

# PRAGMATIC CONTINUOUS DELIVERY

WITH JENKINS, NEXUS AND LIVEREBEL

BY JEVGENI KABANOV, CEO & FOUNDER OF ZEROTURNAROUND

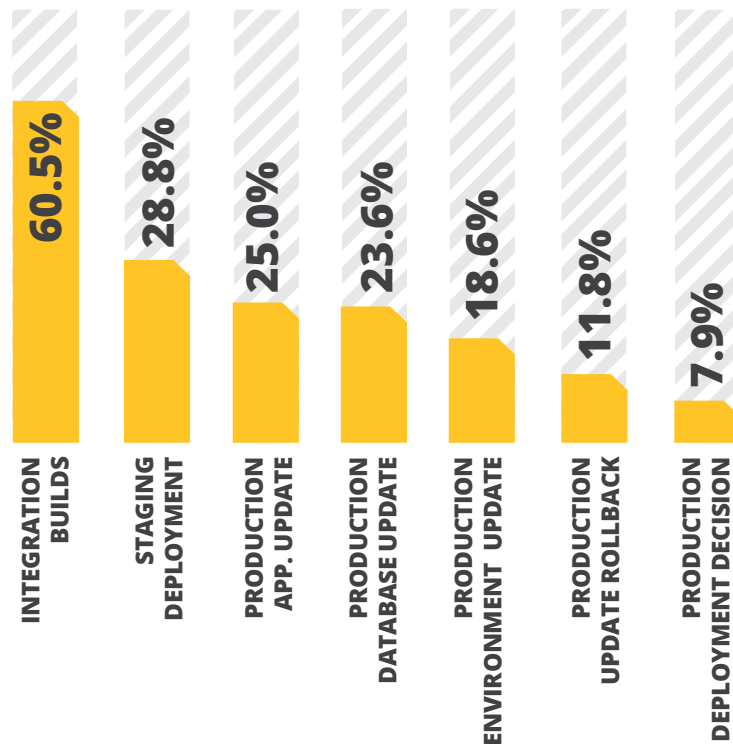
Get started today  
with these tools!

## Manual or Automatic?

In a recent study on Production Deployment processes, results showed that the vast majority of organizations' deployment pipelines still heavily rely on manual operations, which tend to be prone to human error, inconsistencies and generally move things out the door slower.

Out of 1100+ responses ranging from large to small organizations, here is the level of process automation for different roll-out-to-production activities:

### HOW AUTOMATED IS YOUR DEPLOYMENT PIPELINE?



Is it hard to believe that ~75% and more of organizations surveyed still rely on unpredictable, error-prone manual processes for making production application, production database and production environment updates.

There are good reasons why the industry is where it is, and we can start by reviewing those reasons.

# Why is deploying software so hard?

In recent years, we have conducted dozens of interviews and ran two surveys trying to figure out why deploying software to production, something we've been doing for years over and over again, is so darn hard to get fast, automated and painless. We learned that two key problems are responsible for defining the complexity:

## **Downtime:**

Whenever you need to change user sessions, databases & other data stores, or third-party dependencies (e.g. APIs) and cannot take the application offline, it becomes a complicated dance that is *comparable to changing out the members of an orchestra without missing a single beat*.

## **Failure:**

Software is almost unique in this world by having both a high probability of failure (more or less any kind of change can cause it) and a high impact resulting from failure (it is really hard to isolate the repercussions). This means that a lot of software organizations are *built to prevent failure and as a side effect will resist making any changes to their existing infrastructure*.

Despite these and many other technological and organizational issues, most firms want to roll out their software as quickly as possible to their users and customer base. The goal of this paper is to review how to simultaneously increase the velocity of delivering production-ready software, while noticeably decreasing the risk and impact of failure.

# Introducing: Continuous Delivery

Before jumping in to describing the difference in approach it's worthwhile to understand the philosophy of the Continuous Delivery (further "CD") methodology:

## Automate

As often as humans excel at the creative, they appear to suck at doing anything repetitive. In production updates, they are a source of error that should be eliminated as much as possible. The deployment path should be as predictable and repeatable as possible.

## Record

No action can happen without some kind of a trail. In fact, not only should the trail exist, it should be easy to access and navigate through, as well as redundant, like with double-entry bookkeeping.

## Test

Every step of the way, including production, should be probed for failures. Monitoring in both development and production is just a style of testing and should be planned accordingly. Combine A/B testing with gradual rollouts and we have a great way to isolate failure.

## Recover

The moment you know or suspect that something is wrong, begin a predefined recovery process. Recovery is a part of every change and without it the rest of the steps are effectively neutralized.

**This philosophy is incorporated in the central concept of Continuous Delivery (CD):** the automated pipeline leading from development all the way to production.

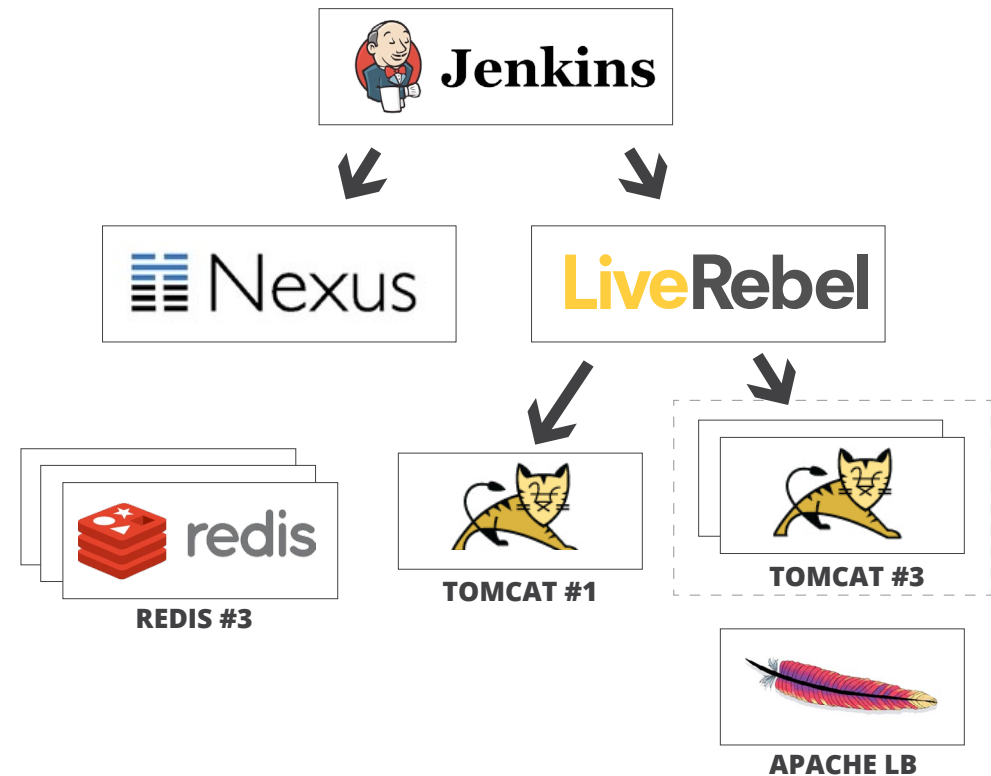
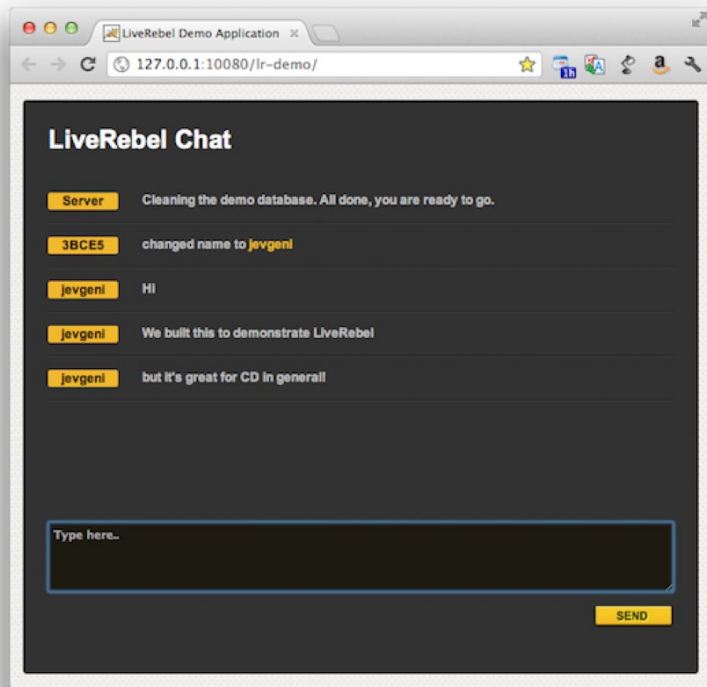


The idea of this pipeline is that every change that a developer makes and commits to the code repository flows through the pipeline creating a record of success along every step of the way. Each stage of the pipeline consists of automated or manual checks that try to enact and catch failure in the change you are trying to commit as early as possible. Once the last check is cleared, the change is release-ready and can be deployed to production at any time.

At this point we would like to introduce an example application & environment and discuss the methodology and its pros/cons.

## Example: Live Chat Application

To demonstrate the do's and don'ts of the CD pipeline we've built a persistent Live Chat application.



This application is deployed in 2 different environments:

### Testing

- 1x Apache Tomcat
- 1x Redis

### Production

- 2x Apache Tomcat
- 1x Apache /w mod\_proxy
- 1x Redis

The production environment is under load using traffic simulated by JMeter (<http://jmeter.apache.org/>), and the other environments are used for testing purposes.

## VCS & Artifact Repository

The starting point of the CD pipeline is the Version Control System. It enables developers to collaborate on the code and manages the changes/patches/commits/changesets (hereinafter called “patches”) that start the pipeline.

Using a Distributed VCS (DVCS) like Git or Mercurial is suggested to allow easier collaboration, including effortless branching and merging. However, there is no requirement to use DVCS and CD can be used with Subversion or even CVS. In the end, every patch figuratively enters the pipeline and results in a release candidate if it passes it successfully.

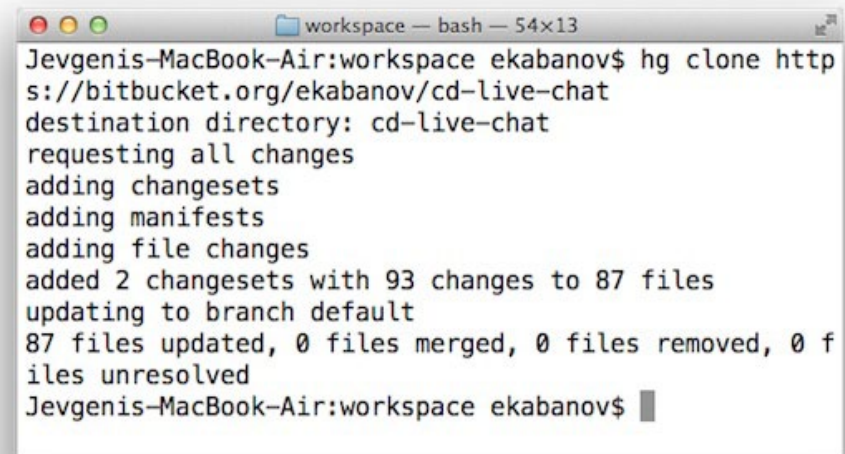
The Artifact Repository - in this case Nexus - serves as the synchronization point in the workflow to allow long-term storage and tracking as well as manual or long-running processes. Every successful completion of the CD stage should deploy an artifact to the corresponding repository.

Using the artifact repository in such a manner lets us easily incorporate manual stages (e.g. manual QA when finished deploys an artifact) and leaves a trail of artifacts that have passed various stages of the CD pipeline. It is also an additional guard against errors in the flow, as the next stage cannot start unless the artifact is deployed to the previous repository.

For our example application we use a Mercurial repository and you can clone it from:

<https://bitbucket.org/ekabanov/cd-live-chat>

For the purposes of this example, we will clone the repository and do local changes that will initiate the CD process.

A screenshot of a terminal window titled "workspace — bash — 54x13". The terminal shows the command "hg clone http s://bitbucket.org/ekabanov/cd-live-chat" being executed. The output of the command is displayed line by line: "destination directory: cd-live-chat", "requesting all changes", "adding changesets", "adding manifests", "adding file changes", "added 2 changesets with 93 changes to 87 files", "updating to branch default", and "87 files updated, 0 files merged, 0 files removed, 0 files unresolved". The prompt "Jevgenis-MacBook-Air:workspace ekabanov\$" is visible at the bottom.

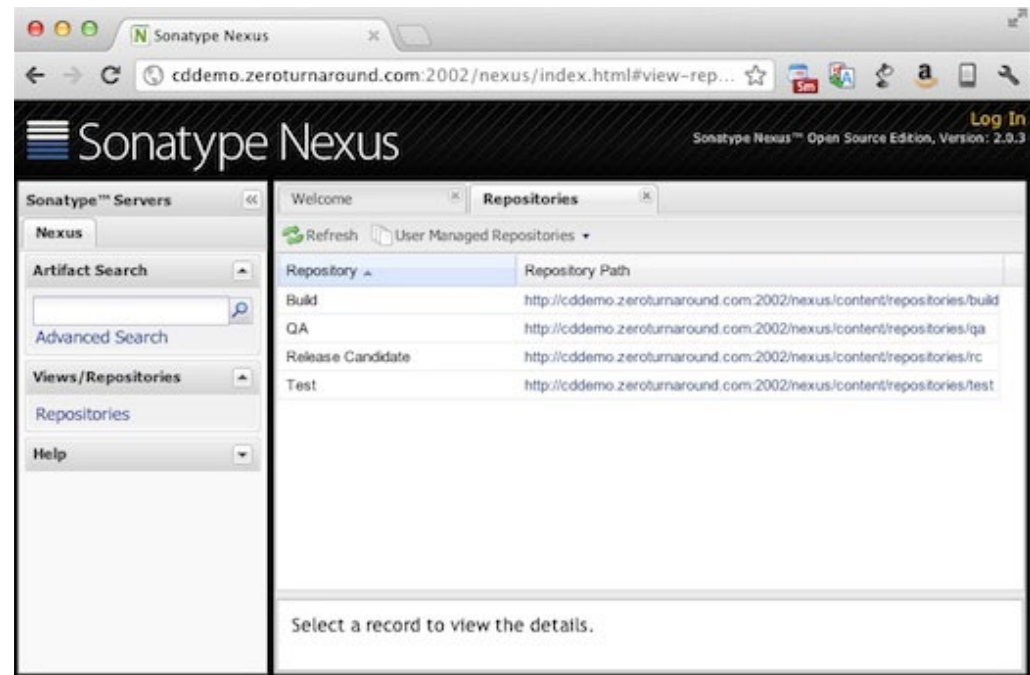
```
Jevgenis-MacBook-Air:workspace ekabanov$ hg clone http s://bitbucket.org/ekabanov/cd-live-chat
destination directory: cd-live-chat
requesting all changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 93 changes to 87 files
updating to branch default
87 files updated, 0 files merged, 0 files removed, 0 files unresolved
Jevgenis-MacBook-Air:workspace ekabanov$
```

**We have also created several local repositories corresponding to each stage of the CD pipeline:**

/repos/build/  
/repos/test/  
/repos/qa/  
/repos/rc/

The “RC” name stands for Release Candidate and reflects the fact that the artifact is ready to be deployed to production, rather than it has been or should be deployed. This is one of the key differences between Continuous Delivery and Continuous Deployment. The latter calls for automatically triggered releases, whereas the former just calls for a release-ready state, delegating the actual release decision to other decision makers.

Note that we do not create the Stage repository. We will not cover the Stage part of the pipeline in this paper, as it is derivative of the Test, QA and Production parts.



Here we use the [Sonatype Nexus](#) repository manager, although others like [JFrog Artifactory](#) and [Apache Archiva](#) will work just as well.

# Orchestration Service

The goal of every CD pipeline should be to enable an automated, trackable workflow. We need an orchestration service to manage that workflow, invoking various stages and managing the decision points. Luckily, such requirements have existed for a long while in the Continuous Integration world, so we can use one of the mature and feature-rich CI servers.

We use the [Jenkins CI](#) server, because it is by far the most popular, [reaching 49% market penetration](#). Popular alternatives include [Atlassian Bamboo](#) and [JetBrains TeamCity](#).

We will initially create only the first three jobs -- **build**, which produces an artifact, **deploy-test**, which deploys it to the Test environment and **automatic-tests**, which runs tests on that environment. The build step is a part of the Build stage, while the next two steps are a part of the Test stage, even though there isn't a great way in Jenkins to bring that out. To have an overview of the step and builds in-progress we will use the [Jenkins Build Pipeline Plugin](#):





# Build Job

The build job starts by doing a checkout of the Mercurial repository:

**Source Code Management**

☐ CVS

☒ Mercurial

Mercurial Version: (Default)

Repository URL:

Branch:

Repository browser: (Auto)

Advanced...

Once that is done, we can run the Maven build (further on we'll show Maven and shell commands without the screenshot):

**Invoke top-level Maven targets**

Maven Version: (Default)

Goals:

Advanced...

Delete

Notice that we pass `$BUILD_NUMBER` to the build. `$BUILD_NUMBER` is a built-in Jenkins environment variable which resolves to the number of the build. By overriding the app version on the command line, we insure that the artifact can be traced back to the build that created it.

Next, we upload the ready artifact to the Build repository:

```
[mvn] deploy:deploy-file
-Durl=http://127.0.0.1:2002/nexus/content/
repositories/build/
-DrepositoryId=build -Dfile=target/lr-demo.war
-Dfiles=build.txt
-Dtypes=txt -Dclassifiers=build-notes -Dpackaging=war
-DgroupId=org.zereturnaround -DartifactId=lr-demo
-Dversion=B${BUILD_NUMBER}
```

Uh-oh! What is this “build.txt” and why is it uploaded with the artifact?

Well, if you remember, **Record** is one of the key tenets of CD. Although the history is preserved in Jenkins, our primary deliverables are the artifacts deployed to stage repositories. Thus we’d like to have the information on the build also present in the repository. Some artifact repositories, such as [Artifactory from JFrog](#), support saving such information out-of-the-box with dedicated Jenkins plugins, but we can also just generate and upload a text file that describes the build.

```
echo “[BUILD]” > build.txt
echo “Build: ${BUILD_NUMBER}” >> build.txt
echo “Jenkins URL: ${BUILD_URL}” >> build.txt
echo “Hg revision: ${MERCURIAL_REVISION}” >> build.txt
echo “Hg log:” >> build.txt
hg log -r ${MERCURIAL_REVISION} >> build.txt
```

Using shell scripts we save information about the build, including the number, Jenkins build URL, Mercurial revision and even the commit message to the “build.txt” file and then upload it to the Maven repository along with the artifact. This means that whenever something fails, the repository will contain all the information about the artifact. The ready “build.txt” looks like this:

```
[BUILD]
Build: 116
Jenkins URL: http://localhost:2001/job/build/116/
Hg revision: 918894d22205c2471bb2220a0db1041f84e94c6
Hg log:
changeset: 10:918894d22205
tag: tip
user: Jevgeni Kabanov <jevgeni@
zereturnaround.com>
date: Fri Jun 01 15:14:41 2012 +0300
summary: Added /success for test purposes
```

## Deploy- Test Job

Cool - now we have an artifact. But for it to be useful, we need to deploy it to the testing environment and get those automated tests crunching. Because multiple builds can run in parallel, we need multiple versions to be deployed side-by-side for the automated and manual tests to progress.

Since we have used **LiveRebel** for staging and production delivery in this article, we will also use it for the tests, although in this case using the **Jenkins Deploy** plugin would work just as well (at least with Apache Tomcat). The benefits of LiveRebel is that it supports a much wider array of application servers and ensures that **OutOfMemoryError** never occurs.

We start by downloading the artifact from the Build repository to guarantee correctness.

```
[mvn] org.apache.maven.plugins:maven-  
dependency-plugin:2.4:get  
-Dartifact=org.zerturnaround:lr-  
demo:B${SOURCE_BUILD_NUMBER}:war  
-DremoteRepositories=cd:::ht  
tp://127.0.0.1:2002/nexus/content/  
repositories/build/  
-Ddest=lr-demo.war
```

We will discuss the origin of the “\${SOURCE\_BUILD\_NUMBER}” variable when we discuss wiring jobs a little later.

**Next we deploy the artifact to the chosen server:**

The screenshot shows the 'Deploy artifacts with LiveRebel' configuration page in Jenkins. The 'Artifacts' section has 'lr-demo.war' entered. Below it, there are checkboxes for 'Only upload artifacts to LiveRebel, no update or deployment will be triggered.' (unchecked) and 'Override artifact liverebel.xml' (checked). The 'Application Name' field contains 'lr-demo-B\${SOURCE\_BUILD\_NUMBER}' and the 'Application Version' field contains 'tmp'. The 'Context path' section has 'lr-demo-B\${SOURCE\_BUILD\_NUMBER}' entered, with a checkbox for 'Use fallback strategy if compatible with warnings' (unchecked). The 'Fallback update strategy' section has 'Offline update' selected (radio button). The 'Servers' section has three options: 'Production 1 (online)' (unchecked), 'Production 2 (online)' (unchecked), and 'Test (online)' (checked). The 'Metadata associated with this archive' field at the bottom contains 'build.txt'.

This deploys the current app build to a unique URL formed like <http://host:port/lr-demo-BXXX>, where XXX refers to the build number. Notice that “build.txt” is attached to the deployment and thus needs to be downloaded.

## Automated- Test Job

This job should run an acceptance test on the test deployment to verify it. We will only use a crude test for demonstration purposes, but in a real app you would use [JUnit](#), [Selenium](#), [Arquillian](#) and other useful tools to make sure that the app version is in good working order.

To start, we will download the WAR and build notes from the previous stage:

```
[mvn] org.apache.maven.plugins:maven-dependency-plugin:2.4:get
-Dartifact=org.zerturnaround:lr-demo:B${SOURCE_BUILD_NUMBER}:war
-DremoteRepositories=cd:::http://127.0.0.1:2002/nexus/content/repositories/build/
-Ddest=lr-demo.war
```

```
[mvn] org.apache.maven.plugins:maven-dependency-plugin:2.4:get
-Dartifact=org.zerturnaround:lr-demo:B${SOURCE_BUILD_NUMBER}:txt:build-notes
-DremoteRepositories=cd:::http://127.0.0.1:2002/nexus/content/repositories/build/
-Ddest=test.txt
```

Next we run the “test” and update the build notes:

```
echo "[TEST]" >> test.txt
echo "Jenkins URL: ${BUILD_URL}" >> test.txt
wget "http://localhost:8080/lr-demo-B${SOURCE_BUILD_NUMBER}/api/success" >> test.txt
echo "Automated Tests Passed!!!" >> test.txt
```

Notice that we change the classifier from “build-notes” to “test-notes”. This is essential to avoid clashes in the local Maven repository.

# Wiring

So far we have defined three Jenkins jobs, but no relations among them. Jenkins still has some way to go before getting awesome at wiring things together; aside from the [Build Flow plugin](#), whose documentation we found tricky, the only way to start the next job is to have either an explicit dependency or an explicit trigger.

Therefore, for the *build* job, we'll add the following post-build step:

The screenshot shows the Jenkins configuration page for a job, specifically the 'Trigger parameterized build on other projects' section. Under 'Build Triggers', the 'Projects to build' field contains 'deploy-test,'. The 'Trigger when build is' dropdown is set to 'Stable'. The 'Trigger build without parameters' checkbox is unchecked. Below this, the 'Predefined parameters' section is expanded, showing a text area with the parameter definition 'SOURCE\_BUILD\_NUMBER=\${BUILD\_NUMBER}'.

We start *deploy-test* explicitly and pass it a `$SOURCE_BUILD_NUMBER` parameter that just uses the current `$BUILD_NUMBER`. This is necessary so we can use the original build number as the version throughout the pipeline.

The *deploy-test* job triggers *automated-tests* similarly:

The screenshot shows the Jenkins configuration page for the 'deploy-test' job, specifically the 'Trigger parameterized build on other projects' section. Under 'Build Triggers', the 'Projects to build' field contains 'automatic-tests,'. The 'Trigger when build is' dropdown is set to 'Stable'. The 'Trigger build without parameters' checkbox is unchecked. Below this, the 'Current build parameters' section is expanded, which is empty.

Note: We don't need to pass the parameter explicitly as *deploy-test* was triggered with the same parameter. The same applies further down the pipeline.

# Delivery Manager

Delivery Management is a comparatively new concept. This is mainly because until now the actual delivery of the application to production without downtime was either impossible or required a lot of manual work and headaches on behalf of the Ops team. Although there are ways to script the deployment in a semi-automated way we decided to use LiveRebel instead for the following reasons:

**No Downtime:** LiveRebel will choose whether to hotpatch the application online or do a rolling session drain, but either way the users can keep on their work.

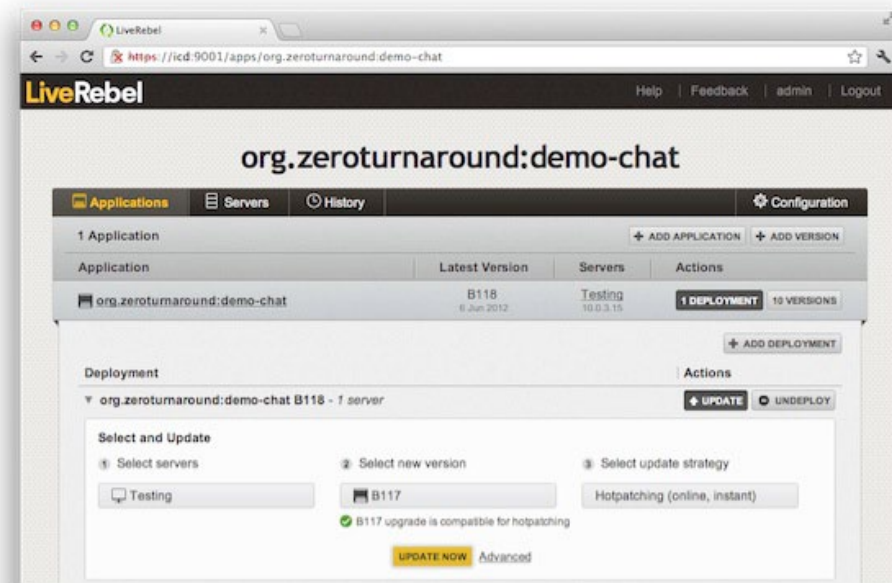
**Failure Management:** LiveRebel always takes the application from consistent state to consistent state and makes sure that any update you do can be rolled back with a single button push.

**Ecosystem Support:** Any app, any server, any source. Not just Tomcat!

**Exact Knowledge:** LiveRebel shows exactly what is deployed, where, how and by whom.

**Zero Configuration:** You can start using LiveRebel inside 5 minutes, without any docs.

This ties well with the tenets of CD, since we can get the production deployment just as predictable and manageable as the rest of the process and eliminate human intervention without fear.



LiveRebel alternatives include [Cargo](#), Jenkins Deploy plugin or just scripting the native tools that containers provide.

## Deploy- Production Job

It's OK if you still have a ton of questions at this point - things are still far from clear. We will discuss the different requirements that will arise in real applications later, but now let's finish our example with production deployment. We are skipping the Staging stage because there is nothing new in it compared to the ones we covered. We will come back to the QA stage later in the document.

We start by downloading the Test build (this should actually be the RC build produced by Staging):

```
[mvn] org.apache.maven.plugins:maven-  
dependency-plugin:2.4:get  
-Dartifact=org.zerturnaround:lr-  
demo:B${SOURCE_BUILD_NUMBER}:war  
-DremoteRepositories=cd:::ht  
tp://127.0.0.1:2002/nexus/content/  
repositories/test/  
-Ddest=lr-demo.war
```

The actual deployment is very easy:

The screenshot shows the 'Deploy artifacts with LiveRebel' configuration page. The 'Artifacts' field is set to 'lr-demo.war'. The 'Context path' is 'lr-demo'. The 'Fallback update strategy' is set to 'Rolling restart'. The 'Servers' section has 'Production 1 (online)' and 'Production 2 (online)' checked. The 'Metadata associated with this archive' is 'rc.txt'.

**Deploy artifacts with LiveRebel**

Artifacts: lr-demo.war

☐ Only upload artifacts to LiveRebel, no update or deployment will be triggered.

☒ Override artifact liverebel.xml

Application Name: lr-demo

Application Version: B\${SOURCE\_BUILD\_NUMBER}

Context path: lr-demo

☒ Use fallback strategy if **compatible with warnings**

☐ Offline update

☒ Rolling restart

☒ Production 1 (online)

☒ Production 2 (online)

☐ Test (online)

Metadata associated with this archive: rc.txt

A more interesting question is how to trigger that deployment.

One way is to use the feature provided by the Jenkins Build Pipeline plugin and add a post-build step to the automated-test job:

The screenshot shows the 'Build Pipeline Plugin -> Manually Execute Downstream Project' configuration page. The 'Downstream Project Names' field is set to 'deploy-production'.

**Build Pipeline Plugin -> Manually Execute Downstream Project**

Downstream Project Names: deploy-production

Delete

Once the rest of the pipeline is complete, Jenkins will show a “Trigger” button on the deploy-production job:

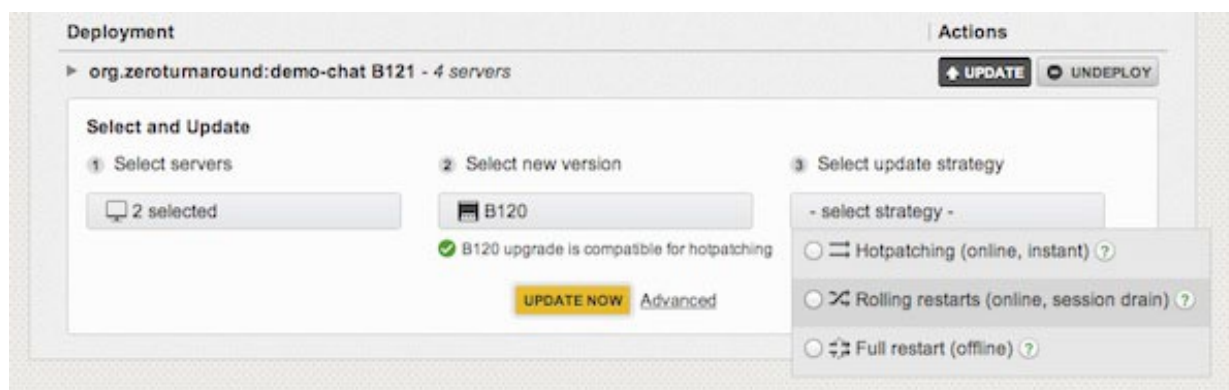


Alternatively you can use the built-in Jenkins REST API and leave the button somewhere more convenient:

[http://icd:2001/job/deploy-production/buildWithParameters?SOURCE\\_BUILD\\_NUMBER=112](http://icd:2001/job/deploy-production/buildWithParameters?SOURCE_BUILD_NUMBER=112)

This also requires us to set up deploy-production as a parameterized build, with SOURCE\_BUILD\_NUMBER as the parameter. A very useful feature of Jenkins is that by adding /api/ to any URL you will get a reference on what you can do with that particular entity.

Finally, LiveRebel’s Jenkins plugin provides an option to only upload the artifact without doing the deploy/update, so you can do the actual update with more control from the LiveRebel UI:





# Manual QA

Although automated tests are getting better and better with every year, there are still good reasons to use the ingenuity of the human mind to test the software. Thus we need to be able to incorporate a manual element in our automated workflow. Here's how we do it.

First we set up a begin-qa job, that is triggered by automatic-test and begins the QA stage.

We start by downloading the test-notes:

```
[mvn] org.apache.maven.plugins:maven-dependency-plugin:2.4:get
-Dartifact=org.zeroturnaround:lr-demo:B${SOURCE_BUILD_
NUMBER}:txt:build-notes
-DremoteRepositories=cd:::http://127.0.0.1:2002/nexus/content/
repositories/test/ -Ddest=build.txt
```

Next we use the [email-ext Jenkins plugin](#) to send a customized email to the QA team (or even better, to their task manager software):

## Here's the full text of the email:

We have prepared an environment for you to test:

[http://host:port/lr-demo-B\\${ENV,var="SOURCE\\_BUILD\\_NUMBER"}](http://host:port/lr-demo-B${ENV,var=)

Please test it thoroughly and IF successful click on this link:

[http://host:port/job/qa-success/buildWithParameters?SOURCE\\_BUILD\\_NUMBER=\\${ENV,var="SOURCE\\_BUILD\\_NUMBER"}](http://host:port/job/qa-success/buildWithParameters?SOURCE_BUILD_NUMBER=${ENV,var=)

Note the strange syntax for inserting environment variables. Note also that we attach the test-notes to the email, thus providing the QA team with full context of the build including the name of the originator. With a bit more effort we could also insert all that information directly into the email.

It is important to correctly set up the trigger in the Advanced settings for this to work:



The qa-success job is triggered by the link in the email as a parameterized build. It downloads the test-notes...

```
[mvn] org.apache.maven.plugins:maven-  
dependency-plugin:2.4:get  
-Dartifact=org.zeroturnaround:lr-  
demo:B${SOURCE_BUILD_NUMBER}:txt:test-  
notes -DremoteRepositories=cd:::ht  
tp://127.0.0.1:2002/nexus/content/  
repositories/test/ -Ddest=qa.txt
```

...creates and uploads the qa-notes along with the artifact to the QA repository

```
echo "[QA]" >> qa.txt  
echo "Manual tests passed!!!" >> qa.txt  
  
[mvn] org.apache.maven.plugins:maven-  
dependency-plugin:2.4:get  
-Dartifact=org.zeroturnaround:lr-  
demo:B${SOURCE_BUILD_NUMBER}:war  
-DremoteRepositories=cd:::ht  
tp://127.0.0.1:2002/nexus/content/  
repositories/test/ -Ddest=lr-demo.war
```

```
[mvn] deploy:deploy-file  
-Durl=http://127.0.0.1:2002/  
nexus/content/repositories/  
qa/ -DrepositoryId=cd -Dfile=lr-  
demo.war -Dfiles=qa.txt  
-Dtypes=txt -Dclassifiers=qa-  
notes -DgroupId=org.zeroturnaround  
-Dpackaging=war -DartifactId=lr-demo  
-Dversion=B${SOURCE_BUILD_NUMBER}
```

## Managing Resources

One of the issues we commonly encounter in different stages of manual and automated testing is the fact that the number of environments is limited while the number of builds that can be in progress at the same time is essentially unlimited. This can be solved by using elastic cloud deployment, but even then it's an unreasonable cost as the process will just bottleneck at a different stage.

Let's start by creating a clean-up-test job that will be triggered by qa-success and would be responsible for removing the deployment we created in deploy-test.

Unfortunately LiveRebel's Jenkins plugin does not support undeployment at the moment, but we can use the Command-Line Interface to the same effect:

```
liverebel/bin/lr-cli.sh -url https://cddemo.zereturnaround.com:9001/ -token 14a3c156-955b-4934-a500-e19404a76945 undeploy -app lr-demo-B${SOURCE_BUILD_NUMBER} -servers ec54877faf343586263cf4c1d7d03088
```

```
liverebel/bin/lr-cli.sh -url https://cddemo.zereturnaround.com:9001/ -token 14a3c156-955b-4934-a500-e19404a76945 remove -app lr-demo-B${SOURCE_BUILD_NUMBER}
```

Limiting the number of deployments is a harder problem. Jenkins does provide a [Throttle Concurrent Builds Plugin](#), but it limits the number of builds that can be run at the same time. However, because no builds run during the Manual QA phase, this doesn't work in our example.

The best solution to be used at the moment is to write a script that would query LiveRebel to find the number of test deployments and delay the build if the max number is up. A companion to this script is a Jenkins job that expires the test deployments after some time. It's a bit clumsy, but it works.

# Business Delivery

The philosophy around Continuous Delivery includes an understanding that new releases should be triggered by the party responsible for the business value that the new version brings. This will probably include at least some staff who you can not exactly expect to find and press a button in Jenkins. Instead we can use email, or even build a custom interface based on the Jenkins REST API.

Let's create a begin-deploy-production job in Jenkins. First we download the qa-notes and mark the build as a release candidate:

```
[mvn] org.apache.maven.plugins:maven-dependency-  
plugin:2.4:get -Dartifact=org.zereturnaround:lr-  
demo:B${SOURCE_BUILD_NUMBER}:txt:qa-notes -DremoteRepos-  
itories=cd:::http://127.0.0.1:2002/nexus/content/  
repositories/qa/ -Ddest=rc.txt
```

```
echo "[RC]" >> rc.txt  
echo "Marked as RC" >> rc.txt
```

```
[mvn] org.apache.maven.plugins:maven-dependency-  
plugin:2.4:get -Dartifact=org.zereturnaround:lr-  
demo:B${SOURCE_BUILD_NUMBER}:war -DremoteReposi-  
tories=cd:::http://127.0.0.1:2002/nexus/content/  
repositories/qa/ -Ddest=lr-demo.war
```

```
[mvn] deploy:deploy-file -Durl=http://127.0.0.1:2002/  
nexus/content/repositories/rc/ -DrepositoryId=cd  
-Dfile=lr-demo.war -Dfiles=rc.txt -Dtypes=txt  
-Dclassifiers=rc-notes -DgroupId=org.zereturnaround  
-Dpackaging=war -DartifactId=lr-demo  
-Dversion=B${SOURCE_BUILD_NUMBER}
```

Next we send an email with subject line like:

```
Release Candidate ready for lr-demo:B${ENV,var="SOURCE_  
BUILD_NUMBER"}
```

and body text like:

The release candidate passed all tests and ready to be de-  
ployed to production.

Just click this link to deploy it to production!

```
http://host:port/job/deploy-production/  
buildWithParameters?SOURCE_BUILD_  
NUMBER=${ENV,var="SOURCE_BUILD_NUMBER"}
```

If you want to do it nicer, you can provide a  
dedicated interface with a big red button or  
even put a physical button on  
management's desk:

You can get one from [Amazon](#),  
but it will only work on Windows OS.



# Conclusions

The most important thing to learn is that it is now fairly easy to build a Continuous Delivery pipeline and most of the complicated workflows in your organization can be modeled using Jenkins, Nexus and LiveRebel.

## The pipeline that we built has the following key aspects:

**Jenkins** is used to orchestrate the overall workflow and create a pipeline that takes patches from VCS and produces release candidate builds that have passed all the hurdles. Scripting Jenkins is essential for more complicated scenarios.

**Nexus** is used to create synchronization points for long-running or manual processes as well as long-term storage of records and artifacts.

**LiveRebel** is used to manage servers, applications, versions and deployments in a centralized and scriptable way. It is responsible for applying changes to environments without a negative impact on users or systems.

**Email** can be used to incorporate manual workflows into the CD pipeline. Alternatively REST or similar APIs can be used to integrate internal tooling into the pipeline in an asynchronous way.

**Text metadata** is a simple, but efficient way to record the log of everything that is important in the life of a build. It should thread through the pipeline and be available everywhere an artifact is deployed, including in production.

With those tools at your disposal, and the Continuous Delivery philosophy tucked into your belt, it isn't hard to build an automated, trackable, testable and recoverable pipeline for production updates that bring continuous advantage your users, your staff or your organization.

For more information, please send your queries to [info@zeroturnaround.com](mailto:info@zeroturnaround.com)



Thanks for reading

Contact Us

[info@zeroturnaround.com](mailto:info@zeroturnaround.com)

**Estonia**

Ülikooli 2, 5th floor  
Tartu, Estonia, 51003  
Phone: +372 740 4533

**USA**

545 Boylston St., 4th flr.  
Boston, MA, USA, 02116  
Phone: 1(857)277-1199