CloudBees®
The Enterprise Jenkins Company

Prepare for the Next Sea Change in IT

Table of Contents

Introduction

Historically we have seen waves of innovation hit the Information Technology industry. Typically, these have happened separately in the areas of infrastructure (mainframe to distributed to virtual), application architecture (monolithic to client-server to n-tier web) and process/methodology (ITIL, etc.). But if you look around, you will see that right now we are in the midst of what is not just another wave in one of these areas, but a complete sea change which encompasses all three areas at once. We are watching the infrastructure space be completely disrupted by lightweight container technology (best represented by Docker). We are seeing application architectures moving to a distributed microservices model to allow value-adding business logic to be quickly added or changed in order to better serve the end-user. And we are seeing the bold new concepts such as “collaboration,” “failing is okay, just fail fast,” “over communicate with feedback,” take over the IT organization in the hot methodology trends of DevOps and continuous delivery (CD). The really interesting part is that these three waves are feeding on each other and amplifying the ultimate effect on IT: the ability to provide more value faster to the business/consumer/user.

In this paper, we will discuss how two of these areas (Docker and CD) are coming together to accelerate the innovation that can happen in microservices based applications. This sea change in I.T. tooling and process has the potential to have a huge impact on all of us. The combination of continuous delivery being executed on applications running in Docker containers will allow us to see, in enterprise I.T., the kind of exponential growth of innovation that we have seen in consumer and mobile applications over the past five years.

The Innovation Catalysts - Docker, Jenkins and Continuous Delivery

In just two years, Docker has grown from nothing to more than 100,000 "Dockerized" applications and close to 1,000 contributors. It has heavily rocked the IT boat and pushed the application architecture wave around microservices-based designs to a new reality. Lightweight container technologies - particularly Docker - are rapidly changing the way we build, deliver and update software. Docker brings a new level of simplicity to defining and creating applications or services by encapsulating them in containers. That simplicity lets **developers and operations personnel use Docker containers as a common currency**, eliminating a source of friction between development and operations. While Docker is an amazing success story, the way that people use Docker in their development and delivery processes is still very much a work in progress. Companies using Docker are discovering how it fits best into their environments and how to use Docker and Jenkins together most effectively across their software delivery pipelines.

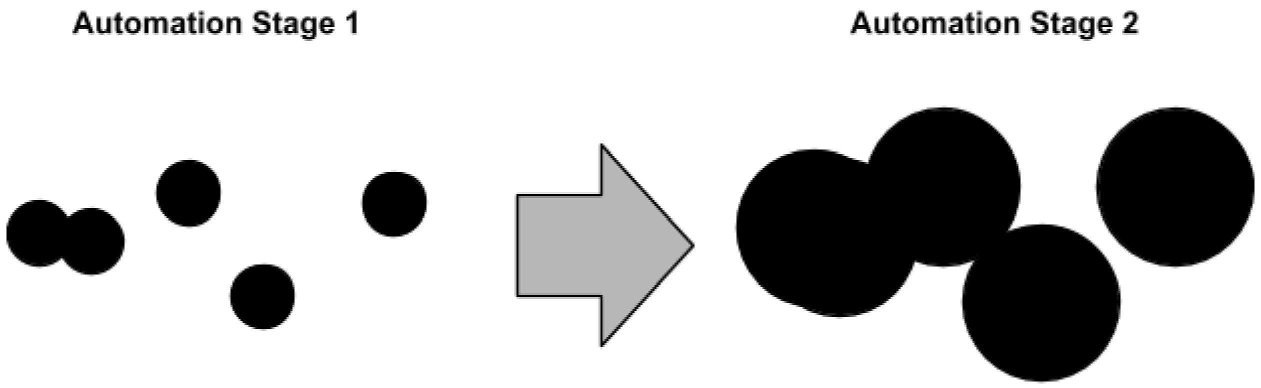
Meanwhile, since its start a decade ago Jenkins has defined what continuous integration (CI) is all about, and has now seen tremendous acceleration in adoption as the market has moved towards DevOps and continuous delivery (CD), the next wave in I.T. operations. Jenkins is currently known to operate on at least 100,000 clusters, 300,000 servers and offers more than 1,000 plugins providing comprehensive integration with third party systems.

As the company helping to drive Jenkins adoption across enterprises, CloudBees has been in the middle of the excitement and discovery process around how to best leverage containers. This experience has informed us of what Jenkins can do better to help people take advantage of Docker, and how to transform the excitement and hard-won discoveries into concrete features in Jenkins that make the end-to-end software delivery process faster, predictable, manageable and drama-free. That's what developers and operations people want.

In this paper, we'll first take a look at the state of the Docker + Jenkins world today. What does the combination bring to the table and how are people using them together? Then we'll take a look at what CloudBees and the Jenkins community have done to make Jenkins the obvious choice to use with Docker to provide CI and CD. Finally, we'll discuss what remains to be done to complete the vision of a world in which Jenkins and Docker, together, bring software development and delivery processes to an entirely new level.

Getting Some Perspective

One of the most interesting things about the Docker phenomenon is how it helps facilitate the way development and operations teams work together, by, among other things, changing the abstraction level from an application binary to a container level. Jenkins has a similar impact on development and operations teams. Jenkins was originally created as a CI server - a server that builds, runs tests and reports results as source code or configuration changes. Today, however, it is very commonly used to orchestrate work across an organization, between teams. The introduction of the Workflow feature in Jenkins has made it simple to define and share the processes involved in a full CD pipeline. Thus, like Docker, Jenkins has raised the abstraction level used by Jenkins users from a build job to a workflow, and it allows those workflows to interact with surrounding systems using a domain-specific language, or DSL. At its heart, a DSL simply allows you to interact with surrounding systems using the familiar nouns and verbs those surrounding systems understand.



Individual pockets are automated, but interactions between them remain manual or messy. Mostly interesting to individual contributors.

Automation expands and starts to interact. The whole thing starts to look like one big automation. You start thinking about relations between those automated units of works. This starts to become very interesting to managers.

From a Jenkins Perspective...

One way to view Docker from a Jenkins perspective is simply as a different, improved packaging approach, in much the same way that applications have been packaged in RPMs or other mechanisms. From that standpoint, automating the package creation, update and maintenance process - particularly for complex projects - is exactly the kind of problem Jenkins was made to address.

Yet, Docker is a lot more than a packaging approach, because it exposes a toolset and API around the container that's being constantly extended by its ecosystem. Because Docker encapsulates both the application and the application's environment or infrastructure configuration, it provides a key building block for two essentials of a continuous delivery pipeline:

- First, Docker makes it easier to test exactly what you deploy. Developers deliver Docker containers or elements that are consumed by other containers and operations deploys those containers. The opportunity to screw up in a handoff or reassembly process is reduced or eliminated. Docker containers encourage a central tenet of continuous delivery - **reuse the same binaries** at each step of the pipeline to ensure no errors are introduced in the build process itself.
- Second, Docker containers **provide the basis for immutable infrastructure**. Applications can be added, removed, cloned and/or its constituencies can change, without leaving any residues behind. Whatever mess a failed deployment can cause is all constrained within a container. Deleting and adding become so much easier that you stop thinking about how to update the state of a running app. In addition, when infrastructure can be changed (and it must change) independently of the applications the infrastructure hosts - a very traditional line between development and operations responsibilities - then there are inevitable problems. Again, Docker's container-level abstractions provide an opportunity to reduce or eliminate the exposure. This gets particularly important as enterprises are moving from traditional virtualization to private or public cloud infrastructure.None of these benefits brought about by the use of Docker appear magically. Your software and infrastructure still need to be created, integrated with other software, configured, updated and managed throughout its lifetime. Docker gives you improved tools to do that, especially when combined with the power of Jenkins to automate these processes.

From a Docker Perspective...

From a Docker perspective, at a very basic level, Jenkins is just another application that should be running in a Docker container. Jenkins itself needs to be updated and often run in a specific environment to be able to test properly. For example, integration tests may require access to backend systems, necessitating creation of a Docker image environment that has strict access controls and strong approval processes for updating. Environment and configuration changes should always result in a Jenkins build to produce new images.

But Jenkins is much more than that, it is the application that passes validated containers between groups in an organization. Jenkins also helps build higher level testing constructs - for example, integration tests may require access to backend systems, necessitating creation of a set of Docker Compose images that Jenkins can bring up, run tests against and bring down. Jenkins can ultimately create gates that make sure that only containers that have been properly pre-tested and pre-approved, make it to the next step. In a world where Docker containers are so easily created and multiplied, the role of such a validation agent can't be minimized.

Taken Together: (Jenkins + Docker) = CD

Today, many Jenkins users take advantage of the combination of Docker and Jenkins to improve their CI and CD processes. They can do this because of Jenkins extensibility and the flexibility in ways in which Docker can encapsulate deliveries. As you might expect**, two of the leading proponents of using Docker and Jenkins together are: the Docker team and the Jenkins team!** The Docker team uses Jenkins and Docker to test Docker. While the Jenkins team has used Jenkins to build Jenkins for a very long time, they also now use Docker as an integral part of the test and delivery process of the jenkins-ci.org web site, in combination with Puppet. Many other people have shared their experience in blogs and brief articles. As this experience has grown, CloudBees and the Jenkins community have identified some areas that would greatly improve the automation and management process when using Docker and Jenkins together. The goal has been to minimize the handcrafting and guesswork involved in figuring out how to make the best use of two tools, in combination. The new capabilities have been released as part of open source Jenkins, together with key integrations into the CloudBees Jenkins Platform. They include:

- Ability for Jenkins to understand and use Docker-based executors, providing improved isolation and utilization
- Easy interaction with Docker image repositories, including Docker Hub, making it possible to store new images built by Jenkins, as well as load images so they can be used as part of a Jenkins CI job
- Rich Jenkins workflow integration with Docker, making it possible to orchestrate builds, tests, deployments of any apps - including but not limited to Docker images, by using Docker as environments
- Extension of Jenkins native fingerprinting capability to enhance tracking of Docker images across development and delivery processes, making it possible to track complete delivery processes, from code to production

The design consideration behind the Jenkins/Docker integration was to consider Docker images as a first-class citizen of any Jenkins operation, from CI activities - where Jenkins has the ability to operate on sanitized and isolated Docker images - to CD activities, where much of the work implies the delicate orchestration of multiple Docker containers along with third-party integrations such as unit testing, load testing, UX testing, etc. The tight integration of Jenkins and Docker means neither feels like an add-on to the other. Instead, they form a cohesive approach to CI and CD that solves problems for both development and operations.

Next Gen CI/CD: Use Cases, Best Practices and Learning

In this section, we will drill down into key use cases related to Jenkins and Docker, and offer some of the best practices for them.

Because Jenkins and Docker are two very flexible technologies, use cases can quickly become confusing: while each use case is based on a specific articulation of Jenkins and Docker, they are doing so in a very different fashion, in order to reach different objectives. For that reason, we have split the use cases into three sections:

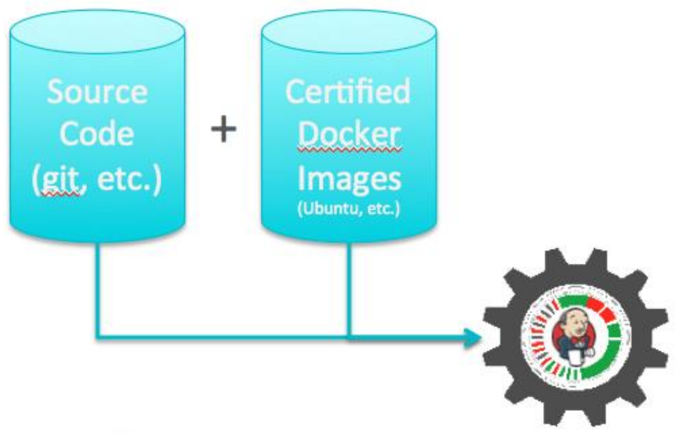
1. **The first section** will focus on building Docker images. This is a trivial use case, but a very important one as it serves as the foundation of anything related to the usage of Docker (on Jenkins or not). Essentially, in a Docker world, anything starts with a container... so they must be built!
2. **The second section** will cover CI use cases and how Docker can help improve CI independently of whether your application will ultimately be deployed as a Docker image or not. In this section, Docker is mostly defered as an underlying (and transparent) layer that enables Jenkins to deliver faster, safer, more secure and more customizable CI.
3. **The last section** will cover typical CD use cases and how you can use Jenkins to orchestrate end-to-end pipelines based on Docker images/applications. Put simply, ***this is the future of software delivery***.

In the Beginning was the Container...

Anything in Docker, will obviously start with the creation of a Docker image. This is the new Lego block of IT. As such, it is essential for the engine that will be responsible for building them to be smart about it, highly customizable, secure and stable. The last point is very important and often dismissed: if everything in your IT is dependent on the creation of Docker images, your Docker builder becomes as critical as your production environment.

Need to fix a security bug in production? If you can't properly rebuild an image as part of your company process, you can't fix your production environment, plain and simple.

Consequently, Jenkins acts as the ideal tool for that. Jenkins has been extensively used in all kind of environments for more than a decade and has proved to be extremely robust and stable. It also features advanced security features (such as CloudBees' Role-based Access Control and clustering features).



Triggers:
- New application code (i.e. feature, bug, etc.)
- Updated certified stack (security fix in Linux, etc.)

The overall philosophy of building containers in Jenkins is based on the idea of tracking the dependencies between the pieces that make up a final application (which could be composed of multiple containers). Whenever part of the source code or one of the source golden images are used to run the image changes, both development and operations have the option to automatically rebuild a new image.

Furthermore, Jenkins is tightly integrated with all Docker repositories on the market, making it possible not only to securely manage credentials used to store generated images, but also to track the entire process, from the initial trigger that initiated the generation of a new image, to the actual location where this image is being used. Full traceability, at all times.

Next Gen CI - Jenkins on Docker

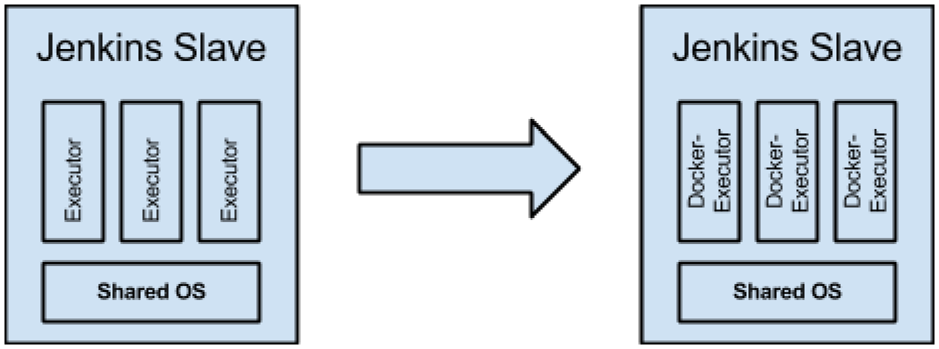
One of the most immediate benefits you can get out of using Jenkins and Docker, is to improve the way you run Jenkins and its build servers. Here, we are talking about how your existing application development - that has nothing to do with Docker (such as mobile apps, desktop apps, etc.) - can still benefit from Docker, which may be more important for many companies.

Companies doing CI maintain a cluster of Jenkins "slave machines," on which a number of virtual slots (aka "executors") are defined, and can be used by a Jenkins master for performing build/test jobs. The number of overall executors in a Jenkins cluster defines how many concurrent jobs will be able to execute at any point in time.

The typical problems with such a setup is that build processes will essentially be sharing resources concurrently. This can create different categories of issues:

- **Transient**: Singleton resources can be requested at the same time by concurrent jobs (network resources, files, etc.). This would typically cause at least one of the concurrent jobs to fail intermittently. (The same job executing at a different time or on a different machine could properly execute.)
- **Persistent**: A build job could make changes to the hosting environment. These changes can disrupt a future execution of that build or of another build.

Both categories of problems incur important costs for the DevOps team: issues have to be debugged and validated, environment have to be regularly debugged and "sanitized." But more importantly, such errors, although not related to actual bugs in the code being tested, lead to teams not fully trusting the CI results. Whenever a job fails, the typical reaction will be "probably not my problem, surely an environment issue, I'll wait to see if it persists." The problem with that attitude is that the more one waits, the more other code changes happen, which are all potentially responsible for truly breaking a build, hence diluting the responsibility of fixing the problem.



To remedy that situation, Jenkins' in-depth Docker integration makes it possible to run your build in its own isolated Docker container, rather than on a simple executor on a shared OS. Even if your application has nothing to do with Docker or will not be delivered as a Docker image itself, it will happily run inside a container. Your testing will behave as if you have a whole computer to yourself, oblivious to the fact that it actually is confined in a jail cell. The Docker-image essentially becomes an ephemeral executor entity that gets efficiently created and discarded 100 times a day.

Using Docker for your CI fixes the above issues - both transient and persistent - as each job executes in a fully virtualized environment that's not visible or accessible by any other concurrent build and each executor gets "thrown away" at the end of each build (or reused for a later build if that's what you want).

Furthermore, some companies are looking to completely isolate teams for confidentiality or IP reasons (i.e., source code/data/binaries from Team A shouldn't be visible to Team B). In the past, the only way to obtain that behaviour was to completely segregate environments (masters and slaves), and possibly implement additional security measures (firewalls, etc.) By basing your CI on Docker, builds executing on slaves are fully isolated and do not pose any risks. Furthermore, the usage of features such as CloudBees' Role-based Access Control makes it possible to share masters as well, by setting proper security rules in place.

Last but not least, IT Ops no longer needs to be in charge of managing build environments and keeping them "clean," a tedious but critical task in a well-run CI/CD environment. Developers and DevOps can build and maintain their customized images while IT Ops provides generic vanilla environments.

For anybody doing CI today, moving to Docker images represents low-hanging fruit that comes with very little disruption, but lots of advantages.

Next Gen CD - Orchestrating Docker

Entering IT Automation with Continuous Delivery

Virtually every IT organization today is driving toward increased automation and exploring ways to achieve CD. There is an incredible array of services, tools and technologies available to these organizations as they strive to meet their goals for faster delivery of higher quality software. Both Jenkins and Docker have been embraced by organizations because they help in specific ways while providing a real foundation to build further. Jenkins, in particular, excels in its ability to integrate with existing systems, and to orchestrate work broadly across multiple organizations. The introduction of native Workflow capabilities within Jenkins was the key to making this happen.

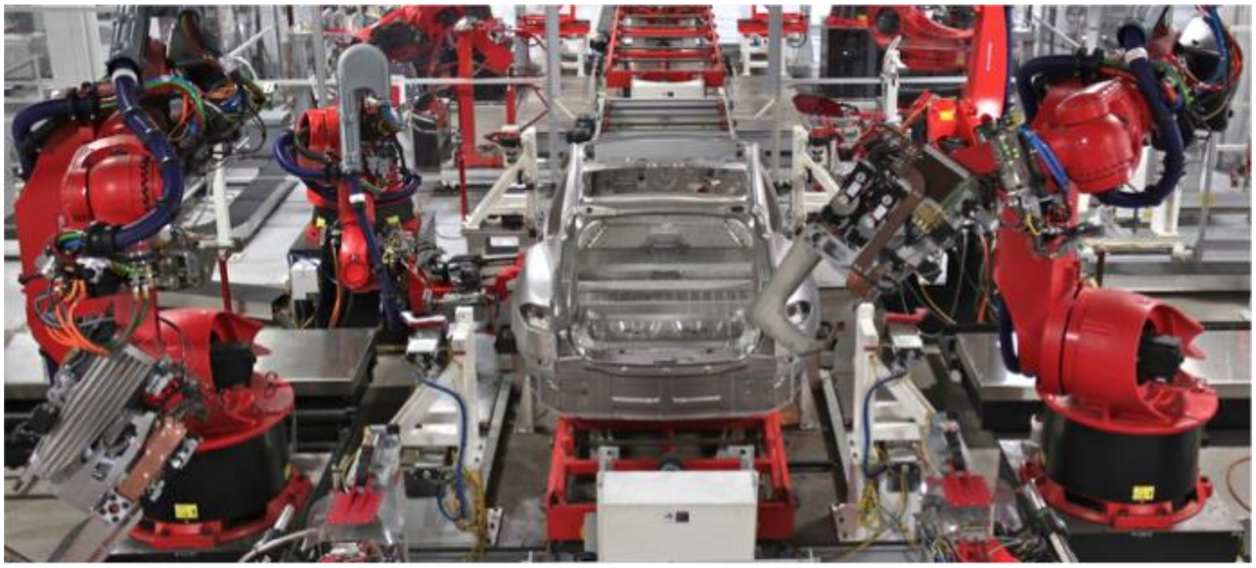
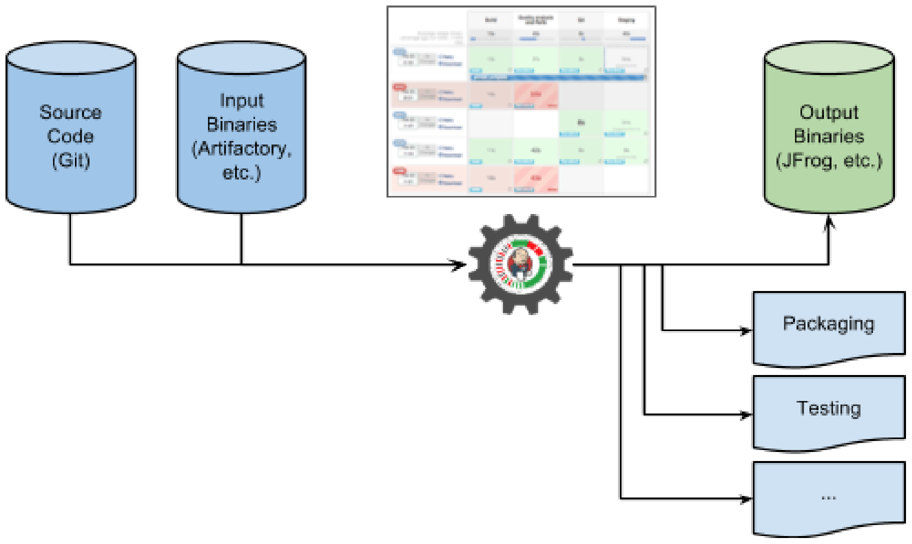


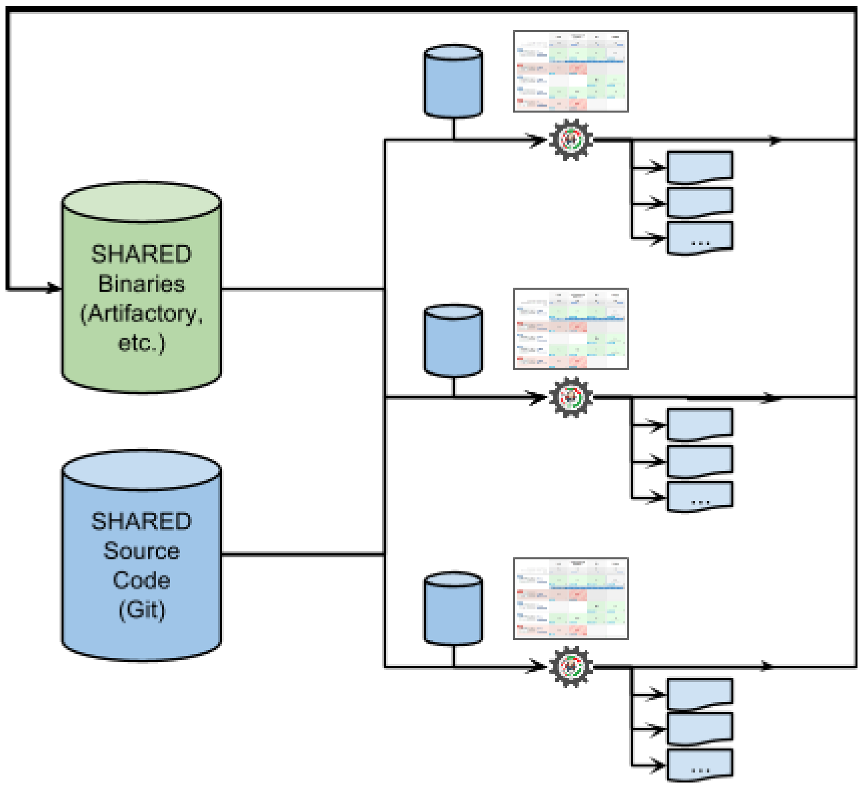
Photo courtesy of Steve Jurvetson via Flickr

One of the key requirements for end-to-end IT automation - real industrialization of IT - is to move work reliably across a complex organization, being able to integrate elements from many contributors and using a vast array of tools. The same pattern of requirements exists for active open source projects like Jenkins and Docker, but the constraints and cultures within enterprises are often the controlling factors in implementation.

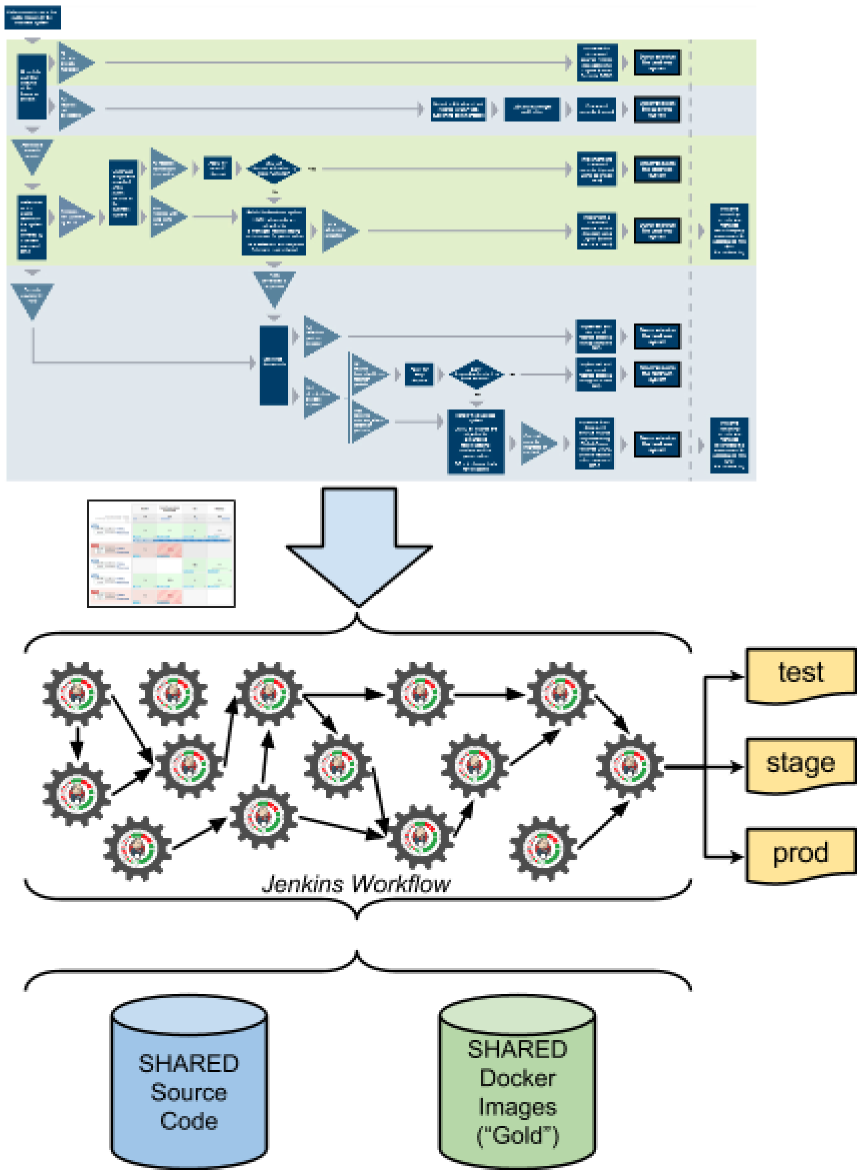
The practices and features for using Docker and Jenkins together, which are outlined in this paper, are the foundation for much broader application as CD practices mature over the coming years. We talked earlier about how automation is evolving. To dig a little deeper, each dot was typically born in a larger organization as a silo of automation.


Source Code (Git)
Input Binaries (Artifactory, etc.)
Output Binaries (JFrog, etc.)
Packaging
Testing
...

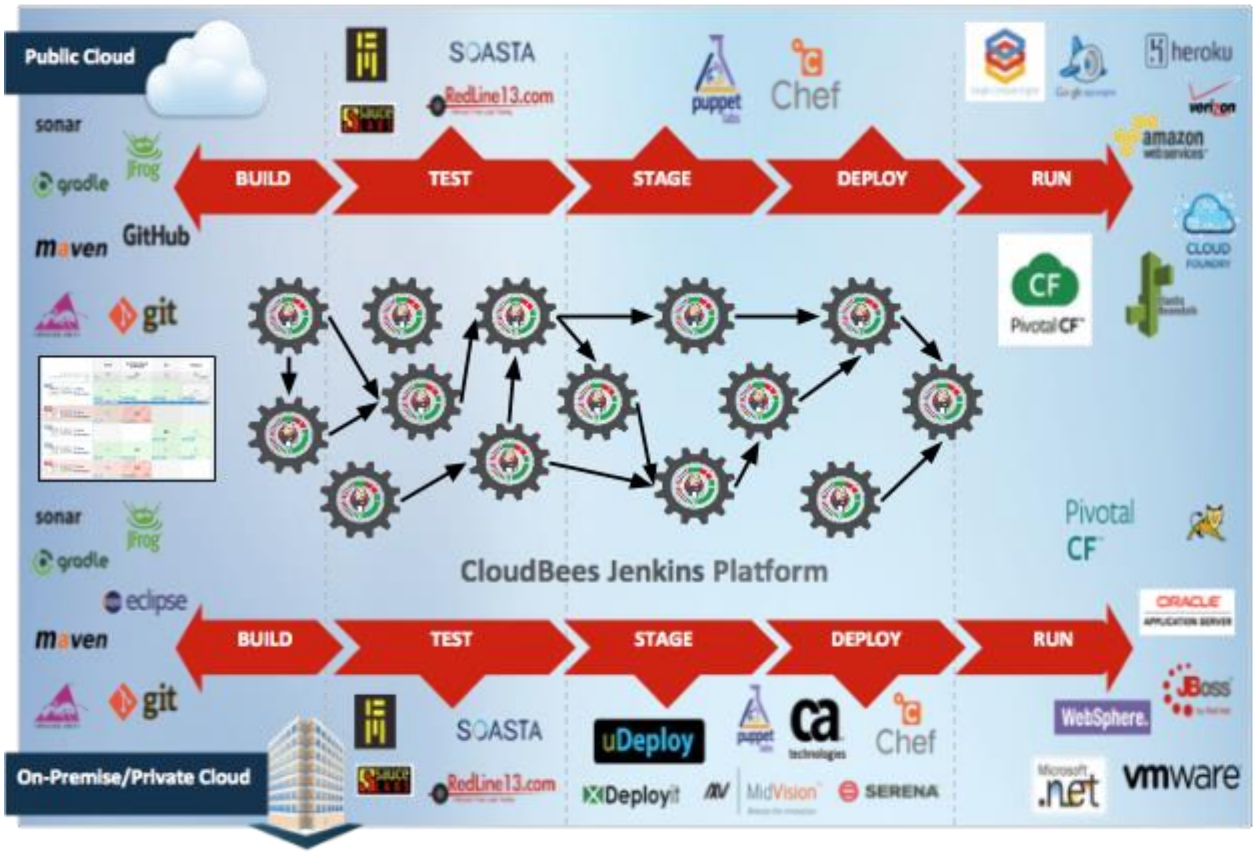
Then, as automation processes grow across an organization, or as the complexity of a delivery increases or dependencies multiply, shared repositories were put in place to provide cross-organizational access and central management.



Jenkins initially played a central role in automating the flow of artifacts between these islands of automation, by providing simple triggers to cause actions when changes took place, along with richer constructs for defining flows. However, what developers and operations users needed was a simpler way to define, version and reuse workflows themselves, not just triggers. In addition, rather than just capturing binaries of libraries and application artifacts, developers and operations users needed simpler ways to capture application containers in environments - Docker's forte.



The combination of Jenkins with Workflow, the new Docker plugins and Docker, provides a new level of abstraction when constructing and operationalizing CD pipelines. These tools help developers and operations personnel speak the same language, share the same processes and meet the same goals as a team. Furthermore, as a pipeline executes, not only can it be spread over the cluster, but it will leverage a key feature of Jenkins: its widespread integrability with hundreds of tools on the market. This is especially important in an enterprise environment where many different tools have typically been accumulated over time and have to be integrated to the overall process



From Images to Gold Images

It is important to fully capture the benefits that Docker brings when added to your continuous delivery practices. If one focuses solely on Docker as a packaging mechanism, one might think the impact will merely be about the "last mile" of how your application gets pushed to production. Yet, since Docker fundamentally improves the way you package, maintain, store, test, stage and deploy your applications, Jenkins makes it possible to capture those improvements all along the application lifecycle and provides key advantages all along your CD pipeline.

In a traditional environment, application source code is stored in a repository and, as Jenkins executes its CD pipeline, it interacts with several tools (Chef, Puppet, Serena) within target runtime environments for testing, initially, followed by staging and production. But the actual "baking" of the application with its environment (operating system, application server, load balancer) is a concern that usually happens relatively late in the process (which also means the environment used along the pipeline stages might vary quite a bit).

In the new CD-with-Docker world, the target runtime environment isn't an afterthought that's left to the IT Ops team at a late stage of the CD pipeline. Instead, the runtime environment is closely associated to the application source code from the start. At the beginning of any CD pipeline, you'll find a set of code repositories as well as a set of binary repositories containing a number of "IT Ops approved" Docker images for the various environment required (operating system, application servers, databases, load-balancers, etc.).

Very early on in the pipeline process, Jenkins will be "baking" the application with its target Docker environment and produce a complete executable application as another Docker image. This will be the runtime version of your application. This runtime will be stored in a company repository that contains the archive of your Docker-ized target runtime applications. You can see your overall CD process as having several code and binary repositories as input and while the pipeline executes, several new Docker images - the applications - to be generated. Those application images might end up being wrapped together as a set of microservices (for example, Kubernetes deployment) or as a traditional monolithic application in one container. Once an application image has been successfully built, it can be stored in the company repository as a "golden image" and serve as a potential candidate for a future deployment to production (remember: CD doesn't automatically push to production - that would be continuous deployment - but makes sure your application is in a release-ready stage at all times).

From a process standpoint, this brings a lot of advantages:

First, since your application gets packaged in its final form very early on, it will travel through all testing and staging steps in that form. This highly reduces the risk of having problems in production not show up in previous steps because of a change in the runtime environment between those two stages.

Second, updating the environment itself is much more formalized, yet simplified. In a typical CD process, the main trigger of a new CD pipeline will be a change in the source code of the application. This will initiate a web of tests, integrations, approvals and so on, aka the CD pipeline. However, in case one wants to update the environment itself (such as patching the operating system), this would happen separately in parallel to the application build process and it is only once the CD pipeline is executed again that the updated bits will be picked up. As we have seen, this could happen late in the pipeline execution, hence not going through all tests with that new environments. With Docker, not only will a code change initiate a CD pipeline execution, but uploading a new Docker base image (such as an operating system) will also trigger the execution of any CD pipeline that is a consumer of this image. And since Docker images can depend on each other, patching an operating system might result in the automatic update of database and application server images, which will in turn initiate the execution of any pipeline that consume those database/application server images! A CD pipeline is no longer just for developers and their source code. Developers and IT Ops now share the exact same pipeline for all of their changes. This has the potential of hugely improving the safety and security of an IT organization: when facing a critical and widely deployed security issue (such as the Heartbleed bug), IT Ops teams often struggle in making sure that absolutely ALL machines in production have been patched. How to make sure that no server gets forgotten? With a Docker-based CD pipeline, any environment dependency is explicitly/declaratively stated as part of the CD pipeline.

In this world where countless Docker images are going through various phases of the CD pipeline and getting copied from one system to another, it becomes very hard to keep track of what processes each of those images went through. Docker images get transformed and change their names all the time as they go through the pipeline. This is where the "traceability" features in Jenkins shine. Jenkins unambiguously keeps track of exactly what images are transformed to what, who made what changes in them, what tests were run, where they were used, who performed any manual approvals and so on. And this all happens regardless of whether they are stored in S3, Docker Hub or a file in NFS. In addition of being a useful trigger condition for automation (i.e. if an image passes these tests, start that process), it is also a treasure trove for a forensic analysis, months or even years after the application has been pushed into production. This information removes a lot of guesswork from troubleshooting and defect analysis, as well as helps you to track the propagation of important changes, such as vulnerability fixes. This can prove very important, for example in the case of a security breach when you need to precisely identify when a specific breach was released out into the wild.

What's Next

As experience with Docker-based applications grow, the industry will quickly evolve to a place where a single container delivers an application or service within a microservices-based architecture. In that microservices world, fleet management tools like Docker Compose, Mesos and Kubernetes will use docker containers as building blocks to deliver complex applications. As they evolve to this sophistication, the need to build, test and ship a set of containers will become acute. Jenkins Workflow Docker DSL is already designed for such a use case. The Jenkins community is working towards supporting Kubernetes in the near term.

Other use cases remain to be discovered: one must learn to walk before running. The heartening thing is that the Jenkins community is on the leading edge of these changes and responds quickly to changes. It is almost as if Jenkins is the one constant thing in the storm of changes that happen around it.

CloudBees, Inc.
2001 Gateway Place, Suite 670W • San Jose, CA 95110 • USA
www.cloudbees.com • info@cloudbees.com