The Deployment Production Line

Jez Humble, Chris Read, Dan North
ThoughtWorks Limited
jez.humble@thoughtworks.com, chris.read@thoughtworks.com, dan.north@thoughtworks.com

Abstract

Testing and deployment can be a difficult and time-consuming process in complex environments comprising application servers, messaging infrastructure and interfaces to external systems. We have seen deployments take several days, even in cases where teams have used automated builds to ensure their code is fully tested.

In this paper we describe principles and practices which allow new environments to be created, configured and deployed to at the click of a button. We show how to fully automate your testing and deployment process using a multi-stage automated workflow. Using this "deployment production line", it is possible to deploy fully tested code into production environments quickly and with full confidence that you can fall back to a previous version easily should a problem occur.

1. Automate your deployment process

It is often the case when working on a large, complex application that it must be deployed into a number of different environments on its journey through into production. In the course of doing this we often spend a great deal of time building the software, getting it to work in the various testing and production environments, and debugging integration problems between the application and other systems.

Often the processes for building, testing and deploying such applications into non-developer environments are manual, and can in extreme cases take days. These processes are usually complex, difficult to repeat reliably, and subject to change during the development process. Sometimes they are documented, but often the documentation is incomplete or out-of-date. In some cases, the information needed to deploy resides in the heads of several key members of staff who need to come together to perform the deployment. It is common for teams to be working on several application streams that share parts of their code base, such as libraries and frameworks.

One solution to these problems is to automate fully the build, testing and deployment process. Automating this process in the early stages of your project is of immediate worth. Not only does it save developers time, but it also helps detect problems with deployment early on in the development cycle, where fixing the problem is cheaper.

Furthermore, automating the entire deployment process embodies a key agile practice – making your code (in this case your deployment scripts) your documentation. One benefit gained from this practice is that your build and deployment scripts capture your deployment and environment testing process, and can be leveraged to give you rapid feedback not just on the integration of the modules of your code, but also on problems integrating with your environment and external dependencies.

However, automating deployment can be complex, time consuming and daunting, and it is not usually clear exactly how to go about it. In this paper we will present four basic principles, look at some practices they motivate, and give examples of how to implement these practices in order to address what we believe are the most common challenges facing automation of the build and deployment process.

These principles are:

1. Each build stage should deliver working software – don't have separate stages for intermediate artifacts such as frameworks.
2. Deploy the same artifacts in every environment – manage your runtime configuration separately.
3. Automate testing and deployment – use several independent testing stages.
4. Evolve your production line along with the application it assembles.

In the following sections, we will examine the motivations for these practices and discuss the practical benefits they deliver.

2. Each build stage should deliver working software

In the process of creating software, we write modules and package them in such a way that we can separate concerns within the application. These modules have dependencies on each other that are arranged such that the build process is as efficient as possible.

Often, however, software has larger scale structures than modules, such as a framework that is shared between several independently deployed parts of an application, or between several distinct applications.

It might seem logical to arrange these dependencies as a series of independent stages in the build process. Thus a change to the framework causes it to be built and tested and an intermediate library to be generated. The library thus built would then be checked in to dependent projects, which would in turn trigger the builds of these dependent applications. In effect, all this is doing is creating a dependency on binaries as well as source code, instead of depending on changes in source code alone to trigger builds. Current continuous integration tools all support triggering a single build from multiple source code repositories, so there is no technical reason why this we should not stick to source-only dependency.

There are two primary reasons why checking in intermediate binaries can cause problems. Firstly it can be inefficient (especially when there is a large amount of change going on in the framework as well as modules that depend on it), and secondly it loses information.

To see how it is inefficient, consider a stream of several parallel workflows, each with several intermediate library builds. Each time a library gets built as a result of a check-in at the beginning of the production line, a whole series of intermediate builds only get triggered once this binary itself is checked in. This means a time lag between checking in your code and the builds of your actual software. Anything that slows down your build process should be avoided, since quick feedback from continuous integration is essential to agile development.

To put this in perspective, one project with separate, dependent builds for each discrete module took just over 30 minutes from a framework source code check in to all dependent modules being built and tested. Moving to a single build took it down to less than 3 minutes for the same code. It turned out that most of the time lost was in building up and tearing down the Ant JVMs to compile and test each module, plus the polling intervals for the different build stages. Taking the same approach on other projects has had a similar, though less dramatic impact.

Another issue with building dependent components in independent stages is that you lose the connection between the initial source code check-in and the triggering of the later build stages. For the later stages, all you know is that a dependent library changed. To find out what source code actually changed and caused your build to trigger (and maybe fail!), you have to trace back through the previous stages. If your framework is under active development, this can become inconvenient and frustrating.

One solution to both these problems is to build only deployable binaries. This means that you have a single build for each application. This build will be kicked off by any change in the source code of any part of that application, including the framework. A check-in of any part of the code will cause the continuous integration server to check out all source code, including that of the framework, and build the entire application in one go, running all associated tests.

This allows the developers to see exactly what changes occurred in a particular build, and hence to determine quickly the exact change that caused the build to trigger. It also delivers the benefit that you need not check in potentially weighty libraries into version control, and that you reduce the number of builds and the length of time it takes to perform them. The disadvantage to this model is that your framework code is compiled and unit tested for each separate application, but in practice this is a negligible overhead.

It is important to note that your source code should still be organized in exactly the same modular way you normally would. This may mean having separate source control projects for your frameworks, common modules and applications. These components may also each have their own continuous integration projects. The important point is that the binaries created by one build stage don't get used as inputs to dependent projects.

The benefit of this approach is greatest when the framework is still developing and evolving at the same pace as the components that use it. As a framework matures and stabilizes, it should then be treated just as any other third party module that the software depends on. This is especially true when the system is large enough that parts of it are upgraded independently.

3. Deploy the same artifacts in every environment

The build should generate one or more deployable artifacts. Components such as intermediate jars should

be rolled into the deployable artifacts within a single build stage. The same artifacts will be deployed into all of your environments, from development through the various UAT and staging environments into production. This ensures that what you test is what actually ends up in production.

One of the primary objectives of configuration management is to get your application up and running on new environments as simply and quickly as possible. It should also be possible to change your configuration at runtime without the application becoming unavailable.

In terms of application configuration, a common anti-pattern is to aim for 'ultimate configurability'. This is often a requirement of new applications, even if only stated implicitly. To avoid this anti-pattern it is important to understand what you want to configure and how often it is likely to change, and devise the simplest possible configuration system that handles these cases.

However, the application binaries are just one part of your configuration strategy. You also have to manage the configuration of your application server stack, databases, network and operating system. In practice it is possible to manage these different components of your environment using the same strategy.

The key to managing configuration is twofold: *separate binaries from configuration*, and *keep all configuration information in one place*.

In the case of your application this means moving its runtime configuration outside of any deployed binaries to a separate place. The same applies to your application server, operating system, database server, etc. It is worth checking that the way the binaries and the configuration have been separated will support the various different environments that need to support your application.

There are two different approaches to doing this. Both have pros and cons, and most of them centre around ease of upgrading. There is often a fear of upgrading the underlying technology of an application, or infrastructure such as the application server, based upon its complexity. However, having a reliable automated system that configures and tests your environments allows you rapidly to gain confidence that an upgrade will not have a negative impact.

The first and often easiest way to separate binaries and configuration is to prepare a stripped down version of each infrastructure component based on the standard installation, and then apply the configuration on top. Examples of this include having a standard operating system or application server build. When you request a new machine to deploy your application to, you know what will be installed, and all you have to do is apply your code and your configuration.

The second approach is to use a script to apply changes to the standard installation, reconfiguring it and moving around files. This is often harder because it requires more logic and control in the application of the configuration deltas, and hence requires more time to implement.

Whichever approach is taken, once you have separated out your configuration, it should be stored in a version control system such as Subversion. It can then be made available either directly from version control, or via LDAP, a RESTful web service, or some other simple, generic method.

Your binaries should also be stored somewhere easily accessible, either on an exported file system or referenced by a URL so that you can create new environments or upgrade your software as simply and quickly as possible.

Another choice that needs to be made in any configuration system is whether to have defaults that can be overridden, or to require all configurations to be explicit. The advantage of the former approach is that it keeps the custom information for each environment small and easy to manage. However, the latter option means you have all the configuration information in one place.

While developing an online user management system for a well-known ISP, we used a combination of these two approaches. The configuration system we used for automatically building and configuring our WebLogic environments applied a set of defaults, with override properties to keep track of where each environment differed from this default. However the application itself had its default configuration compiled in, with each environment having an explicit set of runtime configuration options.

4. Automate testing and deployment

The final stage of the deployment production line is to automate testing and deployment.

There are three principles that drive how this should be done. The first principle is that different types of testing should be independent from each other, with every stage of application testing tagging a particular version of a binary with an "OK" or "fail" stamp. The second principle is to automate fully deploying into all test environments, staging, and even production. Finally, you need to be able to test the environments to which you are deploying.

4.1. Separating out your application testing

There are many types of testing that an application needs to undergo before it can be put into production. Unit, functional, integration and performance tests all need to be executed. Many of these tests can be done in parallel, and not all builds need to undergo all tests. For instance, if a tester wants to carry out exploratory testing, they may only require a build that has passed its automated functional tests, rather than the full performance-testing suite.

Splitting up automated testing into several different suites gives development teams rapid feedback, and is a good use of resources. For example, functional tests can be split up such that a simple set of smoke tests that complete in a couple of minutes are run first. Then a suite that tests the "happy path" of your business logic can be run. Following this, your main functional test suite can be executed, excluding UI tests, which would be separated out into a final suite.

When these tests do pass, the source code that generated the tested binaries should be tagged to indicate this. Tagging the source with the same name as the binary is a simple and effective way of tracking the relationship between the two. The same approach can also be used for environments in which manual testing is performed.

When the continuous integration system has informed the testers that automated testing is complete, they can begin manual testing. Once this is complete, they can then pass or fail that version of the software. If the software passes all manual and automated tests, it is then ready for deployment into staging and production.

One metaphor for this aspect of the build production line is scout badges. As the binaries pass certain testing stages, they earn badges. Badges don't necessarily need to be earned in any specific order, although some badges may define pre-requisites. It's also possible to drop out of the badge certification process early if you decide it's not for you. At the end of the process, if a build has a full set of badges, then it earns a big shiny badge that says it has completed the entire gamut of tests. Once a binary reaches this stage, it can be considered fit for production.

As an easy way to keep track of the various versions of binaries you are generating, the temptation is often to check these binaries into the same version control system you use for your source code and configuration. The problem here is that in many such systems, once you've checked something in, it's very hard to get rid of it. In the case of checking in binaries that are built as part of a continuous integration process, you can have tens of check ins per day.

In one scenario we decided to check the binaries into our Subversion repository from day one. Three months in to the project, we discovered that the repository was 5.5GB in size, but of that only about 250MB was source code.

The easiest way to get around this is to use a simple file system hierarchy to keep your binaries in. Having this file system available as an NFS or Windows share, or even as a URL, allows easy access for your testing systems to the binaries they need to test. Where the test production line is linear, simply copying the file to a new section of this file system once a test on it has passed works well. A more complex system is needed where the production line branches out into parallel stages. Methods we've used include creating files in the same directory as the binaries when they pass, and using an issue tracking system.

Once a binary (or set of binaries) has passed all the required tests, it can then be moved off to a safer location. Binaries that are no longer required can be removed.

4.2. Automate deployment

If left unchecked, deploying an application to a test environment can, over time, become a long, error prone ordeal. This is why you should automate deployment early.

All modern application servers have scriptable remote administration tools. For example, Microsoft's IIS can be administered remotely using Active Directory. Your build and deploy process should leverage these tools.

Bringing up and down clusters of application servers, deploying binaries to these clusters, configuring messaging queues, loading databases and related deployment tasks can all be scripted. Build systems such as Ant provide tools for performing many of these tasks, but if Ant cannot carry out the tasks required, custom scripts can fill the gap.

Once you have this in place, testers can select which build they would like to deploy into a specific environment, and trigger a deployment of that build themselves. Having this "one click deployment" that can be triggered by testers as and when they are ready to test new builds has had a great impact on all the members of the project teams where we have implemented it.

On one large J2EE project, deploying a new build to a testing environment literally required more than a day of a developer's time in order to get a new version of the software deployed and ready for testing. The complexity and pressure involved often resulted in a higher than expected number of human errors in preparing the environment, usually requiring the whole process to start again. Having the whole environment set-up and deployment process automated brought that time down to less than half an hour.

It is a common requirement to have multiple versions of an application working on the same server or cluster at the same time, especially when hardware is at a premium. It is also vital in functional testing where you want to compare the behaviour of different versions of an application.

These issues can sometimes be solved using *slices*. A slice is a single instance of your application server and application, using a preset range of resources on your server (a set of ports, a directory on the file system, a labelled messaging queue, etc.). Once you have separated out your binaries and configuration, it is possible to deploy multiple slices on a machine simultaneously. In this way, when you deploy the latest version of your application, you create a new instance of the application server, assign it an unused range of ports and other resources on the host, deploy your application, and bring it up.

Thus you have multiple versions of your application testable at any time. This strategy can be used on your production environment to ensure that your service is continually available, and to provide an extremely simple failover or back-out strategy to a previous version of your software in case the newest version fails.

There are a few things to be aware of when using this approach. The key one is to ensure that this can be done safely. Not all infrastructure components support a slicing model. Another issue arises when changes need to be made to a component that is shared between slices, although this can usually be worked around with a bit of care.

With the more mainstream adoption of virtualization, a new and compelling paradigm is emerging. In addition to creating environments using scripts, it also becomes possible to create entire "canned" environments, and control them programmatically.

Using a standard virtual machine image makes it easy to have a well-known, consistent environment to use for all your testing. Developers can run local copies of the virtual machine on their desktops, and make sure that their new code runs in the environment as a pre-check in requirement. It also makes it possible to script complex scalability and integration tests. For instance, you could simulate a cluster with a set of virtual machines. You could then fake a catastrophic failure by programmatically bringing down a virtual machine and seeing how the cluster reacts. In the same way, network failures, database connectivity problems and application scalability can be programmatically induced and the application's behaviour tested. We have also used virtual machines to host multiple versions of CruiseControl on one box to avoid resource conflicts, for example when running automated UI tests. However this is a broad topic that is beyond the scope of this paper, so we will leave discussion of this promising field in build engineering.

4.3. Test your environments

Once your environment is configured and your application deployed, you can also script environment testing tasks, sometimes known as *smoke tests* (based on the principle of "switch it on and see if smoke comes out"). These tasks include ensuring you can connect to each of the nodes on a cluster, checking the database is reachable and that your login is valid, making sure the correct version of the application is deployed, sending test messages across message buses and other common integration tests.

Once these tasks are automated, it becomes possible to set up an environment, deploy to it and run integration tests in a fully automated fashion.

The end result of this automation is that your build process extends beyond continuous build and testing. It covers configuring your test environments, testing the environments themselves and deploying your binaries into them.

5. Evolve your production line along with the application it assembles

The diagram below describes a theoretical full production line. Actual implementation of the production line will vary depending on the nature of the application being developed. Just as each application is different and evolving over time, so the build process that supports the application will evolve. The ideas presented here are a guide based on what we have seen in the field. Trying to build a full production line before writing any code is written doesn't deliver any value, so it is important to apply the same amount of pragmatism to build automation as you do to your code.

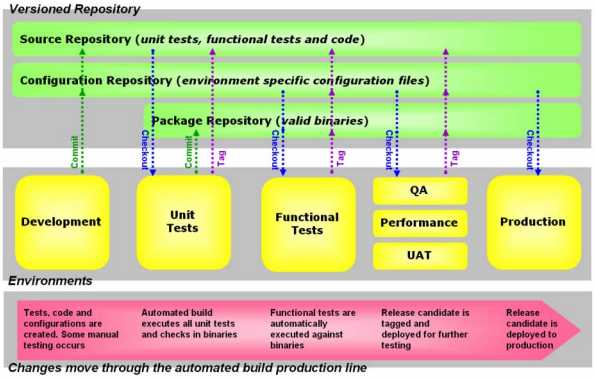

Figure 1: The Deployment Production Line

None of the projects we have mentioned has a complete production line as described above. Either we felt that that they didn't require the full process at their current stage of development, or we came into the project at a later stage, where the cost of fully retrofitting the production line outweighed the benefits.

6. Conclusion

By carefully organizing a project build system so that every stage delivers independent value, we ensure that builds are completed as quickly as possible and that we can link individual builds directly to the source code changes responsible for triggering them. Organizing builds monolithically such that the binary built is unchanged right through to deployment guarantees that what you test is what ends up in production.

The second principle of the build production line is to separate binaries from configuration – for all infrastructure components in addition to your application binaries – and manage configuration using a simple, consistent strategy. Careful application of this principle allows you to upgrade elements of your infrastructure without undue pain.

The next step is to manage application testing on the scout badge model. Your automated deployment system is responsible for setting up and smoke testing environments, and deploying binaries into them for testing.

In order to ensure you can run multiple versions of your application on your servers at the same time, you can use slices and virtualization. These techniques also allow you to deploy a new version of your application into production safe in the knowledge that you can fall back to the previous version easily.

These various principles and practices comprise the deployment production line, a multi-stage, automated workflow of tasks that facilitates multiple teams building and deploying into complex environments. Its emphasis is on simplicity and feedback, using *de facto* tools such as CruiseControl, Subversion and Ant to ensure a consistent, robust and automated build, test and deployment process from development through UAT into production.