Science and Technology Facilities Council

Technical Report
DL-TR-2020-005

Software Outlook CI/CD – Continuous Integration, Deployment, and Delivery: overview and best practices

B Mummery

December 2020

DOI: 10.5286/dltr.2020005

ISSN 1362-0207

Software Outlook

CI/CD – CONTINUOUS INTEGRATION, DEPLOYMENT, AND DELIVERY

Overview and Best Practices

Dr Benjamin Mummery
STFC Hartree Centre
July 2020

Contents

Contents 1
1 Why Use CI/CD? 2
1.1 What is CI/CD and how Does It Work? 2
1.1.1 Continuous Integration 2
1.1.2 Continuous Delivery 2
1.1.3 Continuous Deployment 3
1.2 Best Practices 3
Isolate and Secure Your CI/CD Environment 3
1.2.1 Prioritise your Tests 3
1.2.2 Make the CI/CD Pipeline the Only Path to Production 3
1.2.3 Build Once, and Once Only 3
1.2.4 Keep the Build Fast 3
1.2.5 Use VCS and issue tracking 4
1.2.6 Use on-demand testing environments 4
1.2.7 Test in a scale model of the production environment 4
1.2.8 Every bug-fix commit should come with a test case 4
1.2.9 Adapt your pipeline to your Development Process 4
1.3 Example CI/CD Workflow 4
1.3.1 Local Testing 4
1.3.2 CI Compiling 4
1.3.3 CI Testing 4
1.3.4 Deploy artefact from CI 4
Acknowledgements 5

1 Why Use CI/CD?

The underlying principal of CI/CD is to shorten the cycle of build-configure-deploy-test-release by taking advantage of convenient automation and more complete testing. This brings with it a cornucopia of advantages. For example:

- Smaller, simpler code changes have fewer unintended consequences, and are easier to debug than large monolithic updates.
- As a result, the mean time to resolution (MTTR) for a fault is shorter.
- The amount of time during which an undetected fault can persist in the codebase, potentially causing problems or wasted effort for further development.
- Testing of these changes can be more specific, and therefore more accurate.
- Releases can be more frequent, reducing the time taken to fix errors that make it into the production version.
- More frequent releases mean that changes are incremental; they are smaller and less disruptive for the end user.
- Shorter development cycles mean that the production version cannot fall too far behind the development version. This means that the end-users can easily be involved in, and provide feedback for, ongoing development in a way that remains relevant to that development.

1.1 What is CI/CD and how Does It Work?

CI/CD stands for either "Continuous Integration / Continuous Delivery" or "Continuous Integration / Continuous Deployment". Which of these is the case depends on the exact workflow being employed and, to an extent, who you ask. To try to dispel as much confusion as possible, we'll start by defining what these three terms actually mean.

1.1.1 Continuous Integration

There is a state known as 'merge hell'. Merge hell occurs when the time and effort required to successfully merge a developer's branch onto the main exceeds the time and effort that went into creating it. In other words, divergence from the main has passed the threshold where it becomes easier to redo the work from scratch rather than integrate what has been produced. This is generally regarded as a **very very bad thing**.

Merge Hell is, clearly, a worst-case scenario, but it is the end result of a gradual (or sometimes not so gradual) accumulation of differences from the mainline that will require additional effort to resolve. The longer a developer works on their own branch separately from the mainline branch, the greater the accrual of these differences.

Continuous Integration (CI) is the practice of merging all developers' working copies to a shared mainline with high frequency, up to tens of times per day. This minimises the divergence, avoids merge conflicts, and ensures that relevant changes are adapted to very rapidly.

1.1.2 Continuous Delivery

Under continuous delivery, changes to the code base are immediately passed through a pipeline of tests and integrations up to testing in a production-like environment, *but not including* deployment. This has two main advantages: the pre-deployment codebase is both up-to-date and fully tested, meaning that it can be deployed at any time with very little in the way of restrictions or overhead; and problems with the newly developed code are detected almost immediately and can be corrected without having to update subsequent work.

1.1.3 Continuous Deployment

Continuous deployment is the next step following on from continuous delivery. Under this paradigm, changes that pass the tests under continuous delivery are deployed automatically. Users always have access to the up-to-date codebase and features can be introduced, removed, or adapted to serve evolving needs on a very short timescale, however doing this removes a degree of control from the developers. Whether this is something you want to do depends heavily on your specific case.

Taken together, these ideas form a hierarchy with Continuous Integration providing the foundation for Continuous Delivery, which can be expanded to become Continuous Deployment. How much of this hierarchy it is useful to adopt depends on you specific circumstances.

1.2 Best Practices

Isolate and Secure Your CI/CD Environment

By necessity, the CI/CD system must have access to your codebase and possess the credentials to deploy builds to various environments. As a result, security and safeguards are essential. CI/CD systems should only be deployed to protected, internal networks that are not exposed to outside parties. VPNs and other forms of network access control are valuable tools to aide this security.

1.2.1 Prioritise your Tests

Huge automated testing pipelines, while powerful and useful, can be expensive and time-consuming. Consequently you should aim to test as little as possible. Note that this does not mean *excluding* useful tests in order to lighten the load. Rather, your pipeline should be constructed such that tests that are performed first are either quick and cheap, or are the most likely to fail, or, ideally, both. The aim is to ensure that if your changes fail a test, they do so as early in the pipeline as possible, so as to minimise the "waisted" effort. This is not always a simple thing to do, as where the test should fall in the pipeline depends on, amongst other things:

- the computational cost of running the test
- the likelihood that the test will be failed by any given change
- the overheads entailed in updating the test as required by further development
- the frequency with which updates to the test will need to be made
- what, if any, build processes are required prior to the test being run.

1.2.2 Make the CI/CD Pipeline the Only Path to Production

The best way to break a CI/CD system is to deliberately circumvent it. Code that has been through the pipeline has been demonstrated to build, pass tests, and adhere to whatever standards you have seen fit to require of it. Code that has not, does not. The pipeline protects the validity of the deployment regardless of whether it is a planned release or a quick fix.

1.2.3 Build Once, and Once Only

It is good practice to build at the beginning of the CI/CD pipeline in a clean environment, and then promote the resulting binaries rather than building again at later stages. This decreases computational cost, avoids confusion over where errors may have been introduced, and allows the tested binaries to be archived as part of the repository along with the source code.

1.2.4 Keep the Build Fast

Even if it’s only happening once per pipeline, building is going to be the most common thing your pipeline does. Therefore, it *needs* to be fast. This can be accomplished by taking advantage of horizontal or vertical scaling, for example splitting the build into parts that can be done in parallel.

1.2.5 Use VCS and issue tracking

A Version Control System (VCS) serves as a 'single source of truth', allows tracking of changes, and provides the ability to easily roll back changes. VCS goes so well with CI/CD that VCS hosting services such as GitLab specifically include CI/CD tools. For more information, see our guide on VCSs. Hand-in-hand with this goes the use of issue tracking tools like Jira and Bugzilla to achieve better visibility and coordinate effort across teams.

1.2.6 Use on-demand testing environments

Containers are an invaluable tool, both for reducing and controlling the number of variables in your testing environments and allowing your testing infrastructure to be ephemeral. A container can be spun up when needed, imaged directly from a centrally controlled container image with no additional installation or configuration requirements, and then destroyed when their work is complete, freeing up the resources to be used elsewhere.

1.2.7 Test in a scale model of the production environment

Testing environments can never perfectly replicate the production environment, but every difference invites the possibility for errors to go unnoticed. Differences should therefore be minimised as far as possible, with those aspects that cannot be replicated, for example third-party APIs, services, and other dependencies, analogized by service virtualization.

1.2.8 Every bug-fix commit should come with a test case

When fixing a bug, push a test case that *reproduces* the original bug. This prevents *regression* - the problem of subsequent alterations breaking your fix without anyone noticing.

1.2.9 Adapt your pipeline to your Development Process

Pipelines are not one-size-fits-all, and should be adapted to the specific circumstances in which they find themselves. Do not be afraid to modify the pipeline when it no longer suits.

1.3 Example CI/CD Workflow

1.3.1 Local Testing

- Unit and potentially integration tests run in the developer's local environment before committing to the main.
- limits the impact of bugs / catches them early.
- Incomplete features disabled with e.g. feature toggles.

1.3.2 CI Compiling

- build server compiles code either periodically or per-commit and reports results.

1.3.3 CI Testing

- Unit and integration tests in full context
- Additional static analysis, measure performance profile, extract and format documentation from source code, set up for manual QA processes.

1.3.4 Deploy artefact from CI

- Automate the deployment of CI builds that pass tests.

Acknowledgements

This work made use of computational support by CoSeC, the Computational Science Centre for Research Communities, through its Software Outlook Activity.