

# Continuous Delivery

## How to Touch the Running System

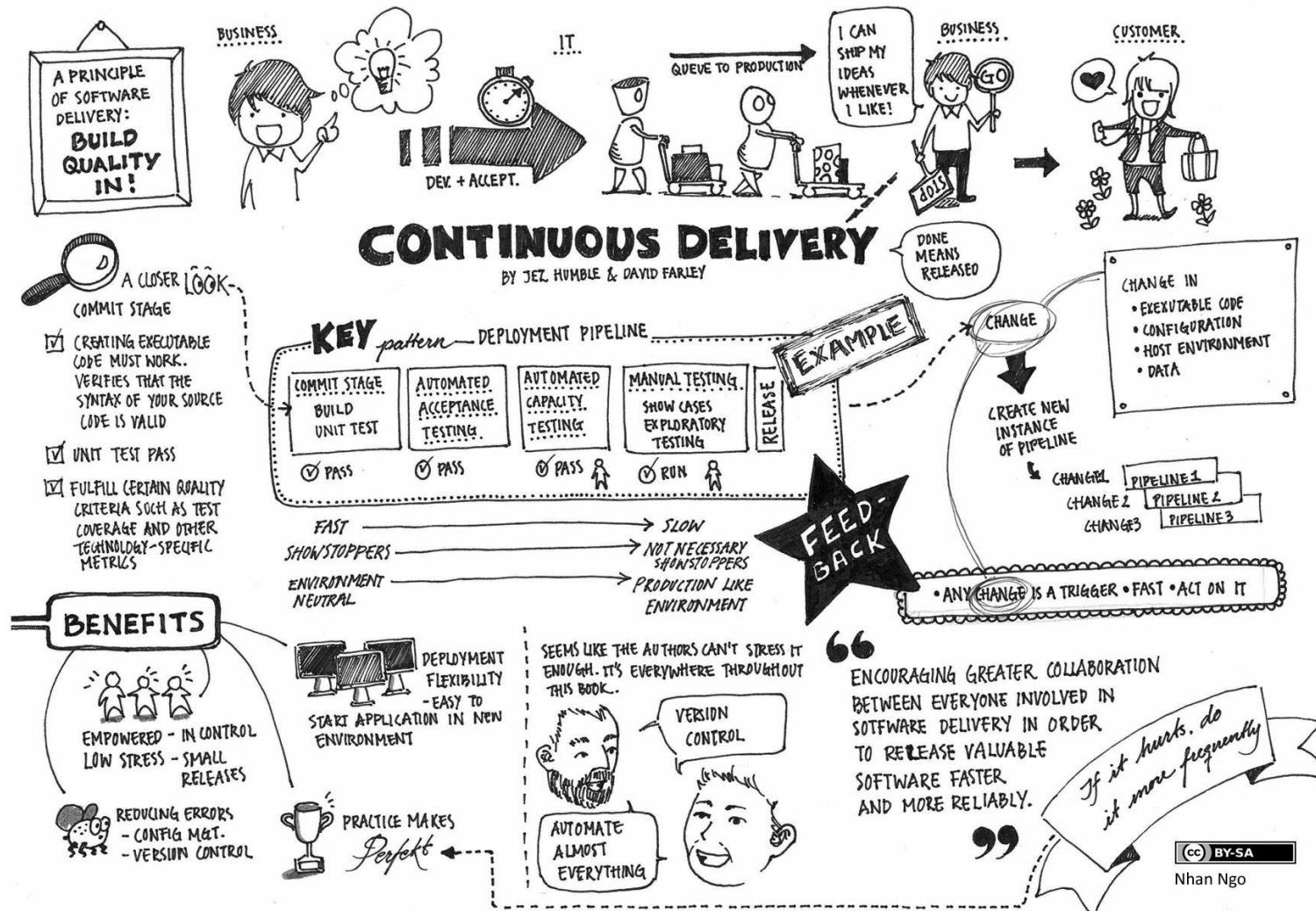


# Aufbau

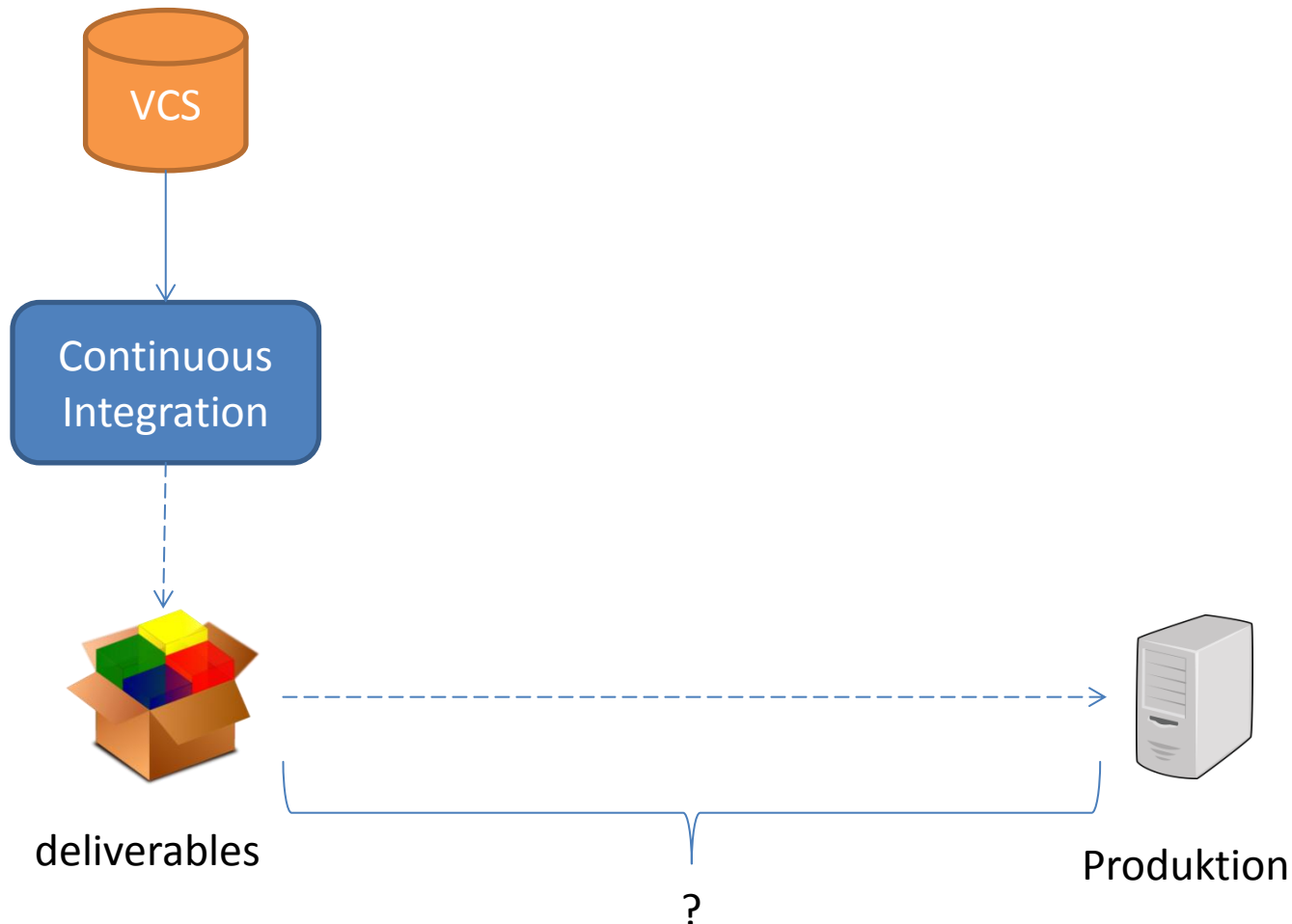
---

1. Was ist Continuous Delivery
2. Die Deployment Pipeline
3. Fazit und Ausblick
4. Literatur und weiterführende Links

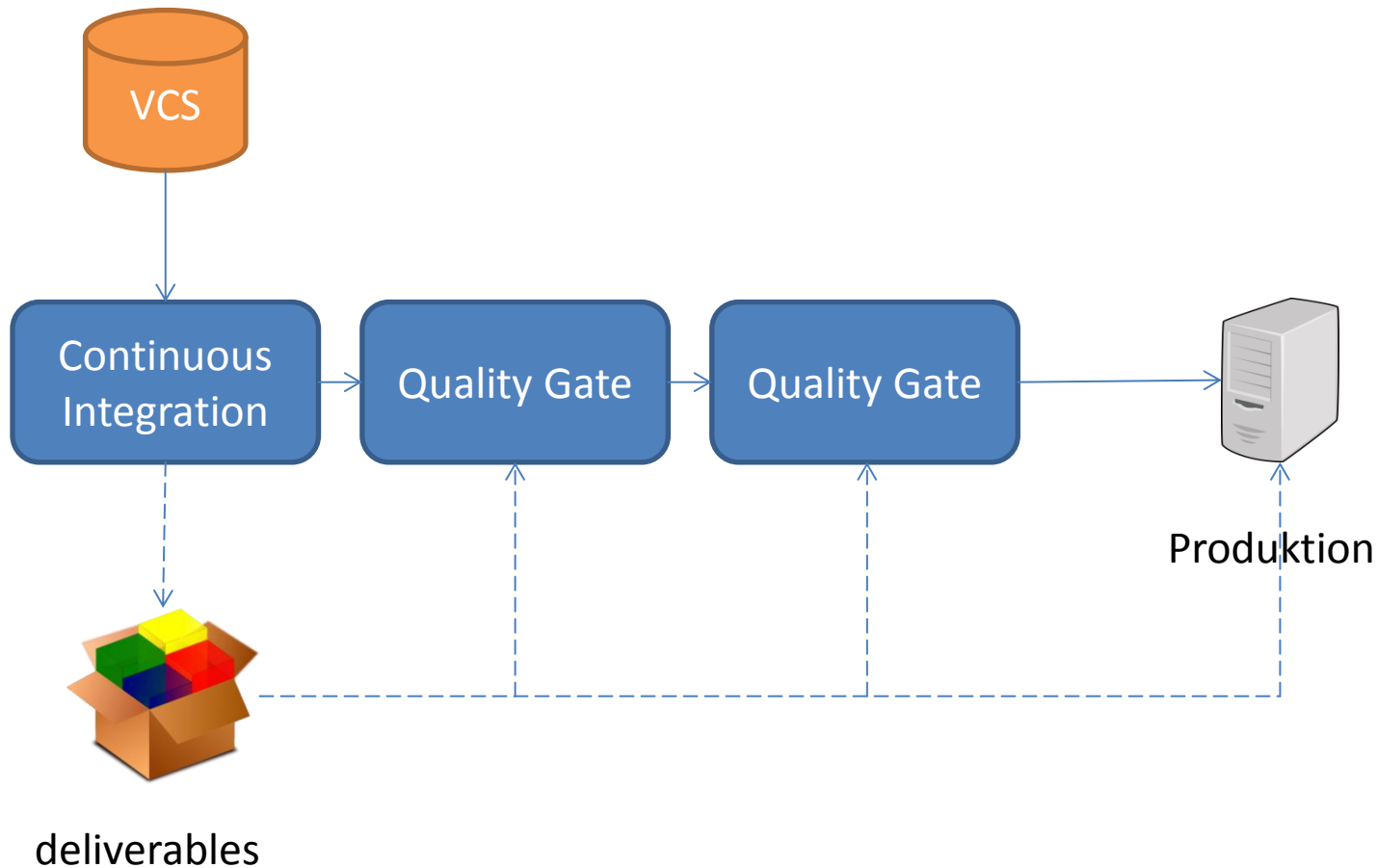
# Was ist Continuous Delivery



# Was ist Continuous Delivery



# Was ist Continuous Delivery



# Was ist Continuous Delivery

---

Continuous Delivery ist eine Vorgehensweise bei der Softwareentwicklung, die sicherstellen soll, dass sich die Software **immer** in einem **auslieferbaren Zustand** befindet und **automatisiert ausgeliefert („deployed“)** werden kann.

Dies wird durch die Einführung einer **hochautomatisierten Deployment Pipeline** erreicht, die sich an Lean-Ideen orientiert.

# Continuous...

## Integration/Delivery/Deployment

---

### Continuous Integration:

- „Software kann immer gebaut werden und erfüllt bestimmte Qualitätskriterien“

### Continuous Delivery:

- “Software **könnte** jederzeit automatisch in Produktion deployed werden“

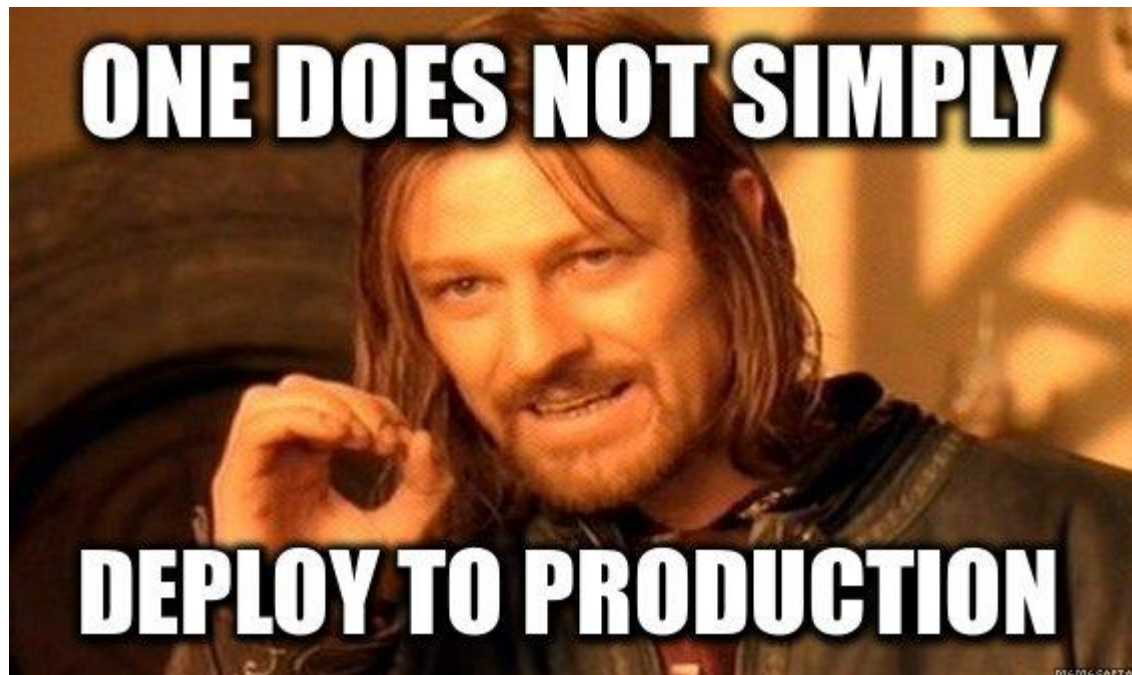
### Continuous Deployment:

- „Jede Änderung, die alle Stages erfolgreich durchlaufen hat, **wird** automatisch in Produktion deployed“

Vgl. Vorlesung „DevOps“

# Warum Continuous Delivery

---



...warum nicht?



# Continuous-Delivery- Antipatterns

---

- Manuelles Deployment in Produktion
- Entwicklung und Tests finden nicht in produktionsähnlicher Umgebung statt
- Manuelle Verwaltung der Konfiguration
- Manuelle Tests

# Zur Erinnerung...

---

*Our highest priority is to satisfy the customer  
through early and **continuous delivery**  
of valuable software.*

– <http://www.agilemanifesto.org/principles.html>

# Begriffsklärung:

## Release vs Deploy

---

- Keine einheitliche Definition, wird synonym verwendet, team-abhängig
- Wir klären für diese Vorlesung:

### **Release (Substantiv):**

- Version einer Software, die für eine Veröffentlichung außerhalb der Entwicklung vorgesehen ist

### **(to) release (Verb):**

- Eine bestimmte Version einer Software (Release) einer bestimmten Nutzergruppe zur Verfügung stellen (beispielsweise „Version 1.2.15 steht zum Download bereit“)
- Ja, man kann ein Release „releasen“

### **Release-Kandidat:**

- Jeder Stand der Software, der als Release in Frage kommt
- Bei Continuous Delivery ist jede Änderung ein potentieller Release-Kandidat

### **Deploy:**

- Installation und Konfiguration einer Anwendung in einer Zielumgebung (aktueller Code auf lokaler VM, Build #5432 auf Staging-System, Version 1.2.15 in Produktion)

# Continuous Delivery Prinzipien

---

- Der Auslieferungsprozess muss wiederholbar und verlässlich sein
- Automatisiere so viel wie (sinnvoll) möglich
- Versioniere alles
- Wenn es wehtut, mach es häufiger
  - Und identifiziere dadurch die Schmerzursache
- „build quality in“
- Fertig heißt deployed
  - Nicht notwendig in Produktion, aber mindestens auf einem produktionsähnlichen Testsystem

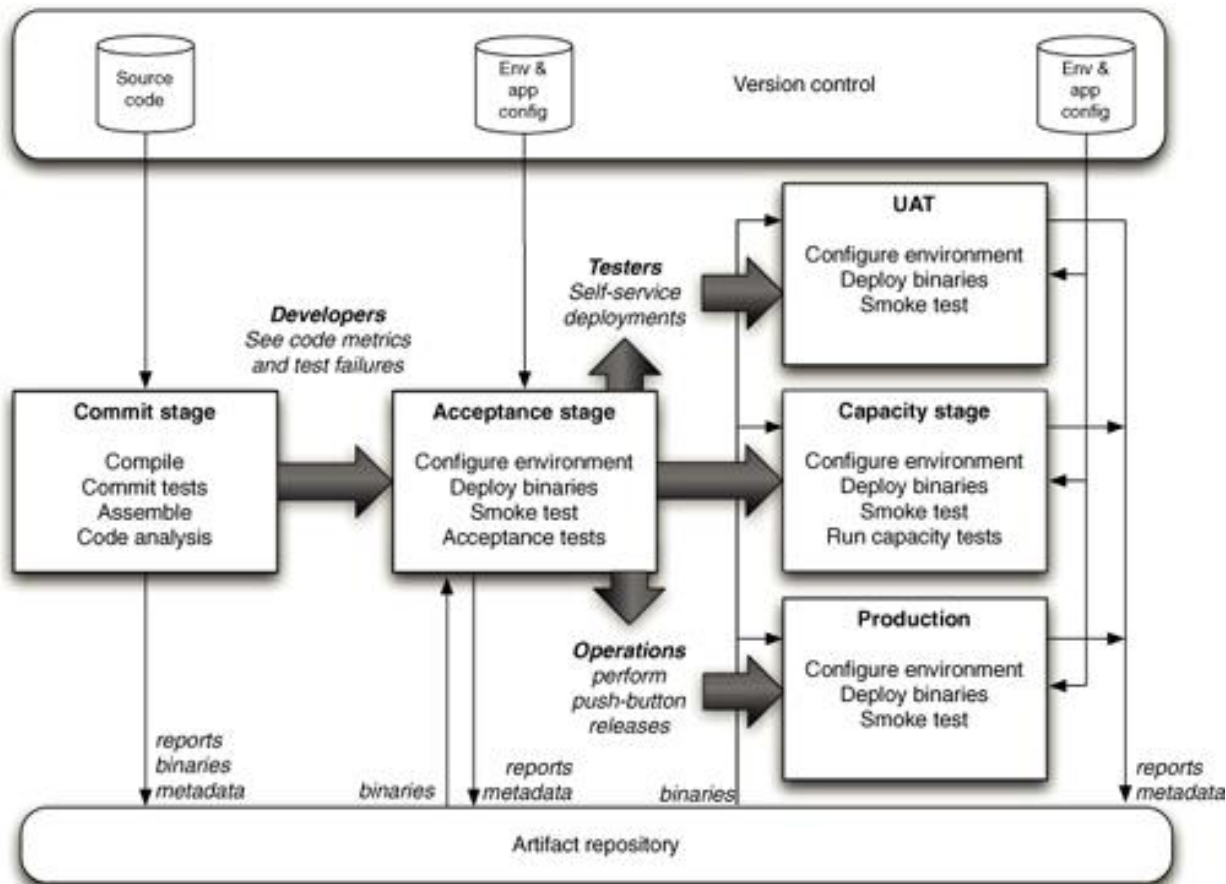
# Continuous Delivery Prinzipien

---

- Jeder ist für den Auslieferungsprozess mitverantwortlich
  - DevOps!
- Verbessere den Prozess kontinuierlich
  - Lean! DevOps!

# Die Deployment Pipeline

## der Kern von Continuous Delivery



# Die Deployment Pipeline

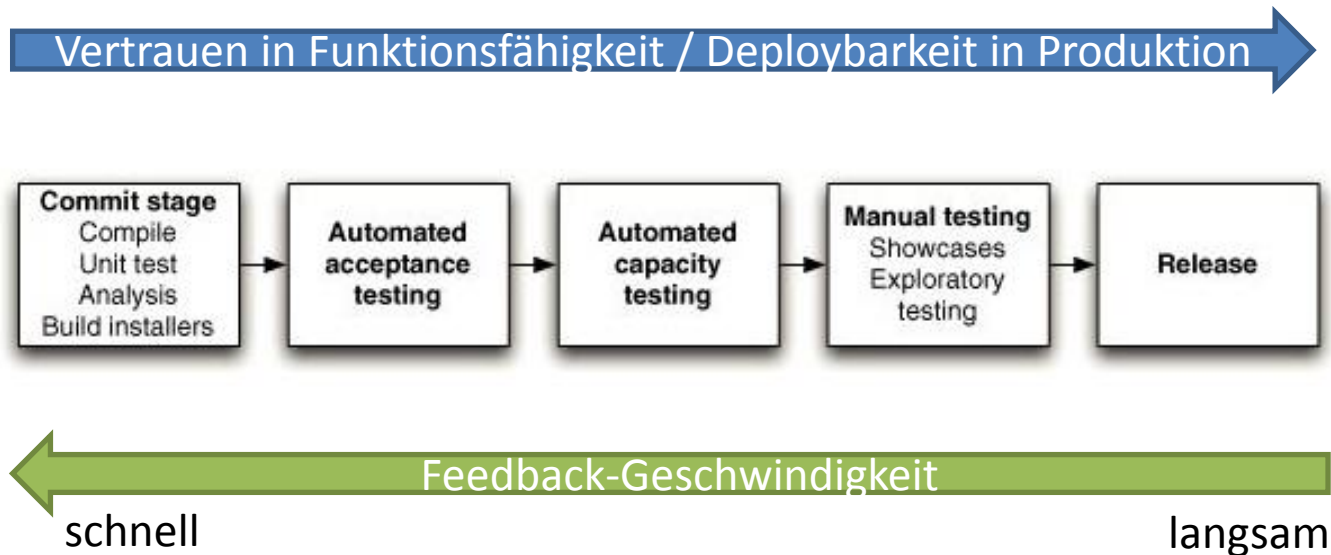
der Kern von Continuous Delivery

---

- Der Code durchläuft mehrere Phasen (Stages)
- Jede Phase bildet ein „Quality Gate“
- Mit jeder erfolgreich abgeschlossenen Phase steigt die Wahrscheinlichkeit, dass die Anwendung bereit für die Produktion ist

# Die Deployment Pipeline

der Kern von Continuous Delivery





# Die Deployment Pipeline

## Bezug zu Lean Manufacturing

---

### **Jidoka (Stop the line):**

- Intelligente Automation
- Prozess bricht selbst ab, wenn ein Fehler auftritt
- Kfr. höhere Kosten, langfristig niedrigere

### **Poka Yoke („unglückliche Fehler vermeiden“):**

- Automatische Mechanismen zu Aufdeckung und Verhinderung von Fehlern

# Die Deployment Pipeline

## Richtlinien

---

- Alle **Artefakte**(deliverables) **werden** in der Commit-Phase **genau ein Mal** gebaut
- Verwende die **gleiche Deployment-Strategie** und Tools für alle Zielumgebungen
- Führe nach einem Deployment **Smoke-Tests** durch
- Führe alle Tests in **produktionsähnlichen Umgebungen** durch

# Die Deployment Pipeline

## Richtlinien

---

- Jede **Änderung** läuft **sofort** durch die Pipeline
- Tritt ein **Fehler** auf, wird die Pipeline **gestoppt**
- Jeder Build schlägt fehl oder produziert einen **Release-Candidate**

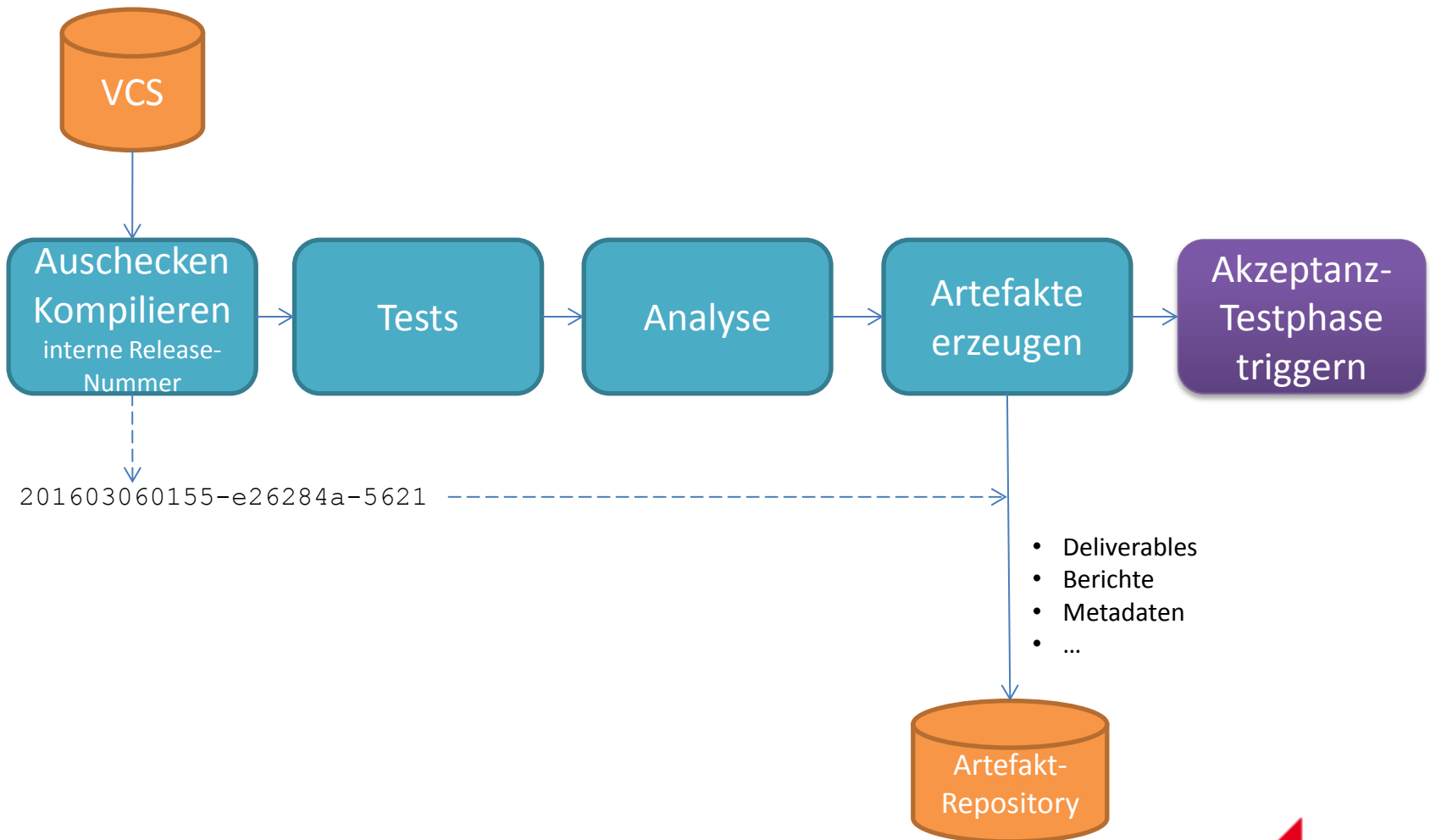
# Commit-Phase

---

## Aufgaben:

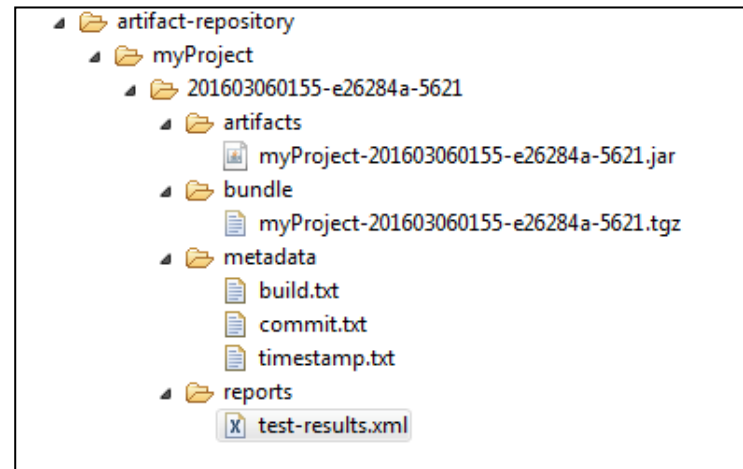
- Kompilieren des Quellcodes (wenn nötig)
- Automatisierte Tests
- Automatisierte statische Analyse
- Eindeutige interne Release-Nummer erzeugen
- Erzeugung aller benötigten Artefakte für die späteren Phasen:
  - Paketierte Anwendung (jar/war/tgz/dll/...)
  - Dokumentation
  - Weitere (Testdaten, ...)

# Commit-Phase



# Artefakt-Repository

- Speichert alle in der Commit-Phase erzeugten Artefakte
- Spätere Phasen können über eindeutige ID auf Artefakte zugreifen
- Einfache Verzeichnisstruktur auf Fileserver/NAS meist ausreichend
- Backup!
- Regelmäßig aufräumen



# Commit-Phase Empfehlungen

---

- Siehe Vorlesung „Continuous Integration“
- Schnelles Feedback (< 10 min)
- Eindeutige, interne Release-Nummer erzeugen (z.Bsp. EnvInject-Plugin für Jenkins)

```
def version = datestring + "-" + commit + "-" + BUILD_NUMBER;  
//201603060155-e26284a-5621  
return [INTERNAL_VERSION: version]
```

# Tests in der Commit-Phase

## Empfehlungen

---

- Hauptsächlich Unit-Tests, Abdeckung  $\geq 80\%$
- Integrationstests
- Ressourcen mocken (bspw. DB)
- Ggf. wenige, ausgewählte Akzeptanztests oder Lasttests



# Akzeptanztest-Phase

---

## Aufgaben:

- Aufsetzen und Konfigurieren einer oder mehrerer Zielumgebungen
- Deployment der Artefakte in den Zielumgebungen
- Ausführen der Tests gegen Zielumgebungen
  - Smoke Tests
  - Akzeptanztests

# Klassifikation von Tests

---

- Unit-Tests
- Integrations-Tests
- Akzeptanztests
- Smoke-Tests

siehe Vorlesung „Unit Tests mit JUnit“

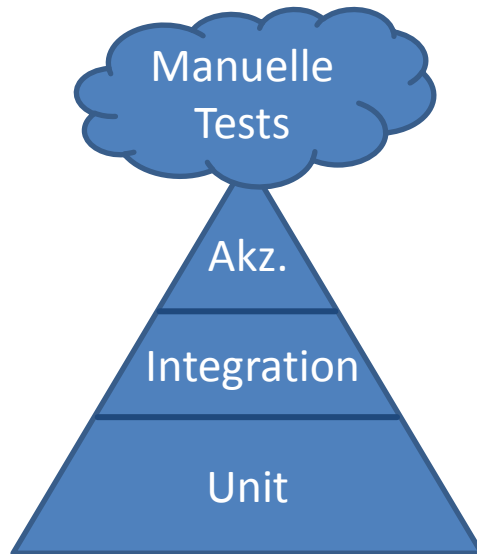
# Smoke Test

- Oberflächliche Prüfung
- Testet, ob etwas grundsätzlich funktioniert, bevor weitere Detailprüfungen vorgenommen werden
- Beispiel:
  - Nach automatischem Deployment in Zielumgebung prüfen, ob Anwendung reagiert, zum Beispiel durch einfachen Request
  - Bei Fehlschlag muss keine Testsuite gegen die Zielumgebung ausgeführt werden

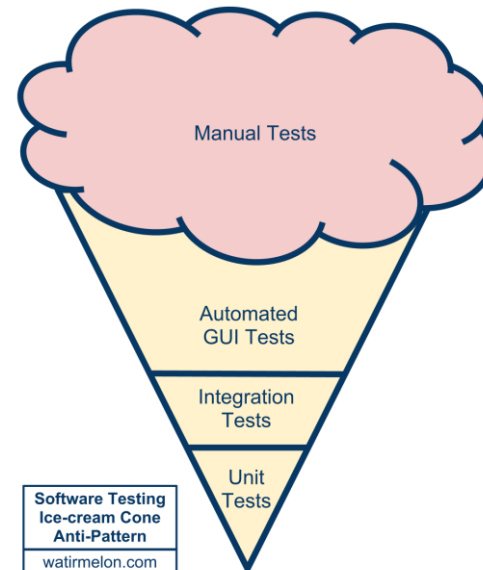


# Akzeptanztest-Phase Empfehlungen

- Verteilung von Tests



Test-Pyramide nach Cohn



**Antipattern!!**

# Akzeptanztest-Phase

## Empfehlungen

---

- Autom. Akzeptanztests sind teuer in Erstellung und Unterhalt
- Es kann sinnvoll sein, keine automatisierten Akzeptanztests zu haben
  - Dann sollten zumindest alle manuellen Tests genau dokumentiert und protokolliert werden
- Wenn es autom. AT gibt: Mindestens den „Happy Path“ der Features testen

# Akzeptanztest-Phase

## Empfehlungen

---

- Produktionsähnliche Umgebungen/Bedingungen verwenden
  - Tests mit mehreren Browsern
  - Tests mit unterschiedlichen Betriebssystemen

# Weitere Test-Phasen

---

Bei Bedarf können der AT-Phase weitere Phasen folgen (sequentiell und/oder parallel).

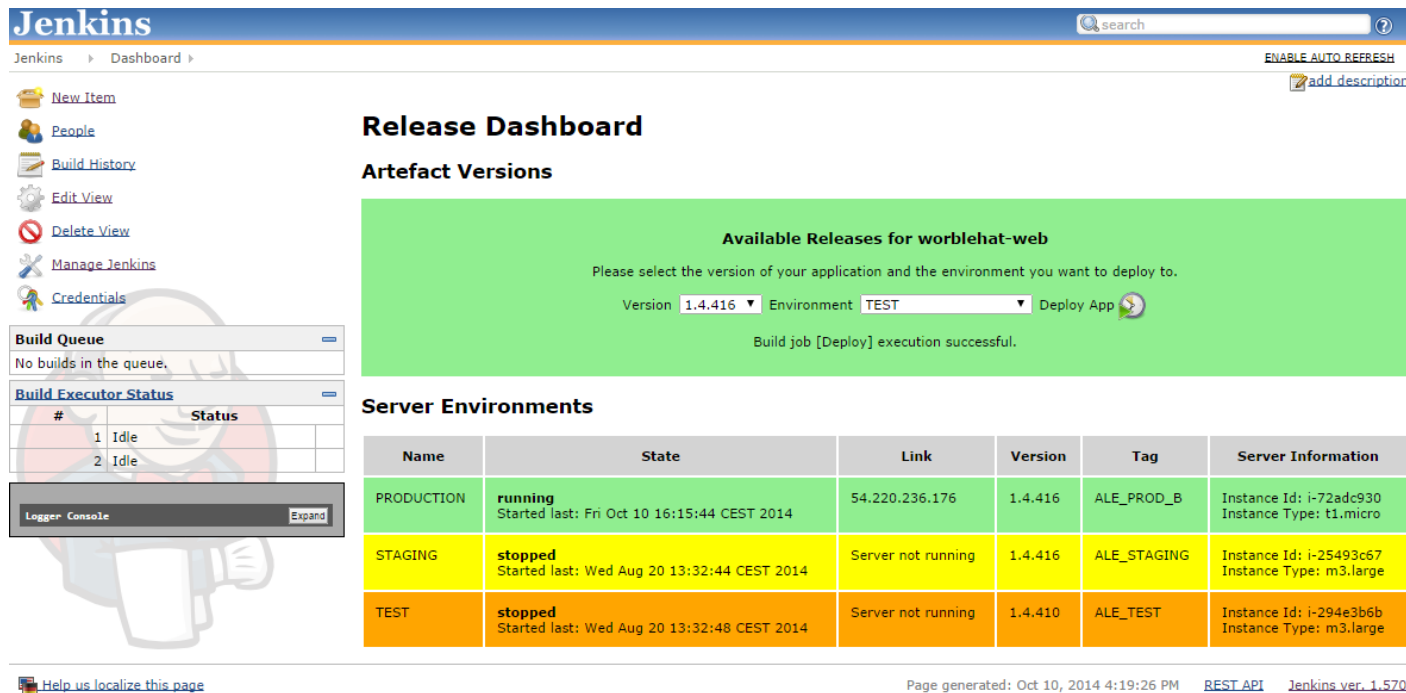
Diese können automatisiert oder Benutzer-gesteuert sein.

Beispiele:

- Exploratory Testing
  - Tester wählt einen release candidate und lässt diesen automatisiert auf einem Zielsystem deployen
- User Acceptance Tests
  - release candidate wird automatisch auf staging-System deployed
- Nichtfunktionale Tests
- Lasttests
- Sicherheitstests

# Deployment in Produktion

- Wird manuell ausgelöst (per Klick auf gewünschtes Release)
- -> aber automatisiert abgewickelt!



The screenshot shows the Jenkins Release Dashboard for 'worblehat-web'. It includes a sidebar with navigation links, a 'Build Queue' section, and a 'Logger Console' at the bottom. The main content area displays 'Artefact Versions' with a green box for available releases and a table for 'Server Environments'.

**Jenkins** search ENABLE AUTO REFRESH add description

Jenkins > Dashboard

**Release Dashboard**

**Artefact Versions**

**Available Releases for worblehat-web**

Please select the version of your application and the environment you want to deploy to.

Version **1.4.416** Environment **TEST** Deploy App

Build job [Deploy] execution successful.

**Server Environments**

Name	State	Link	Version	Tag	Server Information
PRODUCTION	running Started last: Fri Oct 10 16:15:44 CEST 2014	54.220.236.176	1.4.416	ALE_PROD_B	Instance Id: i-72adc930 Instance Type: t1.micro
STAGING	stopped Started last: Wed Aug 20 13:32:44 CEST 2014	Server not running	1.4.416	ALE_STAGING	Instance Id: i-25493c67 Instance Type: m3.large
TEST	stopped Started last: Wed Aug 20 13:32:48 CEST 2014	Server not running	1.4.410	ALE_TEST	Instance Id: i-294e3b6b Instance Type: m3.large

[Help us localize this page](#) Page generated: Oct 10, 2014 4:19:26 PM [REST API](#) [Jenkins ver. 1.570](#)



# Deployment Antipatterns

---

- Dokumentation anstatt Automatisierung
- Unterschiedliche Deployment-Tools für unterschiedliche Umgebungen
- Manuelle Rollbacks
- Manuelle Änderungen an Produktivumgebung möglich

# Deployment in Produktion

## Best Practices

---

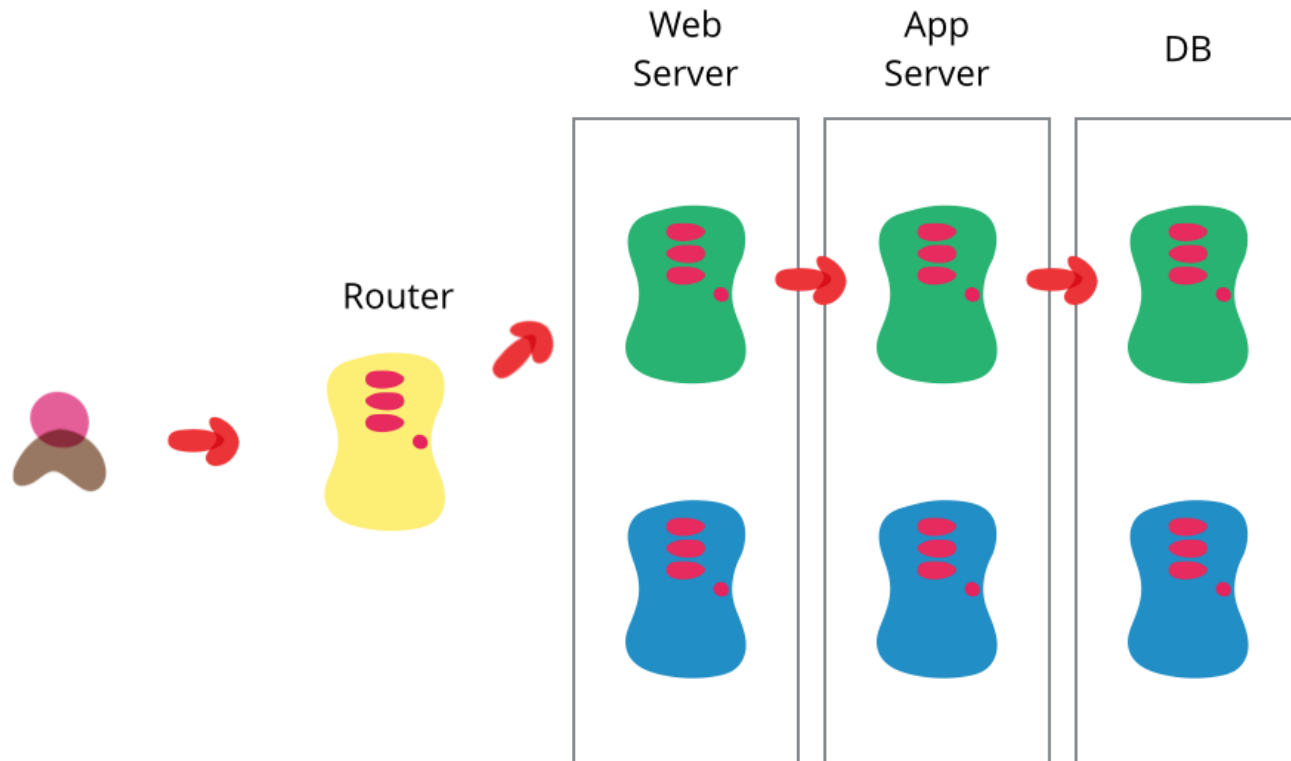
- Automatisierter Rollback-Plan
  - Backup
  - Re-Deploy des zuletzt funktionierenden Releases
- Blue-Green-Deployments
- Canary Releases
- Feature Toggles
- Parallel Code Paths

# Blue-Green-Deployments

---

- Voraussetzung: es existieren min. zwei identische, getrennte Produktiv-Umgebungen („blau“ und „grün“)
- Eine Umgebung ist produktiv, die andere das Staging-System
- Vorgehen:
  - Deployment in „blaue“ Umgebung
  - Umleitung des Verkehrs nach blau
  - „blau“ ist die neue Live-Umgebung
  - „grün“ wird die neue Staging-Umgebung
  - Nächstes Mal: Deployment nach „grün“
  - ...
- Erlaubt schnelles Rollback (wenn Deploy in grün schief läuft -> zurück auf blau schalten und umgekehrt)
- Benötigt zwei Mal identische Ressourcen -> doppelte Kosten

# Blue-Green-Deployments



<http://martinfowler.com/bliki/BlueGreenDeployment.html>

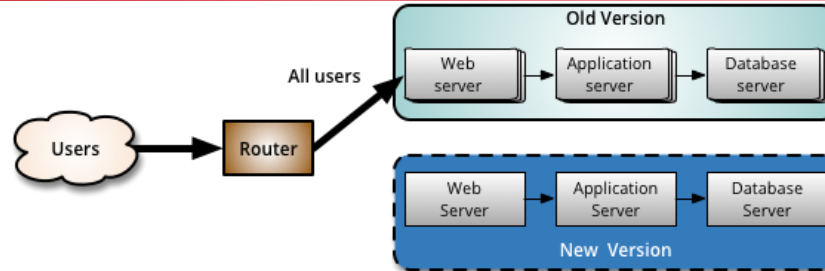
# Canary Releases

---

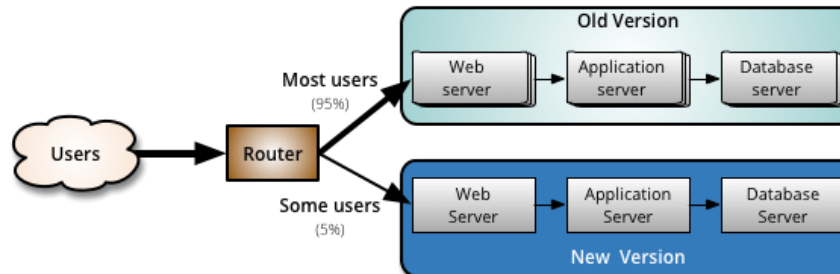
- Ähnlich wie Blue-Green-Deployments
- Beide Versionen der Anwendung sind gleichzeitig im Betrieb
- Schritt für Schritt werden mehr User auf die neue Version geleitet
- Last kann langsam erhöht werden
- Schnelles Rollback möglich
- Höhere Kosten für Infrastruktur
- Benötigt zuverlässigen, kontrollierbaren Routing-Mechanismus

# Canary Releases

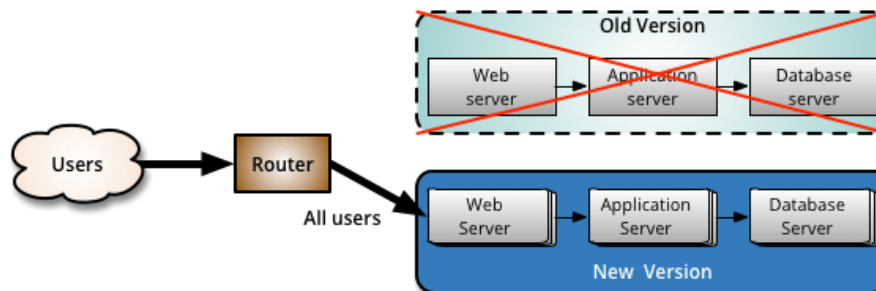
1.



2.



3.



# Feature Toggles

- a.k.a. „Dark Launching“ (facebook)
- Anwendung enthält „Schalter“, um gezielt bestimmte Funktionen zu aktivieren
- Mögliche Lösung, wenn alle im Main-Branch arbeiten (also keine Feature-Branches verwendet werden)
- Problem:
  - Kann schnell unübersichtlich/vergessen werden
  - „Frankensteins and Zombies everywhere“
  - Schafft eventuell mehr Probleme als es löst
- möglichst vermeiden (es sei denn, Sie arbeiten bei facebook, spotify, google, ...)
- Besser: Features in kleine Einheiten aufteilen

*Release toggles are a useful technique and lots of teams use them. However they should be your last choice when you're dealing with putting features into production.*

*Your first choice should be to break the feature down so you can safely introduce parts of the feature into the product.*

*-Martin Fowler*

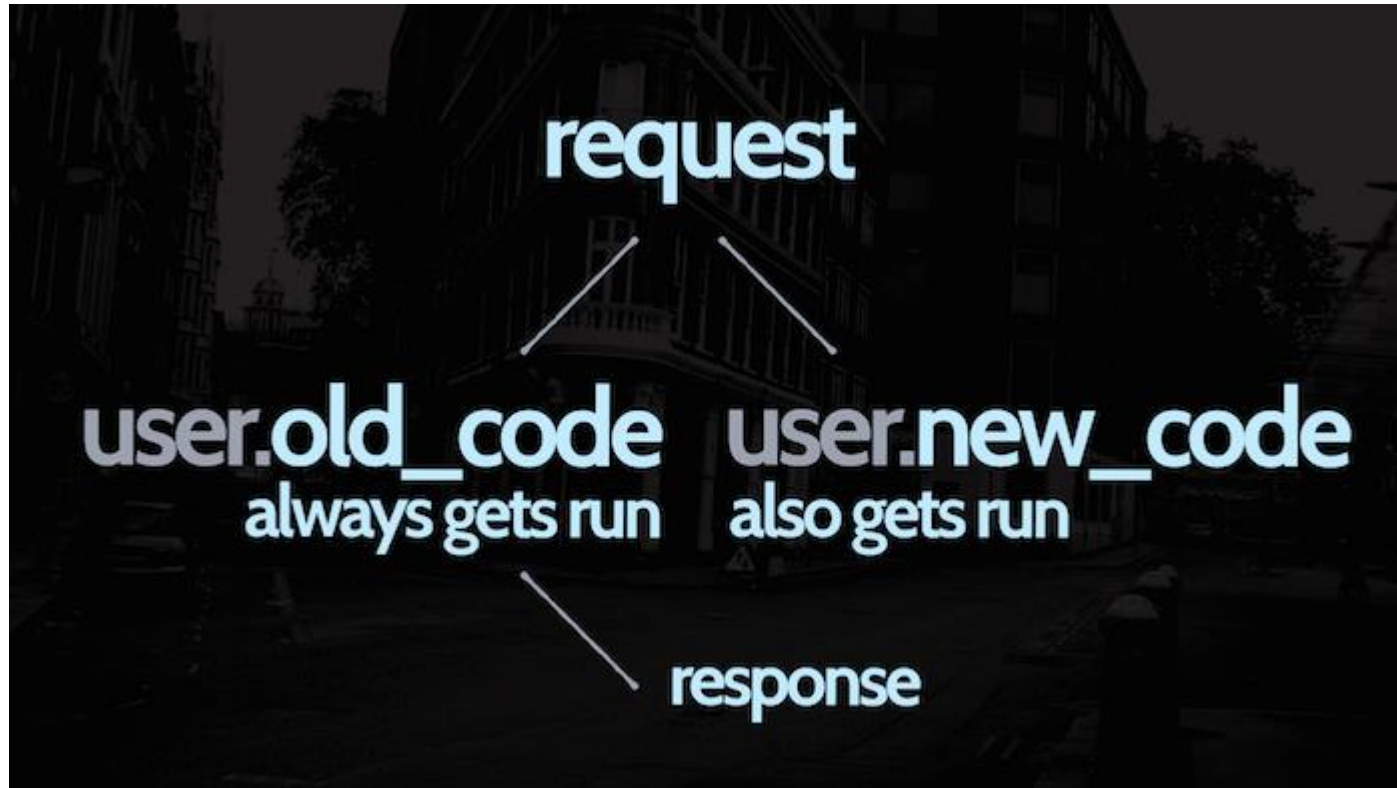
# Parallel Code Paths

---

- Alter und neuer Code werden parallel ausgeführt
- Ergebnisse werden verglichen
- Bei Abweichung: Meldung/Logging
- Wenn keine Abweichungen mehr auftreten, kann der „alte“ Code entfernt werden
- Sehr nützlich zum Prüfen neuer Funktionen/Backing Services/Refactoring
- Siehe <https://github.com/github/scientist>



# Parallel Code Paths



<https://zachholman.com/images/talks/break-nothing>

# Parallel Code Paths

---



```
science "new-auth" do |e|  
  e.control { user.slow_auth }  
  e.candidate { user.fast_auth }  
end
```

always runs and  
returns your result

<https://zachholman.com/talk/move-fast-break-nothing/>

# Parallel Code Paths

```
science "new-auth" do |e|  
  e.control { user.slow_auth }  
  e.candidate { user.fast_auth }  
end
```

can be run as  
a percentage

<https://zachholman.com/talk/move-fast-break-nothing/>

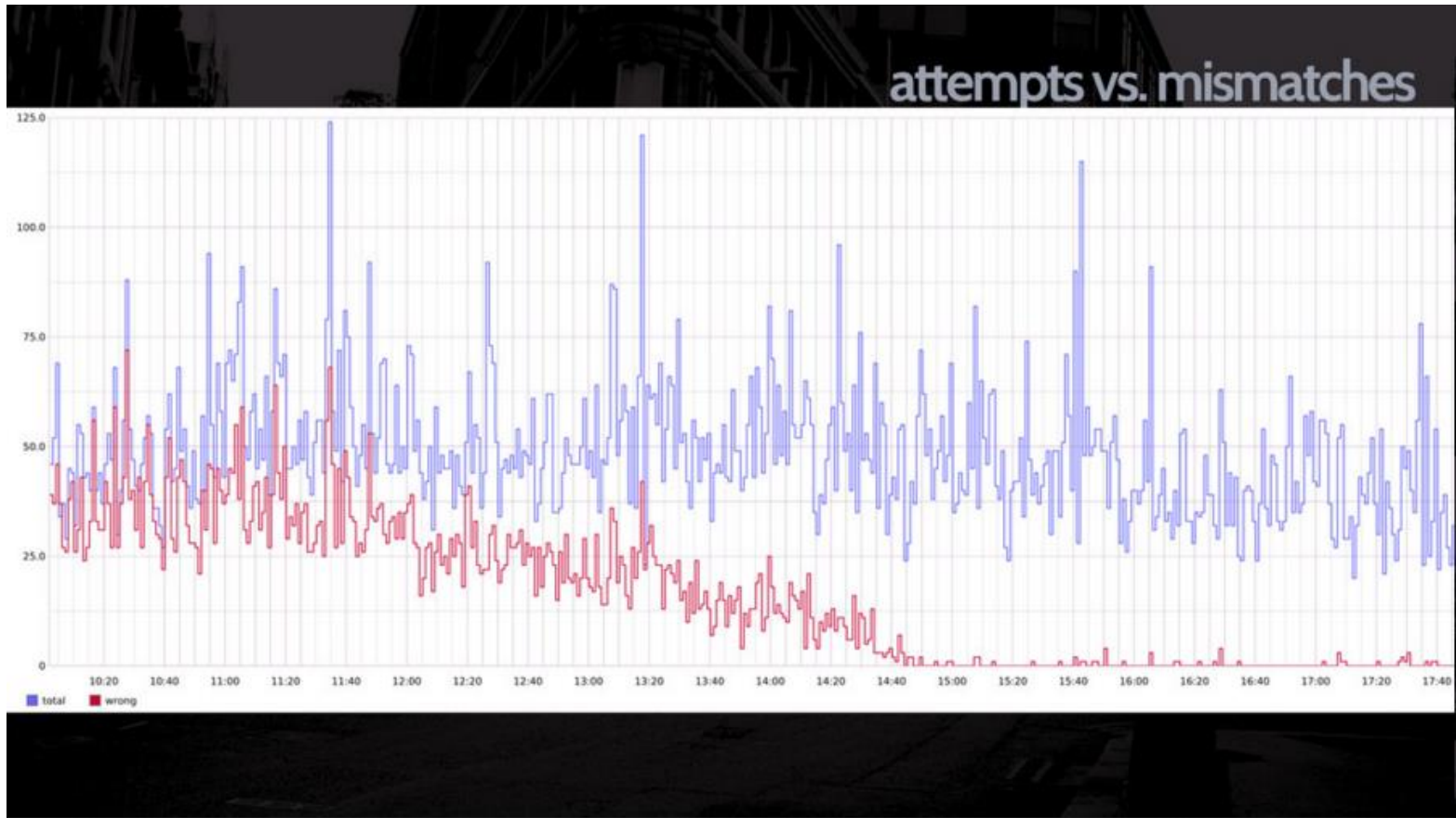
# Parallel Code Paths

```
science "new-auth" do |e|  
  e.control { user.slow_auth }  
  e.candidate { user.fast_auth }  
end
```

collects the  
two results

<https://zachholman.com/talk/move-fast-break-nothing/>

# Parallel Code Paths



<https://zachholman.com/talk/move-fast-break-nothing/>

# Continuous Delivery Tools

---

- Infrastructure as Code-Tools
- Virtualisierung (Vagrant, Docker, ...)
- Build-Tools (maven, ant, cmake, phing, rake, shell, ...)
- Dependency-Management (maven, composer, ...)
- Deployment-Tools (Capistrano, build master, ...)

# Demo

---

- Jenkins mit Pipeline-Plugin
- <https://wiki.jenkins-ci.org/display/JENKINS/Pipeline+Plugin>



# Build Pipeline Plugin

The screenshot displays the Jenkins web interface for the Build Pipeline Plugin. The browser address bar shows `localhost:8082/view/Build%20pipeline/`. The Jenkins header includes a search bar and user information (marcinp | log out). The main section is titled "Build Pipeline: My pipeline" and features a toolbar with icons for Run, History, Configure, Add Step, Delete, and Manage. Two pipeline versions are shown:

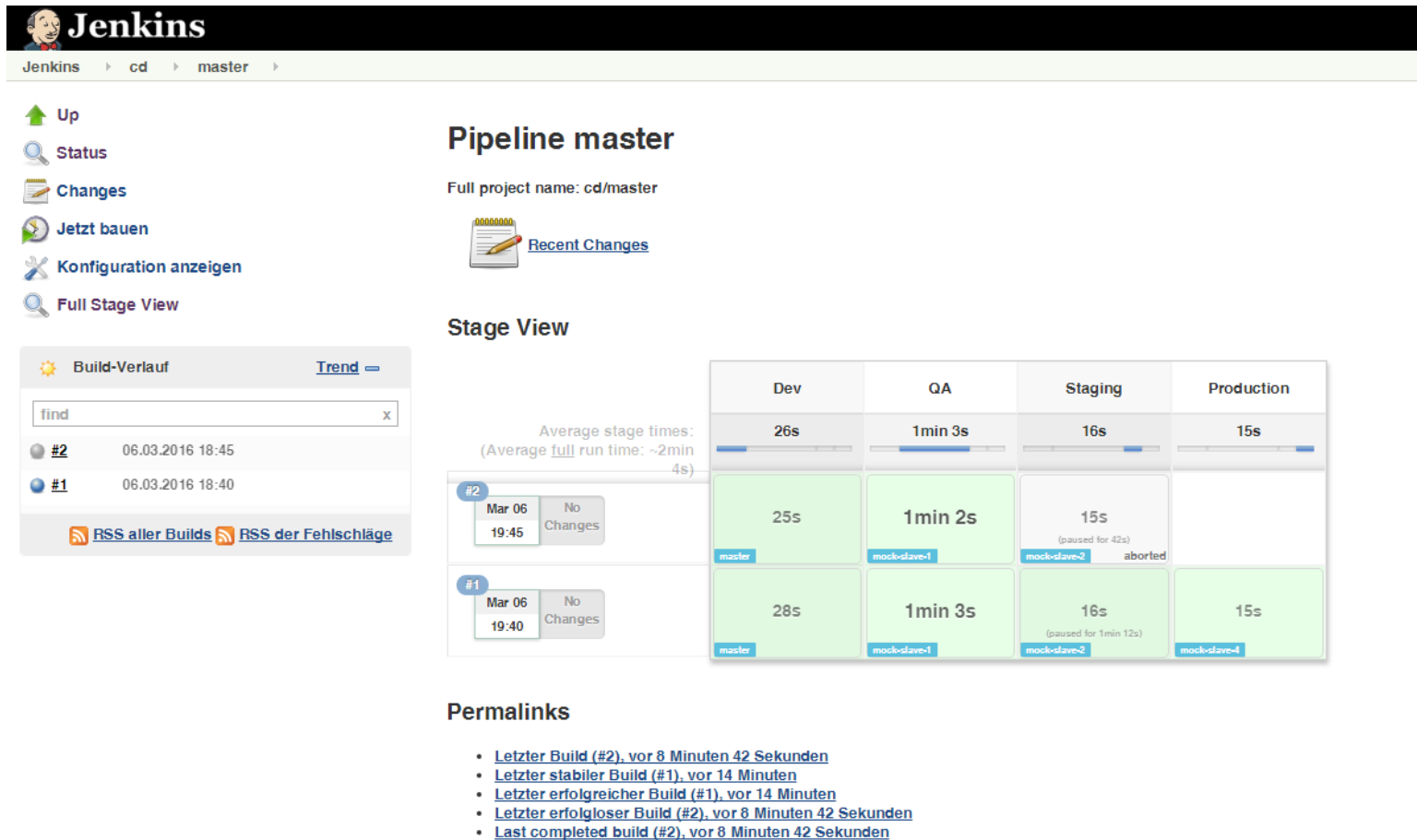
- Pipeline version 8:** No parameters. Steps include Test (Jun 26, 2012 5:30:48 PM, 8 min 10 sec), Release (Jun 26, 2012 5:31:03 PM, 8 min 12 sec), Deploy to Test (Jun 26, 2012 5:31:46 PM, 6 min), Generate docs (Jun 26, 2012 5:31:20 PM, 8 min 9.1 sec), Deploy to Pre-Prod, and Deploy to Prod.
- Pipeline version 7:** No parameters. Steps include Test (Jun 26, 2012 5:26:25 PM, 7 min 10 sec), Release (Jun 26, 2012 5:26:40 PM, 7 min 12 sec), Deploy to Test (Jun 26, 2012 5:27:13 PM, 5 min 13 sec), Generate docs (Jun 26, 2012 5:26:57 PM, 7 min 9.1 sec), Deploy to Pre-Prod (Jun 26, 2012 5:27:31 PM, 4 min 10 sec), and Deploy to Prod (Jun 26, 2012 5:27:46 PM, 4 min 15 sec).

At the bottom, there is a link to "Help us localize this page" and a footer indicating "Page generated: 26-Jun-2012 17:31:39" and "Jenkins ver. 1.470".

<https://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin>



# Pipeline Plugin (ehemals Workflow Plugin)



The screenshot displays the Jenkins Pipeline Plugin interface. At the top, the Jenkins logo and navigation breadcrumbs (Jenkins > cd > master) are visible. The left sidebar contains links for Up, Status, Changes, Jetzt bauen, Konfiguration anzeigen, and Full Stage View. The main content area is titled 'Pipeline master' and shows the full project name 'cd/master'. Below this, there is a 'Recent Changes' section with a notepad icon. The 'Stage View' section displays a table of stage times for two builds (#2 and #1) across four stages: Dev, QA, Staging, and Production. Build #2 is shown as 'No Changes' and Build #1 as 'No Changes'. The 'Permalinks' section at the bottom provides links to the last build, last stable build, last successful build, last failed build, and last completed build.

**Pipeline master**

Full project name: cd/master

**Stage View**

Average stage times:  
(Average full run time: ~2min 4s)

	Dev	QA	Staging	Production
#2	25s	1min 2s	15s (paused for 42s) aborted	
#1	28s	1min 3s	16s (paused for 1min 12s)	15s

**Permalinks**

- [Letzter Build \(#2\), vor 8 Minuten 42 Sekunden](#)
- [Letzter stabiler Build \(#1\), vor 14 Minuten](#)
- [Letzter erfolgreicher Build \(#1\), vor 14 Minuten](#)
- [Letzter erfolgloser Build \(#2\), vor 8 Minuten 42 Sekunden](#)
- [Last completed build \(#2\), vor 8 Minuten 42 Sekunden](#)

<https://wiki.jenkins-ci.org/display/JENKINS/Pipeline+Plugin>

# Exkurs: The Joel Test

---

## **The Joel Test**

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

<http://www.jelonsoftware.com/articles/fog0000000043.html>

# Schritte zur Deployment Pipeline (Empfehlung)

---

- Value stream identifizieren
- walking skeleton erstellen
- Automate build and deployment
- Automate unit tests and code analysis
- Automate acceptance tests
- Automate releases

# Vorteile von Continuous Delivery

---

- Weniger Risiko durch häufige Deployments
  - Deployment wird von der Ausnahme zur Regel
- Schnelles Feedback von Zielgruppen
- Mehr Vertrauen/Sicherheit in Funktionsfähigkeit der Anwendung
- Schnelle, zuverlässige, wiederholbare Prozesse anstatt veraltete Dokumentation
- Forciert Zusammenarbeit (Dev, Ops, QA, ...)

# Nachteile von Continuous Delivery

---

- Oft große Umstellung für Entwicklungsteams
- Hoher initialer Mehraufwand
- Kann mit hohen Kosten verbunden sein (Akzeptanztests, Hardware, Schulungen, ...)

# Fazit

---

- CD /Deployment Pipeline ist ein Best Practice-Muster
  - Muster sind Vorlagen und müssen angepasst werden
  - **Ihre** Phasen können anders aussehen!

*The key test is that a business sponsor could request that **the current development version of the software can be deployed into production at a moment's notice** - and nobody would bat an eyelid, let alone panic.*

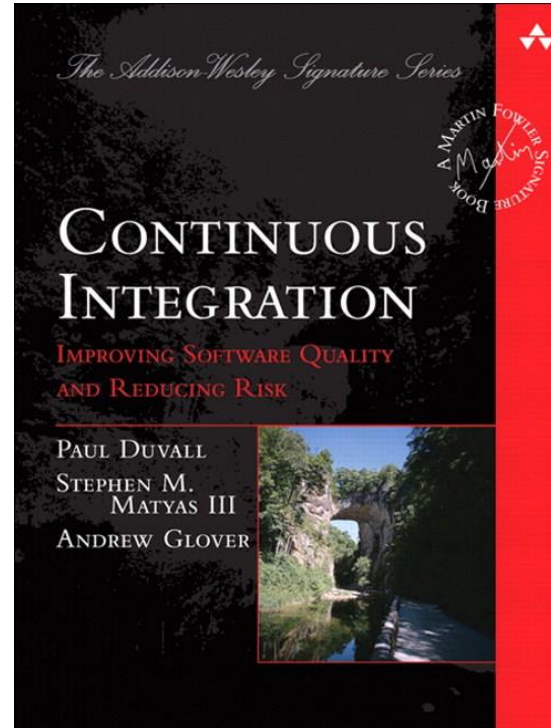
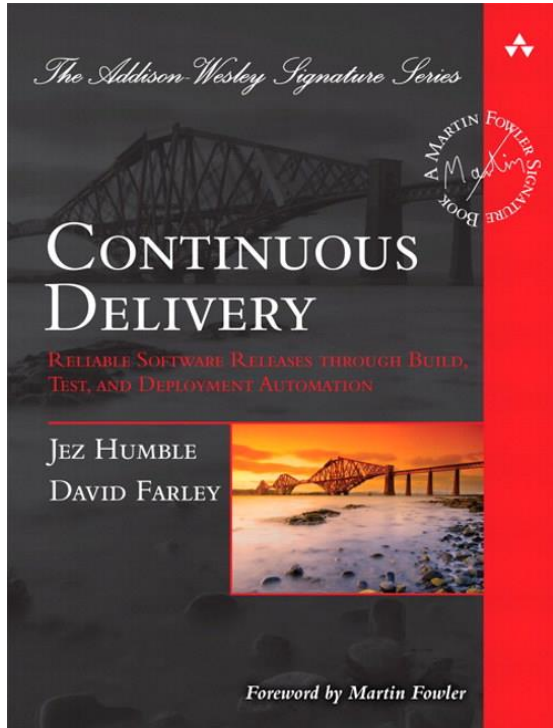
– Martin Fowler

# Ausblick

---

Jenkins 2.0

# Literatur





# Weiterführende Links

---

- <http://www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-1-Grundlagen-2176380.html>
- <http://www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-2-Commit-Stage-2286071.html>
- <http://www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-3-Acceptance-Test-Stage-2457023.html>
- <http://www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-4-Bereitstellen-der-Infrastruktur-2529341.html>