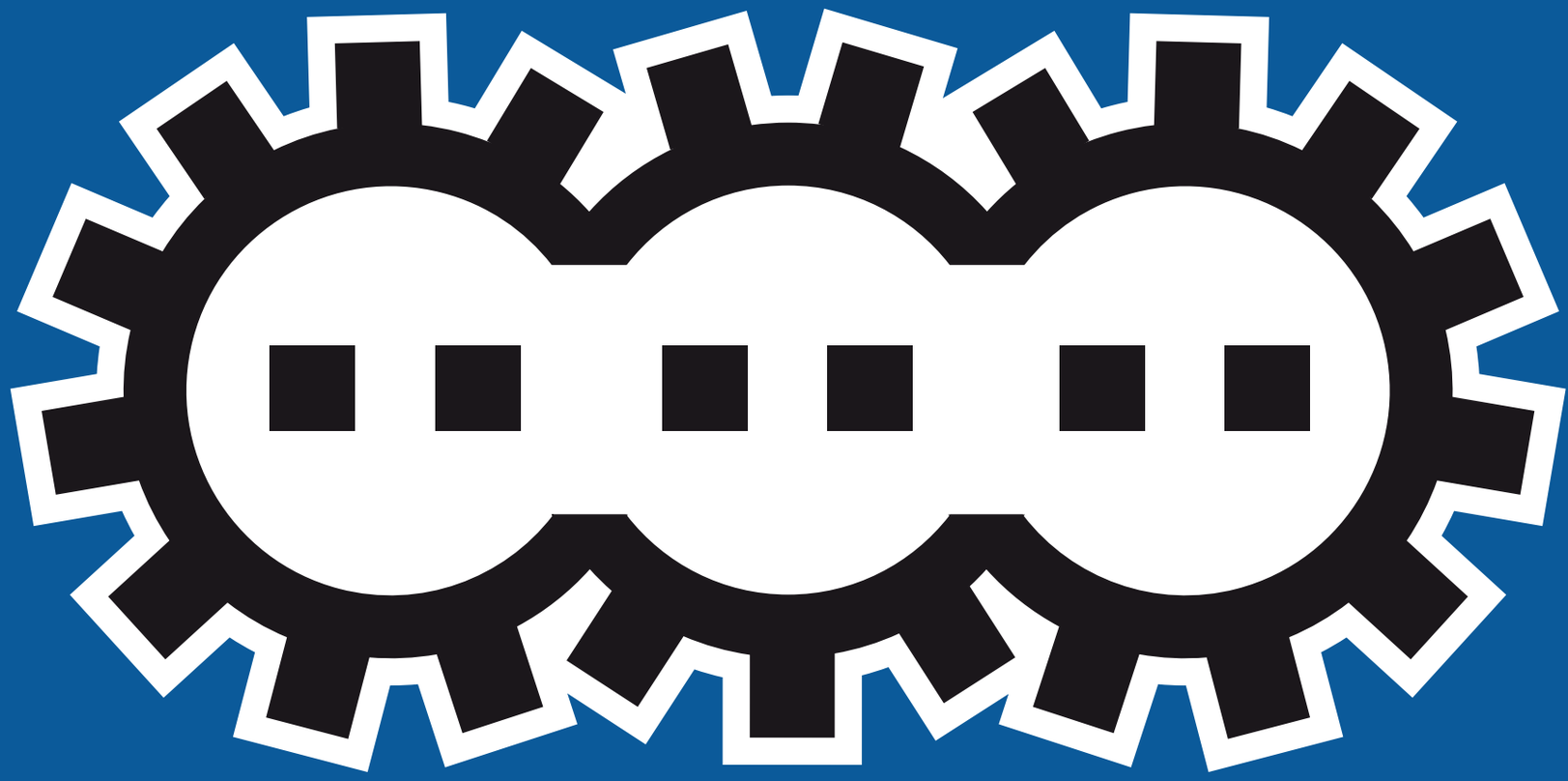
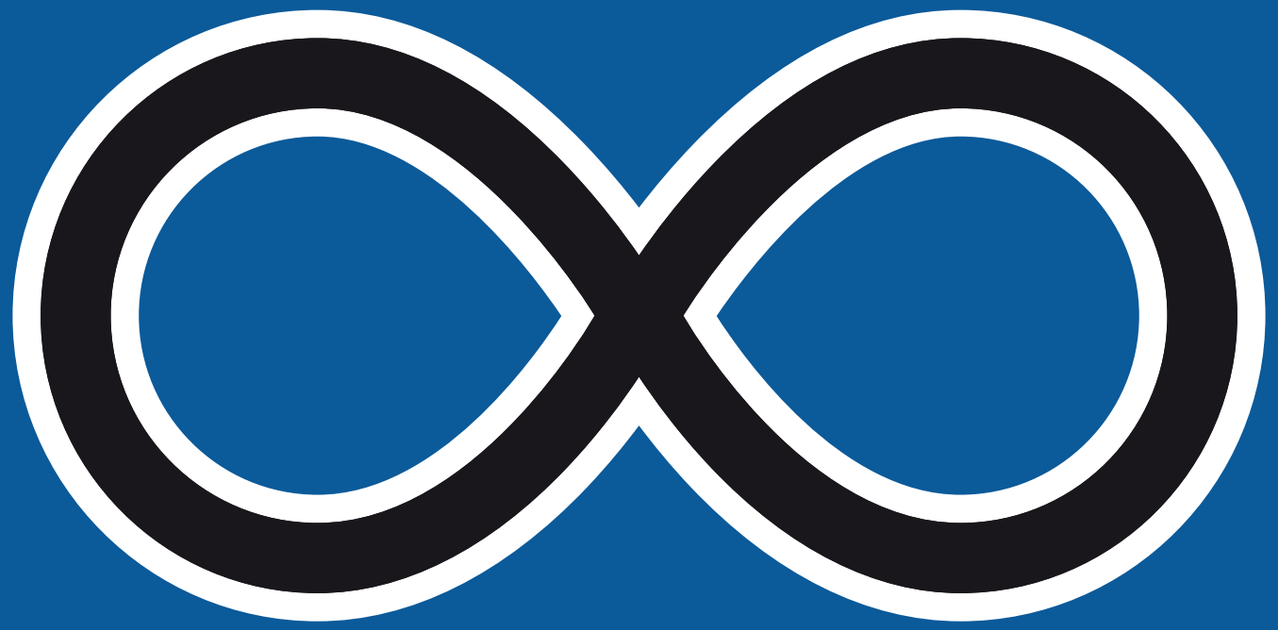


SOFTWAREENTWICKLUNG

IM TEAM MIT OPEN-SOURCE-WERKZEUGEN

10 - Continuous Delivery

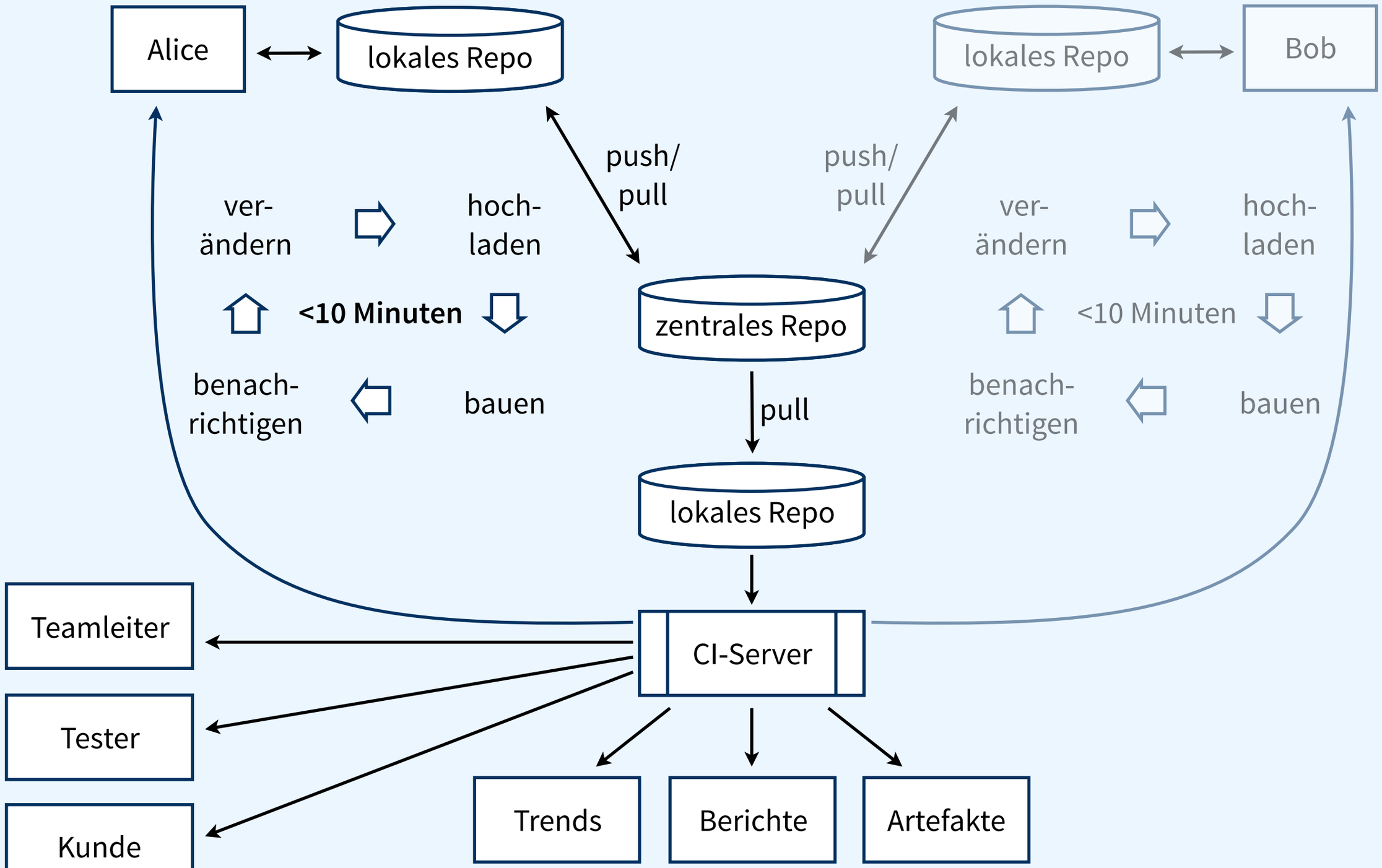
Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen



Wiederholung

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Vogelperspektive



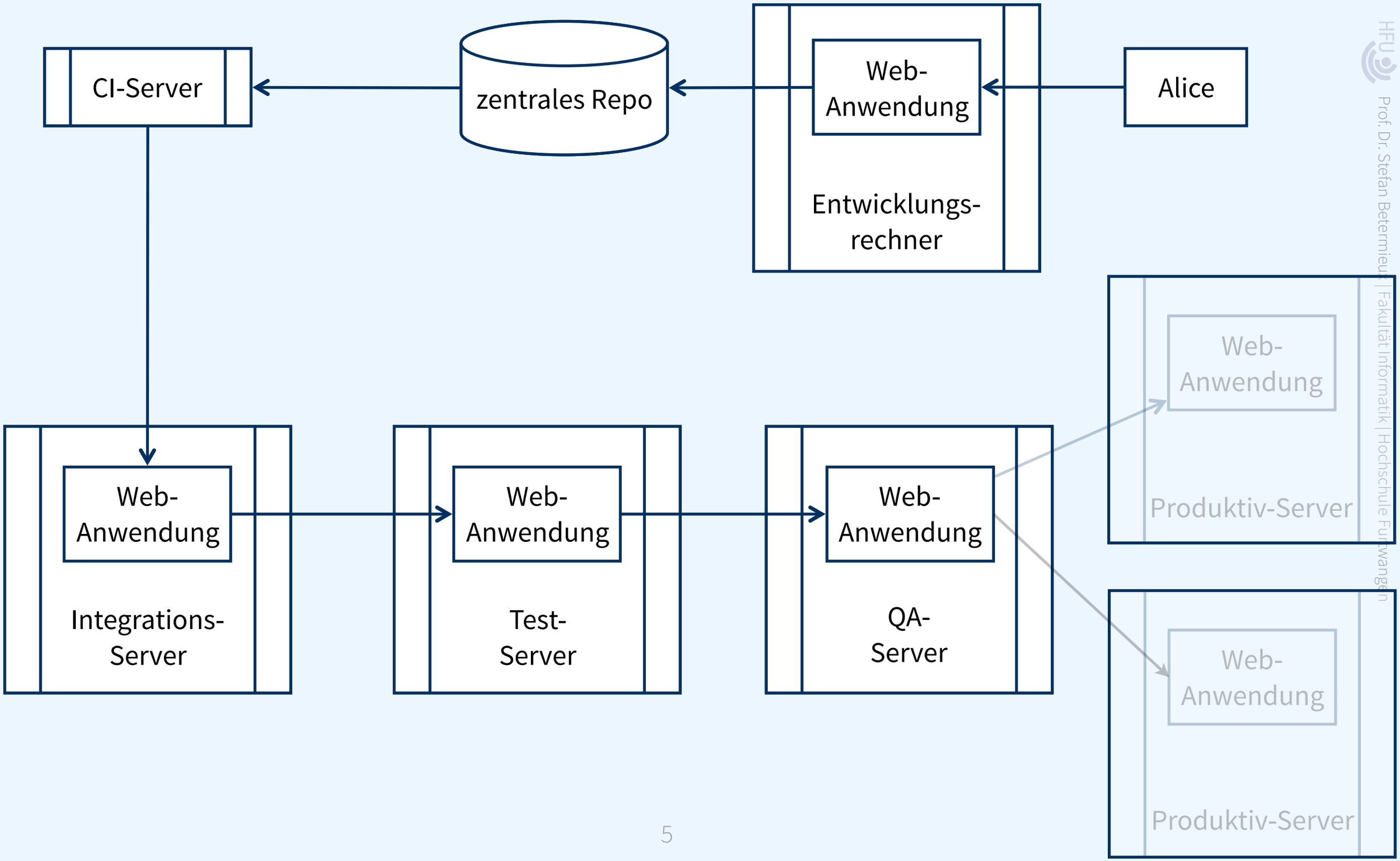
Was ist CI nicht

- Keine Programmiersprache → der CI-Server startet aber eventuell einen externen Compiler
- Kein Build-Werkzeug → der CI-Server startet aber eventuell ein externes Werkzeug
- Keine Versionsverwaltung → der CI-Server fragt aber eventuell bei einem nach aktuellen Änderungen
- Kein Test-Framework
- Kein Artefakt-Repository für erzeugte Artefakte
- Kein einzelnes Produkt
- Keine markengeschützte Methode

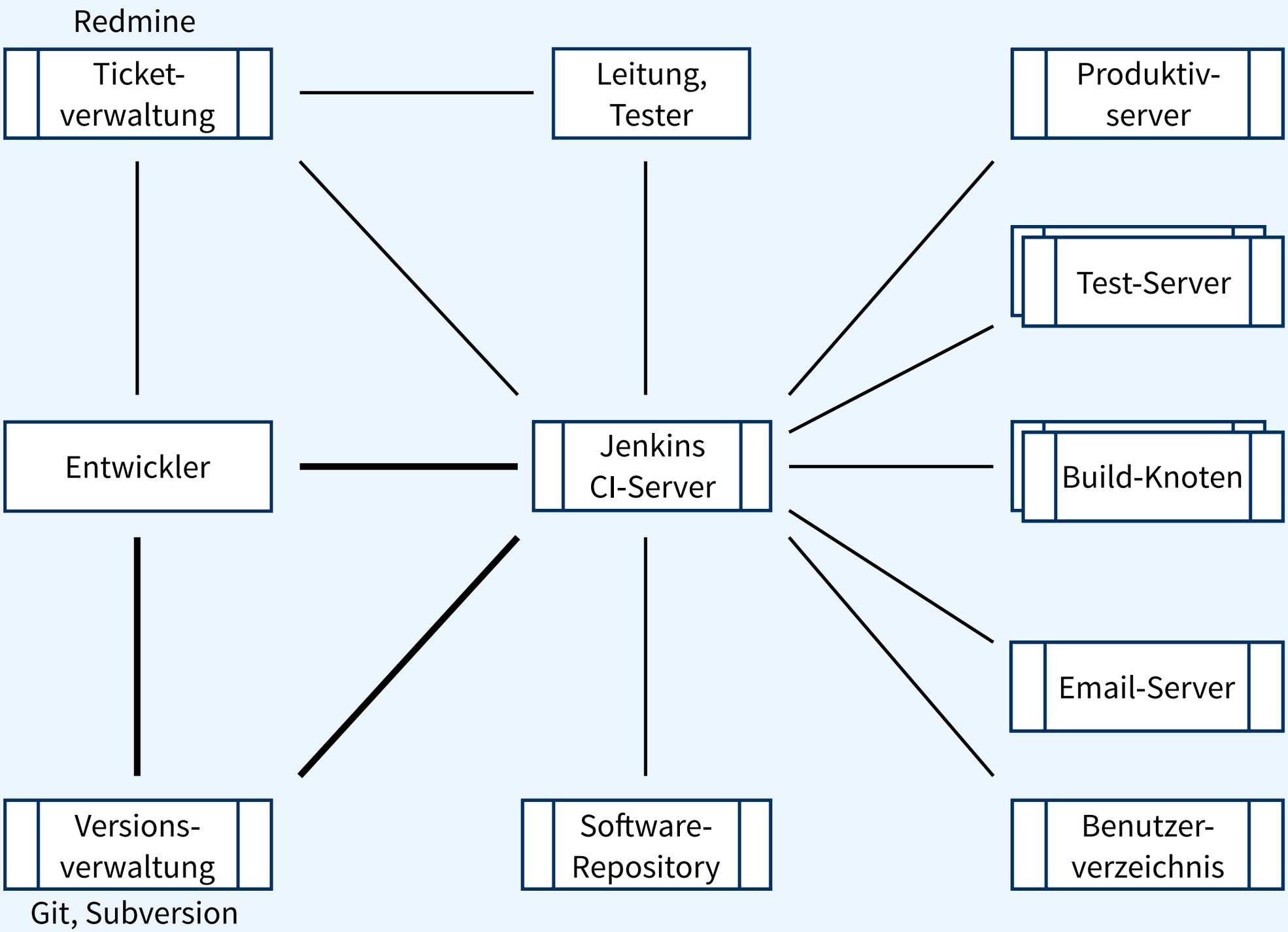
➡ Continuous Integration ist der Dirigent,
der das »Werkzeug-Orchester« leitet!

Quelle: Simon Wiest, Continuous Integration

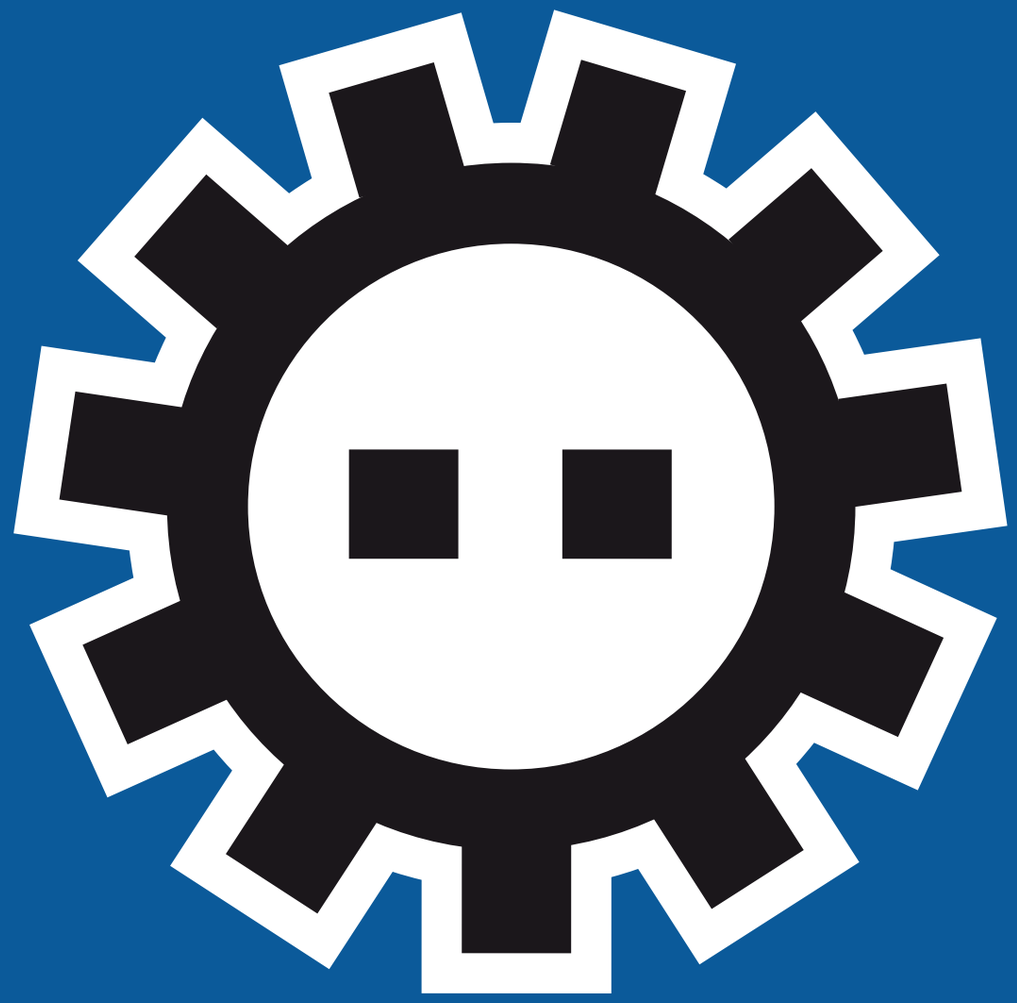
Deployment



Jenkins Systemlandschaft



Quelle: Simon Wiest, Continuous Integration



Motivation

Motivation

- Mit der Continuous Integration können wir Software automatisch bauen und testen
  - schnelle Feedback-Zyklen (<10 Minuten)
- Es fehlt aber noch die Möglichkeit, gebaute und getestete Software automatisch zu installieren (deployment)
  - bisher oft manuell und aufwändig (Stunden oder Tage)

Vergleich

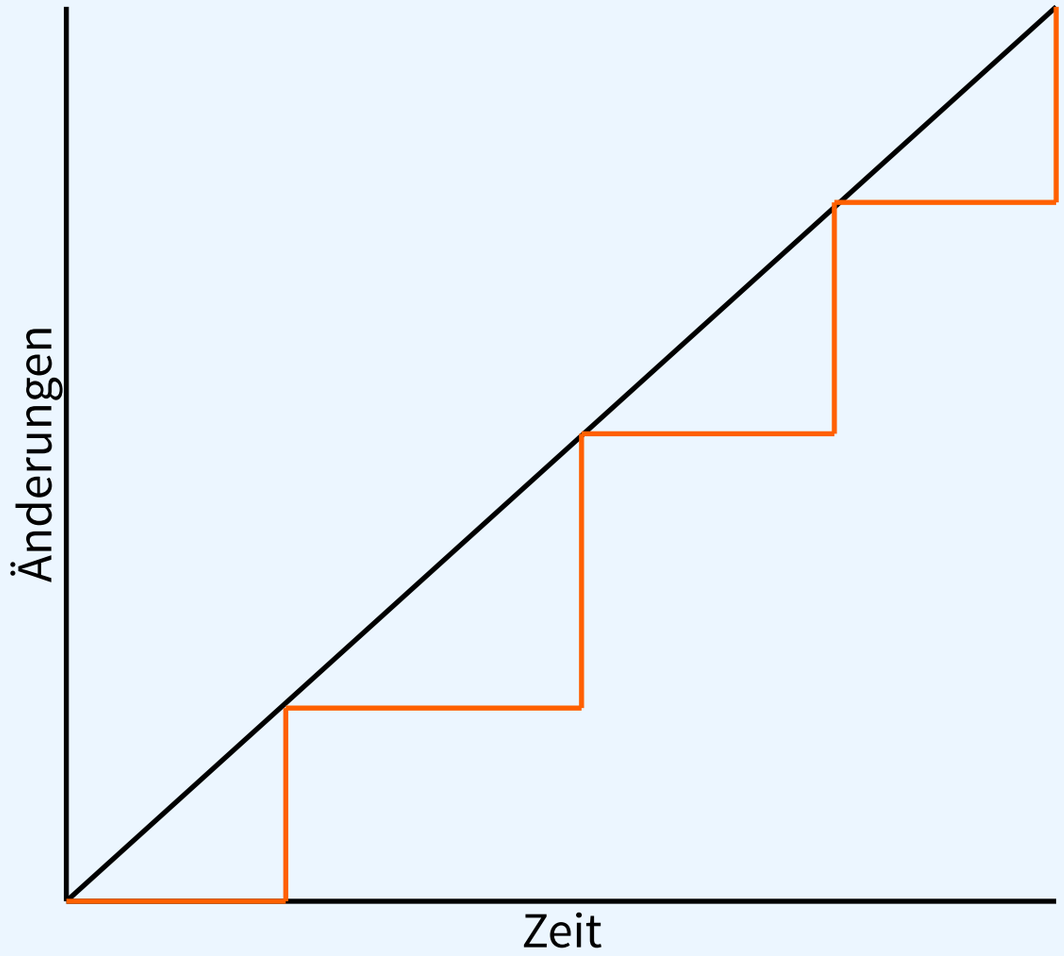
Wo der Fehler erkannt wurde:	Zeit, die benötigt wurde, um Fehler zu korrigieren:
Code Review	1 Stunde
Unit Testing	3 Stunden
Integration Testing	12 Stunden
Beim Kunden	40 Stunden

- Je schneller ein Fehler gefunden wird, desto kürzer ist die benötigte Zeit, den Fehler zu beheben!
- Gilt in ähnlicher Form auch für neue Anforderungen

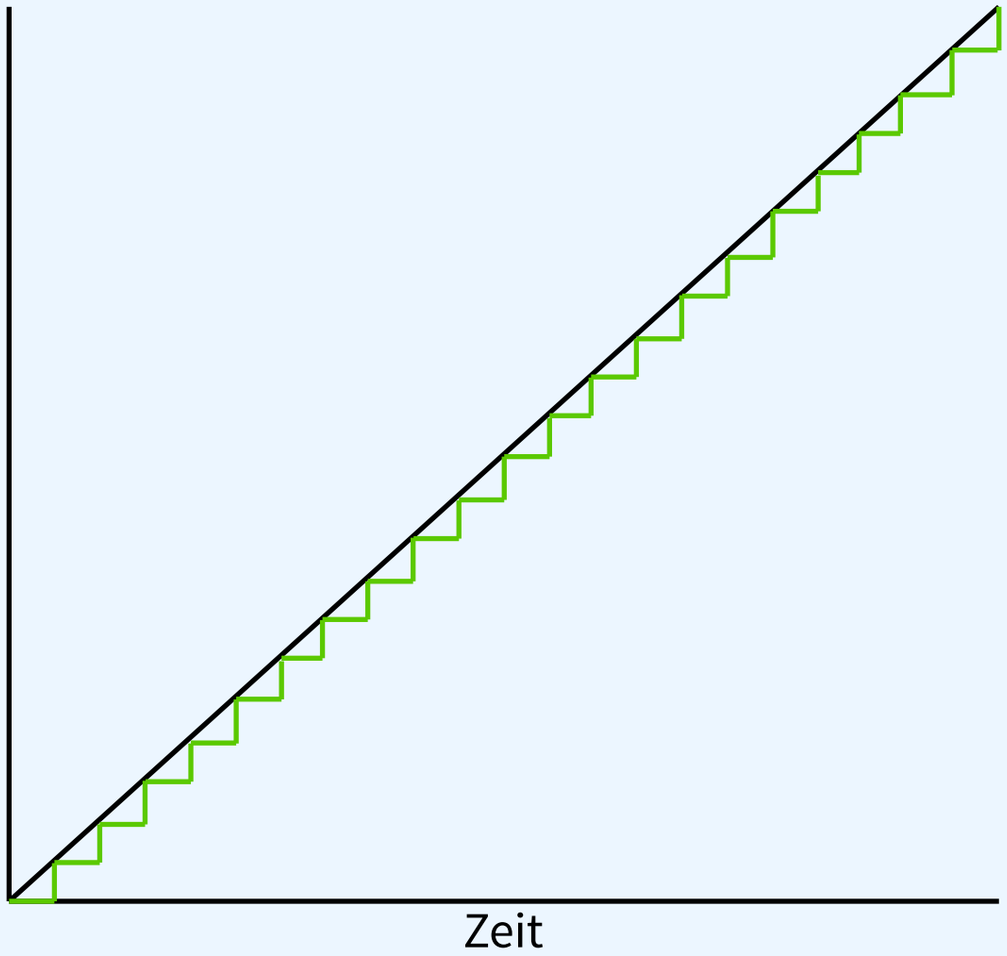
Quelle: Jim Sartain, Senior of Vice President of Quality at McAfee

Releasefrequenz

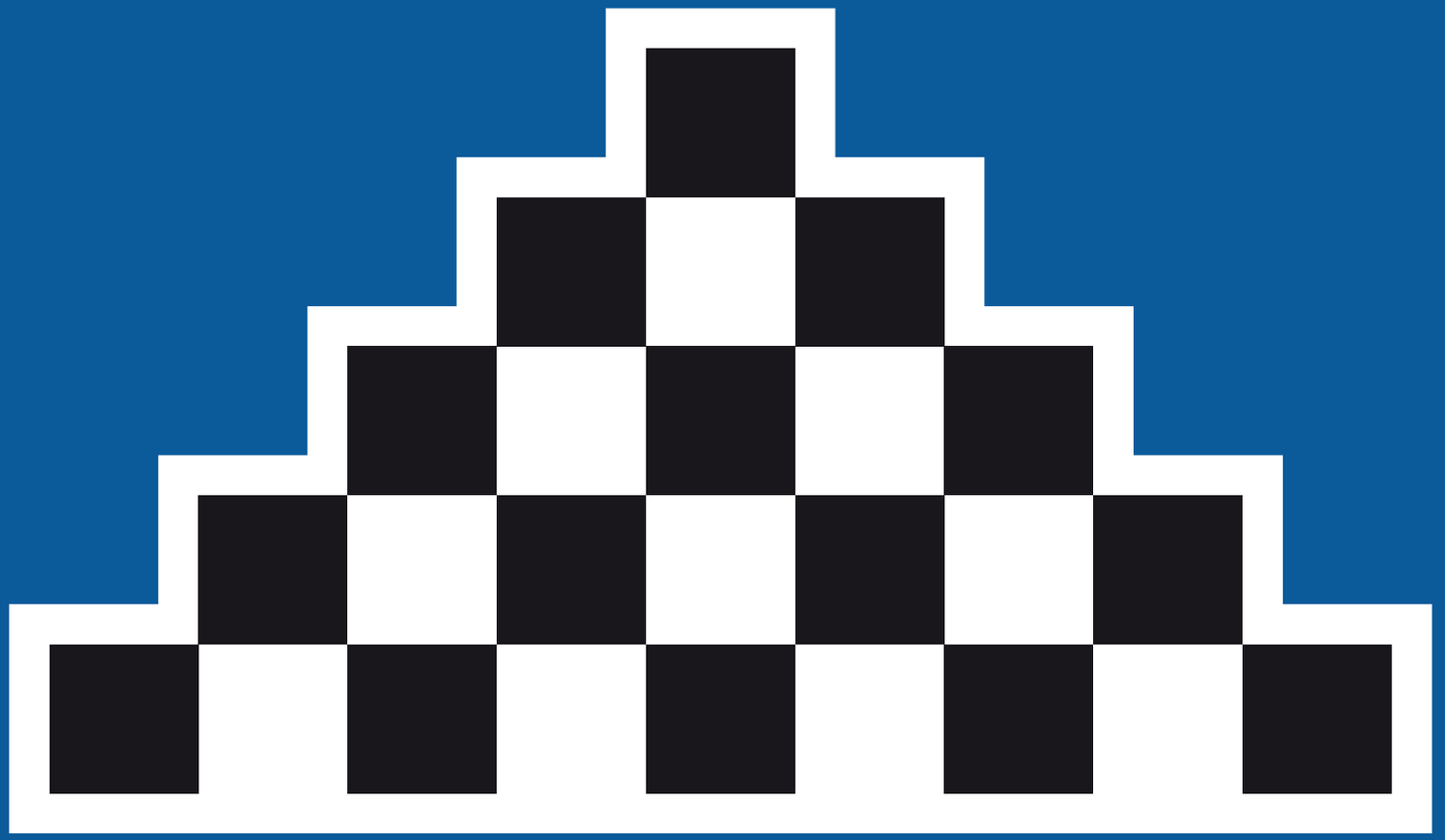
seltenes Deployment
viele Codeänderungen
hohes Risiko



häufiges Deployment
geringe Codeänderungen
geringes Risiko



Quelle: John Allspaw, Ops Metametrics



Grundlagen

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

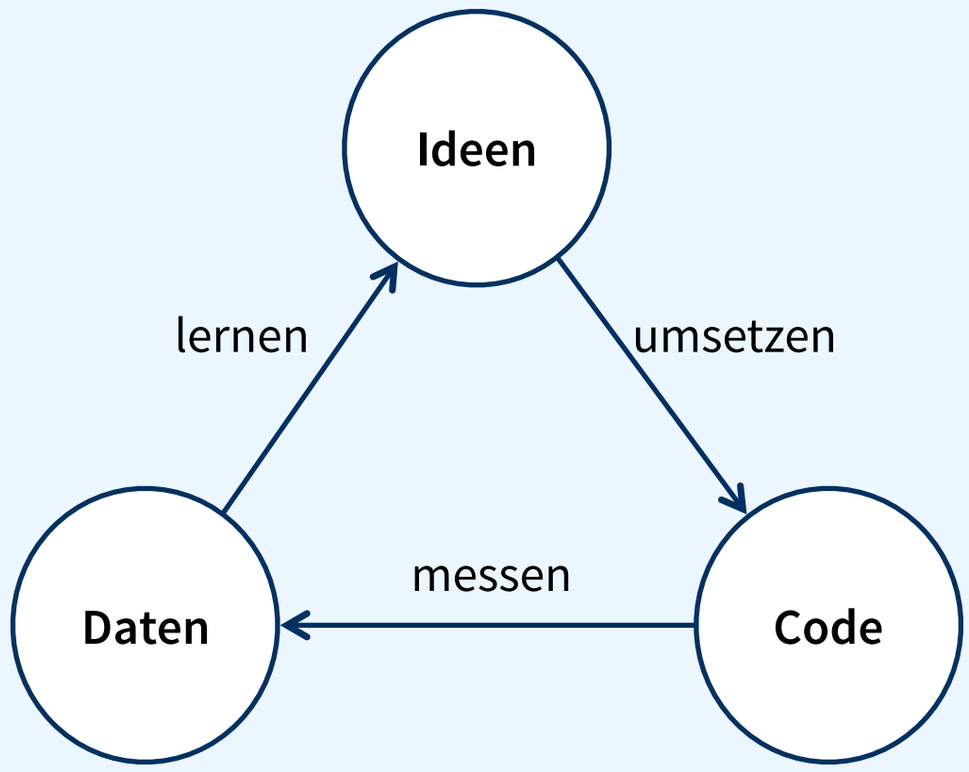
Delivery Pipeline

Innovation

You can't just ask customers what they want and then try to give that to them. By the time you get it built, they'll want something new.

Steve Jobs

- Hypothese aufstellen
- Hypothese einfachstmöglich umsetzen und ausliefern
- Feedback einholen
- Schritte wiederholen

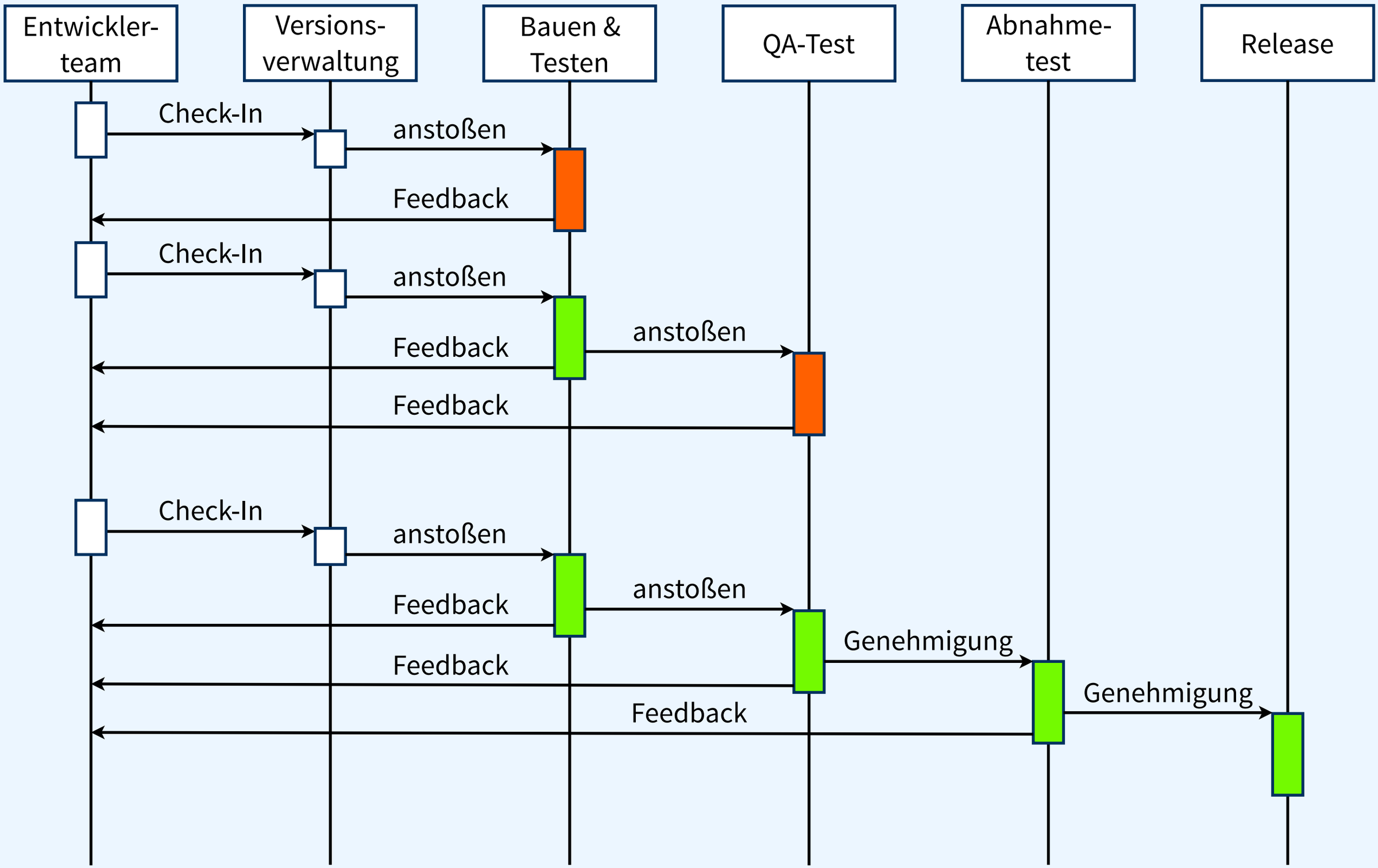


Quelle: Eric Ries, The Lean Startup

Continuous Delivery

- *Continuous Delivery* fasst die Konzepte zusammen:
  - Bauen → CI-Server baut Änderungen zentral und automatisch (*Continuous Integration*)
  - Testen → Unit-Tests werden automatisch ausgeführt und Feedback verschickt (*Continuous Testing*)
  - Installieren → Vom CI-Server erzeugte Projekte automatisch in der Systemlandschaft installieren (*Continuous Deployment*)
- Automatisches Deployment bedeutet nicht,
  - dass erstellte Projekte auf allen Servern bis zur Produktion automatisch verteilt werden
  - sondern dass Projekte mit minimalem Aufwand auf Server verteilt werden können (auf Knopfdruck)

Übersicht

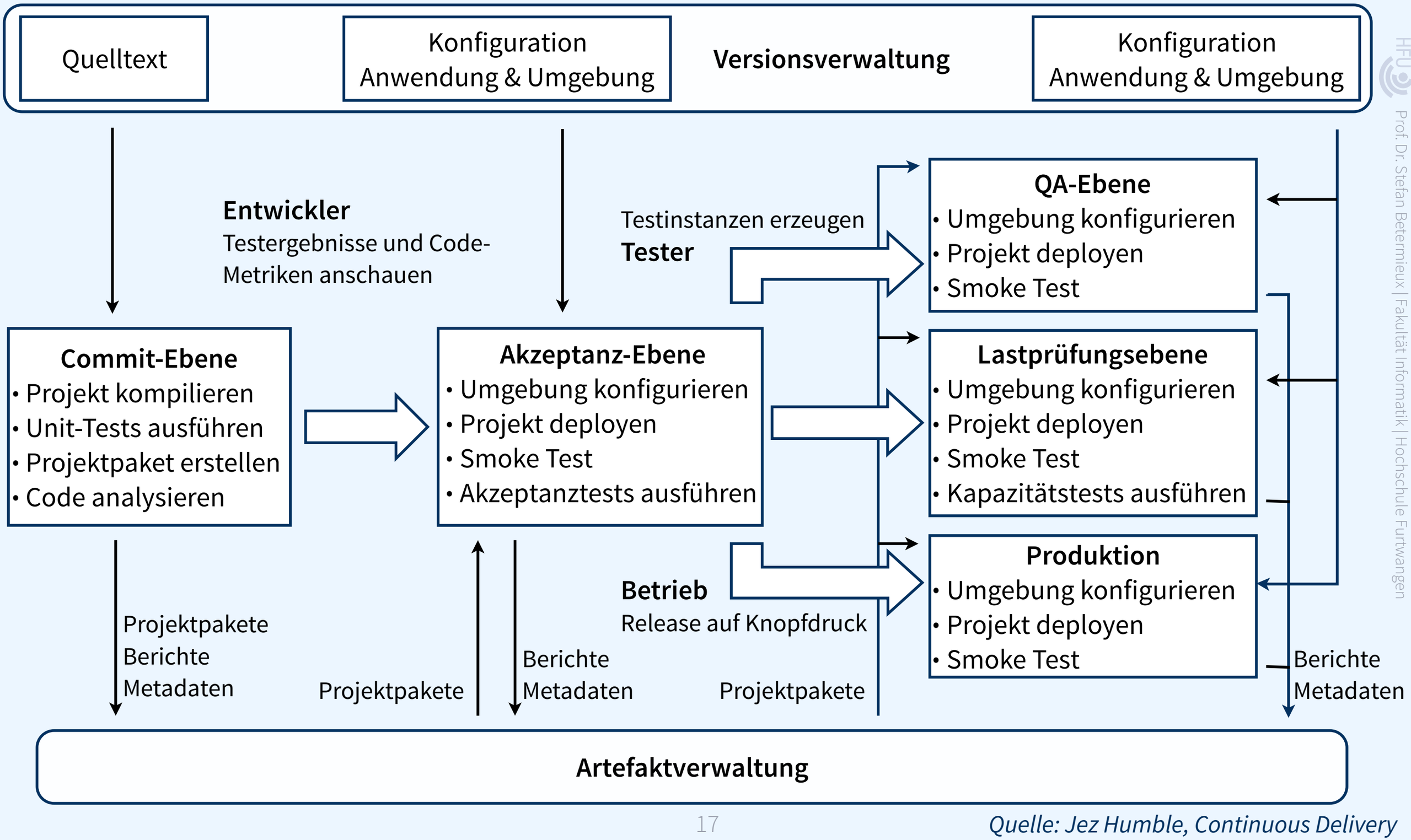


Quelle: Jez Humble, Continuous Delivery

Risiken reduzieren

- Um die Risiken beim Deployment zu reduzieren, sollte das Deployment möglichst häufig durchgeführt werden
- *"How long would it take your organization to deploy a change that involves just one single line of code?"*
  - die Zeit, die ein Kunde mindestens warten muss, bis für ihn überhaupt etwas passiert
- Continuous Deployment möglichst weit automatisieren
  - Systemlandschaft »Deployment Pipeline«
  - umfasst und erweitert Continuous Integration

Deployment Pipeline



Quelle: Jez Humble, Continuous Delivery

Pipeline Idee

- *"Deploy the same artifacts in every environment"*
  - Build-Artefakte werden nur einmal erzeugt
  - es wird getestet, was auch später in der Produktion landet
  - Artefakt-Repository → spätere Vorlesung
- *"Keep everything in version control"*
  - auch Anwendungs- und Infrastrukturkonfiguration
  - stellt sicher, dass in Entwicklung und Betrieb das gleiche Umfeld verwendet wird (App-Server, Datenbank, etc…)
- *"Automate almost everything"*
  - Deployment Pipeline funktioniert nur automatisiert
  - almost → Deployment in die Produktion nur für Mutige

Quelle: iX Developer, Bessere Software, 3/2013

HFU Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Spannungsfeld

Entwicklung

- möchte viele Änderungen schnell umsetzen
  - *Veränderung*

Betrieb

- möchte höhe Verfügbarkeit, also wenig Änderungen
  - *Stabilität*

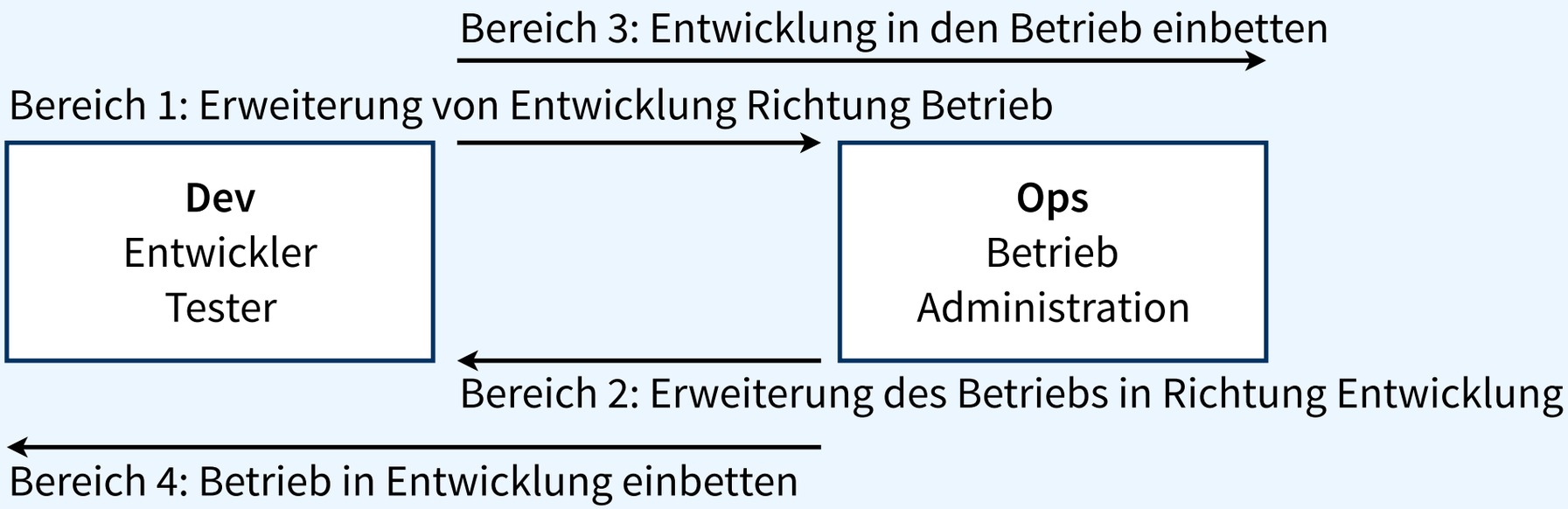
- Beide Parteien sind isoliert und arbeiten oft nur punktuell zusammen
  - die Entwicklung schmeißt ab und zu ein Release über den Zaun
- Wenn ein Fehler in der Produktion hohe wellen schlägt, beginnt häufig das »Blame Game«:
  - Betrieb wirft Entwicklung vor, dass das Release fehlerhaft ist
  - Entwicklung erwidert, dass auf Entwicklungsmaschinen alles funktioniert hat

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

DevOps

DevOps

Development & **Op**erations



- Bereich 1: Entwicklung beschreibt benötigte Infrastruktur
- Bereich 2: Log-Dateien für Entwicklung bereitstellen, Monitoring
- Bereich 3: sinnvoll bei kritischen nichtfunktionalen Anforderungen, z.B. Clustering und Performanz
- Bereich 4: Continuous Delivery, Deployment Pipeline

Quelle: iX Developer, Bessere Software, 3/2013

Kollaboration

DevOps ist nicht nur technisch zu lösen, auch die Kommunikation muss verbessert werden:

- *"Collective Ownership"*
  - alle am Projekt beteiligten und autorisierten Personen dürfen alle Projektdokumente sehen und verändern
- *"Cross-Functional Teams"*
  - Entwicklung, Test und Betrieb sind keine Silos, alle Beteiligten dürfen mit wechselnden Anteilen überall arbeiten
- *"Polyskilled Engineers"*
  - Projektbeteiligte müssen ein Grundverständnis der Arbeit in anderen Bereichen haben
    - » z.B.: müssen Programmierer Datenbanken ändern können

Quelle: Paul Duval, Agile DevOps, IBM developerWorks

Patterns

- Der Bereich DevOps und Continuous Delivery bildet sich gerade erst aus den negativen historischen Erfahrungen
  - kein konkretes Vorgehensmodell
  - viele Erfahrungen, Best Practices und Patterns
- Die nächsten Folien beschreiben einige Patterns beispielhaft
- Am Ende des Abschnitts folgt eine Übersicht aller Patterns
  - viele Ideen
  - wenig Werkzeugunterstützung

Pattern: Chaos Monkey

- Werkzeuge, die zu zufälligen Zeitpunkten einzelne Produktionsressourcen stören oder beenden
  - es wird getestet, ob Recovery-Maßnahmen greifen
  - im Gegensatz zu Murphys Gesetz kann der Chaos Monkey auf Bürozeiten beschränkt werden
- Verschiedene Ausprägungen:
  - Latency Monkey → Erzeugt künstliche Netzwerkverzögerungen
  - Janitor Monkey → Sucht nach momentan unbenutzten Ressourcen und schließt diese
  - Security Monkey → Sucht aktiv nach Sicherheitslücken
  - Conformity Monkey → Terminiert Instanzen, die sich nicht den Vorgaben gemäß verhalten

Quelle: Paul Duval, Agile DevOps, IBM developerWorks

Pattern: Transient Environment

- Entwicklersysteme haben häufig eine lange Lebensdauer, über Jahre wird Software installiert und verändert
  - »funktioniert bei mir« ist keine belastbare Aussage
  - Entwicklerrechner neu aufzusetzen ist kompliziert
  - Änderungen werden häufig nicht auf nachliegende System propagiert (Test- und Produktionssysteme)
- Ein »flüchtiges System« soll diese Probleme lösen:
  - Entwicklungssysteme sind nur begrenzt (z.B. drei Tage) verfügbar und werden dann neu aufgesetzt
  - Entwickler werden aufgefordert, Änderungen an der Infrastruktur technisch zu dokumentieren und allen zugänglich zu machen

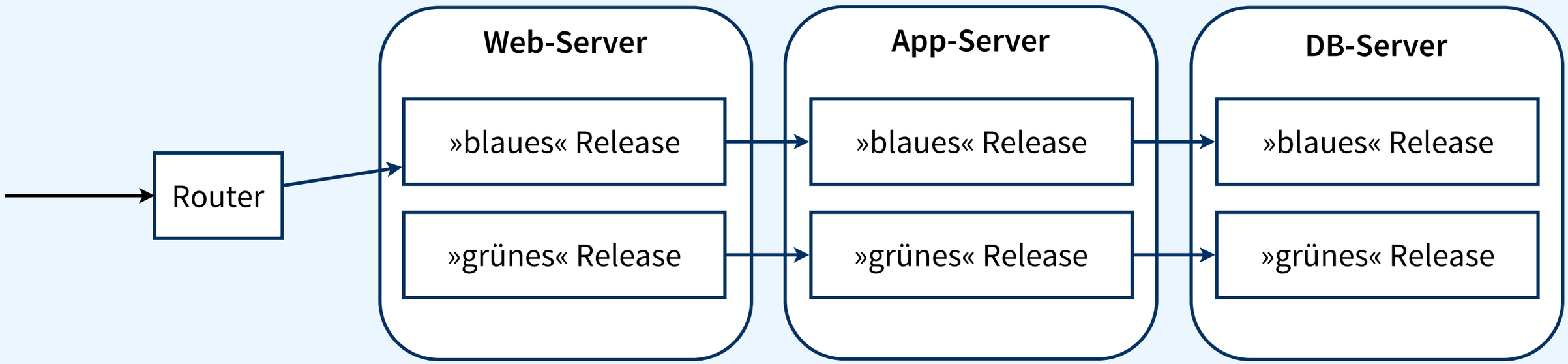
Quelle: Paul Duval, Agile DevOps, IBM developerWorks

Pattern: Dark Launching

- Ein neues Release sollte möglichst wenig Nutzer stören
  - z.B. durch Fehlermeldungen, ausloggen, etc…
- Release ganz ohne Störung ist bei großen Nutzerzahlen utopisch
  - Szenario: neue Anmeldung verhindern, warten bis alle Nutzer die Anwendung verlassen haben → lange Downtime
- Besser die aktiven Nutzer erfassen und das neue Release für einen Zeitpunkt planen, an dem wenig Nutzer erwartet werden
- Positiv: einfach umzusetzen
- Negativ: es werden immer noch (wenige) Nutzer in der Dienstnutzung gestört

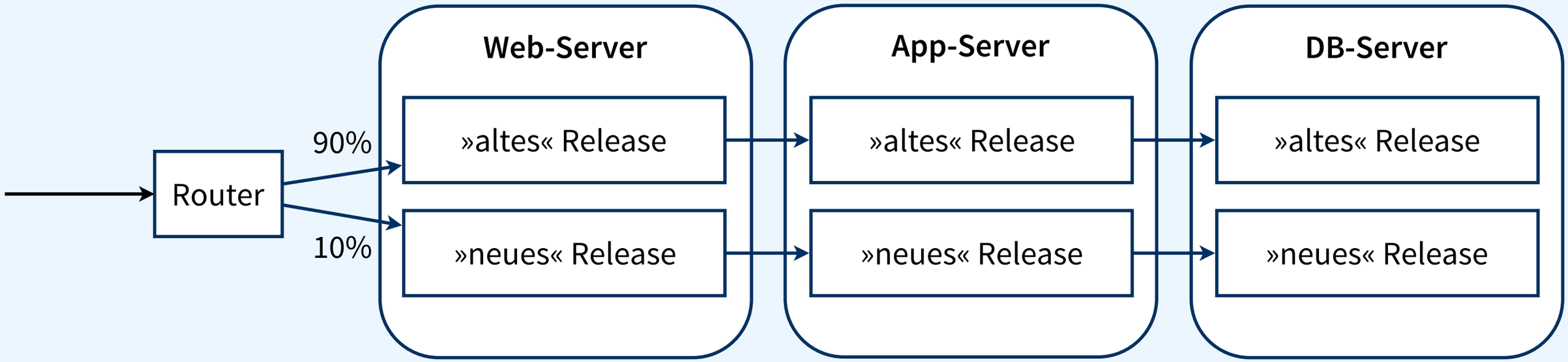
Pattern: Blue-Green Deploy

- Neues Release parallel zum alten Release auf der Produktions-infrastruktur installieren
- Sobald neues Release hochgefahren ist, Nutzer im Router umleiten
- Falls etwas schief geht → zurück zum alten Release
- Positiv: Test auf Produktivsystem möglich, keine Downtime
- Negativ: mehr Hardware-Ressourcen notwendig

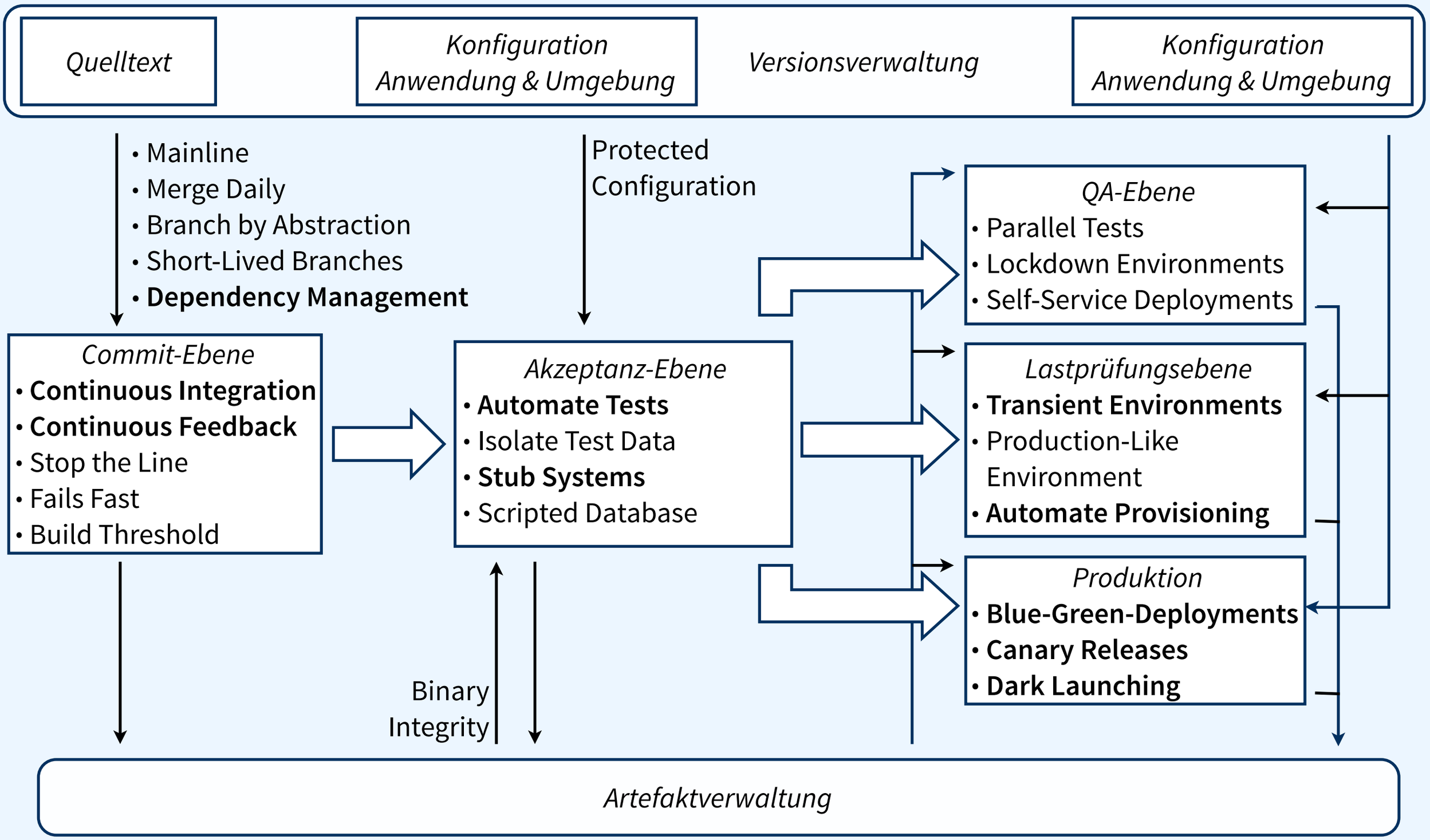


Pattern: Canary Releases

- Ähnliches Setup wie beim »Blue-Green Deployment«
- Nur einen geringen Prozentsatz der Nutzer auf die neue Version umleiten, der große Rest auf der alten Version
- Rückmeldung der Nutzer als Akzeptanztest,
  Anzahl der Rückmeldungen bleibt aber übersichtlich
  - praktiziert von Google, Facebook, etc…
- Ähnliche Vor- und Nachteile wie Blue-Green-Deployment



Patterns Übersicht



Quelle: Dzone Refcardz, Continuous Delivery

Werkzeuge

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Alles Versionieren

- Um Continuous Delivery erfolgreich umzusetzen, ist eine Voraussetzung zwingend erforderlich:
  - alle (Quell-)Dokumente müssen versioniert werden
  - alle (Quell-)Dokumente müssen als Textdatei vorliegen
- Folgende Artefakttypen müssen berücksichtigt werden:
  - Quellcode
  - Konfiguration
  - Datenbanken
  - Infrastruktur
- Binärartefakte nehmen eine Sonderstellung ein

Quelle: Paul Duvall, Agile DevOps - Version everything

Versionierung: Quellcode

- Quellcode ist problemlos versionierbar, da als Textdatei vorhanden
- Etabliertes Vorgehen, fast jeder Software-Entwickler beherrscht die Werkzeuge

```
/GitProjekt $ emacs Application.java
/GitProjekt $ git add Application.java
/GitProjekt $ git commit -m "Anwendung aktualisiert"
[master (root-commit) 32b24bc] Anwendung
aktualisiert
 1 file changed, 5 insertions(+)
 create mode 100644 Application.jav
```

Application.java

```java
/**
 * @author Stefan Betermieux
 */
public class Application {

  public static void main(String[] args) {
    System.out.println("Anwendung gestartert!");
  }

}
```


Quelle: Paul Duvall, Agile DevOps - Version everything

Versionierung: Config & DB

- Konfigurationen enthalten
  - IP-Adressen
  - Server Ports
  - Datei-Pfade
- werden oft im Textformat abgelegt (.properties)
  - leicht zu versionieren

- Datenbanken werden
  - mittels DDL definiert
  - mittels DML befüllt
- Beides textbasierte Formate
  - leicht zu versionieren

Application.properties

```
jboss.home=/usr/local/jboss
jboss.server.hostname=jenkins.example.com
jboss.server.port=8080
jboss.server.name=default
```

Application.ddl

```
CREATE SEQUENCE hibernate_sequence START
WITH 1 INCREMENT BY 1 NO MINVALUE
NO MAXVALUE CACHE 1;
ALTER TABLE public.hibernate_sequence OWNER
TO cd_user;
```


Quelle: Paul Duvall, Agile DevOps - Version everything

Versionierung: Infrastruktur

- Wie können installierte Anwendungen, Rechner, Rechnerverbünde dokumentiert und versioniert werden?
  - Textdokument für den Sysadmin
  - besser mit einer ausführbaren DSL (Domain Specific Language)
- Werkzeuge existieren, die diese DSL auf einem »leeren« System ausführen und alle notwendigen Anwendungen und Vorbedingungen sicherstellen
- Wir betrachten das Open-Source-Werkzeug »Puppet«
  - eine weitere freie Alternative ist »Chef«

Quelle: Paul Duvall, Agile DevOps - Version everything

Puppet

Motivation

- Anwendungen werden auf verschiedenen Plattformen unterschiedlich installiert
  - Ubuntu → `apt-get install apache2`
  - Redhat → `yum install httpd`
  - Windows → Download und Installer starten
- Shellskripte oder Batch-Dateien sind
  - zwar textbasiert und damit versionierbar
  - aber plattformabhängig und schlecht wartbar
- Puppets DSL abstrahiert und findet Gemeinsamkeiten

Puppet: Beispiel

- Das Werkzeug Puppet muss auf dem Rechner installiert sein:
  - am einfachsten über den Paketmanager der Plattform
  - oder von http://puppetlabs.com
- Puppet-Dateien enthalten Regeln, die nach der Bearbeitung durch Puppet auf dem Rechner erfüllt sein sollen
- Regeln sind idempotent → f(x) = f(f(x))
- Struktur einer Puppet-Regel:

```
RESOURCE { NAME:
    ATTRIBUTE => VALUE,
    …
}
```

apache2.pp

```
package { "apache2":
  ensure  => present
}
```

```
/$ puppet apply apache2.pp
Running Puppet with apache2.pp...
notice: /Stage[main]//Package[apache2]/ensure:
ensure changed 'purged' to 'present'
notice: Finished catalog run in 4.64 seconds
/$ puppet apply apache2.pp
notice: Finished catalog run in 0.03 seconds
```

Puppet Regeln

Außer Paketen gibt es noch folgende weitere Ressourcen:

- Dateien →

```
file { '/tmp/hello':
       content => "Hello, world\n"
}
```

- Systemdienste →

```
service { "apache2":
          ensure  => "running"
}
```

- Programme →

```
exec { "apt-get update":
          path => "/usr/bin",
}
```

(Achtung, nicht mehr plattformneutral!)

- Regeln können durch Abhängigkeiten in eine Reihenfolge gebracht werden:

```
service { "apache2":
    ensure  => "running",
    require => Package["apache2"]
}
```

Puppet Beispiel

Tomcat Web-Container installieren und Web-Anwendung deployen:

```
exec { "apt-get update":
  path => "/usr/bin",
}

package { "openjdk-7-jre-headless":
  ensure  => present,
  require => Exec["apt-get update"],
}


package { "tomcat7":
  ensure  => present,
  require => Package["openjdk-7-jre-headless"],
}

service { "tomcat7":
  ensure  => "running",
  require => Package["tomcat7"],
}

file { "/var/lib/tomcat7/webapps/meldeauskunft.war":
  ensure  => "link",
  target  => "/vagrant/meldeauskunft.war",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}
```

Erklärung später!

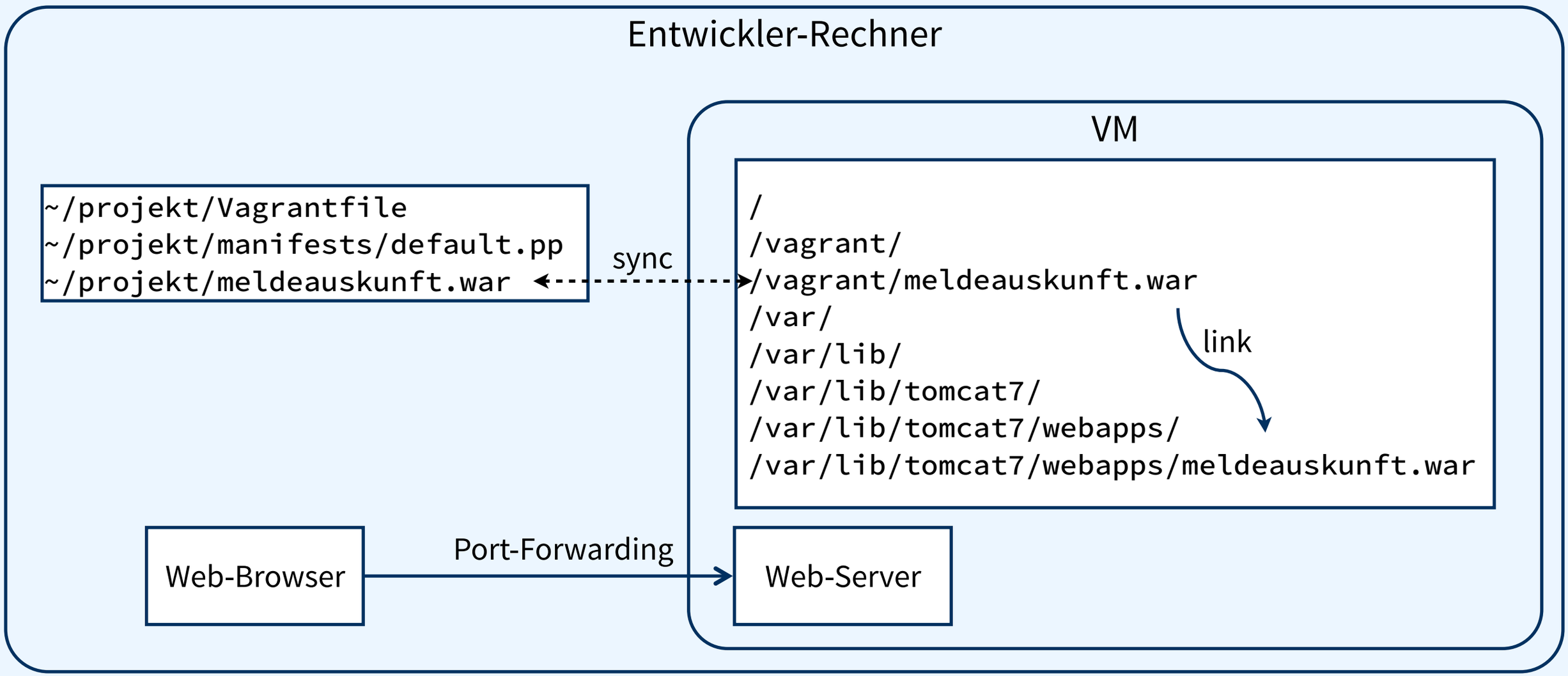
Vagrant

Motivation

- Mit Puppet lassen sich Integrations- und Produktionsserver erstellen
- Es kann mit Puppet aber auch die Laufzeitumgebung für Entwicklerrechner erstellt werden
- Zwei Alternativen:
  - Puppet direkt auf dem Entwicklerrechner ausführen (schlecht)
  - auf dem Entwicklerrechner eine einfache Server-VM starten und diese mit Puppet konfigurieren (besser, da ähnlich Zielserver)
- Da VM ohne GUI funktioniert die Interaktion mittels:
  - Shared-Folders
  - Netzwerk-Ports

Übersicht

```
file { "/var/lib/tomcat7/webapps/meldeauskunft.war":
  ensure  => "link",
  target  => "/vagrant/meldeauskunft.war",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}
```



Zusammenfassung

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Reifegradmodell

Themenfeld	Reifegrad: Basis	Reifegrad: Anfänger	Reifegrad: Fortgeschritten	Reifegrad: Experte
Architektur	• einheitliche Plattform und Technologie	• Projekt in Module aufteilen • API-Management • Bibliotheken verwalten • Datenbanken versionieren	• Keine Branches • Configuration as Code • Feature Hiding • Aus Modulen Komponenten erstellen	• Komponentbasierte Architektur • Metriken erstellen und weiterleiten • Infrastructure as Code
Deployment	• Versionsverwaltung • Automatische Builds • Geplante Builds (CI) • Dokumentiertes manuelles Deployment	• Builds durch Polling der Versionsverwaltung (CI) • Builds werden archiviert • Manuelle Tags in der Versionsverwaltung	• Automatische Builds (Commit Hooks) • Automatische Tags in der Versionsverwaltung • Build Once, Deploy Anywhere • Deploy-Pipeline bis zur Produktion	• Zero-Downtime-Deploys • Master-Slave CI-Server • Zero Touch Continuous Deployments

Quelle: Andreas Rehn, The Continuous Delivery Maturity Model

Danke