Continuous integration

Dr Peadar Grant

April 18, 2024

Contents

1 Continuous integration

Continuous integration technically refers to merging changes back into the main branch as often as possible:

- Idea originated to combine work of developers in teams.
- Key idea (even for a solo developer) is that code is compiled, tested, packaged and deployed automatically on every commit.

Practical usage depends on:

Source control system to record changes and trigger actions.

CI toolset to automate the process

1.1 CI in software development

CI can be used to automate any task, typically:

- Confirming that code compiles correctly without errors (for compiled languages)
- Running automated tests.
- Packaging code or executables into ZIP or other distribution formats (e.g. apk, pkg, MSI installers).
- Generating other artefacts like screenshots, documentation, web pages etc.
- Installing code on test (or production!) environments.
- Announcing updated status using email, Slack, Teams, IRC, Twitter.
- Updating metrics, dashboards, team visualisations etc.

1.2 CI for data science

Some specific ideas for data science:

1. All of the previous ideas from software!
2. Converting markdown to different formats.
3. Generating visualisations using Python from input data.
4. Compiling a LaTeX document to PDF.
5. Data science lecturer creates course plan from topic folders.

1.3 Basic idea

Regardless of how we perform our CI, the basic steps remain the same. The CI tool should:

1. Cleanly clone the repository in full.
2. Run the steps specified.
3. Report the outcome of the steps.
4. Save / upload artefacts created.

Reporting and saving can be done either by the CI tool itself or scripts in the repository.

1.4 Requirements

For any CI to work:

We must be able to run the build step as a sequence of deterministic commands, ideally a single command.

How this works does not matter, but typically:

1. Makefile based on targets / dependencies using `make`.
2. Script in bash, PowerShell, Python, other language.

Unless you can repeatably run your build steps without intervention, you won't be able to run them successfully under CI.

1.5 Artefacts

Normally we want to save some (not all) files generated during the build process.

- Reporting data about testing, performance, sample output etc.
- Files intended for distribution:
  - HTML or PDF from a markdown file
  - Image output from a Python program on a CSV.
  - MSI installer from source code for Windows
  - APK app for an android mobile phone

1.6 CI tooling

There are 3 options:

1. **Roll your own** using git, git hooks and bash / python scripts.
2. **Standalone CI server** like Jenkins, Travis.
3. **Integrated CI solutions** using source control hosts like GitLab, GitHub etc.

Git mainly used with GitLab / GitHub, so integrated option now makes sense in most cases.

2 GitLab CI

To automate actions on commit we need:

Pipeline that defines the steps to perform

Executor environment that will run the steps

Runner service to orchestrate:

1. Creating the executor
2. Running the pipeline

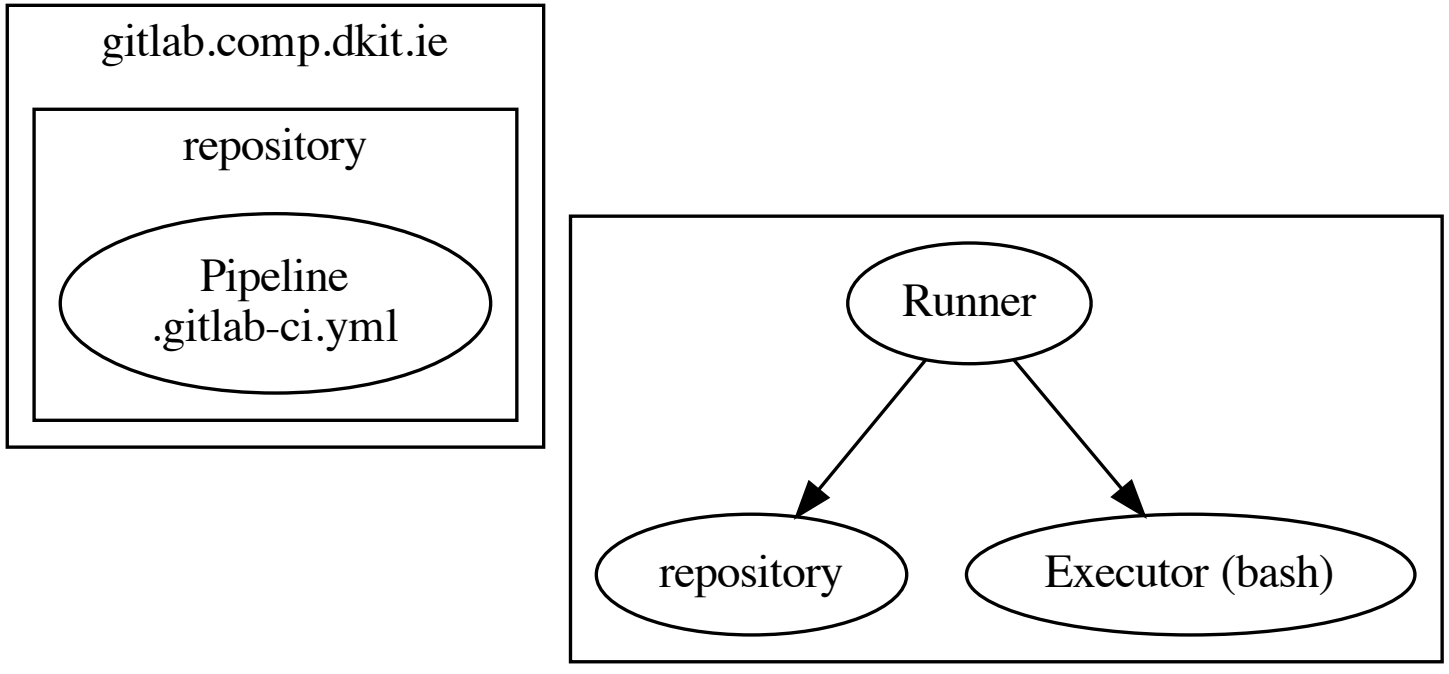

Figure 1: CI environment

2.1 Pipeline

The pipeline defines the commands to execute:

- Stored as file in repository as `.gitlab-ci.yml`
- Pipelines vary in complexity:
  - Simplest pipelines (as we'll use today) just run single (set of) commands when repository updated.
  - More complex pipelines introduce depedencies, stages, rules etc.

2.2 Build machine

Runner and executor are separate to GitLab itself:

- Need to be installed on a suitable computer (here a linux virtual machine) that
- Should be on all the time, so can continuously communicate with GitLab.
- Could be a machine on the network, in the cloud etc.

We will use our XOA Linux instance(s) to host the runner and provide the executor.

2.3 Executor

The executor is the environment used by the runner to run the pipeline. Common executors:

Shell executor uses the default system shell (e.g. bash on linux, PowerShell on Windows)

- Installed already so no further configuration needed.
- Issue: required packages etc need to be installed.

Docker executor uses the docker container system (later)

- Needs to be separately setup.
- Advantage: build environment can be setup in repository.

For today we'll use the shell executor.

2.4 Runner

The runner needs to be installed on our XOA instance:

- The runner communicates with GitLab continuously.
- When new commits are pushed to the repository the runner:
  - Invokes the pipeline using the executor
  - Reports progress back to GitLab for display
  - Captures defined artefacts and sends them to GitLab
- The runner normally is installed as a separate Linux user specifically designated for this purpose.

3 Recommendations

- You must have your build process scripted to begin with.
- You should use the same script(s) to build both locally and in your CI pipeline.

4 Issues

- We are really just using CI here as a trigger to run our script:
  - We could do a lot more in terms of selective reporting if tests or other processes fail.
- Our build environment is not self-contained and is dependent on the machine hosting the runner.
  - Ideally our build environment should be reproducible some way.
  - Consider container, virtualisation or automated package setup.
  - Could consider using the *Docker* runner.