Gut verpackt

Technischer Aufbau einer Continuous-Delivery-Plattform in der Cloud

Marcel Birkner

Continuous Delivery beschreibt die vollständige Automatisierung des Softwareentwicklungsprozesses und verkürzt damit die Lieferzeiten bei gleichzeitig verbesserter Qualität. In diesem Artikel wird der Aufbau einer Continuous-Delivery-Pipeline mit Hilfe von Open-Source-Werkzeugen Schritt für Schritt beschrieben.

Welchen Nutzen Continuous Delivery (CI) für Ihr Unternehmen hat und welche organisatorischen, kulturellen und technischen Voraussetzungen notwendig sind, können Sie in der letzten Ausgabe von JavaSPEKTRUM [Birk13] nachlesen. Dort wurde auch auf die DevOps-Bewegung [Hütt13] hingewiesen, die versucht, eine Brücke zwischen den Entwicklungs- und Betriebsteams zu schlagen, die ja beide am Deployment beteiligt sind. Für die Umsetzung von Continuous Delivery ist es wichtig, dass die sich aus Quellcodeänderungen ergebenden neuen Release-Artefakte „kontinuierlich" vollständig getestet werden. Dazu gibt es eine Reihe von Open-Source-Werkzeugen und -Frameworks, die in einer Continuous-Delivery-Pipeline eingesetzt werden können.

Voraussetzung

Um eine realistische und vollständige Pipeline aufzubauen, benötigen wir erst einmal ein paar Server. Für Testzwecke habe ich mir in der Amazon EC2 ein paar Server-Instanzen eingerichtet. Eine Instanz wird für den zentralen Build-Server verwendet, jeweils eine für die Testumgebung sowie die Staging-Umgebung und je eine für die blaue und grüne Produktionsumgebung (Blue/Green Deployment, [Fowl10]). Kostengünstig lassen sich eigene Server-Instanzen auswählen, konfigurieren und in kürzester Zeit starten. Werden die Server nicht mehr gebraucht, kann man die Instanzen stoppen, ohne dass weitere Kosten anfallen. Für diese Demo verwende ich mehrere Ubuntu-32-Bit-Instanzen mit 3,75 GB RAM und zwei CPU-Kernen. Auf allen Instanzen ist Oracle 7 JDK installiert. Sie können diese Demo auch auf Ihren eigenen Servern oder VMWare-Instanzen nachbauen, ganz unabhängig vom Betriebssystem.

Abbildung 1 zeigt alle Schritte einer Continuous-Delivery-Pipeline, die ich in diesem Artikel vorstellen werde. Los geht's.

Installationen des Jenkins-Servers

Viele Unternehmen verwenden schon seit längerem CI-Server, wie beispielsweise Jenkins/Hudson, CruiseControl, TeamCity oder Bamboo. Anhand von Jenkins möchte ich Ihnen zeigen, wie leicht es ist, eine Continuous-Delivery-Pipeline zu konfigurieren. Der Jenkins-Server ist dabei die zentrale Komponente der Pipeline. Auf ihm werden wir als Erstes alle wichtigen Komponenten für die Pipeline installieren. Zuerst wird Jenkins über den Paketmanager installiert [JenkinsCI]:

```
sudo apt-get install jenkins
```

Nach einer erfolgreichen Installation können Sie im Browser auf Jenkins zugreifen und weitere Plug-ins über den Jenkins-Update-Manager installieren: `http://localhost:8080/pluginManager/available`. Sollten Sie Amazon-EC2-Instanzen verwenden, müssen Sie in der Security Group für Ihre Jenkins-Server-Instanz die Ports 8080, 8081 und 9000 freischalten. Diese werden wir für die nächsten Schritte brauchen.

Folgende Jenkins-Plug-ins sollten Sie installieren:

- Das GitHub-Plug-in wird für die Anbindung an das Versionskontrollsystem (VCS) benötigt. Jenkins unterstützt jedes bekannte VCS und integriert sich somit leicht in jede IT-Landschaft [GitPlugin].
- Das Sonar-Plug-in ermöglicht uns die Integration mit dem „Source Code Quality Management"-Werkzeug (SQM) Sonar von Sonarsource. Dies erlaubt uns, nach jeder Quellcodeänderung eine statische Quellcodeanalyse durchzuführen und positive und negative Trends in der Codequalität frühzeitig festzustellen [SonarPlugin].
- Das Clone-Workspace-SCM-Plug-in ermöglicht uns, in jedem Schritt der Delivery-Pipeline auf den gleichen Versionsstand des

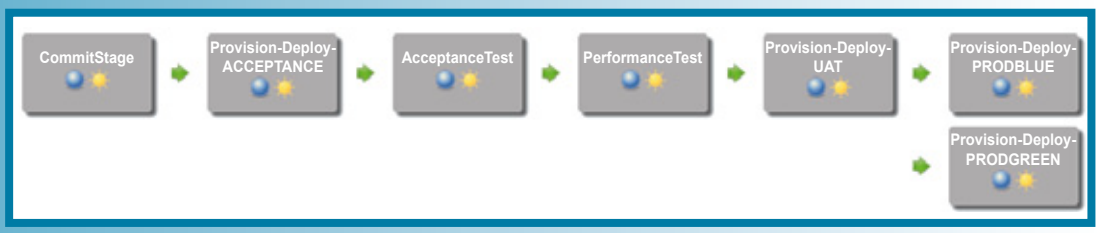

Abb. 1: Darstellung einer Continuous-Delivery-Pipeline, mit Hilfe des Jenkins-Build-Pipeline-Plug-ins

Quellcodes und der Infrastrukturskripte zurückzugreifen. Dieses Plug-in zipped den gesamten Jenkins-Workspace nach der Commit Stage, und stellt den Workspace allen weiteren Pipeline-Stages zur Verfügung [CloneWSPlugin].

- Das Parameterized-Trigger-Plug-in erlaubt es uns, Parameter an einen Jenkins-Job zu übergeben. Dies ist wichtig, wenn wir zum Beispiel die Version des zu installierenden Artefakts über dieses Plug-in übergeben. Dadurch können wir auch ältere Versionen der Anwendung auf die Akzeptanz-Test-Umgebungen deployen [PaTrigger].
- Zuletzt brauchen wir noch das Build-Pipeline-Plug-in [BuildPipeline]. Dieses Plug-in erkennt die Abhängigkeiten der einzelnen Delivery-Pipeline-Jobs und stellt diese grafisch dar (s. Abb. 1).

Nach der Installation der Plug-ins sollte Jenkins neu gestartet werden. Unter `http://localhost:8080/configure` können Sie die Plug-ins für Ihre Umgebung konfigurieren.

Als Nächstes wird Nexus als Artefakt-Repository-Manager installiert [Nexus]. Hierfür reicht auch Artifactory oder Apache Archiva. Jedes gebaute Software-Artefakt wird immer versioniert ins Nexus-Repository hochgeladen und für alle zukünftigen Server-Provisionierungen bzw. Server-Bereitstellungen aus Nexus heruntergeladen. Des Weiteren wird Nexus als Proxy für externe Maven-Repositorys verwendet, wie zum Beispiel für Maven Central. Dadurch müssen Artefakte nur einmal aus dem Internet geladen werden und nicht mehrfach von jedem einzelnen Entwickler. Zudem können eigene Artefakte und Third-Party-Artefakte über dieses Artefakt-Repository in den Entwicklungsteams geteilt werden. Zusätzlich bietet Nexus ein Komponenten-Lebenszyklus-Management an. Dabei werden die verwendeten Artefakte auf bekannte Sicherheitsrisiken und Bugs analysiert, es wird die Ordnungsmäßigkeit von Lizenzen geprüft und alle Daten werden an einem zentralen Ort gesammelt, was die Einhaltung und Kontrolle von Compliance-Vorgaben in Unternehmen erheblich vereinfacht.

Nexus ist mit wenigen Schritten installiert:

```
wget http://www.sonatype.org/downloads/nexus-latest-bundle.tar.gz
tar xvzf nexus-latest-bundle.tar.gz
cd nexus-*/bin
./nexus start
```

Nach dem Starten ist Nexus unter `http://localhost:8081/nexus/index.html` (admin/admin123) erreichbar.

Im nächsten Schritt wird das SQM-Werkzeug Sonar installiert [Sonar]. Zusammen mit dem Build-Breaker-Plug-in prüft Sonar die Quellcodequalität gegen ein Set von Regeln und lässt den Jenkins-Build bei kritischen Bugs brechen. Dabei verwendet Sonar intern statische Quellcodeanalyse-Frameworks wie Findbugs, PMD, Checkstyle und Cobertura:

```
wget http://dist.sonar.codehaus.org/sonar-3.4.1.zip
unzip sonar*.zip
cd sonar*
bin/linux-x86-64/sonar.sh start
```

Sonar ist unter `http://localhost:9000/` (admin/admin) über den Browser erreichbar. Im Sonar Update Center (*Sonar -> Update Center -> Available Plugins*) werden als Nächstes folgende Plug-ins installiert: Build Breaker, PDF Export, Sonar Motion Chart, Timeline, Artifact Size, Sonargraph, Taglist und Radiator.

Damit Jenkins später auf alle anderen Server ohne Log-in zugreifen kann, müssen wir noch die SSH-Schlüssel zwischen Jenkins und allen anderen Servern austauschen. Als Erstes generieren wir auf dem Jenkins-Server ein SSH-Schlüsselpaar:

```
ssh-keygen -t rsa -C "marcel.birkner@codecentric.de"
```

Unter `.ssh/id_rsa.pub` finden Sie den öffentlichen Schlüssel des Jenkins-Servers. Loggen Sie sich jetzt in alle anderen Server ein und hinterlegen Sie den öffentlichen Schlüssel in der `.ssh/authorized_keys`-Datei. Testen Sie den passwortlosen Zugriff vom Jenkins-Server auf die jeweiligen Server und bestätigen Sie die einmalige Aufforderung auf der Kommandozeile, dass der Jenkins-Server in die `known_hosts` eingetragen wird.

Die wichtigsten Voraussetzungen für die Continuous-Delivery-Pipeline sind soweit geschaffen. Jetzt kann der Aufbau der Pipeline beginnen.

Wie ist eine Continuous-Delivery-Pipeline aufgebaut?

Eine Continuous-Delivery-Pipeline kann in folgende sechs Stufen oder Stages (s. Abb. 1) unterteilt werden:

- Commit Stage,
- Bereitstellung der Testumgebung,
- Akzeptanz-Tests gegen Testumgebung,
- Performance-Tests gegen Testumgebung,
- Bereitstellung der Staging-Umgebung (*User-Akzeptanztest-umgebung*, kurz UAT) und
- Bereitstellung der Produktionsumgebung (Blue/Green Deployment).

Die ersten fünf Schritte laufen vollständig automatisiert ab. Nur die Bereitstellung der Blue/Green-Produktionsumgebung sollte manuell durch ein One-Click-Deployment geschehen. Jeder einzelne Schritt ist ein eigener Jenkins-Job, der in den nächsten Abschnitten im Detail erklärt wird.

Commit Stage

In der Commit Stage wird das Versionskontrollsystem (VCS) eingebunden, das den Quellcode und die Infrastrukturskripte enthält. Dieser Job wird mit einem *Build Trigger* konfiguriert, sodass er bei Quellcodeänderungen sofort automatisch startet. In diesem Job wird der Quellcode der Anwendung kompiliert und bei Fehlern wird eine Benachrichtigung an das Entwicklerteam gesendet. Des Weiteren werden alle Unit-Tests ausgeführt, um die Basis-Funktionalität der Anwendung zu prüfen. Danach wird die statische Quellcodeanalyse mit Sonar gestartet. Durch das Build-Breaker-Plug-in kann in Sonar definiert

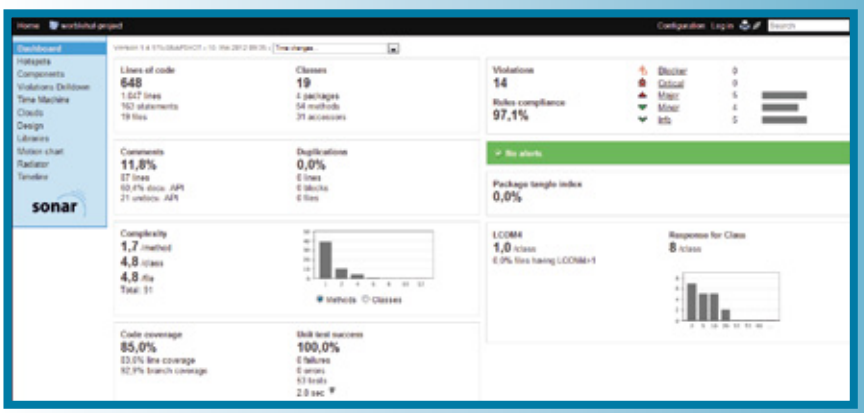

Abb. 2: Sonar Dashboard

werden, welche Fehler zu einem Abbruch des Jenkins-Jobs führen. Wenn zum Beispiel eine kritische FindBugs-Regel verletzt wird, sollte auch der Jenkins-Job mit einem Fehler abbrechen. Wenn auch hier keinerlei Fehler auftreten, wird das Artefakt getagged, versioniert und in das Nexus-Artefakt-Repository als Release-Kandidat hochgeladen.

Zuletzt wird unter *Build Other Jobs* der Name des nächsten Jenkins-Jobs eingetragen. In diesem Fall handelt es sich um den Job für die Provisionierung der Testumgebung, der im folgenden Abschnitt beschrieben wird. Mit Hilfe des Clone-Workspace-Plug-ins wird der aktuelle Stand des Workspaces gepackt und für alle weiteren Jobs zugreifbar gemacht. Dadurch ist sichergestellt, dass alle nachfolgenden Jobs der Pipeline auf die gleiche Version der Dateien zugreifen. Zwischenzeitliche Änderungen im VCS haben somit keine Auswirkung auf den aktuellen Durchlauf.

Provisionierung der Testumgebung

Im nächsten Schritt wird der Release-Kandidat auf einer Testumgebung voll automatisiert deployt. Damit alle Umgebungen immer nachvollziehbar konfiguriert werden, sollte ein Konfigurationsmanagement-Tool wie zum Beispiel Puppet oder Chef verwendet werden [Puppet]. Mit Puppet beschreibt man den gewünschten Zielzustand eines Servers deklarativ. Der zentrale Bestandteil sind Puppet-Manifest-Dateien, in denen Ressourcen beschrieben werden, die von Puppet konfiguriert werden. Zu den Core-Ressourcen-Typen gehören notify, file, package, service, exec, cron, user und group [PuppetTypes]. Das folgende Puppet-Skript legt über den Ressource-Type „user“ einen „tomcat“-User an, wenn er auf dem System noch nicht vorhanden ist:

```
user { 'tomcat':
  ensure => 'present',
  home => '/opt/tomcat/',
  shell => '/bin/bash',
}
```

Auf die gleiche Art und Weise lassen sich Datenbanken, Applikationsserver und weitere Server-Konfigurationen installieren. Puppet verwendet dabei die Paketmanager des jeweiligen Betriebssystems, um weitere Komponenten zu installieren. Die Puppet-Skripte laden zudem die gewünschten Software-Artefakte aus dem Nexus-Repository. Alle Puppet-Skripte werden wie auch der Quellcode in das VCS eingecheckt. Dadurch ist jederzeit nachvollziehbar, wer was wann am Quellcode oder an den Serverkonfigurationen geändert hat. Somit lassen sich Konfigurationsprobleme schneller beheben und die Server jederzeit mit einem vorherigen Versionsstand neu aufsetzen. Da Jenkins die Puppet-Skripte auf die jeweiligen Server kopiert und via SSH mit dem root-User aufruft, muss in der `/etc/sudoers`-Datei folgender Eintrag gemacht werden:

```
Defaults:ec2-user !requiretty
```

Wer Puppet nicht unter dem root-User ausführen möchte, kann dafür auf dem Server auch einen eigenen User mit den passenden Rechten einrichten. Mit dem folgenden Skript werden alle Puppet-Skripte auf den Testserver kopiert und danach wird Puppet via SSH auf dem Testserver aufgerufen:

```
echo "Copy Puppet files"
scp -r puppet/ ec2-user@$instance:
cpCmd="cp -r /home/ec2-user/puppet/* /etc/puppet/“
ssh ec2-user@$instance "sudo su --session-command='$cpCmd' root"

echo "Execute Puppet"
ppCmd="puppet apply /etc/puppet/manifests/site.pp"
ssh ec2-user@$instance „sudo su --session-command='$ppCmd' root“
```

Nachdem die Puppet-Skripte durchgelaufen sind, ist die Testumgebung vollständig provisioniert und der Release-Kandidat erfolgreich deployt.

Bei Anwendungen, die eine Datenbank verwenden, empfehlen sich Tools wie Liquibase oder DbMaintain. Datenbankänderungen werden bei Liquibase in XML-Changesets geschrieben und bei der Provisionierung einer Umgebung ausgeführt. Liquibase unterstützt auch einen Rollback-Mechanismus. Bei komplexeren Datenbankänderungen und Migrationen empfiehlt es sich, einen eigenen Datenbank-Updater zu schreiben (zum Beispiel mit einem JdbcTemplate), der beim Starten der Anwendung über eine Versionstabelle den Stand der Datenbank prüft und danach alle neuen Schema- und Datenänderungen durchführt.

Als Letztes wird in der Jenkins-Job-Konfiguration unter *Build Other Jobs* der *Akzeptanz-Test*-Job eingetragen.

Akzeptanz-Tests

Nachdem die Testumgebung mit der neuesten Version der Anwendung provisioniert wurde, werden die Akzeptanz-Tests ausgeführt. Dazu wird ein eigener Job in Jenkins eingerichtet, der die Akzeptanz-Tests gegen die Testumgebung laufen lässt. Für Akzeptanz-Tests von webbasierten Anwendungen lassen sich jBehave, das Robot-Framework oder Canoo WebTests einsetzen. Für das Testen von SOAP oder REST-Webservices ist SoapUI zu empfehlen [jBehave, SoapUI].

Mit einer wachsenden Basis an Akzeptanz-Tests wird die Anwendung automatisch auf Regressionen getestet, da immer alle Tests ausgeführt werden. Somit ist sichergestellt, dass neue Features keine bestehenden Funktionen brechen. Sollten Fehler während der Ausführung der Tests auftreten, bricht die Pipeline sofort ab und der Fehler wird dem Entwicklungsteam sofort mitgeteilt.

Performance-Tests

Wenn alle Akzeptanz-Tests erfolgreich durchlaufen wurden, haben wir ein produktionsreifes, getestetes Software-Artefakt. Für viele Anwendungen sind nicht-funktionale Anforderungen oft genauso wichtig wie neue Funktionen. Daher ist es sinnvoll, die Anwendung auch auf ihre Performance-Änderungen hin zu testen. Als Entwickler wollen wir sicher sein, dass die neuen Funktionalitäten die Anwendung in ihrer Gesamt-Performance nicht verschlechtern, sondern vielleicht sogar verbessern. Deshalb wird ein *Performance-Test*-Job in Jenkins angelegt. Diese Stage lässt vollautomatisch Performance-Tests gegen die Testumgebung laufen und prüft, ob die letzten Änderungen Auswirkungen auf die Performance-Schwellwerte haben (z. B. Antwortzeiten). Für Performance-Tests kann jMeter eingesetzt werden. Zusammen mit dem Chronos-Plug-in erhält man in Jenkins eine grafische Übersicht der jMeter-Performance-Berichte (s. Abb. 3).

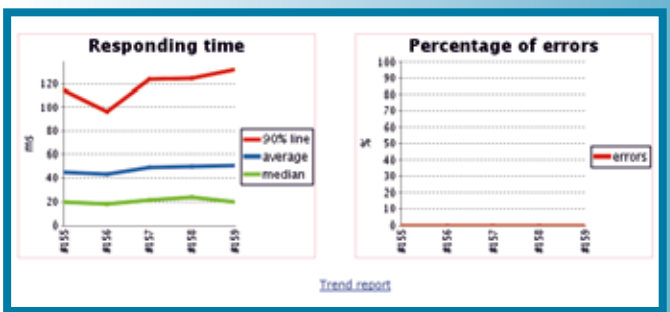

Abbildung 3: Jenkins-Chronos-Plug-in für jMeter-Berichte

Diese jMeter-Tests sollten nur als Indikator für die Performance-Entwicklung der Anwendung gesehen werden und können komplexe Last- und Performance-Tests nicht ersetzen. Bei kritischen Anwendungen mit vielen Hundert Servern sind größere Infrastruktur-Setups und längere Laufzeiten der Tests notwendig, um eine realistische Aussage der Anwendungsperformance zu erhalten. Auch solche Testumgebungen können automatisiert mit den gleichen Skripten provisioniert werden.

Provisionierung der Staging-Umgebung
Im letzten vollständig automatisierten Schritt wird die Staging-Umgebung provisioniert. Dafür werden die gleichen Puppet-Skripte wie für die Testumgebung verwendet. Auf der Staging-Umgebung können nun manuelle Tests durchgeführt werden, die nicht durch automatisierte Tests abgedeckt werden können. Zusätzlich kann der Fachbereich seine neuen Features auf der Staging-Umgebung testen. Wenn auch dieser Schritt erfolgreich war, steht einem Deployment in die Produktion nichts im Wege.

Provisionierung der Produktionsumgebung (Blue/Green Deployment)
Für das Deployment in die Produktion empfiehlt sich die Blue/Green-Strategie. Um diese umzusetzen, werden alle Server der Produktion gedoppelt. Der erste Teil ist aktiv und der zweite im Hot-Standby. Über einen Loadbalancer wird geregelt, welche Umgebung aktiv ist. Dadurch ist es möglich, eine neue Version der Software zuerst auf der Hot-Standby-Umgebung zu deployen und vom Fachbereich über ein internes Netzsegment abnehmen zu lassen. Wenn alles okay ist, werden neue Anfragen nach und nach auf die neue Umgebung umgeleitet. Sollte es Probleme mit der neuen Version der Software geben, wird der Loadbalancer wieder auf die alte Umgebung umgeschaltet. Für die Deployments in die Produktionsumgebungen werden wiederum die gleichen Skripte verwendet, wie für die Test- und Staging-Umgebungen. Durch mehrfache Deployments in die Testumgebungen ist sichergestellt, dass auch die Puppet-Skripte einwandfrei funktionieren.

Weitere Tools

In den letzten Jahren sind immer mehr Tools erschienen, die sich der DevOps-Problematik angenommen haben. Sie versuchen, Probleme bei Deployments und beim Betrieb von Anwendungen zu verringern. Zusätzlich unterstützen diese Tools Berechtigungskonzepte, die gerade in großen IT-Landschaften notwendig sind.

VMWare arbeitet am vFabric Application Director, der Deployments in VMWare-virtualisierten Umgebungen und der Amazon EC2 erleichtert. Von XebiaLabs gibt es mit Deployit ein weiteres spannendes Tool, das auf ein eigenes internes Repository für die Verwaltung von Deployment-Artefakten setzt. Für Amazon EC2 gibt es mit OpsWork ein eigenes Tool, das für die Provisionierung der EC2-Instanzen fertige Chef-Recipes bereitstellt.

Fazit

Für viele Unternehmen werden existierende Open-Source-Tools ausreichen, um ihre Deployment-Prozesse mit einer Continuous-Delivery-Pipeline zu automatisieren. In kurzer Zeit können oft lästige manuelle Aufgaben automatisiert und die Anzahl der Fehler, die durch manuelle Schritte entstehen, reduziert werden. Durch die Protokollierung aller Jenkins-Jobs entsteht zusätzlich ein Audit-Protokoll, das Dev- und Ops-Teams bei der Fehlersuche zuverlässig unterstützt. Für Unternehmen mit komplexen Teamstrukturen und diversen Abhängigkeiten zu externen und internen Anwendungen lohnt es sich, ein Auge auf den Markt der DevOps-Anwendungen zu werfen.

Ich hoffe, Sie konnten einen guten Überblick gewinnen, wie Sie eine Continuous-Delivery-Pipeline mit Open-Source-Tools aufbauen. Unter [CDDemo] finden Sie ein Webinar mit der Demo einer funktionierenden Delivery-Pipeline.

In einem weiteren Artikel über Continuous Delivery werde ich auf Anti-Patterns, die eine Umsetzung erschweren, eingehen und CD-Anforderungen an Anwendungen beschreiben, damit sie für Continuous Delivery bereit sind.

Literatur und Links

[AppDirector] VMWare Application Director, `http://www.vmware.com/products/application-platform/vfabric-application-director/features.html`

[Birk13] M. Birkner, Einführung in Continuous Delivery, in: JavaSPEKTRUM, 2/2013

[BuildPipeline] Jenkins-Build-Pipeline-Plug-in, `https://code.google.com/p/build-pipeline-plugin/`

[CDDemo] Continuous Delivery Webinar, `https://blog.codecentric.de/2012/11/continuous-delivery-webinar/`

[CloneWSPlugin] Jenkins-Clone-Workspace-Plug-in, `https://wiki.jenkins-ci.org/display/JENKINS/Clone+Workspace+SCM+Plugin`

[Deployit] XebiaLabs Deployit, `http://www.xebialabs.com/`

[Fowl10] M. Fowler, BlueGreenDeployment, 1.3.2010, `http://martinfowler.com/bliki/BlueGreenDeployment.html`

[GitPlugin] Jenkins-Github-Plug-in, `https://wiki.jenkins-ci.org/display/JENKINS/GitHub+Plugin`

[Hütt13] M. Hüttermann, Agiler Baukasten für DevOps, in: JavaSPEKTRUM, 1/2013

[jBehave] `http://jbehave.org`

[JenkinsCI] Jenkins-CI-Server, `http://jenkins-ci.org/`

[Liquibase] `http://www.liquibase.org/`

[Nexus] Nexus-Repository, `http://www.sonatype.org/nexus/`

[OpsWorks] AWS OpsWorks, `http://aws.amazon.com/opsworks/`

[PaTrigger] Jenkins-Parameterized-Trigger-Plug-in, `https://wiki.jenkins-ci.org/display/JENKINS/Parameterized+Trigger+Plugin`

[Puppet] `http://www.puppetlabs.com`

[PuppetTypes] Puppet Types, `http://docs.puppetlabs.com/references/stable/type.html`

[SoapUI] `http://www.soapui.org/`

[Sonar] Quellcodequalitätsmanagement, `http://www.sonatype.com/`

[SonarPlugin] Jenkins-Sonar-Plug-in, `http://docs.codehaus.org/pages/viewpage.action?pageId=116359341`

Marcel Birkner ist Software Consultant bei der codecentric AG. Sein Fokus liegt auf Java, Spring, Cloud-Technologien und allem rund um die Automatisierung von Softwareentwicklungsprozessen.
E-Mail: marcel.birkner@codecentric.de