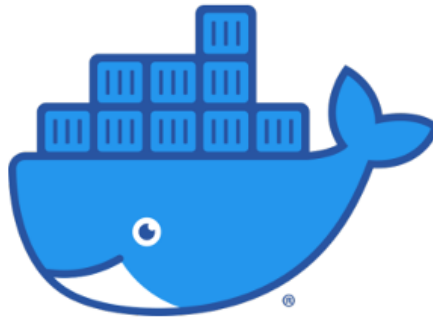


Robotic Sea Bass

Learn assorted topics in robotics, AI, programming, and more.

Continuous Integration with GitHub, Docker, and Jenkins



In our previous post on [Testing Python Code](#), we created unit tests for a simple Python project using [PyTest](#). Here we build on this work by discussing how we can take those tests from running locally on our own machines to something more scalable.

NOTE: While the previous post was specific to Python, you will find that the material in this post has nothing to do with programming language. Continuous integration is a general software engineering practice.

The code used in this post is available at <https://github.com/sea-bass/python-testing-ci>.

Introduction to CI

If you're developing in a "bubble" — that is, doing everything on your machine — you likely kept tweaking your development environment until your tests passed and that was good enough. Chances are if you handed the code to someone else, whether it's another developer on your project or an end user, there will be something *a little* different in their environment that may cause things to fail for them. At this point you have two options.

- **Option A:** Say “*it works on my machine*” and let others figure it out. This is how you get your software engineer card revoked.
- **Option B:** Make sure your (and all contributors’) changes are frequently built and tested on a clean environment to reduce the chances of this happening.

Continuous Integration (CI) is a software engineering practice to bring together the contributions of multiple developers on a project and automatically perform necessary tasks such as building and testing. The idea is for these tasks to run right as these contributions are made — hence continuous — with the goal of detecting issues as early as possible.

Source control tools such as [Git](#) are already a partial solution to continuous integration. Hosting your code on a server like [GitHub](#) or [GitLab](#) is what enables multiple developers to contribute to the same code repository. However, a Git server by itself doesn’t actually build and test the code — it just hosts it.

The following video explains the Continuous Integration / Continuous Delivery (CI/CD) workflow very nicely. If you don’t want to read the rest of this section, just watch this.

Professional Guides: Continuous Integration Continuous Delivery



To recap the video, the basic idea of CI is as follows:

1. Developer pushes changes to a repository hosted on some server

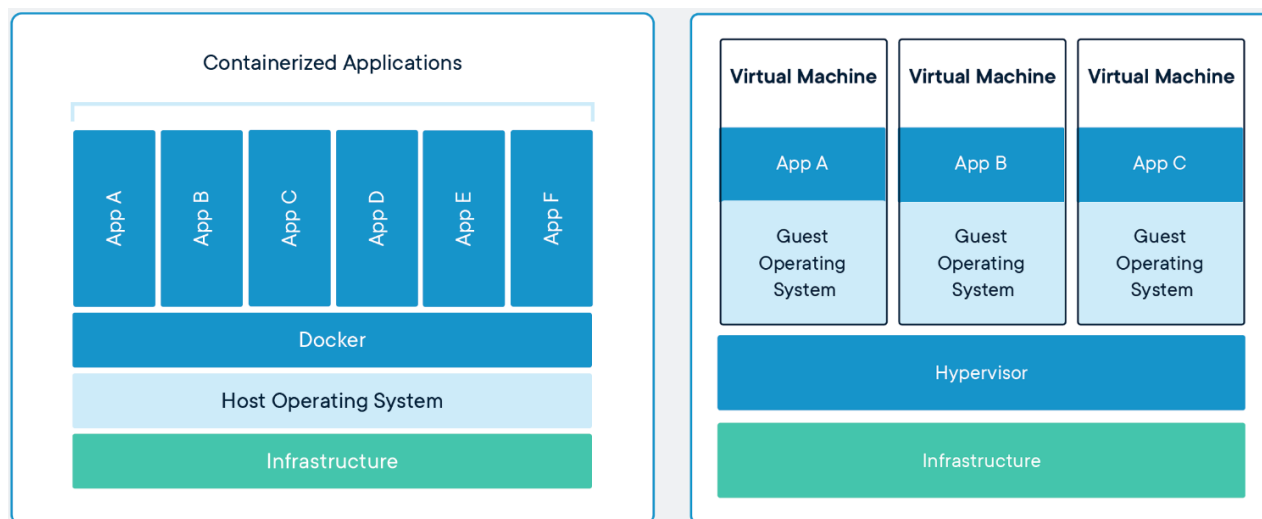
2. Server registers this change and starts a CI job. This typically involves **building** and **testing** the project.
3. Developer gets results back from the server. If there is a failure, it needs to be addressed.

The fact that now rely on a server to build the code in a sterile, isolated environment means that we can't get lucky with "works on my machine". Your project *has* to successfully build on the server before the tests are run... and then those tests have to pass. Only then can your code be considered ready for the world to see.

Let's take another try at this reproducibility question. What are some easy ways to provide an entire development environment as needs to be configured for your project? Or what if your CI server is tasked with testing different projects on different operating systems, or anything else that could lead to conflicts across environments?

The creation, maintenance, and deployment of such isolated environments motivates the use of virtualization tools like *containers* or *virtual machines* (VMs). To give an extremely high-level summary of these two approaches:

- **Virtual Machines** perform virtualization at the physical hardware level, which lets you run completely different operating systems than the host machine.
- **Containers** perform virtualization at the application level, which uses the host machine's operating system kernel under the hood.



Containers vs. Virtual Machines

[Source: <https://www.docker.com/resources/what-container/>]

The big takeaway is, unlike VMs, with containers you cannot virtualize any and every operating system configuration from your host machine. However, if you can manage with

just containers, they are much faster and memory-efficient than VMs.

For more details, you can refer to [the Docker page](#) where the image above came from, or also [this page from IBM](#).

Running Unit Tests in a Container

For our example, we'll be using a container since it's less overhead than a VM and containers are sufficient since both my development environment and CI "server" will be using Ubuntu 18.04.

Specifically for this example, and also because it's by far the most popular containerization tool, we will be using [Docker](#).

Typically, all the steps needed to assemble a Docker image are written in a Dockerfile. This is literally a text file with the name "Dockerfile". The Dockerfile looks as follows.

```
1 # Define the base image
2 FROM ubuntu:18.04
3
4 # Install required packages
5 RUN apt-get update \
6     && apt-get upgrade\
7     && apt-get install -y --no-install-recommends \
8         python3 \
9         python3-pip \
10        python3-setuptools
11
12 # Copy this repo to a folder in the Docker container
13 COPY . /app
14
15 # Set the work directory
16 WORKDIR /app
17
18 # Install all the required packages
19 RUN pip3 install -r python_requirements.txt \
20     && pip3 install .
```

In plain English, the sections in the Dockerfile would roughly translate to:

1. Start with a pre-made standard image for Ubuntu 18.04
2. Install additional system requirements (in this example, Python 3)
3. Copy the files from the GitHub repository
4. Set the working folder at startup to the location where we copied our files
5. Install the Python packages needed to run the code, as specified by a requirements file (we discuss this in a [previous post](#))

Assuming you've installed Docker, you can now build this Docker image locally on your machine to do some preliminary testing. The following command will build the image in the current folder (assuming that's where your Dockerfile is) and give it an output name of **testing-ci**.

```
docker build -t testing-ci .
```

We can check that our image was created by typing

```
docker images
```

```
sebastian@type-v2:~/robotic_sea_bass/testing_ci/python-testing-ci$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
testing-ci	latest	e182856b45d4	5 minutes ago	350MB
inst_methodc	latest	99a5ad075244	10 days ago	18.6GB
ubuntu	18.04	c3c304cb4f22	3 weeks ago	64.2MB
rabbitmq	3	cf7986a5b16f	5 weeks ago	154MB
mongo	4.0	6182b91a1dbe	7 weeks ago	418MB
jsbroks/coco-annotator	workers-stable	407961237e57	2 months ago	2.39GB
jsbroks/coco-annotator	webserver-stable	d8d30601bcce	2 months ago	2.4GB
nvidia/cuda	10.1-devel-ubuntu18.04	9e47e9dfcb9a	5 months ago	2.83GB
nvidia/cuda	10.2-devel-ubuntu18.04	af2eaa345ab7	5 months ago	2.91GB
python	3.6.9	5bf410ee7bb2	5 months ago	913MB

Ignore the other images on there... but really, having multiple images for different projects is partially why Docker is useful!

Then, we can start a container based on this image. Read that again. A *container* is an instance of an *image*. To run the tests using the Docker image as the execution environment, you can do this in one shot as follows.

```
docker run testing-ci pytest
```

Or, you can use the interactive (**-it**) flag to get access to a terminal where you can run the tests.

```
docker run -it testing-ci
root@CONTAINER_ID:/app# pytest
```

```
sebastian@type-v2:~/robotic_sea_bass/testing_ci/python-testing-ci$ docker run -it testing-ci
root@6ef9d910a04d:/app# pytest

=====
test session starts
platform linux -- Python 3.6.9, pytest-5.4.1, py-1.8.1, pluggy-0.13.1 -- /usr/bin/python3
cachedir: .pytest_cache
metadata: {'Python': '3.6.9', 'Platform': 'Linux-4.15.0-99-generic-x86_64-with-Ubuntu-18.04-bionic', 'Packages': {'pytest': '5.4.1', 'py': '1.8.1', 'pluggy': '0.13.1'}, 'Plugins': {'html': '2.1.1', 'cov': '2.8.1', 'metadata': '1.8.0'}}
rootdir: /app, inifile: pytest.ini
plugins: html-2.1.1, cov-2.8.1, metadata-1.8.0
collected 13 items

tests/test_matrix_operations_basic.py::test_numpy_version <- ../home/sebastian/robotic_sea_bass/testing_ci/python-testing-ci/tests/test_matrix_operations_basic.py PASSED [ 7%]
tests/test_matrix_operations_basic.py::test_addition <- ../home/sebastian/robotic_sea_bass/testing_ci/python-testing-ci/tests/test_matrix_operations_basic.py PASSED [ 15%]
tests/test_matrix_operations_param.py::test_numpy_version PASSED [ 23%]
tests/test_matrix_operations_param.py::test_addition_exact[a0-b0-expected0] SKIPPED [ 30%]
tests/test_matrix_operations_param.py::test_addition_exact[a1-b1-expected1] SKIPPED [ 38%]
tests/test_matrix_operations_param.py::test_addition_close[a0-b0-expected0] PASSED [ 46%]
tests/test_matrix_operations_param.py::test_addition_close[a1-b1-expected1] PASSED [ 53%]
tests/test_matrix_tools.py::TestMatrixTools::test_numpy_version PASSED [ 61%]
tests/test_matrix_tools.py::TestMatrixTools::test_addition_exact[a0-b0-expected0] SKIPPED [ 69%]
tests/test_matrix_tools.py::TestMatrixTools::test_addition_exact[a1-b1-expected1] SKIPPED [ 76%]
tests/test_matrix_tools.py::TestMatrixTools::test_addition_close[a0-b0-expected0] PASSED [ 84%]
tests/test_matrix_tools.py::TestMatrixTools::test_addition_close[a1-b1-expected1] PASSED [ 92%]
tests/test_matrix_tools.py::TestMatrixTools::test_shape_mismatch PASSED [100%]

----- generated xml file: /app/latest_test_results.xml -----
----- generated html file: file:///app/latest_test_results.html -----

----- coverage: platform linux, python 3.6.9-final-0 -----
Coverage HTML written to dir coverage_results
```

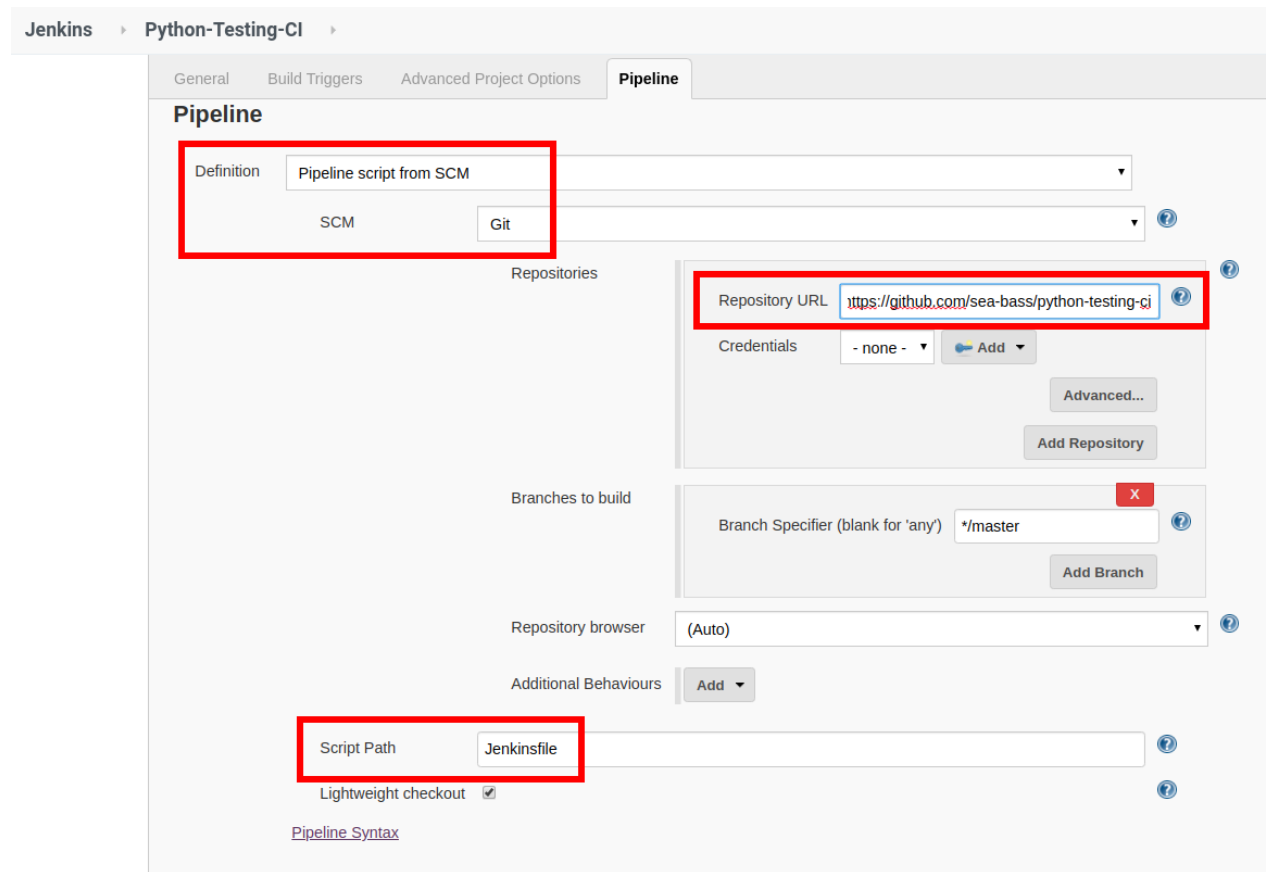
... our tests still pass inside a Docker container!

This is one step towards setting up continuous integration, since the beauty of containerization is that we can provide this same Dockerfile to the CI server and have it automatically build the image, start a container, and run our tests every time we push to the repository.

Building and Testing on a CI Server

Next we want to set up a CI server that will do the automated building and testing for us. There are many CI tools available — some free and open-source, some not. The most popular ones I've personally seen used in the robotics community are [Jenkins](#), [TravisCI](#), and [CircleCI](#). We will be using [Jenkins](#).

I don't intend for this to be a full Jenkins tutorial, but below is a screenshot of a Jenkins pipeline I've set up to tie into my GitHub repository.



The main thing that brings this all together is the creation of a Jenkinsfile that describes the steps to be taken when we run a continuous integration job.

Our Jenkinsfile will use the Dockerfile (yes, really) we created in the previous section. The Jenkinsfile for this example contains 3 major pieces:

1. Telling Jenkins to build a Docker image from the Dockerfile provided in the GitHub repository.
2. Running the unit tests using PyTest (remember, the Docker image was already set up to start in the correct working directory).
3. Recording the JUnit-style XML file generated from PyTest so the test results show up in Jenkins. For more information, see [this link](#).

```

1 pipeline {
2     agent { dockerfile true }
3     stages {
4         stage('Tests') {
5             steps {
6                 sh '/bin/bash -c "pytest"'
7             }
8         }
9     }
10    post {
11        always {
12            junit 'latest_test_results.xml'
13        }
14    }
15 }

```

Now, I don't have a dedicated server, so I used [ngrok](#) to establish a tunnel from a specific port on my localhost (where Jenkins is being served) so that GitHub can send a request to Jenkins when it registers. I won't go through the details here, but [this blog from CloudBees](#) has all the information if you want to try this yourself.

NOTE: This is not a very secure approach at all, so feel free to try things out with it and then promptly shut down ngrok!

```

ngrok by @inconsheveable (Ctrl+C to quit)
Session Status      online
Account             Sebastian A Castro (Plan: Free)
Version             2.3.35
Region              United States (us)
Web Interface        http://127.0.0.1:4040
Forwarding           http://abcdefg.ngrok.io -> http://localhost:8080
Forwarding           https://abcdefg.ngrok.io -> http://localhost:8080

Connections         ttl    opn    rt1    rt5    p50    p90
                    5      0      0.00   0.00   5.01   5.19

HTTP Requests
-----
POST /github-webhook/ 200 OK
POST /github-webhook/ 200 OK
POST /github-webhook/ 200 OK
POST /github-webhook/ 200 OK
POST /github-webhook/ 200 OK

```

ngrok was set up to create a tunnel from localhost:8080 (where Jenkins is running), where "abcdefg" would on your end be whatever string ngrok generates at the time.

The screenshot shows the GitHub repository settings for 'python-testing-ci'. The left sidebar contains navigation links: Options, Manage access, Branches, Webhooks (highlighted), Notifications, Integrations, Deploy keys, Secrets, Actions, Moderation, and Interaction limits. The main content area is titled 'Webhooks / Manage webhook'. It includes a description: 'We'll send a POST request to the URL below with details of any subscribe format you'd like to receive (JSON, x-www-form-urlencoded, etc). More documentation.' Below this is a 'Payload URL *' field containing 'http://abcdefg.ngrok.io/github-webhook/'. A 'Content-Type' dropdown is set to 'application/x-www-form-urlencoded'. There is an empty 'Secret' field. Under 'Which events would you like to trigger this webhook?', the 'Just the push event.' option is selected. The 'Active' checkbox is checked, with the note 'We will deliver event details when this hook is triggered.' At the bottom are 'Update webhook' and 'Delete webhook' buttons.

Once you have an Internet visible URL for your Jenkins server, you can create a webhook in GitHub that is triggered when pushing to the repository.


Now that we've set up the integration between GitHub and Jenkins, we expect that every time we push to the GitHub repo, a request will be sent to Jenkins to run a CI job.

On this first push, the build failed because I had an error in my Dockerfile... so the Docker image could not be created correctly.

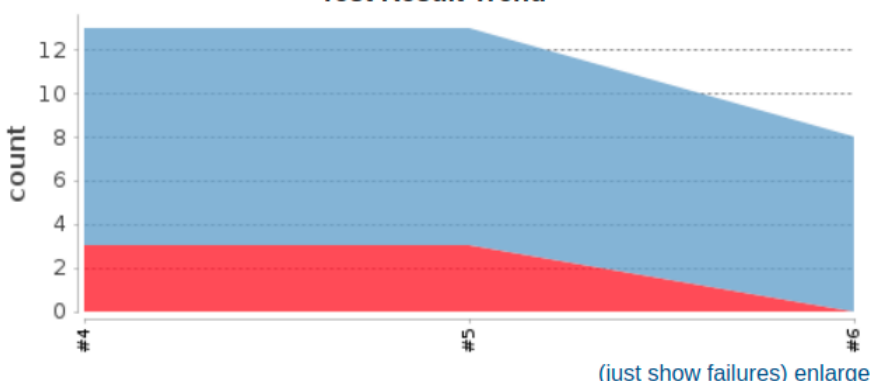
Jenkins Python-Testing-CI enable auto refresh

Pipeline Python-Testing-CI

[add description](#)
[Disable Project](#)

 [Recent Changes](#)

Test Result Trend



Run	Blue (Passing)	Red (Failing)
#4	12	3
#5	12	3
#6	8	0

[\(just show failures\)](#) [enlarge](#)

Stage View

Average stage times:

Declarative: Checkout SCM	Declarative: Agent Setup	Declarative: Post Actions
958ms	22s	394ms
958ms	22s failed	394ms

#7
May 12 12 19:28 12 commits

The failure was in the “Agent Setup” stage, meaning the build failed.

After fixing the build, I ran the tests but one of them failed, which marks the entire testing stage as a failure (as it should). You’ll see this below as Run #10.

Finally, I “fixed” this by marking the problematic test as “skipped”, and everything passes in Jenkins. You’ll see this below as Run #11. Notice in the trends graph that the “red” (failure) bit was converted to “yellow” (skipped), while all the other passing tests are denoted by “blue”.

Jenkins Python-Testing-CI enable auto refresh

Pipeline Python-Testing-CI

[add description](#) **Disable Project**

[Recent Changes](#)

Test Result Trend

(just show failures) [enlarge](#)

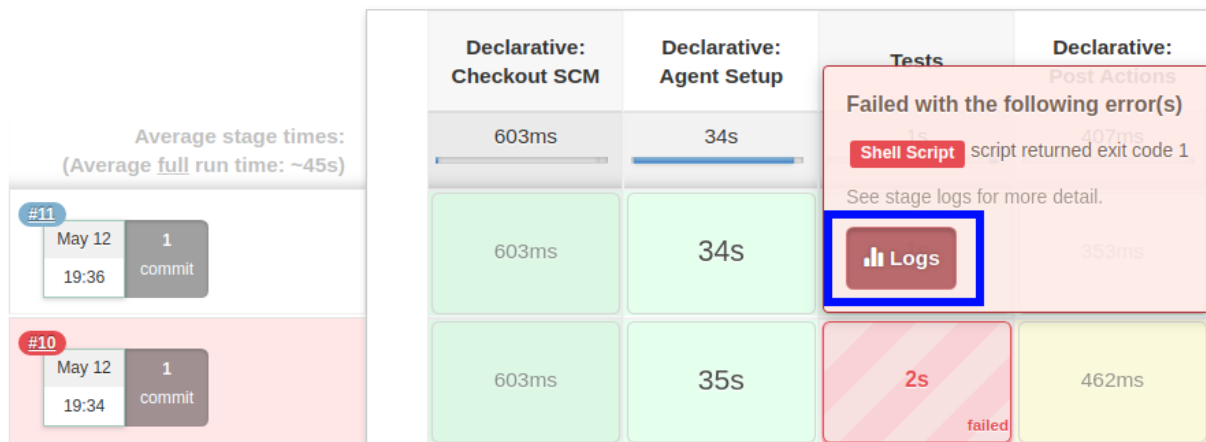
Stage View

Average stage times: (Average full run time: ~45s)

	Declarative: Checkout SCM	Declarative: Agent Setup	Tests	Declarative: Post Actions
Average	603ms	34s	1s	407ms
#11	603ms	34s	1s	353ms
#10	603ms	35s	2s failed	462ms

Notice now there is a new column showing the tests. This means the build passed and we actually got to the testing stage.

One last comment: After you run a job, Jenkins gives you access to log data. You will find this extremely important to figure out why things failed and how you can fix things for future runs.



Some light reading abounds!

Summary

So that’s a high-level overview of continuous integration. Obviously as you move from something like this simple example to a more realistic project involving many people, a release cycle, and actual end users who don’t want their tools broken, CI becomes much more useful.

I cannot stress enough how important it is to have a dedicated server if you’re serious about deploying CI/CD for your work. If you need more motivation, having a server constantly online will let you embed CI build status badges in your repository READMEs!



You’ve probably seen these around. Now you know what they are! [This video](#) shows how to do this for the Jenkins/GitHub combo.

Again, all the code is available at <https://github.com/sea-bass/python-testing-ci>. Note that to recreate everything you will need to do a lot of the Jenkins and GitHub setup on your end. Please feel free to reach out if you are trying this, or something similar, and run into issues. It was a lot of trial-and-error for me to get all the pieces together as well!

Sebastian Castro **May 12, 2020** **Software Development**
 continuous integration, Docker, GitHub, Jenkins, testing

Robotic Sea Bass, proudly powered by WordPress.