Best Practices for CI in Python

Moshe Zadka – https://cobordism.com

Acknowledgement of Country

Belmont (in San Francisco Bay Area Peninsula)

Ancestral homeland of the Ramaytush Ohlone people

0.1 What is CI?

Before diving into the best practices of Continuous integration, it's a good idea to understand what it is. Specifically, breaking down CI into its component parts.

What is Continuous Integration made of?

The gears in the machine

Before having CI is even expected, we usually expect that it is possible to run a local build. This is not strictly a requirement, but it a highly desirable property. If a build fails, not being able to simulate locally is unpleasant.

As a working assumption, there is already a way to run, locally:

- Lint
- Test
- Package

(Usual) Requirements for Continuous Integration

Local "build":

Lint

Test

Package

Assuming these steps can be at each point of the source code, when does the server run them, and on which source code versions?

In the old days, it was the "nightly build". This might seem ridiculous to modern ears, but at some point, having a nightly build at all was considered advanced.

The next step up is automatically building on a merge to the main development branch. This means that a broken main branch is immediately detected, and, in general, can be attributed to one change.

The modern way is to run the build on a *suggested* patch. Some systems call it a "try" run, some call it running the CI on a pull request or a merge request.

In all of those cases, the advantage is that there is feedback on a proposed change *before* it goes it. Often, patches with a failing CI cannot be applied (or "merged"), or sometimes need special approval to merge.

There is other exotica: running builds on "merge trains" and then only running the build on intermediate steps if the build fails, running the build against a fast-forwarded version of the patch or similar.

Continuous Integration: Run Build on Server

When?

Classic: Nightly

Modern: Merge to main

Advanced: Suggested patch (Pull Request, Merge Request)

Exotica (Merge trains and more)

The focus of the talk will be on “patch builds” (or “PR builds” or “MR builds” or “RR builds”). In those cases, the main use of the CI is as an automated gate: “is the patch good?”

Often, the CI being “green” will be a prerequisite for a human reviewing it. As above, in some cases, a reviewer can override the CI and approve a “red” patch.

This should be a rare occasion. If red approval happens too often, the CI is not delivering enough value.

Continuous Integration: Patch Builds

Focus of talk

Automated gate: ”is patch good?”

The continuous integration is a form of computation. The input is a source code snapshot, and the output is the CI results.

The thing that runs the computation is called a “runner”. Another popular term is a “worker”.

Usually, a centeralized or semi-centralized “coordinator” will send jobs to the workers and runners. It will collect, and backup, the results.

Continuous Integration: Runners

Architecture:

- CI coordinator
- CI runners

Most modern frameworks collect live logs, to allow for real-time feedback. Post run, they retain the log “forever”, possibly with a configurable retention policy.

Continuous Integration: Build logs

Modern frameworks:

Collect live from runners

Retain ”forever”

0.2 What is Good CI?

Given that this is what a CI system *is*, what is a good CI? Especially in the context of Python.

This does not help achieve it, but it does paint the goal to achieve.

What makes CI good?

Paint the target

Ideally, a CI is *accurate*. When it says a patch is good, it is good. When it says a patch is bad, it is bad.

CI criteria: accuracy

Is the answer correct?

When a patch fails, good feedback is *actionable*. What was the problem? How can it be fixed? How can it be reproduced locally?

CI criteria: actionability

If patch is not good, how clear is it how to fix?

How to reproduce locally?

Whether good or bad, fast feedback is good feedback. The situation is not as symmetric as that, though.

Red feedback promptness is not important. Green feedback can be semi-automated: the next step is to ask for a review, or, if pre-reviewed, can be automerged on success.

Red feedback, however, needs to be fixed. This is part of the developer's loop, and should be optimized with care.

CI criteria: promptness

How long does it take to answer?

Finally, cost. Cost is important to consider because many of the other criteria can be improved using costs.

Actionable feedback takes more resources. Running tests more than once makes them more accurate. Run tests in parallel, and on stronger machines, makes them faster.

CI criteria: cost

Mostly the runner compute cost

0.3 Improving accuracy

How do you improve the accuracy of the CI pipeline? One of the main sources in inaccuracies, both false alarms and missing alarms, are different environments.

When at all possible, run tests inside containers. Pinned container images that are custom-built. They do not need to be *built from scratch*, but they need to be custom-built and pinned.

Include the version of Python you want, and any non-Python-library dependencies, in the container.

CI accuracy: use containers

Container images with:

Version of Python

Other non-Python dependencies

Pin the image tag!

Another source of inaccuracy is different versions of PyPI libraries. Many programs have hundreds or even thousands of recursive dependencies.

All CI runs, including tests, linting, docs, type checking, and more, should use pinned dependencies (requirements.txt, or alternatives when using other virtual environment managers).

You should upgrade the pins – in a dedicated patch, that does nothing else. There are services to do that, but doing it manually is better than YOLOing it.

CI accuracy: pin versions

Test against pinned dependencies

Upgrade pins in a dedicated patch

(There are services)

Finally, monitor and improve your test quality. Tests should be written to detect real issues, and avoid detecting spurious problems.

Improving unit tests is its own, ten hour, talks. From the perspective of a CI system, the most important thing is *monitoring*.

Can you tell which tests are flakey? Can you tell how many bugs pass the test suites? How are you tracking that?

CI accuracy: test quality

Monitor and improve test quality

(This is a whole 'nother talk)

0.4 Improving actionability

Many test/lint runners keep verbosity, by default, to a happy medium. In CI, the trade-offs are different.

Set verbosity to the maximum it will allow. If this makes logs completely unreadable, use tricks like tee | filter and allow downloading the raw logs. Failures in CI, especially those that do not reproduce locally, are horrendous to debug.

Anything that makes them easier is worthwhile.

CI actionability: set verbosity to 11

Use verbosity options in test runners/linters/etc.

Logs can be filtered more easily than unfiltered

When tests fail, they should fail with relevant details. Sure, False is not True, but you already knew that. What parameters made the False? Which one of them was constant?

Some frameworks, like pytest, will help: but they are not magic. When writing tests, force them to fail, and check that they are verbose.

Exceptions you raise yourself get no special treatment. Make sure that they contain enough information so that when they appear in the logs, they have information to help diagnose the problem.

You are also probably better off setting logging levels to high verbosity, and directing them to a file that can be downloaded. This can be invaluable in tracking down issues.

CI actionability: test failure verbosity

Test assertion failure in test for verbosity

When raising exception, add details!

Before starting the test, help understand the context by spewing your guts to the log. Talk about the enviornment variables, the platform, and anything else that might be useful.

As a useful corrollary, avoid putting secrets in environment variables, especially persistent secrets.

CI actionability: environmental details

Spew details on environment to logs

Environment variables

Platform

Etc.

(Don't put secrets in environment)

0.5 Improving promptness

There is nothing more frustrating than waiting for a computer. Time waste is death my a thousand cuts.

Any primitives the CI system itself has to cache downloads locally should be used (or at least tested). When building containers, use container layer caching. This can be non-trivial to set up, in order to cache against a registry. It is worth it.

Cache downloads from PyPI locally. "Locally" is, as always, a matter of degree. If it helps, it helps. No need to litigate what "here" means like you're on an episode of Sesame street.

CI promptness: caching

Use CI primitives to cache downloads

Container layer caching(complicated but worth it)

Local PyPI and Container caching proxies

Use pooling primitives to reuse connections and downloads for anything that needs to connect remotely. While in general unit tests should avoid connectivity, sometimes this is unavoidable. In those cases, inter-test pooling can help quite a bit.

CI promptness: pooling

Reuse connections, downloads, etc.

Think carefully how to break up tests

Many CI systems allow "automatically merge when pipelines complete successfully". Because of this, a *failed* pipeline's time is more "expensive" than a "successful" one.

There are systems that will order tests based on the likelihood that they will fail. At the very least, run tests that use the code that has changed first.

The same is true for linting: run linting systems that are fast and check trivial things first. This improves the chances that a failure will be reported faster.

CI promptness: fail fast

Order tests based on likelihood to fail

Open source solutions exist

CI primitives can be used to parallelize runs. Distinct tox (or equivalent) targets can be run in parallel.

Correctly building smaller tox targets can improve things further. For example, splitting the targets into tests per subsystems will allow those tests to parallelize.

CI promptness: parallelize runs

Use CI primitives to parallelize independent runs

Running tests on systems with extra CPUs and using internal tester parallelization to parallelize runs can also achieve speed-ups.

CI promptness: parallelize tests

Use tester primitives to parallelize independent runs

Mock out slow parts. This includes using tools like eatmydata or tmpfs to mock out hard-drive slowness. Run local servers in irresponsible configurations. Mock out network connectivity completely.

CI promptness: mock slowness

Mock out slow things

0.6 Improving cost

This is the part that will make the business happy: you're asking for less money! The easiest way to make computers do things faster is to make them not do things.

Configure the CI system to kill useless runs. If someone has pushed to an existing patch, any existing runs should be cancelled or killed. Those results will not matter to anyone.

If directly configuring the CI systems is impossible, consider adding a specialized "assassin" which will check for out of dateness and kill the relevant processes.

CI cost: kill useless runs

Example: Commit added to patch

Sometimes, it makes sense to stop runs early if they fail. Sometimes it doesn't – it means that people can fix "one problem at a time".

Part of it depends on the fidelity of "local emulation". This can be configured as an opt-out: allow marking specific patches, or repositories, as "keep runs to the bitter ends because it's too hard to emulate locally".

CI cost: stop runs early

Do you need to finish if it fails?

(Sometimes! Trade-offs)

While some of the things that are done for performance cost *more*, mocking out often reduces *cost*, as well as *time*. Even if it takes the same amount of time, sometimes mocking out backends reduces costs enough to be worth it.

CI cost: better tests

Examples: Better stubbing/mocking instead of real services

0.7 Summary

Engineering does not have solutions, only trade-offs. Usually, these are multi-dimensional trade-offs.

The most common dimensions of CI quality to trade off are:

- Effort
- Quality
- Cost

If you put a lot of effort in, you can increase quality *and* reduce cost. By spending money, you can often reduce effort while *still* improving quality. Finally, if you can sacrifice the test quality, you can reduce both effort and cost.

Realistically, you will care to some amount about all three. It is worthwhile to have a rough estimate of how much.

Effort and cost are at least somewhat natural to compare. In most modern cases, both are paid for monthly by the business, in the form of salary and cloud services respectively.

Quality is harder to quantify, but is the most important. Quality is the *value* you get out of the CI system, the reason to spend the money.

CI quality: trade-offs

Effort

Quality

Cost

Decide!

Because of that, it makes sense to *measure* the CI quality. Measure the CI accuracy, actionability, and promptness.

This will have you solve a harder problem, but one that needs to be solved. *What do you value? How much*?

CI quality: measure

Given trade-offs, is this ok?

Now that you have measurements of costs and quality, you know how much effort it is worthwhile to spend. Check which parameters are furthest off from where you want them, and start making things better.

CI quality: improve

What needs to be better?

This is not a one-time effort. Your priorities will change. Your tests will change. Your systems will change.

Keep measuring. Keep tracking. Keep aligning.

CI quality: repeat

Keep your eye on the ball!