The Deployment Pipeline

(Extending the range of Continuous Integration)

Dave Farley 2007

Continuous Build

A core practice in Agile development projects is the use of Continuous Integration. CI is a process by which builds and extensive suites of automated test cases are run at the point that software is committed to the version control system.

This practice has been in use on projects for many years and provides a high degree of security that at any given point the software under development will successfully build and pass its unit-test suite. This significantly increases confidence that the software serves its purpose. For many, some would say most, projects this is a huge step forward in the quality and reliability of the software ultimately delivered.

In complex projects though, the potential for problems doesn't stop at the point at which the code will compile and pass unit tests.

However good unit test coverage, by its nature, it tends to be close to the solution of the problem. In some circumstances unit tests can get into a state where they prove only that the solution is the same solution that the development team envisaged, rather than that it is one that meets the requirements.

Once the code is built, it must be deployed, for most modern software built by teams of developers this is not a simple case of copying a single binary to the file system. Instead it often involves the deployment and configuration of a collection of technical pieces, web-servers, databases, application servers, queues and others in addition to the software itself.

Such software usually has to go through a reasonably complex release process, visiting a variety of deployed environments on its progress towards production. It will be deployed to development machines, QA environments, performance test environments, production staging environments before finally making it into production.

In most projects most, if not all, of these steps will include a significant degree of manual intervention. People will manually manage configuration files they will manually tailor the deployment to suit the environment to which it is being deployed. There is always something to forget. *"It took me two hours to find that the development environment stores its template files in a different location to production"*.

Continuous Integration helps, but in its most common incarnation it is misnamed, it should be called "Continuous Build", what if it really applied to the whole release process?

Beyond Continuous Build

The teams I have been working on have been taking this to heart for the past couple of years and have been building end-to-end Continuous Integration release systems that will deploy large complex applications to whichever environment we choose at the click of a button. This approach has resulted in a dramatic reduction in stress at the point of release, and a significant reduction in problems encountered. During the process of establishing these end-to-end CI environments, we have discovered a fairly general abstraction of the build process that helps us hit the ground running, and allows us to build fairly sophisticated build systems rapidly at the start of our projects.

The process is based on the idea of a release-candidate progressing through a series of gates. At each stage the confidence in the release candidate is enhanced. The objective of this approach is to develop this level of confidence in a release candidate to a point at which the system is proven ready for release into production. True to agile development principles this process is started with every check-in, every check-in, in effect, being a viable release candidate in its own right.

As the release candidate proceeds through the process, some of the gates through which the release candidate may pass are essential to most projects and some are more tailored to meet specific project needs.

We commonly refer to this process, this sequence of gates, as a Deployment Pipeline.

Full Lifecycle Continuous Integration

Figure 1 shows a typical full lifecycle Deployment Pipeline and captures the essence of the approach. This process has proven, through experience on a number of different projects, to be fairly generic, although it is, as with all agile practices, normally tailored and tuned to meet specific project needs.

The process starts with the developers committing changes into a source-code repository, the Continuous Integration system, typically Cruise Control on our projects, responds to the commit by triggering the CI process. In this case it compiles the code, runs commit-tests and if they all pass creates assemblies[1] of the compiled code and commits these assemblies to a managed area of storage.

[1] In this instance assemblies is meant to refer to any grouping of compiled code, these will be .NET assemblies, Java jars, WAR and EAR files. A key point is that these assemblies do not include configuration information. We want the same binary to be run-able in any environment.

Managing Binaries

A fundamental of this approach is that each step in the process should move the release candidate forward in its progress toward full release. One important reason for this is that we wish to minimize the opportunities for errors to creep into the process.

For example, when we only store source code, we have to re-compile that source code each time we wish to deploy, if we are about to run a performance test and need to first re-compile the source-code, then we run the risk of something being different in the performance test environment, perhaps we inadvertently used a different version of the compiler, or linked with a different version of a library. We want to eliminate, as far as we can, the possibility of us inadvertently introducing errors that we should have found at the commit-test or functional-test stages of the Deployment Pipeline.

This philosophy of the avoidance of re-work within the process has several side-benefits. It tends to keep the scripts for each step in the process very simple and it encourages a clean separation of environment-specific stuff and environment-neutral stuff[2].

However, care needs to be taken when managing binaries in this manner. It is too easy to waste vast amounts of storage on binaries that are rarely if ever used. In most cases we compromise. We avoid storing such binaries in version control systems, because the overheads are simply not worth the benefits. Instead we have started using a dedicated area of a shared file-system, we manage this as a rolling binary repository, archiving binaries into version-lab-labelled, compressed images, so far we have written our own scripts for this stuff, not yet having found a suitable off the shelf alternative.

These binary images are tagged with the version information of the source-code they were built from. It is easiest to think of the build-tag as a release-candidate identifier, it is used to represent the association between all the source-code, all the binaries and all the configuration information, scripts and anything else it takes to build and deploy the system.

This collection of ‘managed-binaries’ represent a cache of recent builds, past a certain point we delete the old binaries, if we later decide that we need to step back to a version for which the binaries have been removed, we must re-run the whole Deployment Pipeline for that source-tag, which remains safely in the version control system, but this is a very rare event.

[2] The process implicitly discourages the compilation of binaries for specific deployment targets. Such deployment-target specific binaries are the antithesis of flexible deployment, yet are common in enterprise systems.

The Check-in gate

The Deployment Pipeline is initiated by the creation of a release candidate. This candidate is created implicitly when any developer commits any change to the version control system.

At this point code will be compiled and a series of commit-tests will be run. This collection of commit-tests will normally include all unit-tests, plus a small selection of 'smoke-tests', plus any other tests that prove that this check-in does indeed represent a viable release candidate, one that is worth the time to evaluate further.

The objective of these commit-tests is to fail fast. The check-in gate is interactive; the developers are waiting for a pass before proceeding to their next task. In view of this speed is of the essence to maintain the efficiency of the development process. However, failures later in the Deployment Pipeline are more expensive to fix, so a judicious selection of additions to the commit-test suite, beyond the essential unit-tests, is often valuable in maintaining the efficiency of the team.

Once the commit-tests have all passed, we consider that the check-in gate has been passed. Developers are now free to move-on to other tasks, even though later stages in the Deployment Pipeline have yet to run, let alone pass.

This is more than anything else a process-level optimization. In an ideal world where all acceptance tests, performance tests, and integration tests will run in a matter of seconds there is no advantage to pipelining the CI process. However, in the real world these tests always take a long time to run, and it would be a massive blow to the productivity of a development team to have to wait until all had completed successfully before being able to move on.

Treating the commit-test build as a check-in gate frees the team to move on with new tasks. However, the team is expected to closely monitor the outcome of the release candidate that results from their check-in through the rest of its life-cycle. The objective is for the team to catch errors as soon as it can and fix them, while allowing them to get on with other work in parallel with lengthy test runs.

This approach is only acceptable when the commit-test coverage is sufficiently good to catch most errors, if most errors are being caught at later stages in the pipeline it is a good signal that it is time to beef up your commit-tests!

As developers we will always argue for the fastest commit cycle, in reality this need must be balanced with the check-in gate's ability to identify the most common errors we are likely to introduce. This is an optimization process that can only work through trial and error.

Start the design of your commit-test suite by running all unit-tests, later add specific tests to try and trap common failures that you see occurring in later stages of the pipeline.

The Acceptance Test Gate

Unit tests are an essential part of any agile development process, but they are rarely enough on their own. It is not uncommon to have an application where all unit tests pass, but where the application doesn't meet the requirements of the business it was developed to serve.

In addition to unit tests, and the slightly broader category of commit-tests, the teams I work with rely heavily on automated acceptance tests. These tests capture the acceptance criteria of the stories we develop, and prove that the code meets those acceptance criteria.

These are functional tests, that is they exercise the system end-to-end, though often with the exception of any interactions with external systems outside our control. In those cases we generally stub such external connection points for the purposes of our acceptance test suite.

We like to bring the creation and maintenance of these acceptance tests into the heart of our development process, with no story deemed complete until its acceptance criteria are tested by an automated suite created by the developers and added to the acceptance test suite. We tend to expend some energy on ensuring such tests are very readable, even to non-technical people, but that is perhaps less fundamental to our CI process, and so is outside the scope of this article.

Acceptance tests are run in a controlled environment dedicated to the task, and monitored by our CI management system (usually Cruise Control).

The acceptance test gate is a second key point in the life-cycle of a release candidate. Our automated deployment system will only deploy release candidates that have passed all acceptance tests. This means that it is not possible to progress any release candidate beyond this stage into production unless all acceptance criteria are met.

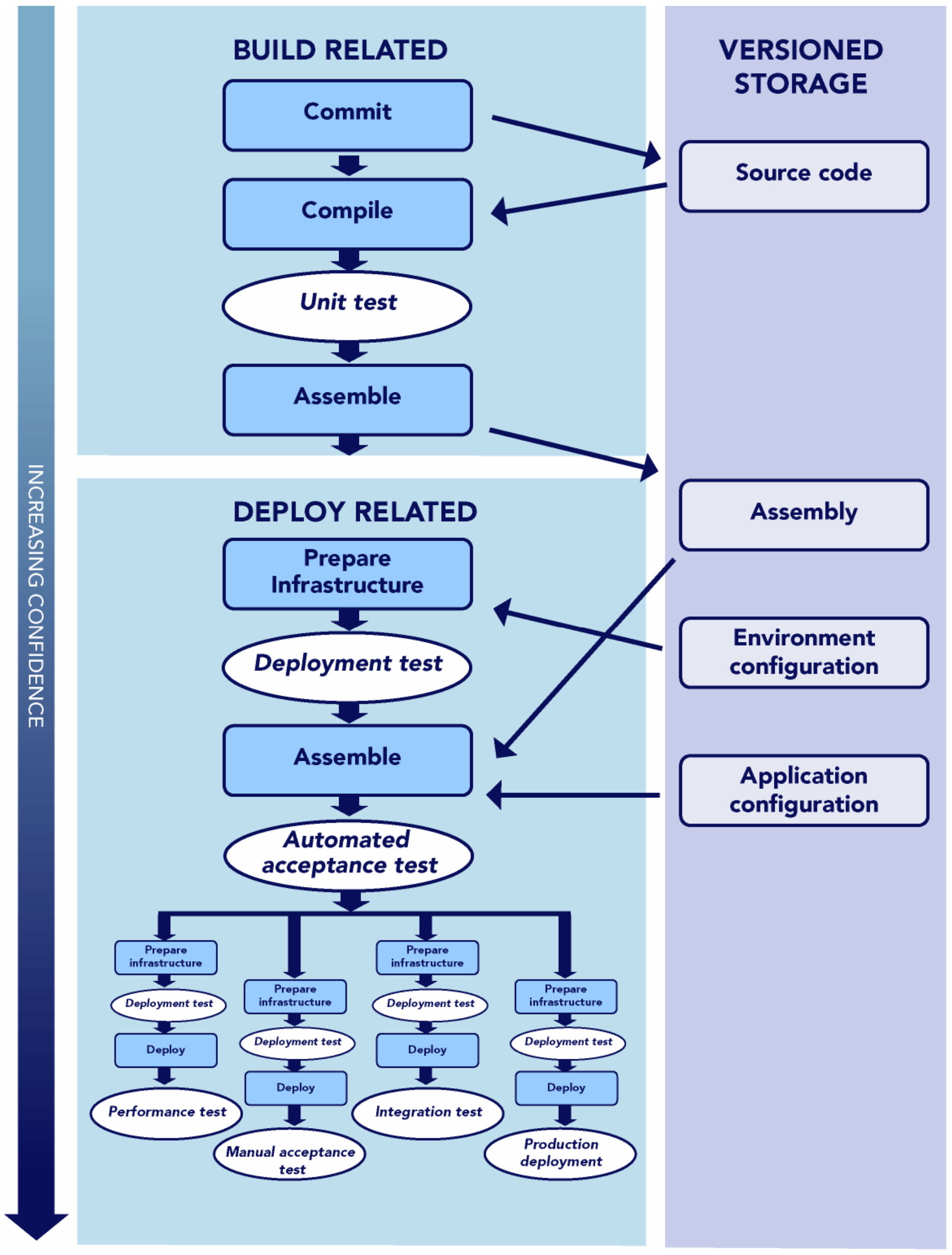

Figure 1 - Deployment Pipeline

Preparing to Deploy

In some circumstances it may make sense to automate the deployment of everything associated with an application, but for large scale enterprise applications this is rarely the case. However if we could automate the management and configuration of the

entire infrastructure, it would eliminate the cause of many errors, specifically the manual deployment and configuration process so common in enterprise-scale systems. This being the case, the attempt is worth some effort, and even if we only partially succeed we will usually eliminate many sources of common errors.
We have adopted a pragmatic approach to this problem we will often rely on a standard server image, application servers, message brokers, databases and so on. These images will represent some form of 'snap-shot' of a deployed, configured system installed and configured with a base-level configuration.

Such images can take many forms, whatever is needed or convenient for the project. Often we will have a database script that will establish a starting schema and a dump of data that will populate it. We may have standard OS installations, or application-server configurations that can be deployed and established as part of the commissioning process for any server we decide to deploy to, it may even be as simple as a copy of a folder tree to a file system, so we always have the same structure in-place.

Whatever the nature of the 'image' the intent is to establish a common base-line configuration so that subsequent changes can be maintained as a collection of deltas from it.

Often we will maintain a collection of such images so that new environments can be set up quickly leaving little room for human error.

This raw infrastructure is not deployed each time we deploy the software, in most cases it is laid-down at the point at which we commission some new environment, and then rarely touched.

However, each time we deploy the application, we will reset the infrastructure to as close to this base-state as is possible, in order to establish a known-good starting point for the remainder of the deployment.

Once this baseline infrastructure is in place, it may prove helpful to run a suite of simple deployment tests. These tests are intended to confirm that the basic infrastructure is in-place and ready to be configured to the specific needs of the application and specific environment. Typically these tests will be very simple representing an assertion that the basic plumbing is in place, for example ensuring that the DBMS is present and the web-server is responding to requests.

If these tests fail we know that there is something wrong with our image or the hardware.

Once these tests pass, we know we are ready to deploy the application. The application-specific deployment scripts are run to copy our assemblies, which were built and tested during the commit stage of the process, from their area of managed storage to the correct locations.

In addition to simply copying binaries, our scripts will, where necessary, start and stop any application servers or web-servers, populate databases with schemas, or updates as appropriate, perhaps configure the message broker and so on.

In essence deployment is a five stage process, with four stages for each individual deployment:

1. Third-party infrastructure is installed, often from an image where practicable. *This is only done at commissioning time for a new server environment.*

2. The infrastructure is cleaned to a known-good start-state.
3. Deployment tests confirm the infrastructure is ready for the software.
4. Application assemblies are deployed.
5. The infrastructure is configured appropriately to the needs of the application.

We divide our build/deploy scripts into small pieces to keep them simple, as with any other well-factored software. Each is focussed on a specific task and relies as much as possible on clearly defined inputs.

Subsequent Test stages

As stated earlier the acceptance-test gate is a key milestone in the project lifecycle. Once passed the release candidate is available for deployment to any of a variety of systems. If the candidate fails this gate it is effectively un-deployable without significant manual effort – this is a good thing, because it maintains the discipline of only releasing code that has been thoroughly tested and, as far as our automated test suite is able to, proven to work.

Progress through the Deployment Pipeline to this point has been wholly automated, if the release candidate has passed the previous stage of the process, it is promoted to the next stage and that stage is run.

In most projects that approach doesn’t make sense for the remaining stages in the process, and so instead we make the following stages optional, allowing any release candidate that has passed acceptance testing to be selected for deployment to either manual user acceptance testing, performance testing or, indeed, deployment into production.

For each of these deployments the steps described in the “Preparing to Deploy” section are performed, helping to ensure a clean deployment. By the time a release reaches production it will have been successfully deployed using the same techniques several times, and so there is little concern that anything will go wrong.

In my last project, each server that could host our application had a simple web-page that provided a list of available release candidates from which one could be selected and the ability to optionally re-run the functional test suite and/or the performance test suite in that environment. This provided a high degree of flexibility in our ability to deploy our system any time we wanted to wherever we wanted with very little fear of inadvertently introducing errors in the process.

The degree of automation involved, or not, in promoting release candidates through these subsequent stages is perhaps the most variable part of this otherwise fairly generic process. On some projects it makes sense to always include a performance test-run, on others it may not. The details of the relationships between the different, post-acceptance test, stages and whether they are selected manually or run automatically is not really a problem providing the scripts that manage the CI process are well factored.

Automating the Process

Figure 2 shows a simple map of the scripts used for the automation of a Deployment Pipeline. Each box represents a process stage, each line within a box represents an individual script, or build-script target, that fulfils that function.

In most projects using this approach the first two process gates, check-in and acceptance, are initiated by a Continuous Integration management application like Cruise Control.

One of the important benefits of this approach to organising the build scripts is that each script, or script element is focussed on doing one, relatively straight-forward, thing well rather than trying to manage the entire build process in one complex step. This is a very important gain, in ensuring that the build process is itself manageable and amenable to change as the project evolves and matures.

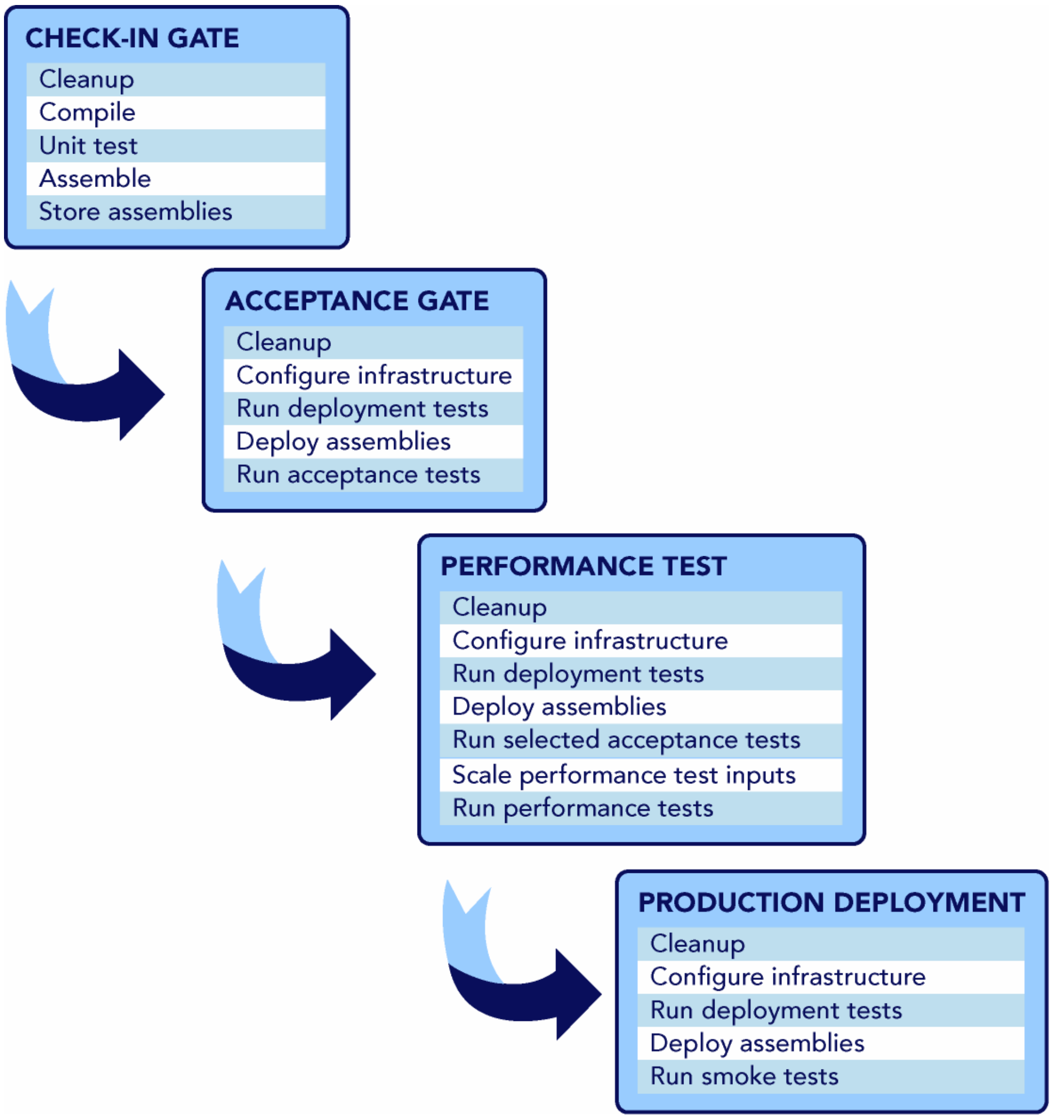

Figure 2 – Example Process Steps

The detail of these scripts is not within the scope of this article, and is, in reality, too project dependant to be of much interest, but we have found in a number of different projects that when we apply this kind of high-level structure to our build processes we get reliable, repeatable, trustworthy deployment, allowing us to deploy in seconds or minutes what previously took days, and often fraught weekend days at that!

Conclusion

If your organisation is not yet using a CI approach to your build start tomorrow, it is the single most effective means of improving the reliability of your systems that we have found.

Extending the range of CI to effectively eliminate as many sources of human error as possible has enormous benefits, not just in productivity, but also in the quality of the deliverable and in the lowering of stress at the point of delivery into production.

Find our more about CruiseControl Enterprise at
http://studios.thoughtworks.com/cruisecontrol