Florian Tobusch – Sommersemester 2018 |
Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

# Continuous Deployment

## Keine Angst vor Releases

# Gliederung

- Einführung
  - Begriffsklärung
  - Firmenbeispiele
  - Vorteile von Continuous Deployment
- Deployment Pipeline
  - Commit Stage
  - Akzeptanztests
  - Kapazitätstests
  - Explorative Tests
  - Produktion
- Fazit
- Literatur

# Einführung

# Begriffsklärung

| Continuous Integration (CI) | Continuous Delivery (CD) | Continuous Deployment |
|---|---|---|
| • Gemeinsames Repository<br>• Automatisch kompilieren und bauen<br>• Unittests | • Beinhaltet CI<br>• Durchlauf weiterer Test-phasen<br>• Entwickler bestimmen Deployment-zeitpunkt selbst | • Beinhaltet CD<br>• Jede Änderung im Code kann automatisch in Produktion gehen |

# Firmenbeispiele

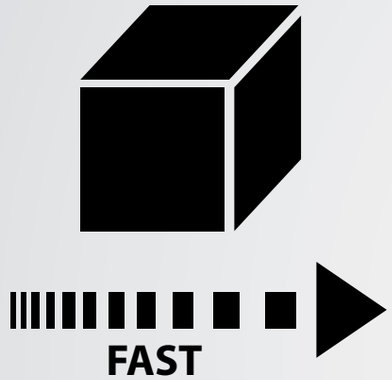
## Google-Web-Server-Team

- »No changes without tests«


## CSG International

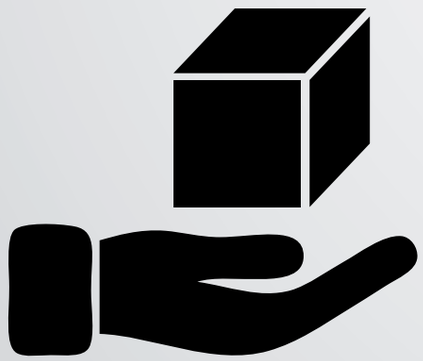
- Tägliche Deployments
- Produktionsnahe Testumgebungen

# Continuous Deployment – Vorteile




**Kurze Durchlaufzeiten**

**Weniger Risiko, mehr Qualität**

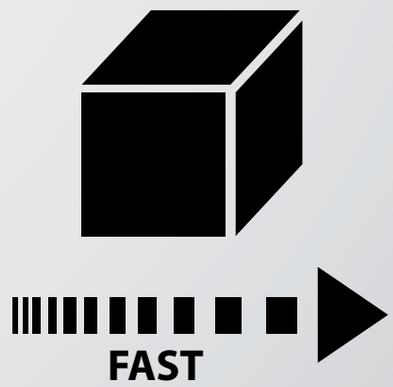


**Mehr Verantwortung für die Teams**

**Schnelles Feedback**

# Kurze Durchlaufzeiten

- Weniger manuelle Arbeitsschritte durch Automatisierung
- Schnellere Veröffentlichung von neuen Features
- Verbesserung der Time-To-Market



# Risiko reduzieren / Qualität erhöhen

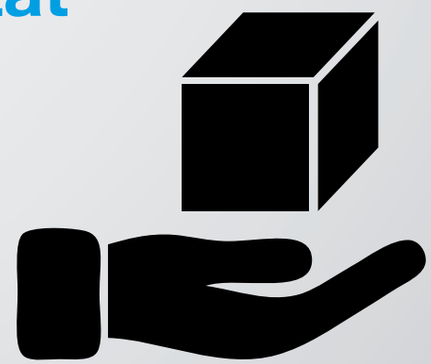
- **Automatisierung bringt Sicherheit**
- **Reproduzierbare und nachvollziehbare Prozesse**
- **Einführung von häufigeren Deployments**
- **Somit kleine, überschaubare Änderungen pro Deployment**

# Schnelles Feedback

- **Durch Automatisierung schneller Durchlauf der Pipeline**
- **Häufige Durchläufe der gesamten Pipeline schaffen regelmäßiges Feedback (Release nicht nur einmal pro Monat/Quartal)**

# Mehr Verantwortung

- Team ist verantwortlich für den Zustand der Deployment Pipeline
- Team ist verantwortlich dafür, dass keine fehlerhafte Software in Produktion gelangt
- Keine weitere Instanz außer dem Team prüft Code-Änderungen, bevor sie in Produktion gelangen
- Mehr Verantwortung kann zu besserer Qualität der Anwendung führen



# Deployment Pipeline

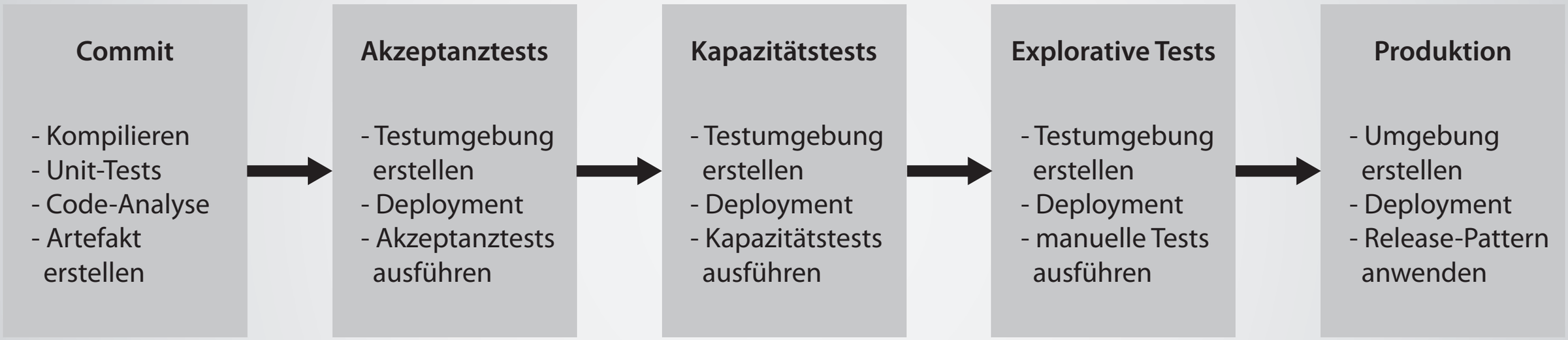
# Pipeline – Allgemein

- Selbe Binaries (Artefakt) für alle Phasen, inkl. Produktion
- Gleicher Deployment Prozess für alle Phasen
- Testumgebungen sollten nahezu der Produktionsumgebung entsprechen
- Schnelle Tests an den Anfang der Pipeline, langsame Tests nach hinten verschieben
- Fehlerhafte Artefakte gelangen nicht zur nächsten Phase

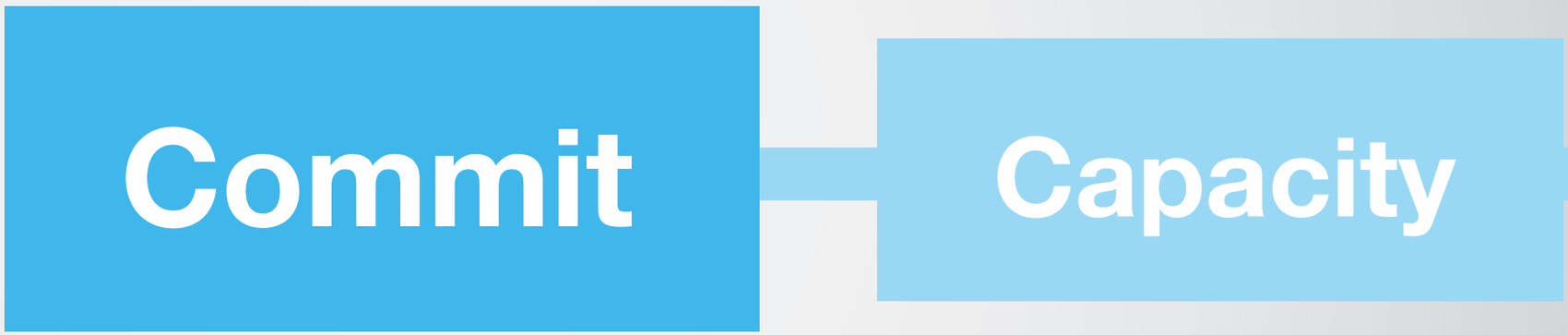
Pipeline-Tools: Jenkins, Bamboo, TravisCI

Build-Tools: Maven, Gradle

# Phasen der Pipeline



# Commit Stage

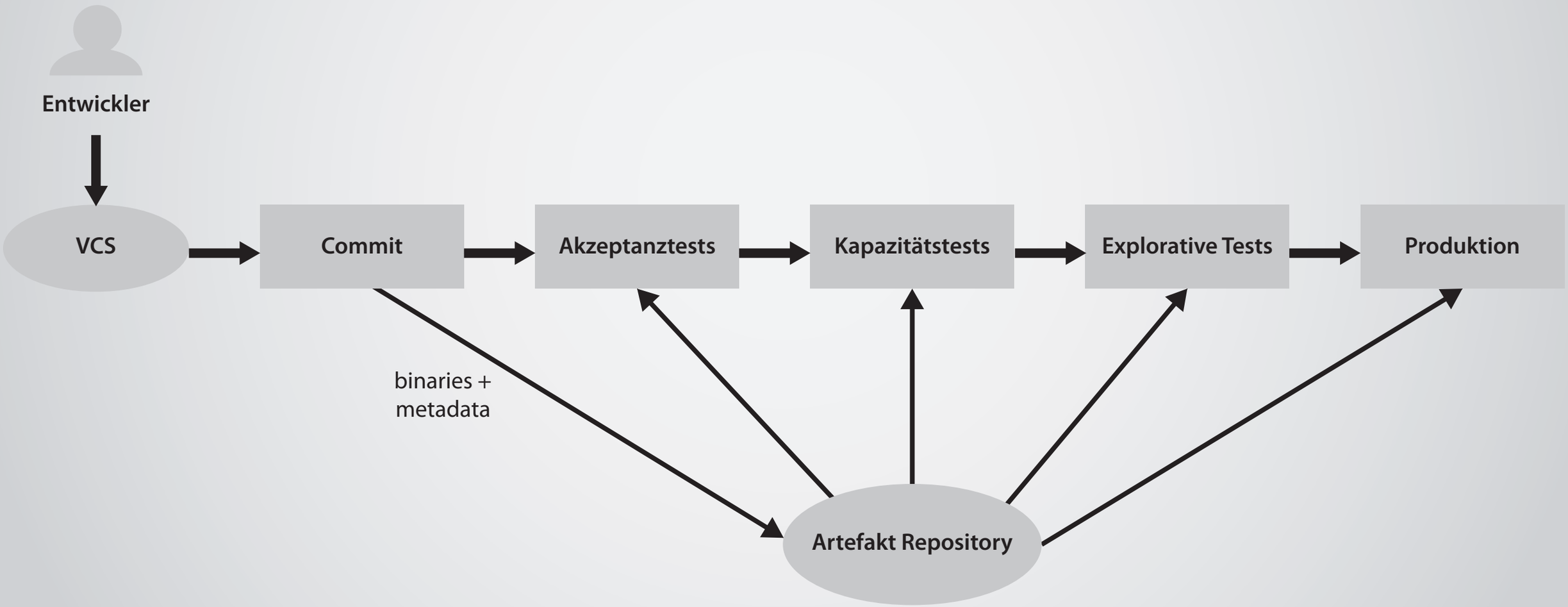


# Commit Stage

**Tasks der Commit Stage:**

- **Kompilieren**
- **Tests ausführen (z.B. Unittests)**
- **Codeanalyse (z.B. Testabdeckung, Komplexität, ...)**
- **Artefakt erstellen (WAR, EAR)**

# Artefakte managen




**Commit** | Acceptance | Capacity | Explorative | Production

# Akzeptanztests

nmit

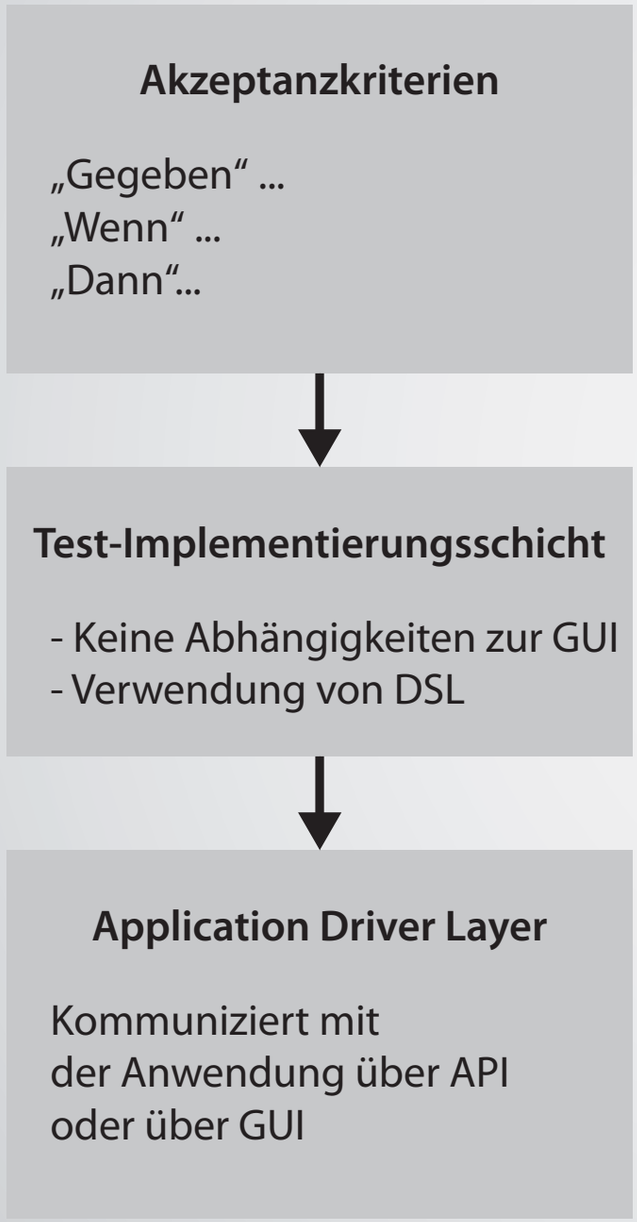
Acceptance

Capacity

# Akzeptanztests

- Überprüfung auf abstrakterer Ebene
- Tests unabhängig von konkreten Implementierungen
- Anwendung wird als ganzes getestet, keine einzelnen Einheiten
- Tests prüfen, ob die Akzeptanzkriterien der Stories/Requirements erfüllt sind
- Basis zur Abnahme durch den Kunden

# Akzeptanztests erstellen




- **Textuelle Beschreibung der Tests (auch für Kunden lesbar)**
- **Behavior Driven Development (BDD)**
- **Nur die »Application Driver Layer« kennt die konkrete Implementierung der Applikation**

**Tools: JBehave, Cucumber**

# Akzeptanztests – Beispiel

```
1  Szenario: Kunde registriert sich erfolgreich
2  Gegeben ein neuer Kunde mit
3  E-Mail eberhard.wolff@gmail.com
4  Vorname Eberhard Name Wolff
5  Wenn der Kunde sich registriert
6  Dann sollte ein Kunde mit
7  der E-Mail eberhard.wolff@gmail.com existieren
8  Und es sollte kein Fehler gemeldet werden
```

# Akzeptanztests – Beispiel

```
1  //Implementierung von »Gegeben«
2  @Given(»ein neuer Kunde mit E-Mail $email Vorname
         $vorname Name $name«)
3  public void gegebenKunde(String email,
         String vorname, String name){
4          kunde = new User(vorname, name, email);
5  }
```

# Kapazitätstests

tance

**Capacity**

Explorative

# Kapazitätstests

- Performance der Applikation testen
- Mit verschiedenen Akzeptanztests realistische Auslastung erzeugen
- Tests über GUI: anfällig, entsprechen dem Vorgehen der Nutzer
- Tests über API: stabiler, entsprechen nicht dem Nutzervorgehen

Vorteil der Kapazitätstests:

- Veränderungen der Performance sind direkt auf Codeänderungen rückführbar

# Kapazitätstests

**Schwierigkeiten:**

- **Produktionsumgebung realistisch nachbauen (Abhängigkeiten, Hardware, Datenbanken, ...)**
- **Passenden Schwellwert für Erfolg/Misserfolg der Tests definieren (nicht zu aggressiv, nicht zu schwach)**

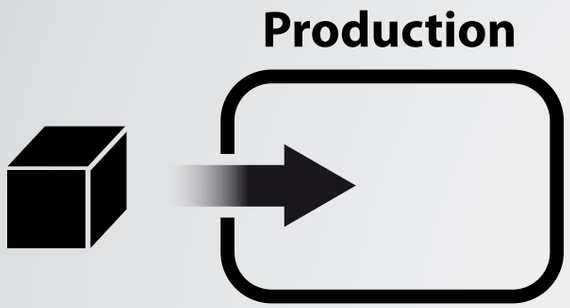
# Explorative Tests

# Explorative Tests

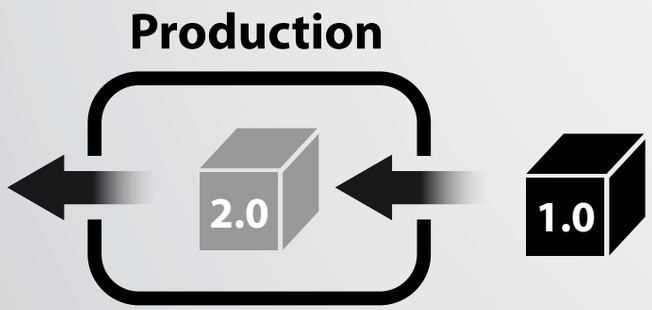
- Manuelle Tester mit fachlichem Wissen testen mögliche Schwachstellen
- Bedienbarkeit und Design nur manuell testbar
- Gefundene Fehler in Zukunft durch schnelle automatische Tests überprüfen lassen
- Tests sind teuer und zeitintensiv

# Produktion

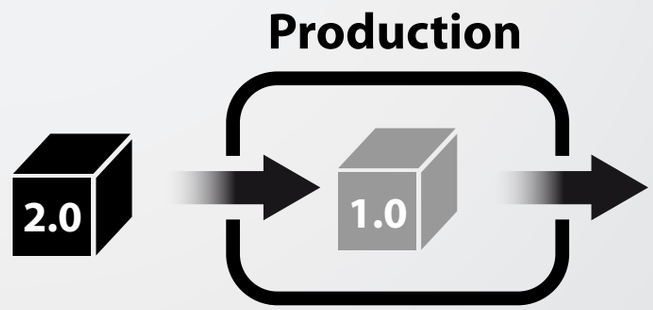
# Rollout, Rollback, Roll Forward




Rollout

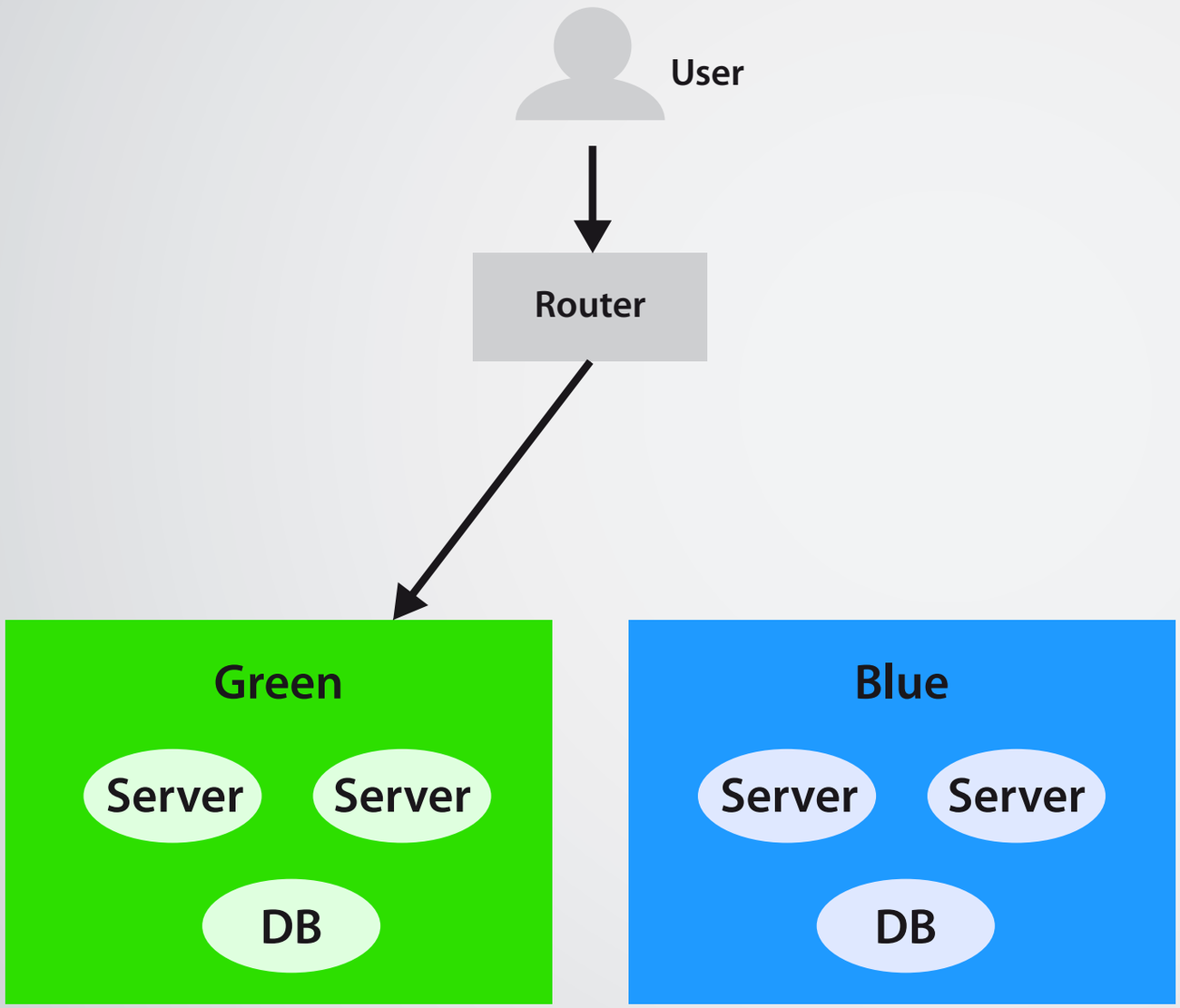



Rollback




Roll Forward

# Blue-Green-Deployment




- 2 identische Produktions-umgebungen
- Nur eine Umgebung ist live
- Wechsel der Umgebungen durch den Router

# Blue-Green-Deployment

*Lösungsansätze für das Handling von Datenbanken*

**Synchronisation:**

- **Synchronisation der Datenbanken beim Wechsel der Umgebung**

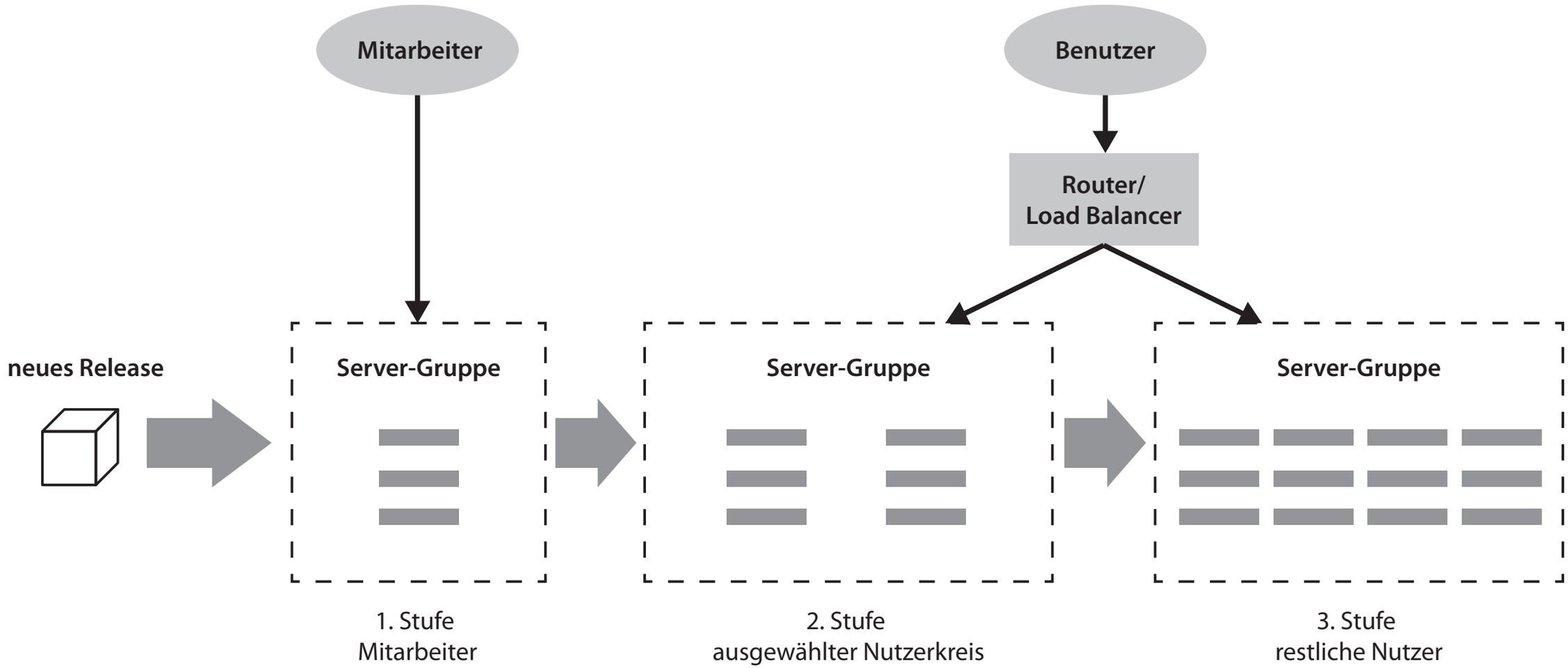
**oder Entkopplung:**

- **Datenbank-Release unabhängig vom App-Release**
- **Versionierung der Datenbank hierfür notwendig**

# Canary Releasing

- Neues Feature nach und nach für alle Nutzer ausrollen (z.B. zuerst nur für Poweruser oder Mitarbeiter)
- Bei Fehlern nur kleiner Nutzerkreis betroffen
- Produktionsumgebung somit auch für realistische Tests nutzbar

# Canary Releasing



# Feature Toggle

- Neues Feature wird deaktiviert in Produktion deployt
- Aktivierung des Features ohne neu zu deployen
- Verwaltung aller Toggles durch eigenen Service oder Konfigurationsdatei

Kombination mit Canary Releasing sinnvoll

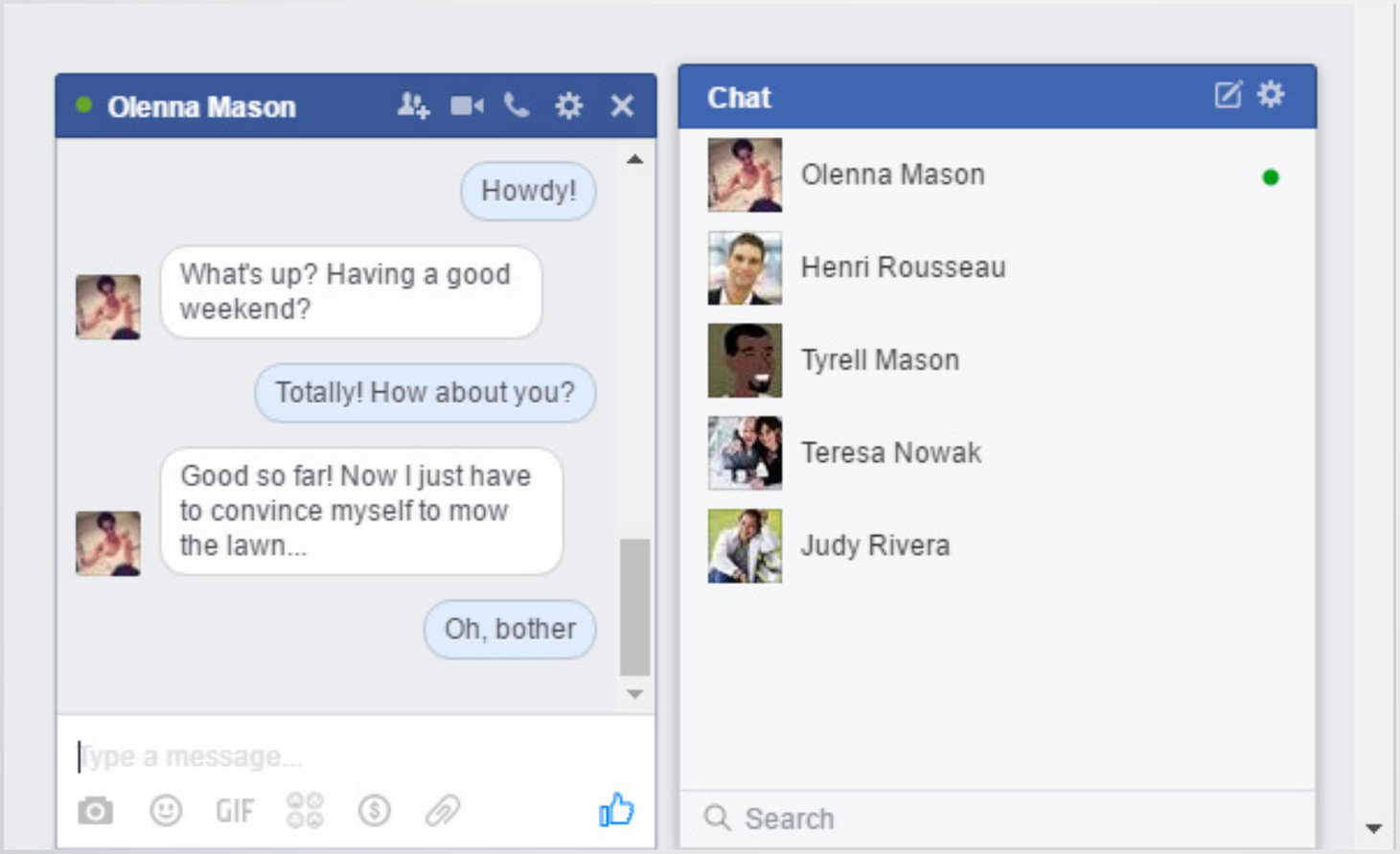
# Dark-Launch

- baut auf den Feature Toggles auf
- Feature wird aktiviert, aber ist nicht für Nutzer über die GUI sichtbar
- Dark-Launches werden für realistische Tests verwendet (z.B. neue Suchalgorithmen auf Performance testen)

Kombination mit Canary Releasing sinnvoll

# Dark-Launch – Beispiel

## Beispiel: Facebook-Chat




Bildquelle:

https://www.gcflearnfree.org/facebook101/chat-and-messages/1/ [06.06.18]

# Fazit

# Vielen Dank

# Literatur


Gene Kim, Jez Humble, Patrick Debois, John Willis (1. Auflage, 2017). Das DevOps Handbuch. Heidelberg: O'Reilly.

Jez Humble, David Farley (2011). Continuous Delivery. Boston: Pearson Education.

Eberhard Wolff (2. Auflage, 2016). Continuous Delivery – Der pragmatische Einstieg. Heidelberg: dpunkt.Verlag.