2015 IEEE/ACM 3rd International Workshop on Release Engineering

# Continuous Delivery with Jenkins

Jenkins Solutions to Implement Continuous Delivery

Valentina Armenise
CloudBees
Brussels, Belgium
valentina.armenise@gmail.com

*Abstract*— This paper illustrates how Jenkins evolved from being a pure Continuous Integration Platform to a Continuous Delivery one, embracing the new design tendency where not only the build but also the release and the delivery process of the product is automated. In this scenario Jenkins becomes the orchestrator tool for all the teams/roles involved in the software lifecycle, thanks to which Development, Quality&Assurance and Operations teams can work closely together.

Goal of this paper is not only to position Jenkins as hub for CD, but also introduce the challenges that still need to be solved in order to strengthen Jenkins' tracking capabilities.

## I. Introduction

The Time To Market of a product critically affects its success especially when talking about technologies, which have to be delivered while they are still new. In such environment, then, what really differentiates the product on the market is not only the product itself, or its quality, but also the speed at which it can evolve: for a company, taking the product to market fast means to win over the competitors and being always aligned with new tendencies.

The concept of Continuous Integration (CI) was a first step that significantly sped up the lifecycle of a product, pushing developers to commit/integrate more frequently to a shared repository, triggering automated unit-tests after each commit; as direct consequence, this helped to detect problems right after a bad commit and reduced the necessity of back-tracking to individuate the issue in changes happened far away in time. Detecting issues sooner means reducing the integration time, developing cohesive software faster and increasing the productivity.

In the past there has been a strict separation of roles among Development (Dev), Quality&Assurance (QA) and Operations (Ops) people due to the fact that each team was very specialized on its own tools and managed a different - and usually isolated - phase of the product lifecycle: passing from one phase to the next one required complex processes, manual steps and approvals, which make the end-to-end lifecycle too slow for the business expectations.

The continuous need of improving the lifecycle across the different phases and teams and, especially, the need of reducing the Time To Market, was what brought up the concept of "Continuous Delivery" (CD) which, as natural extension of CI, advocates the automation of the end-to-end tests, of the release processes and of the deployment steps. To make this happen, cutting down the strict separation of roles amongst the different teams was essential and led to the definition of the "DevOps" concept, which refers to the new role born from the need of close collaboration between Development and Operations staffs along all the software lifecycle. In the CD scenario, indeed, the several technical profiles (Dev, QA and Ops teams) involved in the product lifecycle need to collaborate and communicate faster, requiring a unique orchestrator for managing their different team-specific tools (i.e. Application Servers, Puppet/Chef, Sonar, IDE, etc.).

Intention of the paper is to describe how Jenkins, thanks to its flexibility and ecosystem, was one of the first candidates to cover this role and embrace this new concept, quickly transitioning from being a Continuous Integration tool only to a Continuous Delivery one: thanks to the development of new plugins, indeed, Jenkins was able to provide a solution for chaining jobs together, promoting their execution according to specific strategies and for maintaining traceability of the produced binaries.

### A. The Jenkins Platform

Jenkins is an Open Source (OSS) CI Platform, whose initial objective has been the automation of the build and test processes [1].

The build system is completely written in Java and is easily extensible thanks to its plugins architecture and to extension points left into its object model.

This makes of Jenkins a highly customizable and flexible tool, able to cover many possible scenarios and requirements thanks to the thousands of plugins developed by its huge Open Source Community.

## II. From Continuous Integration to Continuous Delivery with Jenkins

The traditional way of managing software development, known as Waterfall model, it's a logical and sequential model which has the big flaw that does not easily apply to the reality: one of the key concepts of this model, indeed, is the definition of frozen requirements in a preliminary and isolated state. This preliminary step would initiate the waterfalls of

978-1-4673-7070-7/15 $31.00 © 2015 IEEE
DOI 10.1109/RELENG.2015.19

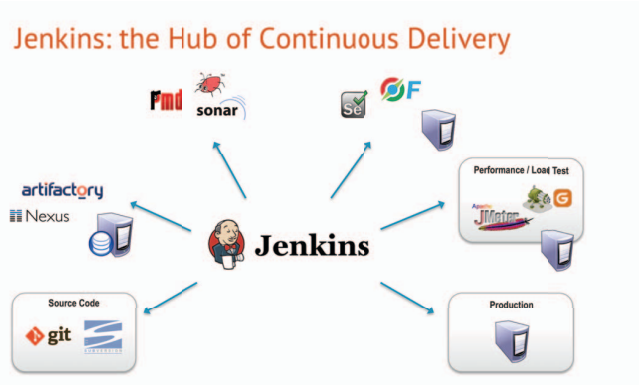


Fig. 1. Jenkins as Orchestrator Tool

subsequent steps, which could take weeks or months to terminate, involving design, implementation, verification, and deployment. During the time needed to go through the whole waterfall of steps, the requirements likely change, due to the volatility of the market, to new competitors, or simply to an improvement of the original project.

Realising that the traditional approach was failing too often, in 2001 a small group of people gathered to discuss the new values required by the software development. These values were collected in the "Agile Manifesto" [2] and encourage the rapid response to changes, as opposed to frozen requirements, and the collaboration across teams, as opposed to phase-isolation. Since then, companies have moved many steps towards an "Agile Transformation", trying to be always more dynamic, competitive, responsive to changes than ever before.

Among all the methodologies applied to concretize the Agile principles, Scrum and Extreme Programming (XP) are probably the most popular ones, claiming the need of small development cycles and cross functional teams. While the first is more a product development framework [3], the latter prescribes engineering practices to follow during the software development, which encourage constant code integration, test automation and code refactoring [4]. XP was the first methodology that fully embraced the concept of Continuous Integration (CI), advocating the need of committing and testing small changes [4] rather than big ones, reducing the risk of bugs and facilitating the debug at the same time.

CI literally started to change how companies looked at Build Management, Release Management, Deployment Automation, and Test Orchestration [5] and several tools emerged in order to fulfill these new expectations. Among these, Jenkins, born as small hobby project of K.Kawaguchi, suddenly became the most used CI tool, thanks to its plugins architecture and OSS nature, which motivated many developers around the world to contribute to its features. Thus, thanks to its elasticity and flexibility, Jenkins was able to continuously adapt to new needs, providing always more functionalities and integrations with external tools and technologies, completely fulfilling the ongoing CI revolution.

However, if CI started as automation of the development-phase only, pretty soon the revolution embraced the test-phase of the QA team and the deployment into various environment of the Ops team [6], involving the whole lifecycle of a product and introducing a new concept: Continuous Delivery (CD). Jenkins was again enhanced and extended, and moved from being a CI tool solely to a CD platform, allowing Dev, QA and Ops teams to work closely together using the same orchestrator (Fig.1) and embracing the key points of CD [7]:

- Collaboration across all the teams involved in the product-lifecycle
- Extensive automation of the delivery process

Implementing CD means being able to chain the different steps involved in the cross-team pipeline and automate their execution [8]: from the checkout of the code and the unit-tests, to the static code-analysis, to the performance tests, to the release of the binaries until the deployment into test/staging/production. In Jenkins this is accomplished thanks to plugins that make it possible to chain jobs together, promote the execution of jobs and allow human manual intervention. On top of that, graphical visualization plugins make it possible to monitor the execution of the pipeline, providing an aggregated view of the different steps, together with the results of their execution.

## III. Jenkins and Chef/Puppet Traceability

Puppet and Chef are the most commonly used IT Automation Tools used by DevOps to set up the infrastructure, speeding up the process of installing the required software, middleware and various dependencies. From a technical point of view, integrating Jenkins with Puppet/Chef is no more difficult than establishing a SSH connection: the Puppet Master and the Chef Server will ensure that the managed servers are configured properly and eventually deploy the artifact generated by Jenkins.

However, although Jenkins can keep track of all the files generated by a build - assigning them a MD5 checksum that will be stored in the Jenkins' fingerprint database [9] - as the artifact leaves the Jenkins environment, to be deployed by Puppet/Chef, the traceability will be lost:

- Where does the artifact come from?
- Which was the build that generated the artifact?

Traceability is a key aspect of software development allowing the verification of corporate compliance to processes and the backtracking to the root-cause of issues. Without this information, diagnosing a problem in the deployed artifact is quite complex and would require deep investigation from developers, who don't have any visibility of what is the build that generated the corrupted binaries: traceability is where the integration between Jenkins and Puppet/Chef brings value.

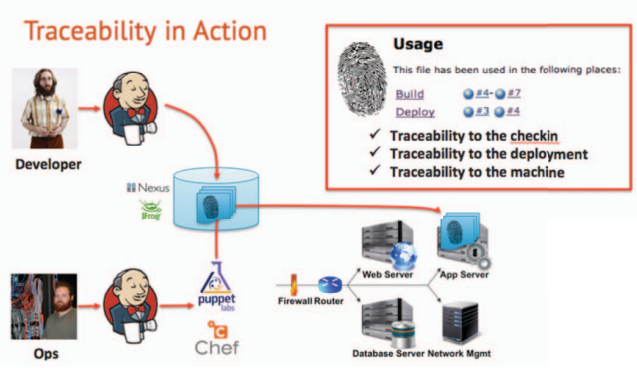


Fig. 2. Traceability in Action

The Puppet and Chef plugin - built on top of the Notification plugin - makes Jenkins capable of receiving deployment notifications coming from Puppet or Chef, containing the fingerprint of the deployed artifact. Jenkins, thus, will be able to identify the fingerprint in the report and look up for the MD5 checksum in its database: after successful mapping, the deployment information (when, where) will be added to the build [9].

Thanks to this mechanism, for each produced artifact, it is now possible to trace where this was deployed, when it was deployed and also the results of tests run against it, giving DevOps full visibility on the history of an artifact and giving them an easy way to debug a potential error (Fig.2).

This further extension of traceability takes Continuous Delivery with Jenkins and Puppet/Chef to a level higher than ever before.

As concrete example of the above, the Jenkins community itself (jenkins-ci.org) builds the Jenkins infrastructure with a continuous delivery pipeline orchestrated by Puppet and Jenkins, implementing a real and complete case-study for anyone willing to follow the flow and to promote Continuous Delivery as the key-point of an Agile Environment.

## IV. Jenkins and Workflow

With CD, Jenkins has become the most-used orchestrator for the different phases of the product lifecycle: from the checkout of the code and the unit-tests, to the static code-analysis, to the performance tests, to the release of the binaries until the deployment into test/staging/production. To make this possible, Jenkins needed to be enhanced in order to provide capabilities to support different orchestration strategies and visualization tools. This resulted in a wide set of plugins which allow the implementation of complex and not sequential pipelines, involving different jobs (one for each phase) and promotion strategies. Although these several plugins (i.e. Build Flow, Continuous Delivery Pipeline) work together and, generally, do not conflict with each other, looking for the right way to implement the pipeline, choosing the right plugin amongst many, and using a different plugin for each specific functionality, tend to be painful and time consuming and result in long chains of jobs, difficult to manage.

The new Workflow engine, was born in order to give a comprehensive answer to this problem by providing the possibility of implementing all this features inside a single plugin and of chaining different CD phases within a single workflow definition [10]. In order to accomplish this, the plugin reinvents the traditional way of scripting, by promoting a new Groovy Domain Specific Language (DSL) highly flexible and customizable. In the Jenkins Environment, this script-based plugin not only benefits of all the Jenkins' advantages and plugins but even enhances them, providing additional functionalities that contribute to automate a CD pipeline and to reduce the Time To Market. One of these is the possibility of inserting "checkpoints" along the execution of the groovy script, rendering the entire pipeline completely re-triable from anywhere [11].

When scripting a pipeline or when using the standard Jenkins Plugins to implement a flow, a single phase/script cannot be broken down to its individual steps (i.e. a single test for a test-suite). If something goes wrong during the execution, the entire phase needs to be re-executed. Of course this leads to enormous waste of time, considering that a build or a test-suite could take hours or even days to complete. This "restartability" feature of the Workflow plugin, then, gives the great advantage of being able to re-try the pipeline from the last successful checkpoint rather than re-executing the failed phase/pipeline from the beginning: in a scenario where the Time To Market really influences the success of a product, this functionality is key for reducing the delivery time.

Although Automation is what Jenkins aims to, CD pipelines can rarely be totally automated: QA or DevOps people may want to keep some sort of control over the whole workflow, being able to manually approve or not the final delivery. For this reason, the Workflow Plugin introduces another interesting functionality that makes the script "pausable" until human intervention: with this feature even the very last part of the product lifecycle can be scripted together with the rest of the pipeline, still ensuring its execution only after manual approval.

The above - together with the Workflow's capability of surviving machine's failure, providing a dynamical graphical visualization tool to control the execution of the flow (Fig.3), giving easy access to each step's logs, facilitating the debug - makes of the Workflow Plugin a revolutionary tool that not only speeds up a CD pipeline, but also improves the productivity of each single person involved in the product lifecycle.


Fig. 3. Workflow Plugin Stage View

## V. Challenges

Although the above solutions represents big steps for enhancing the implementation of CD pipelines in Jenkins, there are two main flaws which still need to be addressed and which acquire more importance when moving from CI to CD:

- How to do versioning of artifacts that must be continuously shippable?
- How to trace the environment in which the artifact was created?

### A. Versioning of Continuously Shippable Artifacts

With the advent of CI, the release cycle became much smaller than what it used to be in the Waterfall approach, causing the revision version to change at every commit, several times per day.

In order to avoid the overkill of the number of release versions and branches, the concept of snapshot was adopted, indicating a current development copy prior to a release: all the snapshot-versions prior to the same release would be identified by the same version identifier compromising the traceability of the artifact.

If in CI this problem could be minimized by the use of workarounds (i.e. Maven timestamps when releasing to a repository, Jenkins fingerprints for binaries, Git hash of the commit), with CD the trade-off to pay is somewhat more significant. With the concept of continuous delivery, indeed, all the artifacts produced must be shippable into production: an artifact without a unique identifier, and thus not traceable, must not be deployed and, as consequence, should not exist in continuous delivery/deployment pipelines.

Obviously, this creates a flaw in CD, where the artifact is generated and continuously released/shipped after every commit, nullifying the concept of intermediate versions and creating the need for a new mechanism, able to assign a unique revision version - human readable and sortable - to each produced binary: it is for this reason that build tools like Maven need to migrate from the concept of snapshot to a new versioning process that combines iniquity and human readability.

### B. Tracking of the Build Environment

As said before Jenkins can maintain the traceability of the artifacts inside its ecosystem (thanks to its fingerprint database) or outside (when using Puppet or Chef). However, while it is possible to trace when and where the artifact was generated or deployed, instead there is no way to capture the environment used to generate the binaries: knowing which Jenkins job generated the artifact can surely help the diagnosis in case of application problems, but does not help in case of problems specific to the build environment (i.e. which version of java was used to generate the binary? which compiler?).

Although there have been attempts to address these challenges and several solutions have been proposed, best practices on the subjects still need to be defined.

## VI. Conclusion

The ongoing transformation in the software industry, which requires fast response to changes and thus, across team communication and collaboration, brought Jenkins to extend its functionalities to impersonate the role of orchestrator for the all and different roles involved in the product lifecycle.

The need of managing the whole Continuous Delivery pipeline required the implementation of new features in Jenkins, with the objective of facilitating the creation of complex workflows, allowing the traceability, reducing the Time To Market and improving the productivity: the Notification Plugin and the Workflow Engine Plugin, described above, were two key responses, in Jenkins, to the new business expectations.

However, although these functionalities manage to position Jenkins as the hub of CD pipelines, further steps need to be taken in order to make Jenkins able to fully embrace the CD revolution: indeed, the lack of best practices for versioning continuously shippable artifacts and for tracing the build environment introduces a gap that has to be filled in order to further strengthen Jenkins Continuous Delivery capabilities.


[1] J.F.Smart, *Jenkins: the Definitive Guide*. United States: O'Really Media, 2011.

[2] K. Waters, "What is Agile? (10 Principles of Agile*)", All About Agile,* 2007. [Online]. Available:http://www.allaboutagile.com/what-is-agile-10-key-principles [Accessed Jan.10, 2015].

[3] J. Sutherland, *The art of doing twice the work in half the time*. New York: CROWN BUSINESS, 2014.

[4] K.Beck and M.Flower, *Planning extreme programming*. Addison Wesley, 2000

[5] K. Kawaguchi, "Jenkins Architecture", *wiki.jenkins-ci.org*, 2012. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Architecture[Accessed Jan.10, 2015].

[6] C.Le Clerc, "Pre-requisites for a successful enterprise Continuous Delivery implementation", *Betanews,* January 2015. [Online], Available: http://betanews.com/2015/01/13/pre-requisites-for-a-successful-enterprise-continuous-delivery-implementation/ [Accessed Jan.17, 2015]

[7] M. Fowler, "Continuous Delivery", *martinfowler.com*, 2013. [Online]. Available: http://martinfowler.com/bliki/ContinuousDelivery.html [Accessed Jan 06, 2015].

[8] J.Humble and D.Farley, *Continuous Delivery*. Crawfordsville, Indiana: Addison-Wesley, 2010

[9] CloudBees, "CloudBees Partners with Chef Software and Puppet Labs to provide Continuous Delivery Traceability with Jenkins", *cloudbees.com*, 2014. [Online]. Available: https://www.cloudbees.com/press/cloudbees-partners-chef-software-and-puppet-labs-provide-continuous-delivery-traceability[Accessed Jan.07, 2015].

[10] CloudBees, "Workflow Plugin", *cloudbees.com*, 2014. [Online]. Available: https://www.cloudbees.com/press/cloudbees-and-jenkins-community-extend-continuous-delivery-leadership-jenkins-workflow[Accessed Jan.07, 2015].

[11] CloudBees, *Jenkins Enterprise by CloudBees 14.11 User Guide*. Boston, MA: Cloudbees, 2011. [E-book] Available: http://jenkins-enterprise.cloudbees.com/docs/user-guide-docs/index.html