot.yml` inside the `workflows` folder.
4. Paste the following code into the `TagBot.yml` file:

```
name: TagBot

on:
  schedule:
    - cron: 0 * * * *

jobs:
  TagBot:
    runs-on: ubuntu-latest
    steps:
      - uses: JuliaRegistries/TagBot@v1
        with:
          token: ${ secrets.GITHUB_TOKEN }
```

5. Now, every time you create a release for your package on JuliaRegistry, it will automatically update the repo to reflect the newest release.

Testing code before merging Pull Request

Sometimes, we need to run tests on a Pull Request before merging it to see if the build fails. This can be done using matrix testing so that cross-platform compatibility is maintained. To do so, we must write a testing script that tests all the code on the repo, and checks if it runs as expected. This process of writing a `tests.jl` or `runtests.jl` script is different for each repo, so you'll have to figure that out yourself.

Assuming the testing script has been written and can test the entire repo for errors, we can now set up an automated workflow for the repo. To set the automated cross-platform testing^[5] up, follow these steps:

1. Place the testing script (`tests.jl` or `runtests.jl`) in the root of the repository.
2. Create a `.github` folder in the root of your repo, and create a `workflows` folder inside this `.github` folder.
3. Create a `testing.yml` file in this `workflows` folder.

4. Now, let us write the code for `testing.yml`

First off, we need to define the name of the action at the top of the script like this:

```
name: Run tests
```

Now, we need to schedule this testing. In most testing scenarios, you'll want to run this on every push to master, and every Pull Request:

```
on:  
  push:  
    branches: master  
  pull_request:
```

Let us finally set up the `job` for our script. Since this is a testing activity, we will enclose all or steps in a `test` block. To enable cross-platform testing, we use something called Matrix Testing. This runs the code on different operating systems and Julia versions to ensure not only cross-platform compatibility, but also backward-compatibility.

```
jobs:  
  test:  
    runs-on: ${{ matrix.os }}  
    strategy:  
      fail-fast: false  
      matrix:  
        julia-version: ['1.0', '1.1', '1.2', '1.3', nightly]  
        os: [ubuntu-latest, windows-latest, macOS-latest]  
    steps:  
      - uses: actions/checkout@v1.0.0  
  
      - name: "Set up Julia ${{ matrix.julia-version }}"  
        uses: julia-actions/setup-julia@v1  
        with:  
          version: ${{ matrix.julia-version }}  
  
      - name: Test exercises  
        run: julia --color=yes --check-bounds=yes --project -e using Pkg; Pkg.test(coverage=true)
```


Notice how we set the `runs-on` to different operating systems using a `matrix`. Let us now look at the steps: First, we checkout our repo so our workflow can access it, Then, we set up a Julia environment with a matrix specified version, and then, we finally run our testing script that test our package.

This completes the workflow for cross-platform testing.

For security reasons, it is advised to use commit hashes instead of versions while checking out our repo in our GitHub Action. So, our checkout step will be

```
steps:  
  
  - uses: actions/checkout@*commit hash*  
  
  - name: "Set up Julia ${ matrix.julia-version }"
```

The action will now use the code the way it was after the commit with that commit hash. For more information on how using the version could be a security risk, you can refer to [this blogpost](#).

References

Cover page image by Pixabay on [Pexels](#)

- [1] - Definition of [Continuous integration from Wikipedia](#)
- [2] - [Windows as a service, as an example of CI/CD](#)
- [3] - [Linux as an example of CI/CD](#)
- [4] - [Obstacles of CD](#)
- [5] - [The standard testing script for Packages](#)