



# Version Control, GIT, CI/CD

Dániel Hajnal, Collin Forslund, Elias Oezcan, Jan Wichmann, Sebastian Kiebert

Public

Juni, 2023



# Agenda

1

## **Version Control**

Versionierung die Codeänderungen verwaltet

---

2

## **GIT**

Verteilt Versionsverwaltung von Dateien

---

3

## **Continuous Integration**

Ständige Zusammenführung und automatische Prüfung von Codeänderungen

---

4

## **Continuous Delivery und Deployment**

Automatisches Ausrollen und Aktualisieren von Software.

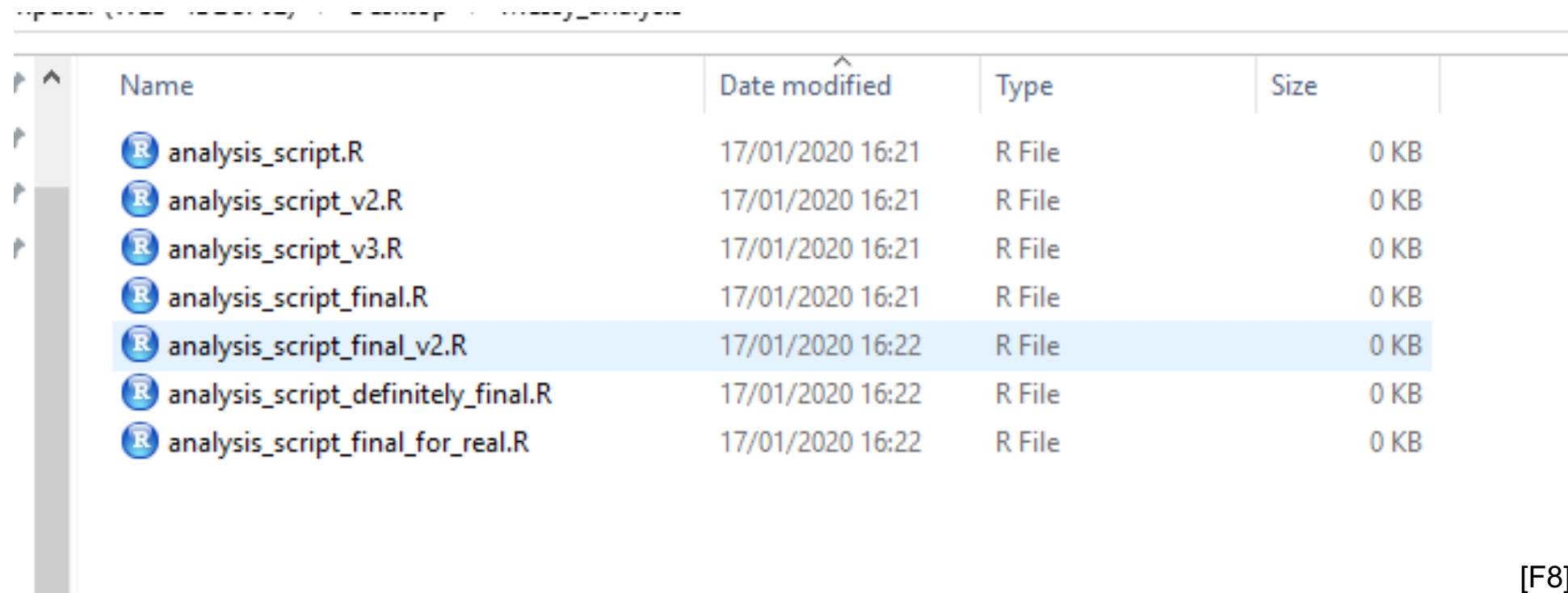
# Version Control System (VCS)

- zur Verfolgung und Verwaltung von **Änderungen** an Dateien
- Funktionen:
  - *reversibility*: Wiederherstellung eines vorherigen (funktionierenden) Zustandes
  - *concurrency*: gleichzeitige Bearbeitung durch mehrere Entwickler
  - *history*: Protokollierung der Änderungen zur besseren Übersicht








# Grundbegriffe

- *repository*: Speicherort für alle Dateien, Verzeichnisse, Historie und Metadaten des Projekts
- *version*: Zustand des Projekts zu einem bestimmten Zeitpunkt
- *working directory*: lokale Kopie des Projekts
- *checkout*: Kopieren einer *version* aus dem *repository* ins *working directory*
- *check in*: Übertragen von Änderungen aus *working directory* in *repository* → Erstellung einer *version*
- *branch*: Verzweigung → parallele Bearbeitung mehrerer *versions* möglich
- *merge*: Zusammenführung zweier *branches*

# Naive Versionenverwaltung



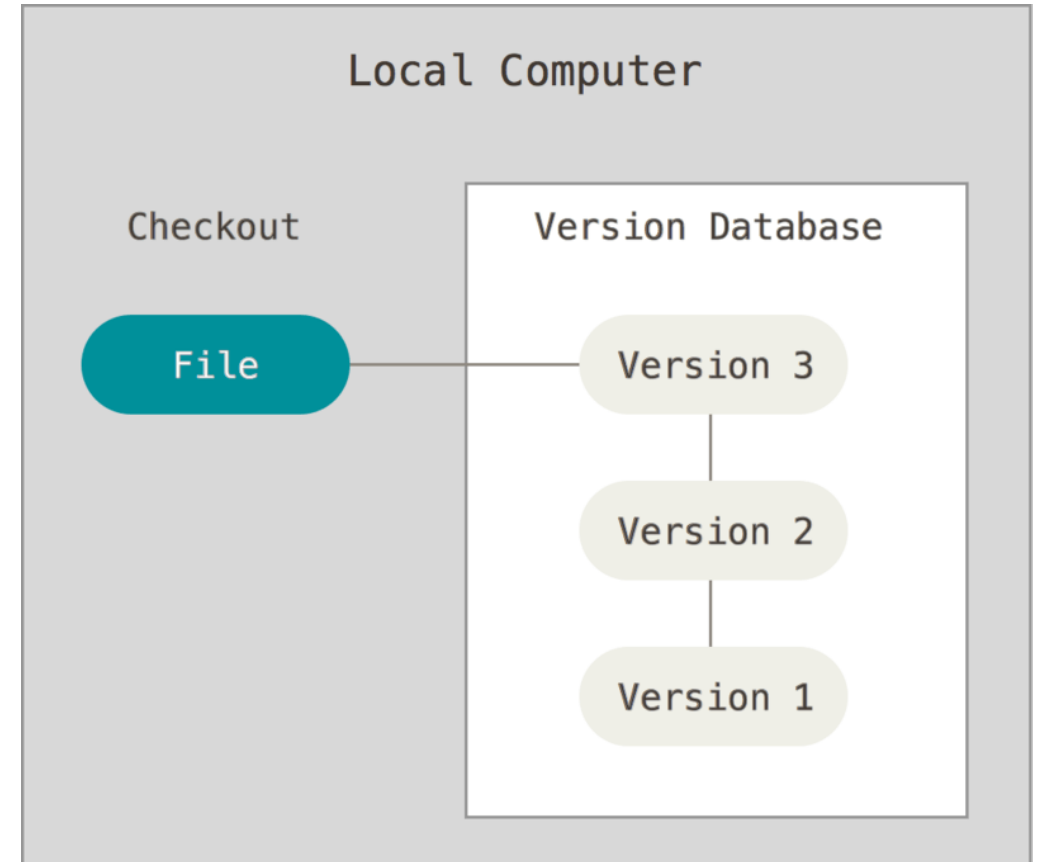
The image shows a screenshot of a file explorer window. The title bar at the top reads "My Computer - Windows Explorer". Below the title bar is a table with five columns: "Name", "Date modified", "Type", "Size", and an empty column. The table lists seven R script files, each preceded by a blue circular icon containing a white 'R'. The file "analysis\_script\_final\_v2.R" is highlighted with a light blue background. To the left of the table is a vertical sidebar with a grey bar and several small upward-pointing arrows.

Name	Date modified	Type	Size	
 analysis_script.R	17/01/2020 16:21	R File	0 KB	
 analysis_script_v2.R	17/01/2020 16:21	R File	0 KB	
 analysis_script_v3.R	17/01/2020 16:21	R File	0 KB	
 analysis_script_final.R	17/01/2020 16:21	R File	0 KB	
 analysis_script_final_v2.R	17/01/2020 16:22	R File	0 KB	
 analysis_script_definitely_final.R	17/01/2020 16:22	R File	0 KB	
 analysis_script_final_for_real.R	17/01/2020 16:22	R File	0 KB	

[F8]

# Local VCS

- einfache Verwaltung der Versionen auf lokaler Datenbank
- nur Speicherung von Deltas (= Änderung zur vorherigen Version) → Effizienz
- reverse-delta: noch effizienter
- Beispiele: SCSS (1972), RCS (1982)

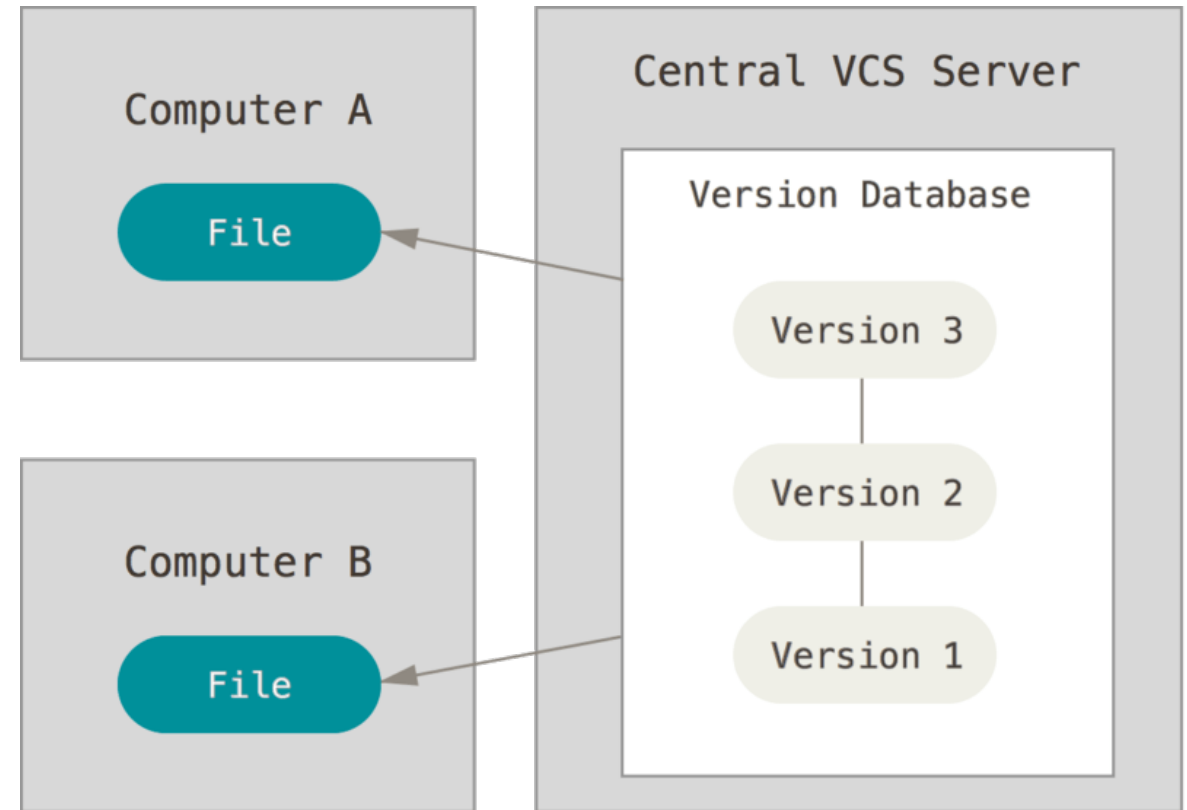


[F9]

[8]

# Centralized VCS

- Client-Server Prinzip
- ein zentrales Repository
- File-Locking reguliert Kollaboration
- Beispiele: CVS (1986), Subversion (2000)

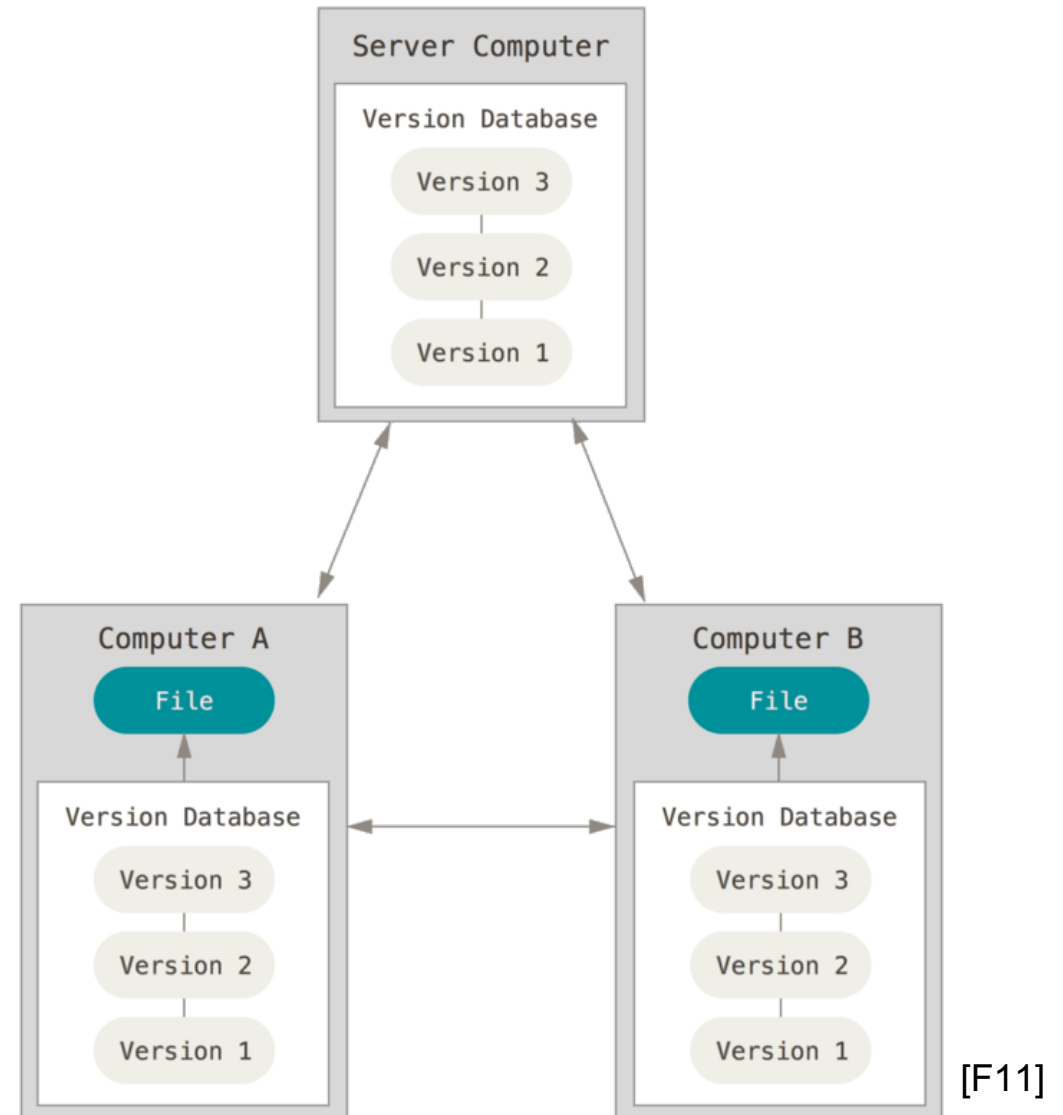


[F10]

[8]

# Distributed VCS

- jeder verfügt über eigenständige Kopie des repositories
- merge conflict statt file conflict
- Vorteile:
  - Backup
  - offline möglich
  - schnellere Operationen





# Git Fakten

- Frei benutzbare Software
- Versionenverwaltung
- Initiiert durch Linus Torvalds



[F12]

# Git Fakten



- git ist „de facto Standard“
- git ist flexibel (Projektgröße)
- git ist Open Source

[F12]

# GitHub

**Code hosting platform für Version Control und Kollaboration**



**100+ Mio**  
Entwickler

**4+ Mio**  
Organisationen

**330+ Mio**  
Repositories

# GitHub

**Code hosting platform für Version Control und Kollaboration**



Veröffentlichen  
von Repositories

Zugriff konfigurieren  
→ Lese- Schreibberechtigung

Erleichtert  
Kollaborationen

# GitHub

**Code hosting platform für Version Control und Kollaboration**



Veröffentlichen  
von Repositories

Zugriff konfigurieren  
➔ Lese- Schreibberechtigung

Erleichtert  
Kollaborationen

# GitHub

**Code hosting platform für Version Control und Kollaboration**



Veröffentlichen  
von Repositories

Zugriff konfigurieren  
→ Lese- Schreibberechtigung

Erleichtert  
Kollaborationen

[1]

# GitHub

**Code hosting platform für Version Control und Kollaboration**

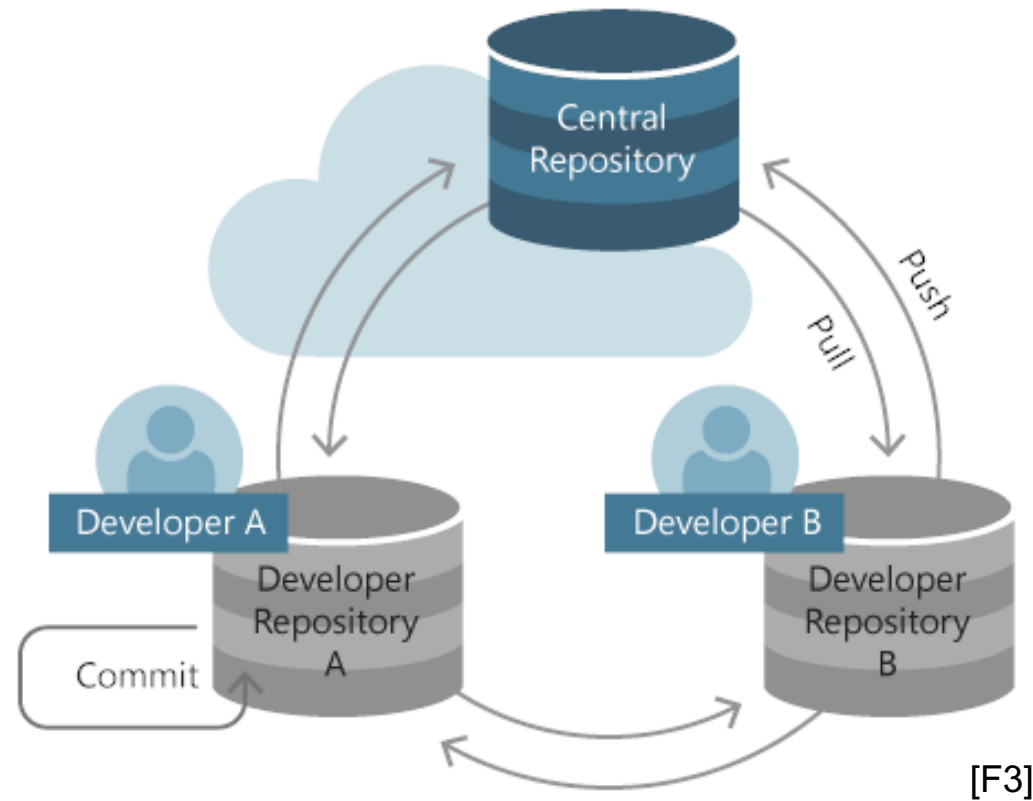


Erfolg → Große Anzahl an Nutzer + einfache Integration von git

# Repository

Speicherort für Dateien und Dokumente

Jeder lokaler Rechner  
der Daten speichert  
kann als Repository  
genutzt werden



Repositories  
sind unabhängig

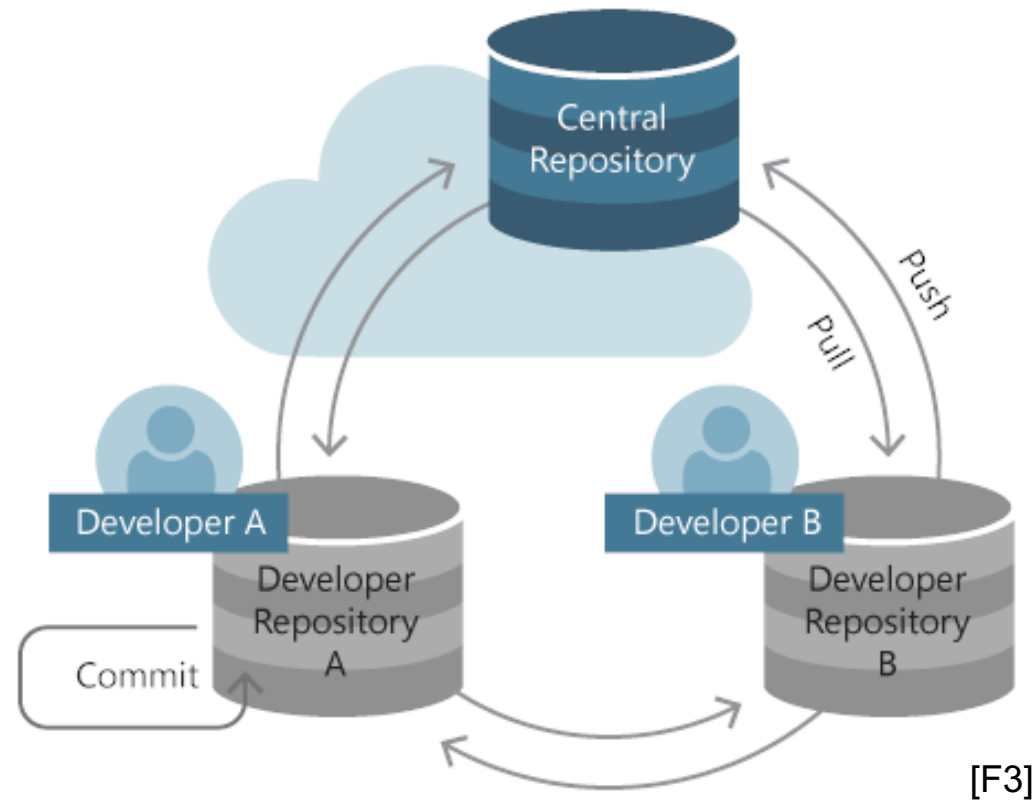
[2]



# Repository

Speicherort für Dateien und Dokumente

Jeder lokaler Rechner  
der Daten speichert  
kann als Repository  
genutzt werden

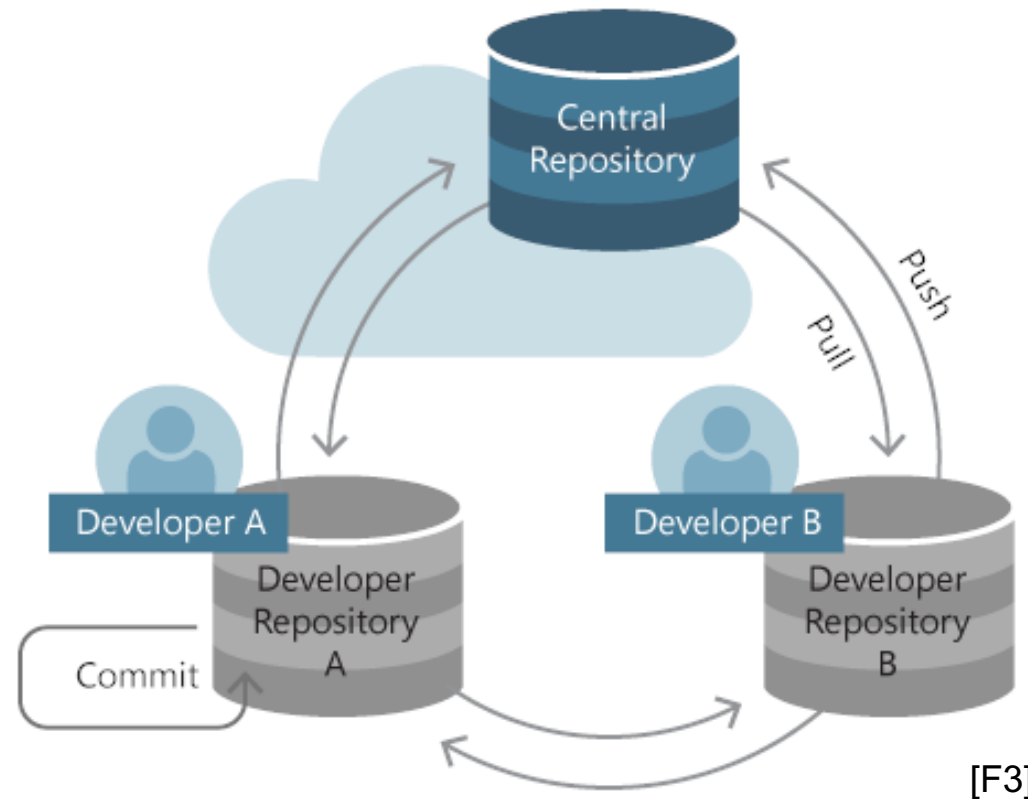


Repositories  
sind unabhängig

# Repository

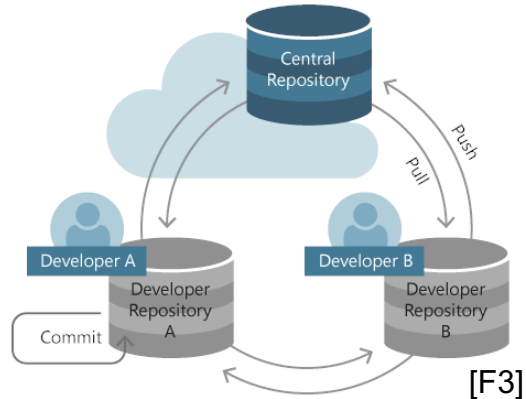
Speicherort für Dateien und Dokumente

Jeder lokaler Rechner  
der Daten speichert  
kann als Repository  
genutzt werden



Repositories  
sind unabhängig

## Initialisieren eines Repository



Jedes lokale Verzeichnis kann unter Versionskontrolle gestellt werden

➔ wird dadurch ein git Repository

Bestehendes Verzeichnis unter Versionskontrolle stellen

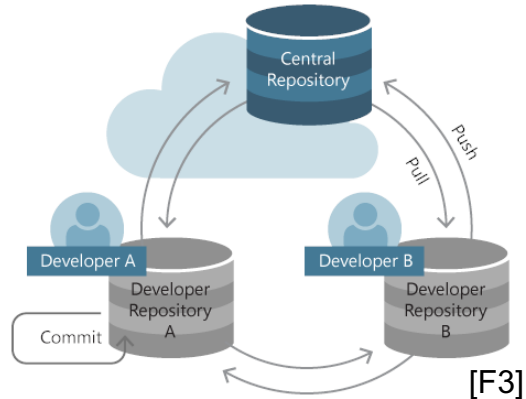
„git init“ erstellt subdirectory „.git“  
➔ Repository Skeleton

commits realisieren Versionskontrolle  
➔ version control

git Repository **clonen**

„git clone <Source>“  
➔ kopiert alle Daten von bestehendem git Repository

## Initialisieren eines Repository



Jedes lokale Verzeichnis kann unter Versionskontrolle gestellt werden

→ wird dadurch ein git Repository

Bestehendes Verzeichnis unter Versionskontrolle stellen

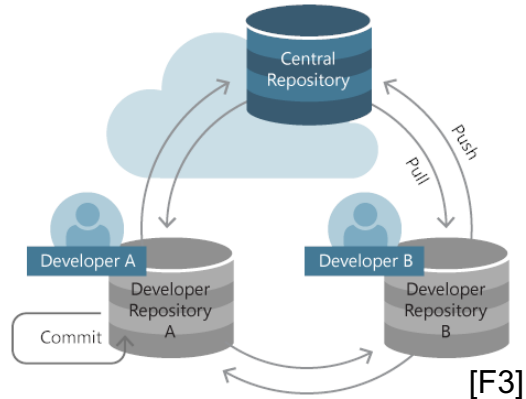
„**git init**“ erstellt subdirectory „**.git**“  
→ Repository Skeleton

**commits** realisieren Versionskontrolle  
→ version control

git Repository **clonen**

„**git clone <Source>**“  
→ kopiert alle Daten von bestehendem git Repository

## Initialisieren eines Repository



Jedes lokale Verzeichnis kann unter Versionskontrolle gestellt werden

➔ wird dadurch ein git Repository

Bestehendes Verzeichnis unter Versionskontrolle stellen

„git init“ erstellt subdirectory „.git“  
➔ Repository Skeleton

commits realisieren Versionskontrolle  
➔ version control

### git Repository **clonen**

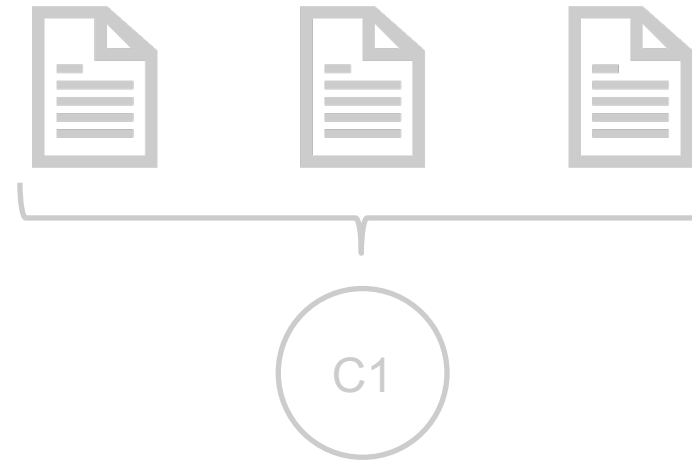
„git clone <Source>“  
➔ kopiert alle Daten von **bestehendem git Repository**

# Commit

To commit(engl. festlegen)

**commit** → stellt einen Snapshot  
des (lokalen) Repositories dar

muss **Message** beinhalten

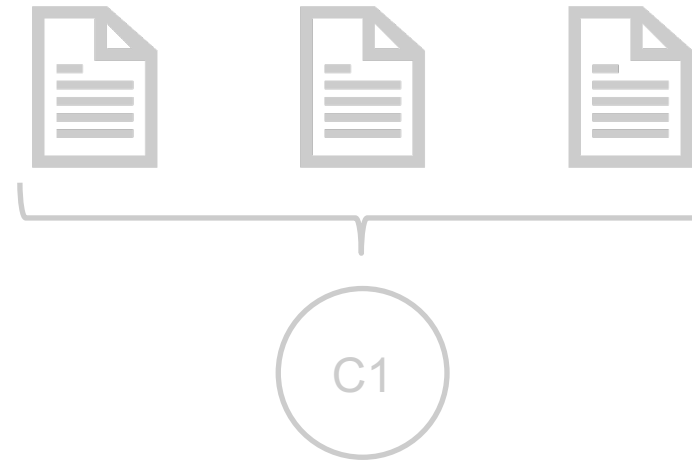


# Commit

To commit(engl. festlegen)

**commit** → stellt einen Snapshot  
des (lokalen) Repositories dar

muss **Message** beinhalten

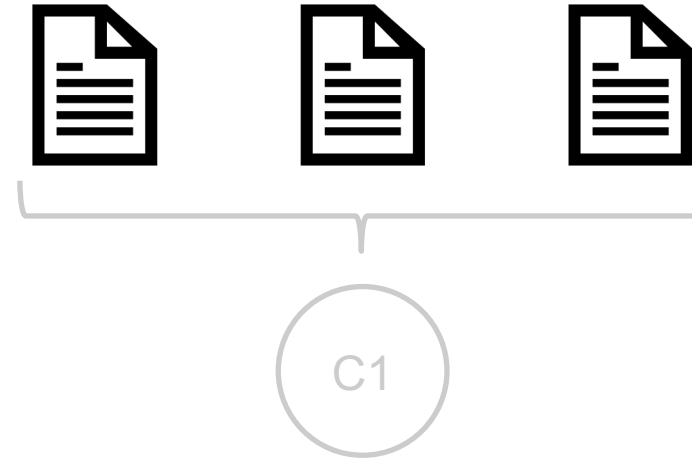


# Commit

To commit(engl. festlegen)

**commit** → stellt einen Snapshot  
des (lokalen) Repositories dar

muss **Message** beinhalten



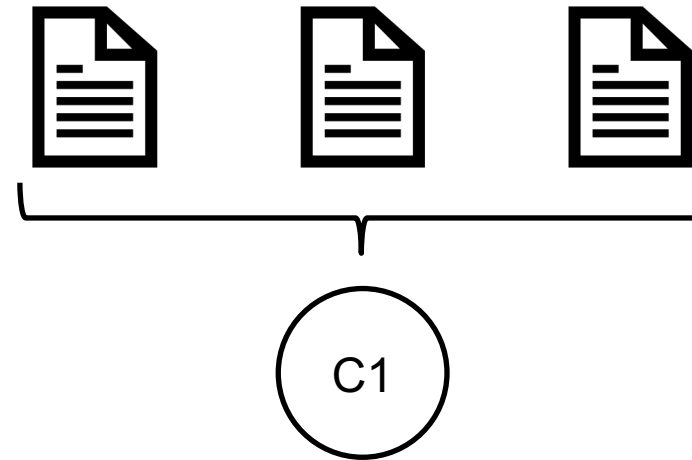


# Commit

To commit(engl. festlegen)

**commit** → stellt einen Snapshot  
des (lokalen) Repositories dar

muss **Message** beinhalten

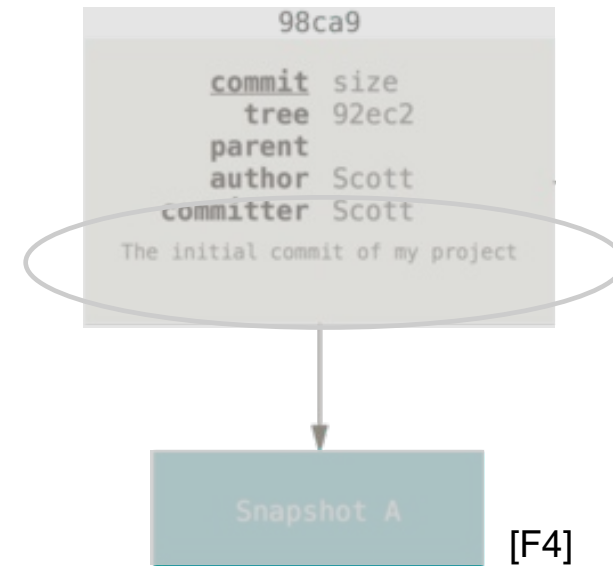


# Commit

To commit(engl. festlegen)

**commit** → stellt einen Snapshot  
des (lokalen) Repositories dar

muss **Message** beinhalten

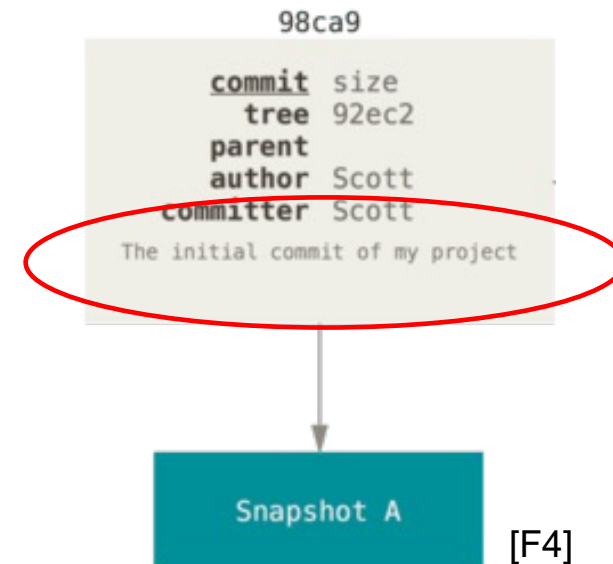


# Commit

To commit(engl. festlegen)

**commit** → stellt einen Snapshot  
des (lokalen) Repositories dar

muss **Message** beinhalten

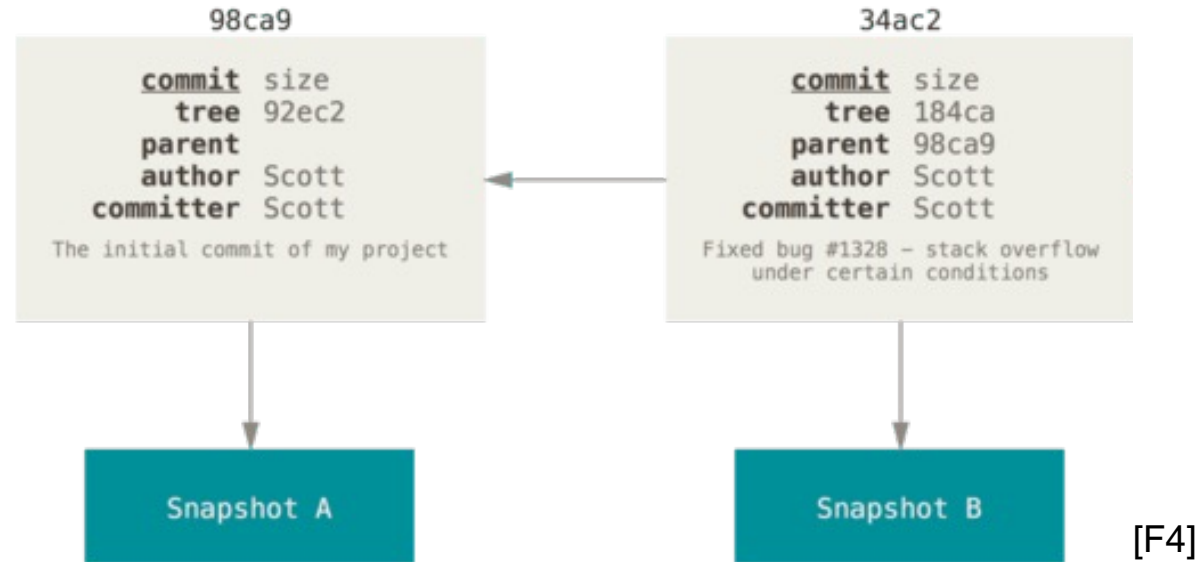


# Branches

Git speichert Daten als geordnete **Serie von Snapshots**

git commit

- **commit object** beinhaltet meta-daten
- snapshot **staged content**
- pointer zu **vorangegangenen commits**



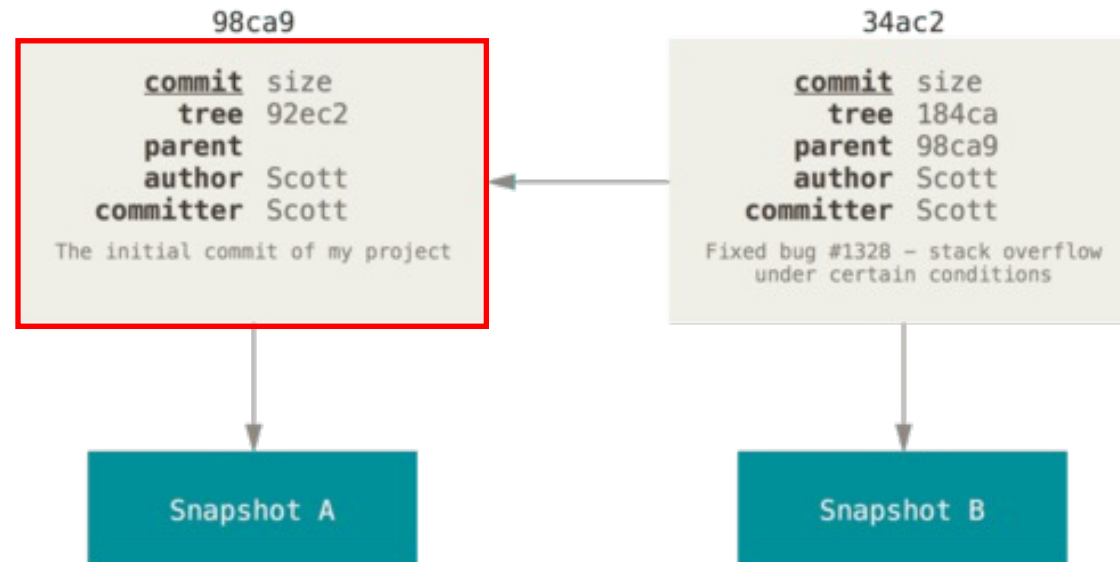
[2]

# Branches

Git speichert Daten als geordnete **Serie von Snapshots**

**git commit**

- **commit object** beinhaltet meta-daten
- snapshot **staged content**
- pointer zu **vorangegangenen commits**



[F4]

[2]

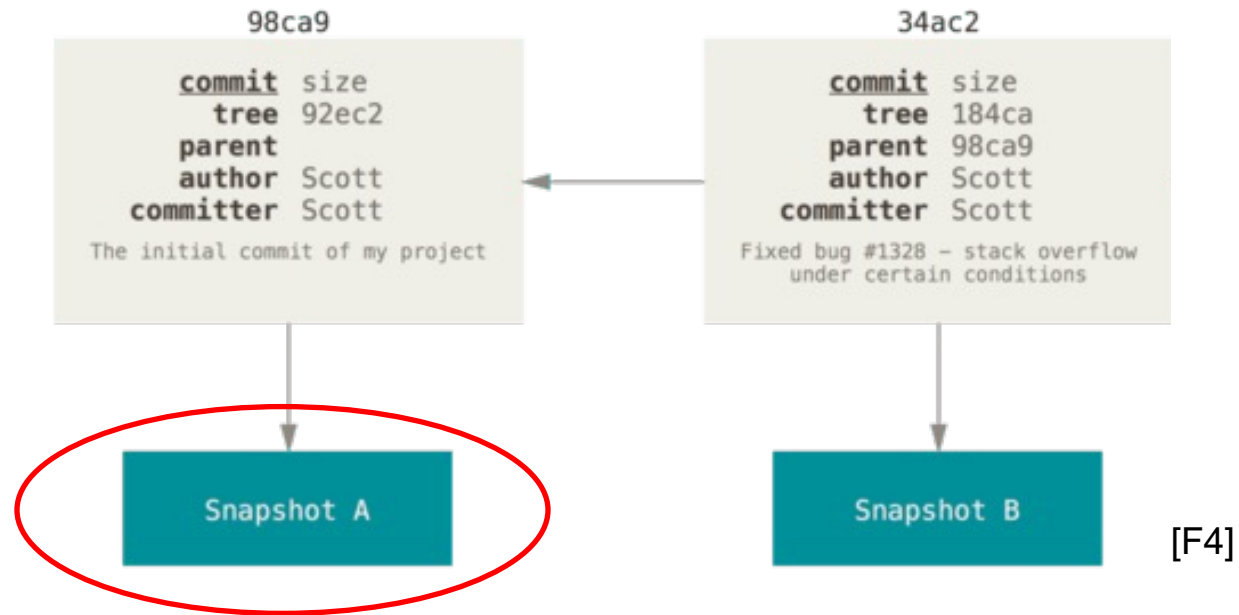
# Branches

Git speichert Daten als geordnete **Serie von Snapshots**

**git commit**

→ **commit object** beinhaltet meta-daten

- **snapshot staged content**
- pointer zu **vorangegangenen commits**



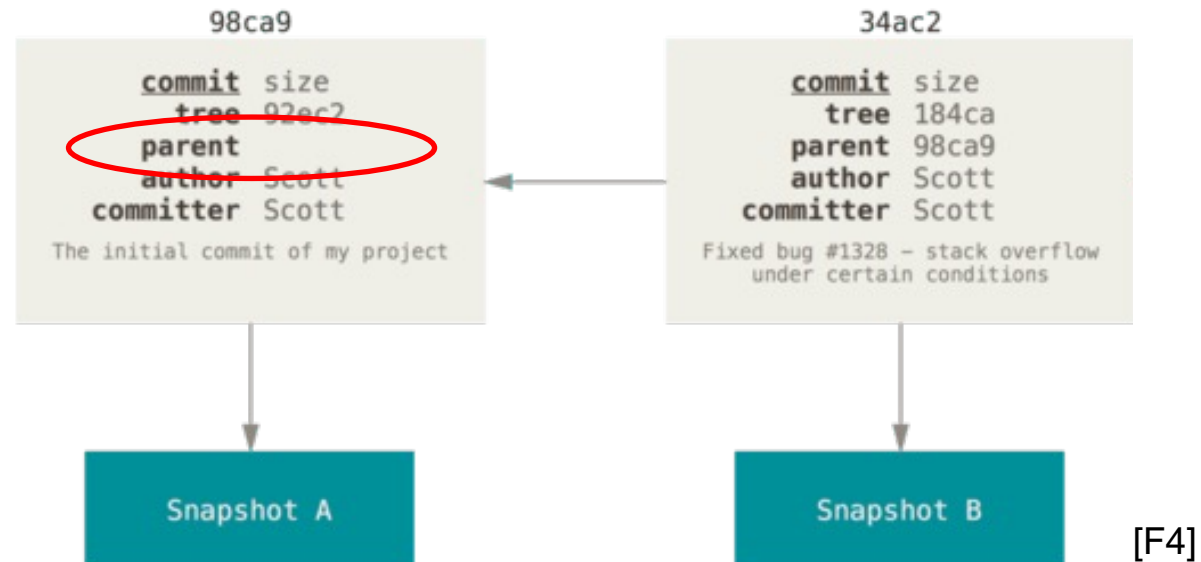
[2]

# Branches

Git speichert Daten als geordnete **Serie von Snapshots**

**git commit**

- **commit object** beinhaltet meta-daten
- snapshot **staged content**
- **pointer zu vorangegangenen commits**

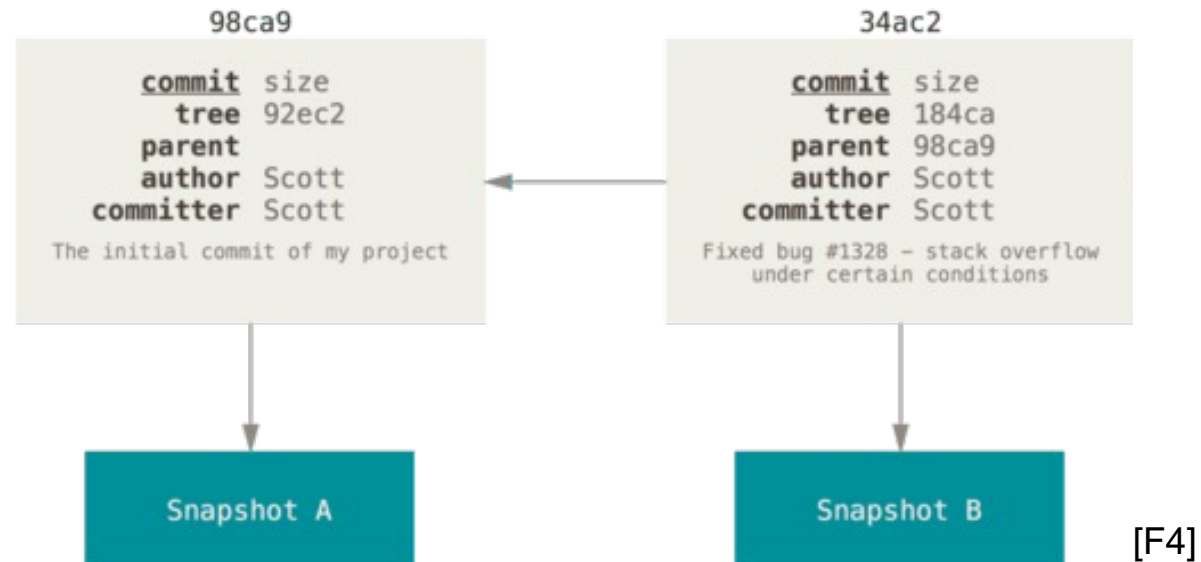


[2]

# Branches

Git speichert Daten als geordnete **Serie von Snapshots**

Wie werden daraus  
nun branches?

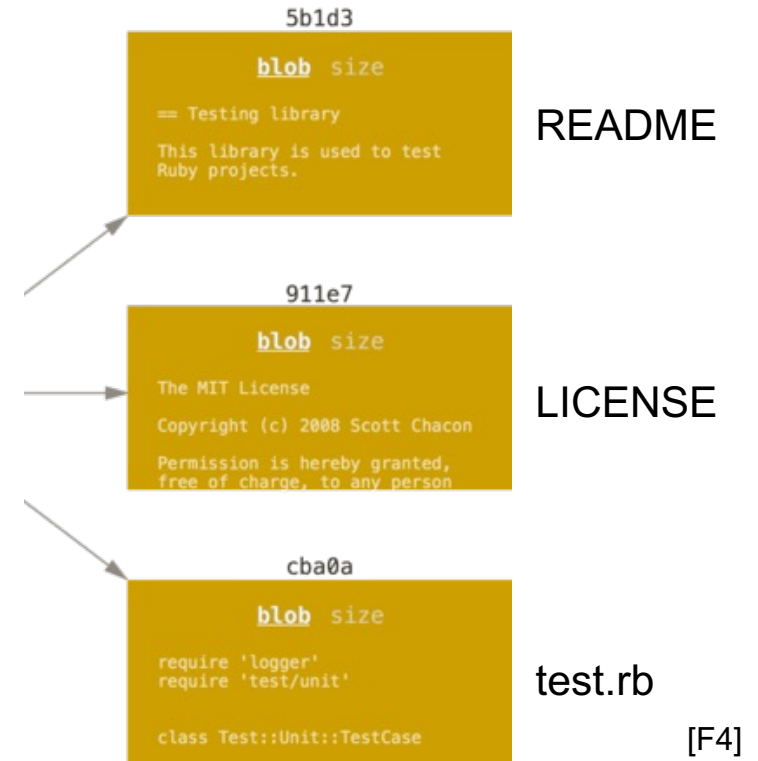


[2]



# Branches

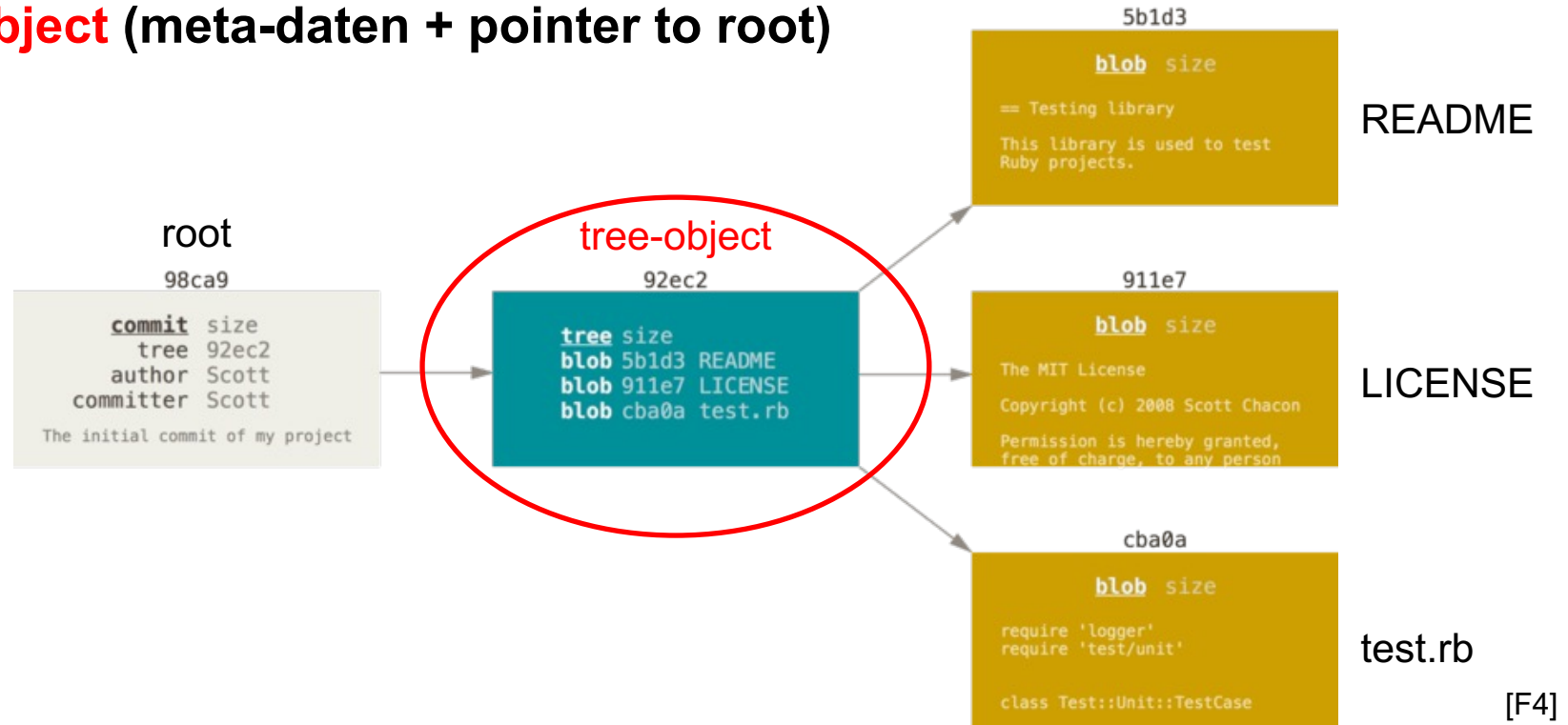
Beispiel: 3 lokale files  
→ **commit**



[2]

# Branches

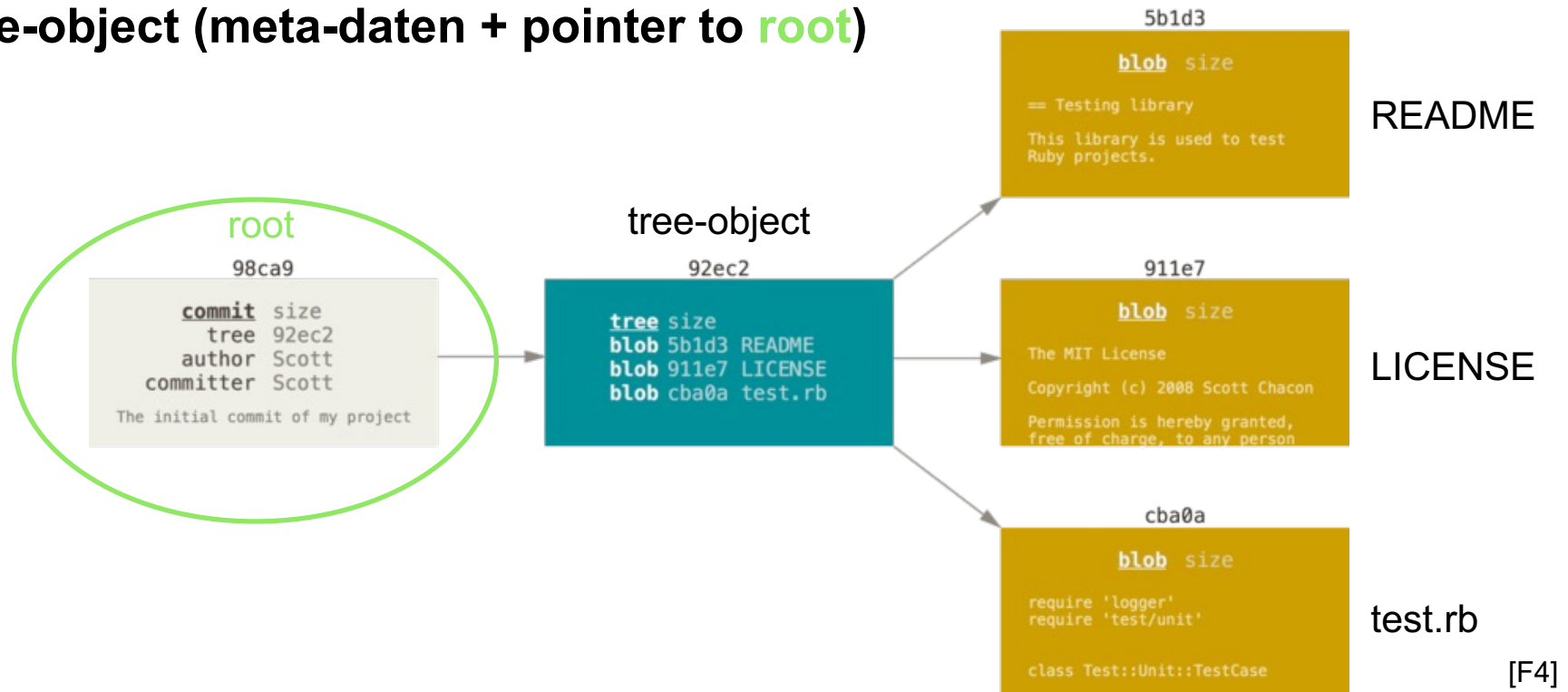
Git erzeugt ein **tree-object** (meta-daten + pointer to root)  
im git Repository



[2]

# Branches

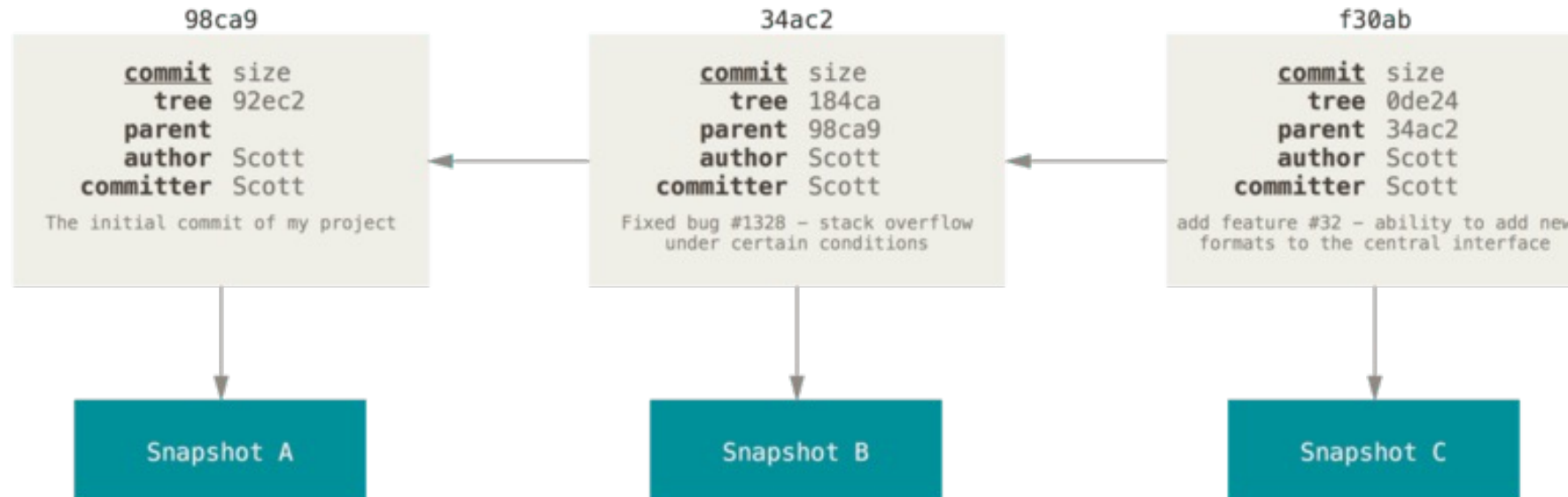
Git erzeugt ein **tree-object** (meta-daten + pointer to **root**)  
im git Repository



[2]

# Branches

Weitere commits → Pfadstruktur

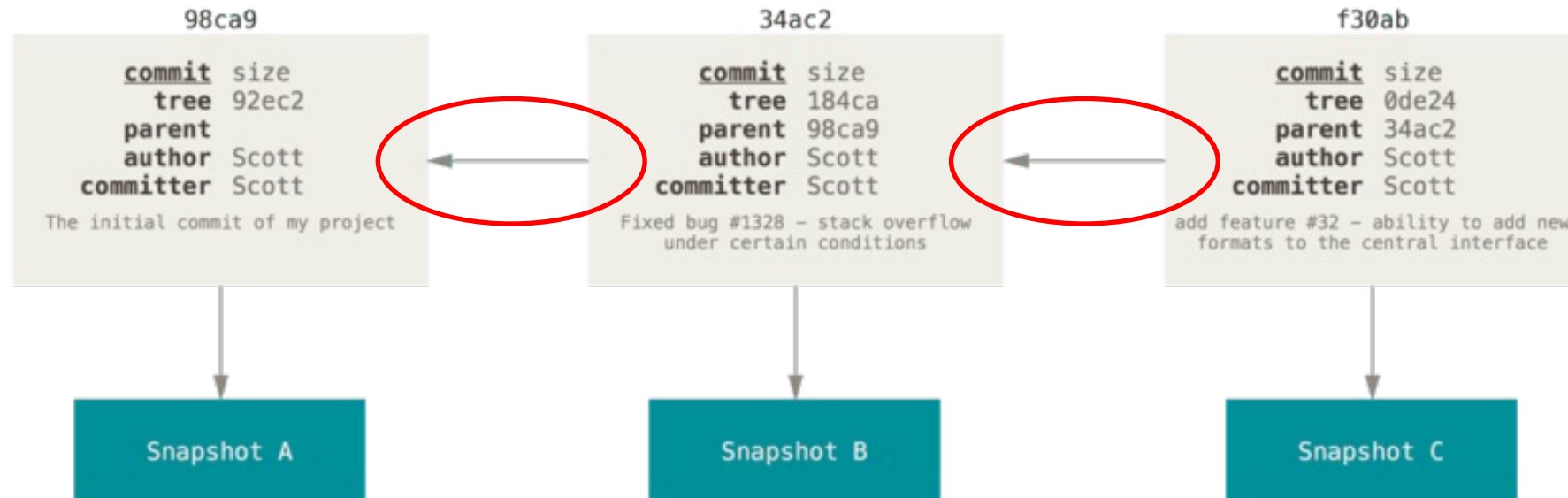


[F4]

[2]

# Branches

**Pfadstruktur** durch Pointer → Sprünge zu vergangenen Zuständen möglich

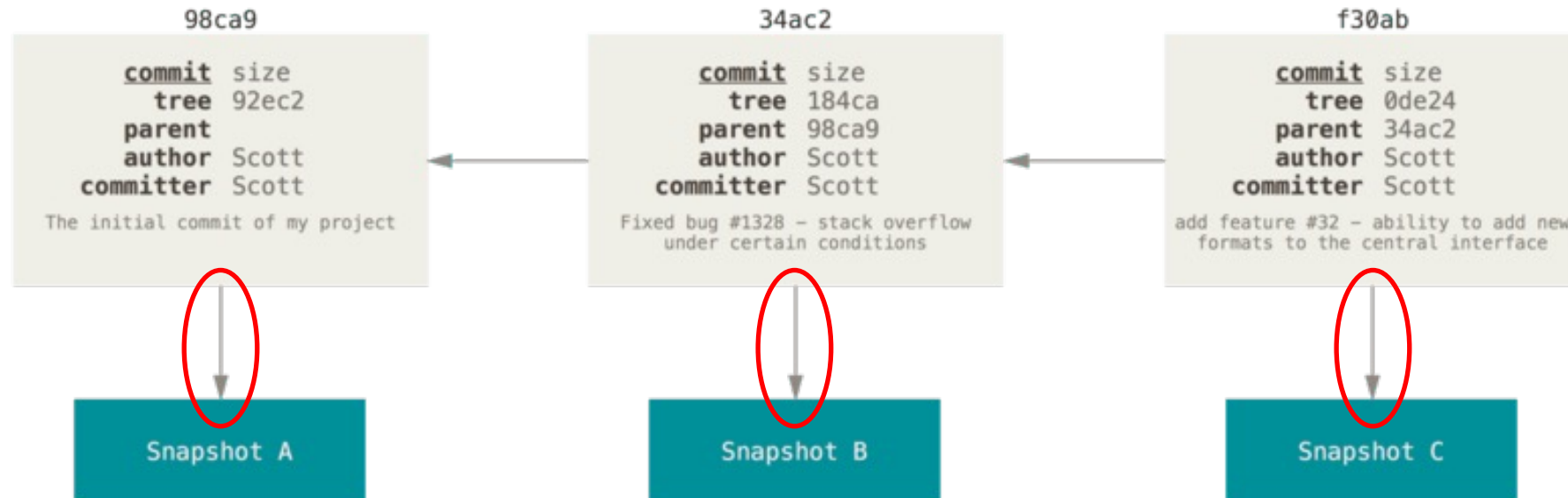


[F4]

[2]

# Branches

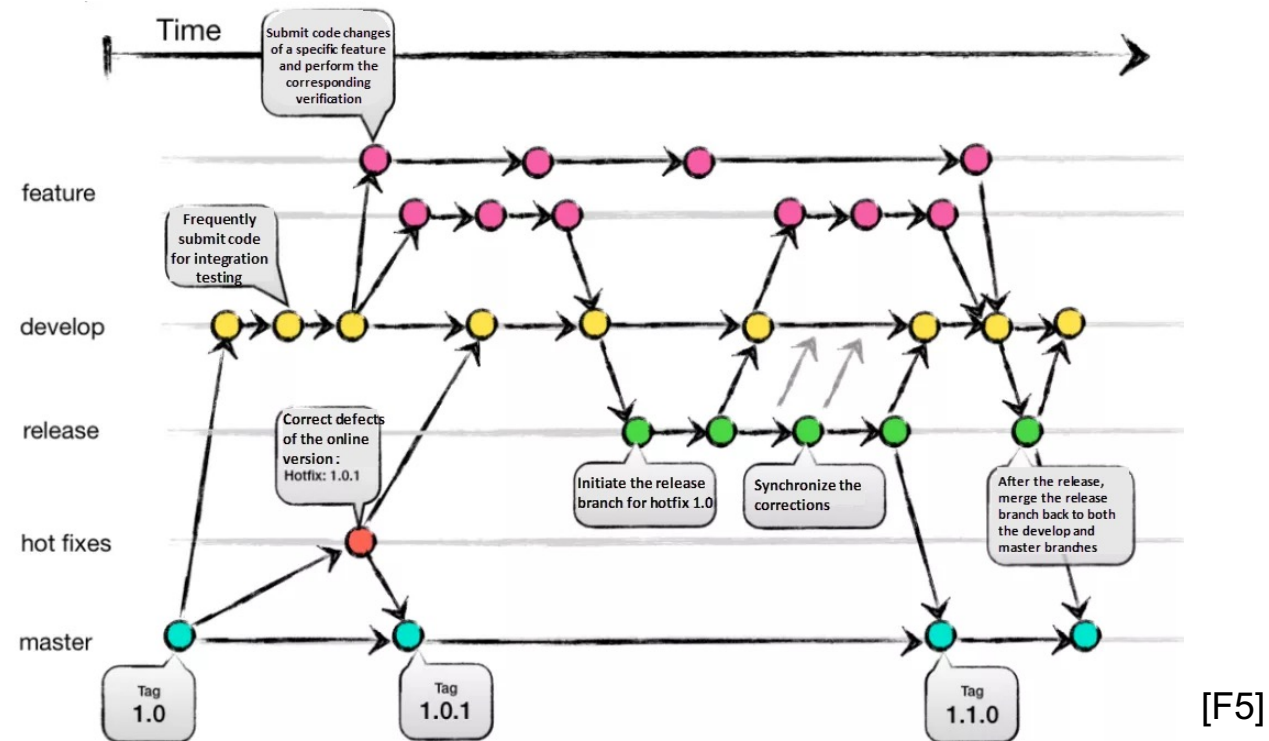
**Pfadstruktur** durch Pointer → Sprünge zu vergangenen Zuständen möglich



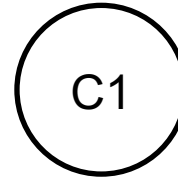
[F4]

[2]

# Branches



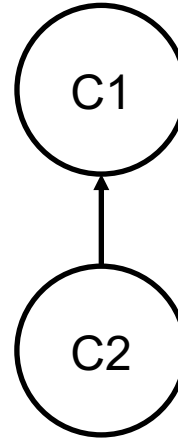
# Beispiel für branching & merging



Commite Änderungen  
→ erzeuge commit **C2**



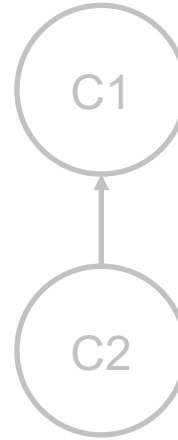
## Beispiel für branching & merging



Was wenn wir ein neues feature entwickeln wollen ohne den  
aktuell funktionieren Zustand zu gefährden?

[2]

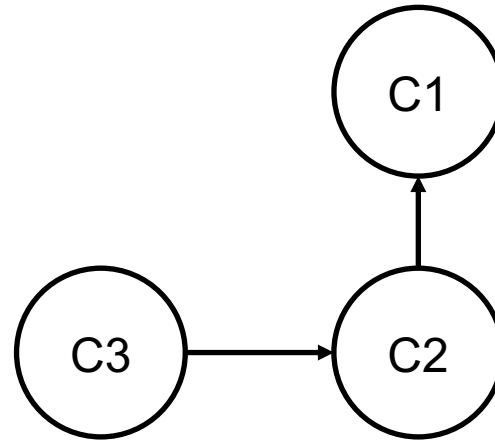
## Beispiel für branching & merging



Was wenn wir ein neues feature entwickeln wollen, ohne den aktuell funktionierenden Zustand zu gefährden?

[2]

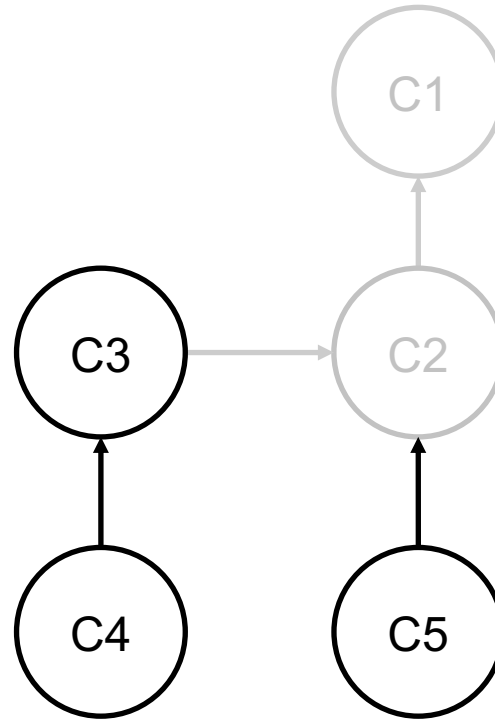
## Beispiel für branching & merging



**Neuer branch** mit  
erster Rohversion des  
neuen features

[2]

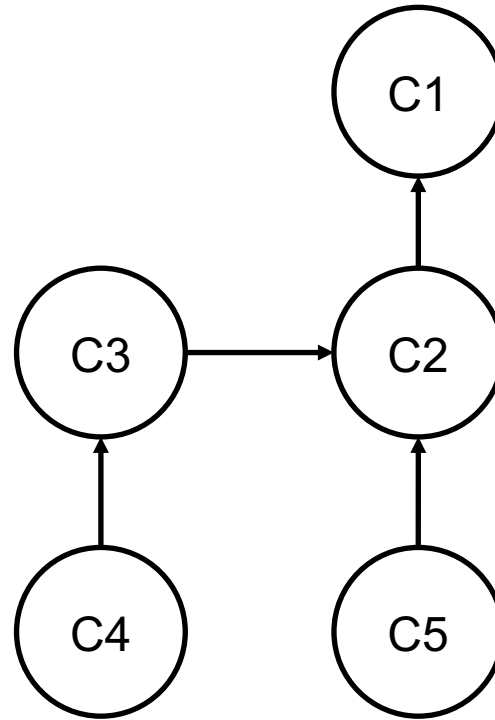
## Beispiel für branching & merging



Entwickeln  
➔ neue Zwischenzustände  
entlang des branches +  
weiterer commit

[2]

## Beispiel für branching & merging



Neues feature funktioniert..  
Was jetzt?

[2]

# Merge

Um unabhängige Änderungen Daten konsistent zu speichern  
→ **mergen**

**Vergleich der Daten → 2 Fälle**

**Disjunkte- vs inkonsistente Änderungen**

# Merge

## 1. Fall disjunkte Änderungen → bestehender code wird einfach erweitert

lib/compose.js

View file @e71062d

<pre>@@ -12,19 +12,27 @@ function setWritability(obj, writable) { 12   }); 13 } 14</pre> <div>origin//their changes Use this</div> <pre>15 function mixin(base, mixins) { 16   base.mixedIn = base.hasOwnProperty('mixedIn') ? base.mixedIn : 17   []; 18 19   for (var i = 0; i &lt; mixins.length; i++) { 20     if (base.mixedIn.indexOf(mixins[i]) == -1) { 21       module.exports = { 22         mixin: mixin 23       }; 24     } 25   } 26 }</pre>	<pre>@@ -12,19 +12,27 @@ function setWritability(obj, writable) { 12   }); 13 } 14</pre> <div>HEAD//our changes Use this</div> <pre>15 function mixin(base, mixins) { 16   base.mixedIn = Object.prototype.hasOwnProperty.call(base, 'mi 17   xedIn') ? base.mixedIn : []; 18 19   for (var i = 0; i &lt; mixins.length; i++) { 20     if (base.mixedIn.indexOf(mixins[i]) == -1) { 21       setWritability(base, false); 22       mixins[i].call(base); 23       base.mixedIn.push(mixins[i]); 24     } 25   } 26 27   for (var i = 0; i &lt; mixins.length; i++) { 28     if (base.mixedIn.indexOf(mixins[i]) == -1) { 29       module.exports = { 30         mixin: mixin 31       }; 32     } 33   } 34 }</pre>
--	---

[F6]

[2]

# Merge Conflict

## 2. Fall inkonsistente Änderungen → merge conflict

Merge Revisions for /Users/jetbrains/PhpstormProjects/MyProject/src/logoff.php

↑ ↓ ↕ Apply non-conflicting changes: >> Left >>> All <<< Right ✎ Do not ignore >> No changes. 1 conflict

Uncommitted changes from stash	Result	Changes from Remote
<pre>&lt;?php session_start(); \$_SESSION = array(); if (isset     (\$_COOKIE     [session_name()])) {     setcookie     (session_name(),     '', time() - 500,     '/'); } session_destroy(); header("Location: logon.php"); exit;</pre>	<pre>1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10</pre>	<pre>&lt;?php session_start(); \$_SESSION = array(); if (isset     (\$_COOKIE[session_name()     ])) {     setcookie(session_name     (), '', time() - 600,     '/'); } session_destroy(); header("Location: logon.php"); exit;</pre>

Accept Left Accept Right Abort Apply

[2]

[F7]



# Resolving merge conflicts

Wie sollte die resultierende Datei aussehen?

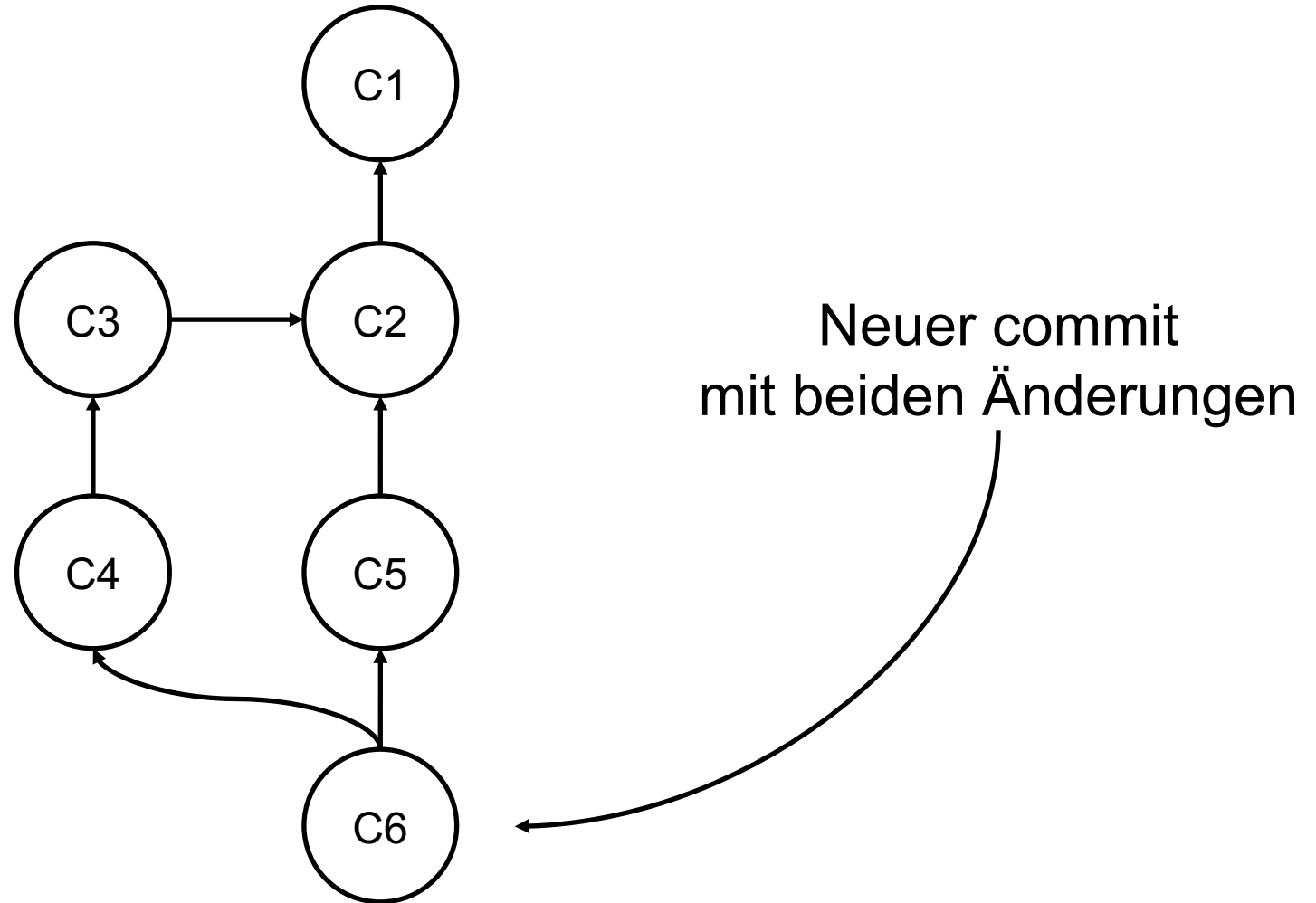
```
<?php ✓
session_start();
$_SESSION = array();
if (isset
($_COOKIE[session_name()
])) {
    setcookie(session_name
(), '', time() - 600,
'/');
}
session_destroy();
header("Location:
logon.php");
exit;
```

## Fehlende Eindeutigkeit

- ➔ Entwickler muss entscheiden wie der conflict aufgelöst wird
- ➔ nur mit aufgelösten conflicts konsistente Datensicherung möglich
- ➔ andernfalls kein commit mit mehreren parents (dringend benötigt für multi-user projects)

[2]

# Resolving merge conflicts



[2]

# Branches

In a nutshell: **Ein branch ist eine geordnete Menge an commits.**

## Vorteile

- Sprünge zu vorangegangenen Zuständen sind einfach  
→ leichter workflow
- Branches sind light weight  
→ häufiges **branchen** möglich (schnelle Prototypen)
- Umfangreiche Menge an keywords erlaubt komplexes mergen und branchen von commits
- Mehrere Entwickler können losgelöst voneinander entwickeln + konsistent zu gemeinsamen Projekt beitragen

# Branches

In a nutshell: **Ein branch ist eine geordnete Menge an commits.**

## Vorteile

- Sprünge zu vorangegangenen Zuständen sind einfach  
➔ leichter workflow
- Branches sind light weight  
➔ häufiges **branchen** möglich (schnelle Prototypen)
- Umfangreiche Menge an keywords erlaubt komplexes mergen und branchen von commits
- Mehrere Entwickler können losgelöst voneinander entwickeln + konsistent zu gemeinsamen Projekt beitragen

# Branches

In a nutshell: **Ein branch ist eine geordnete Menge an commits.**

## Vorteile

- Sprünge zu vorangegangenen Zuständen sind einfach  
→ leichter workflow
- Branches sind light weight  
→ häufiges **branchen** möglich (schnelle Prototypen)
- Umfangreiche Menge an keywords erlaubt komplexes mergen und branchen von commits
- Mehrere Entwickler können losgelöst voneinander entwickeln + konsistent zu gemeinsamen Projekt beitragen

# Branches

In a nutshell: **Ein branch ist eine geordnete Menge an commits.**

## Vorteile

- Sprünge zu vorangegangenen Zuständen sind einfach  
→ leichter workflow
- Branches sind light weight  
→ häufiges **branchen** möglich (schnelle Prototypen)
- Umfangreiche Menge an keywords erlaubt komplexes mergen und branchen von commits
- Mehrere Entwickler können losgelöst voneinander entwickeln + konsistent zu gemeinsamen Projekt beitragen

# Branches

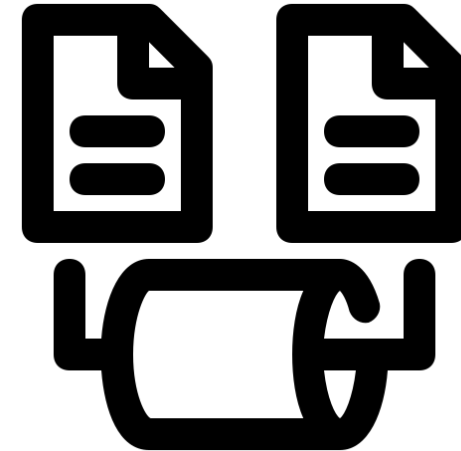
In a nutshell: **Ein branch ist eine geordnete Menge an commits.**

## Vorteile

- Sprünge zu vorangegangenen Zuständen sind einfach  
→ leichter workflow
- Branches sind light weight  
→ häufiges **branchen** möglich (schnelle Prototypen)
- Umfangreiche Menge an keywords erlaubt komplexes mergen und branchen von commits
- Mehrere Entwickler können losgelöst voneinander entwickeln + konsistent zu gemeinsamen Projekt beitragen

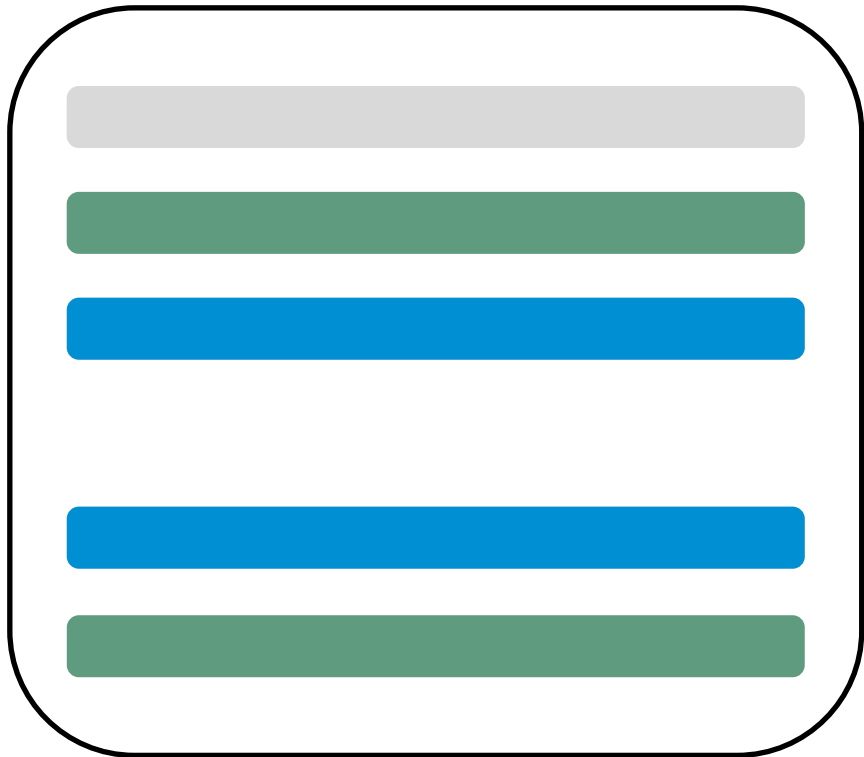
# CI/CD

- DevOps-Methoden, die darauf abzielen, den Entwicklungs- und Bereitstellungsprozess von Software zu verbessern
- Sie beziehen sich auf verschiedenen Phasen einer automatisierten Software-Release-Pipeline
- Den gesamten Lebenszyklus der zu entwickelnden Anwendung durch agile Entwicklungsmethoden zu verwalten und optimieren

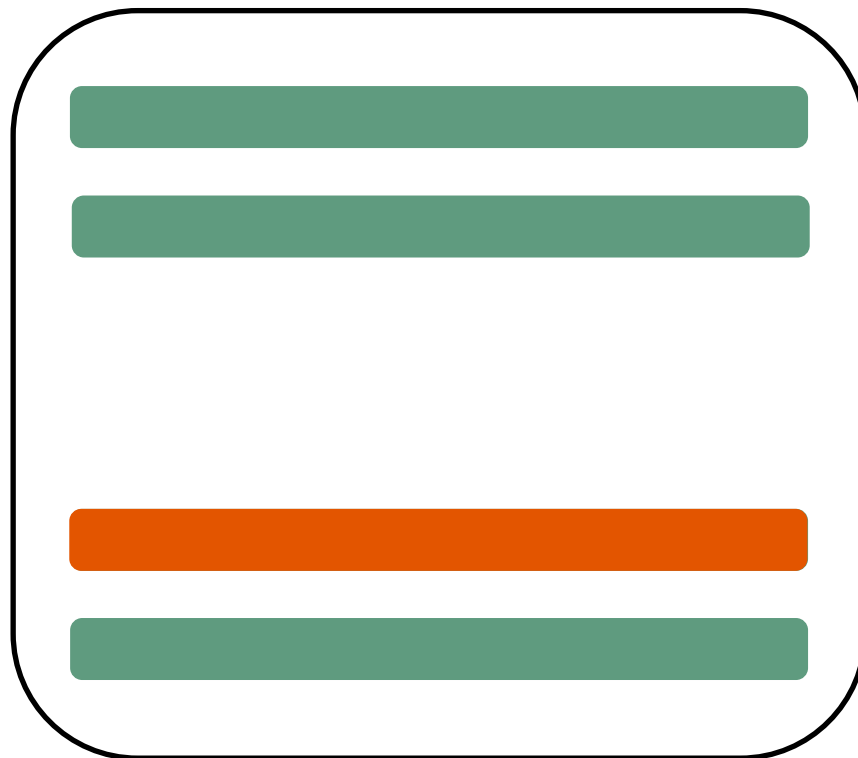
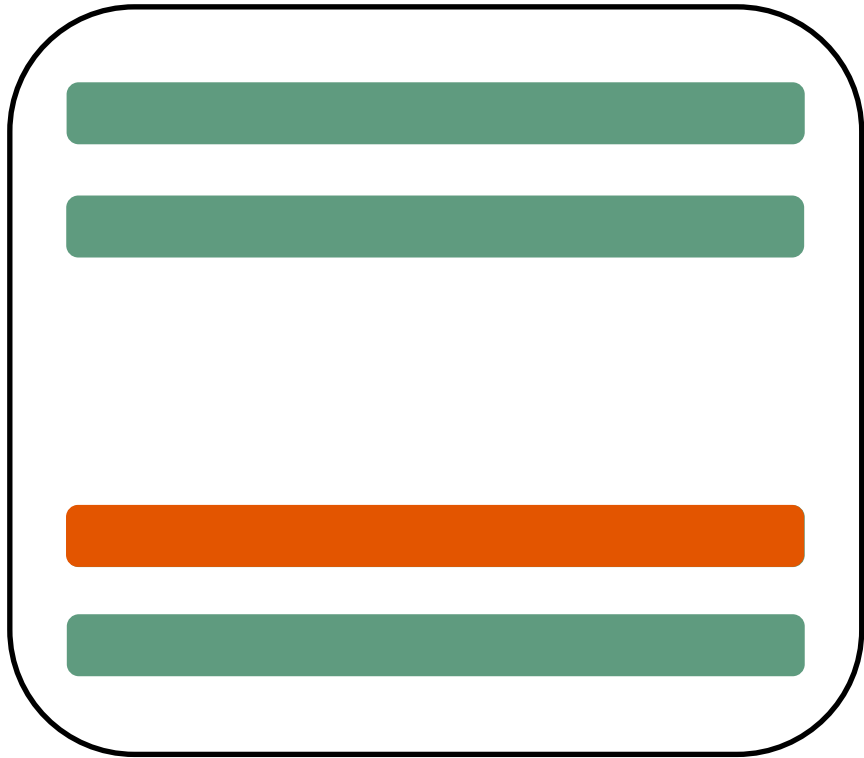




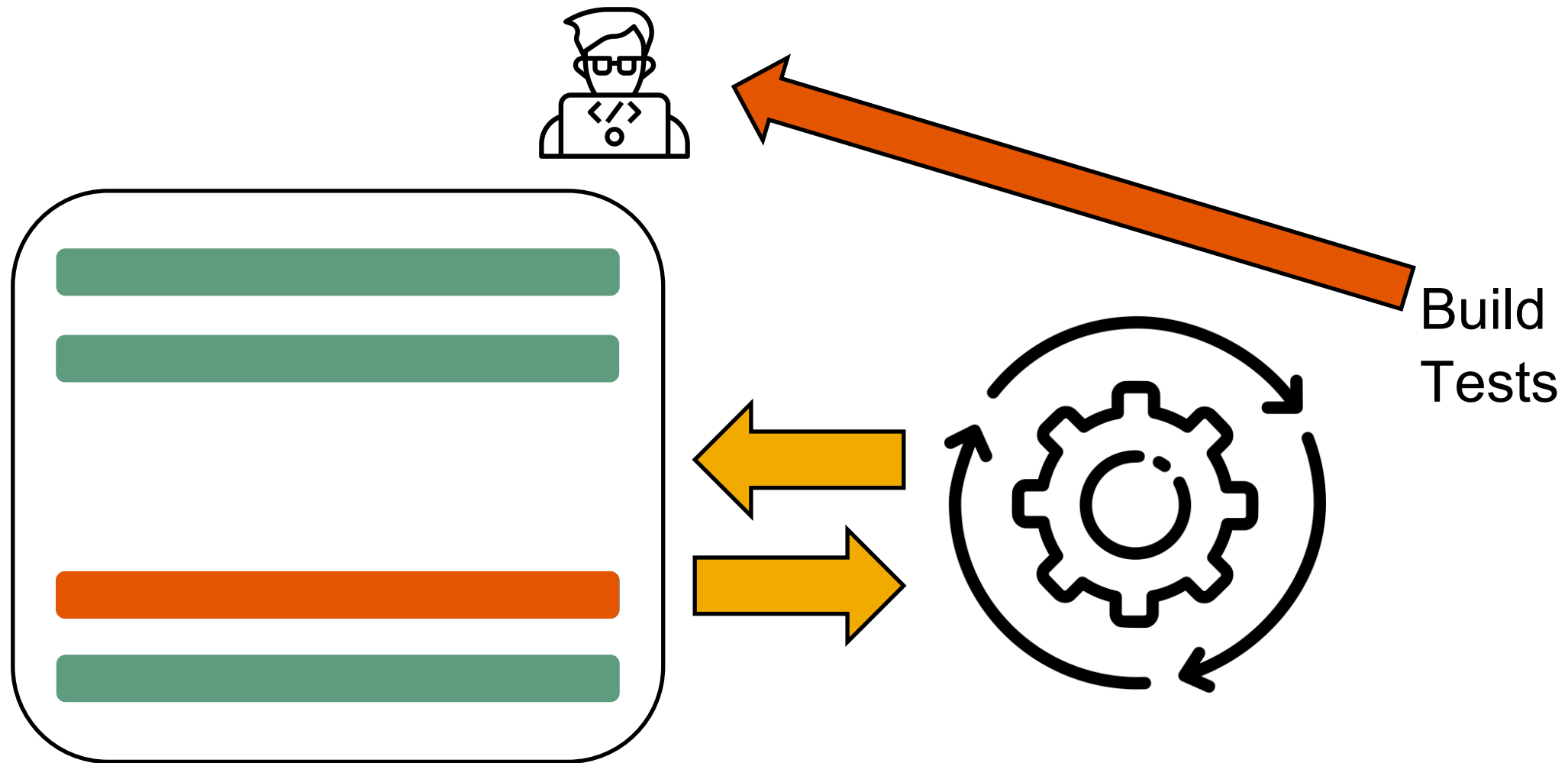
# Ohne Continuous Integration



# Mit Continuous Integration

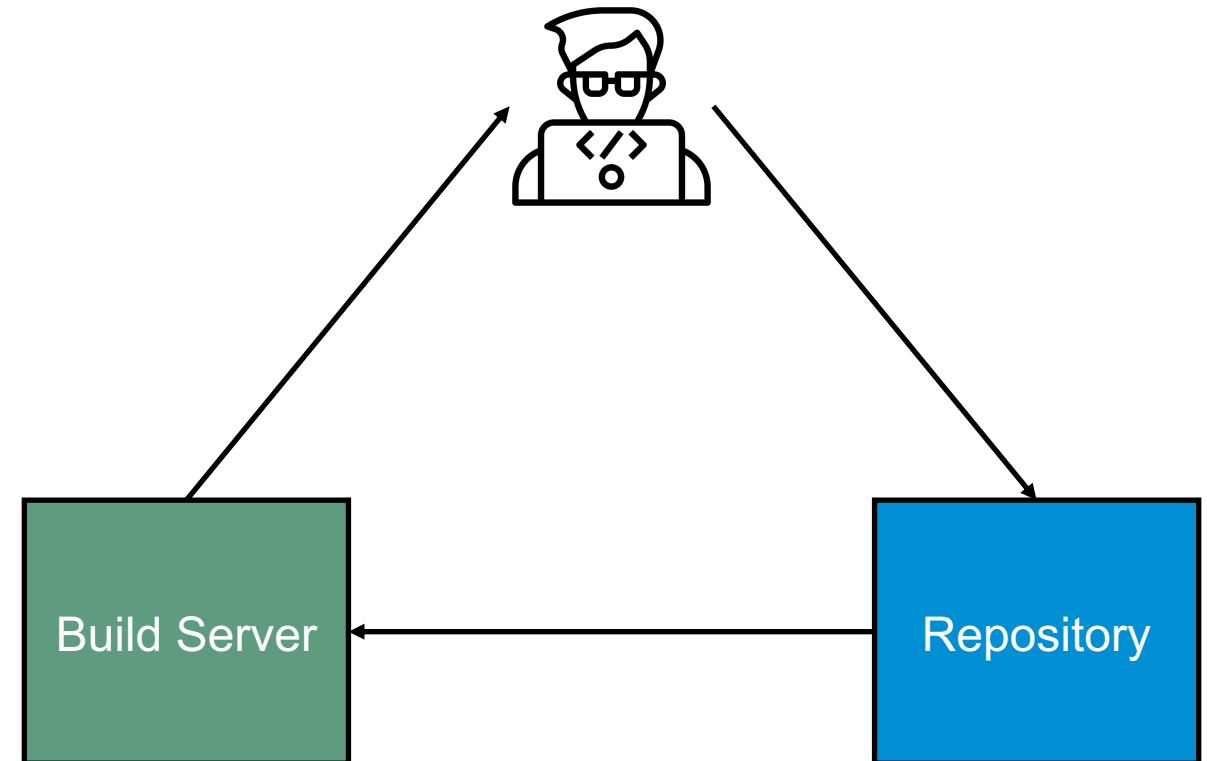


# Mit Continuous Integration



# Ablauf von Continuous Integration

- Ein gemeinsames Repository wird festgelegt
- Vor der Integration sollten lokale Testskripte ausgeführt werden
- Entwickler führen regelmäßig push oder pull requests durch
- Änderungen werden gebaut und getestet
- Entwickler erhalten Feedback
- Bei Fehlern wird eine Behebung durchgeführt

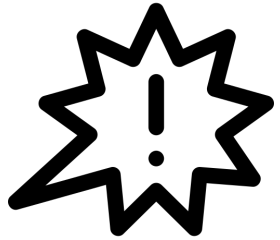


[5]

# Best Practices der Continuous Integration

- Verwenden eines Versionskontrollsystems für Code-Integration
- Regelmäßige Integration von Codeänderungen
- Testorientierte Entwicklung
- Einrichtung automatisierter Build- und Testprozesse
- Schnelle Fehlerbehebung

# Vorteile von Continuous Integration



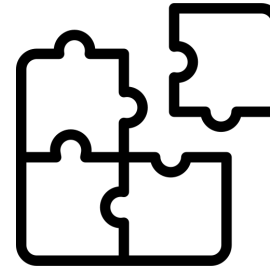
Vermeiden von  
Merge-  
Konflikten



Frühes  
Erkennen und  
schnelles  
Finden von  
Fehlern



Skalierbarkeit



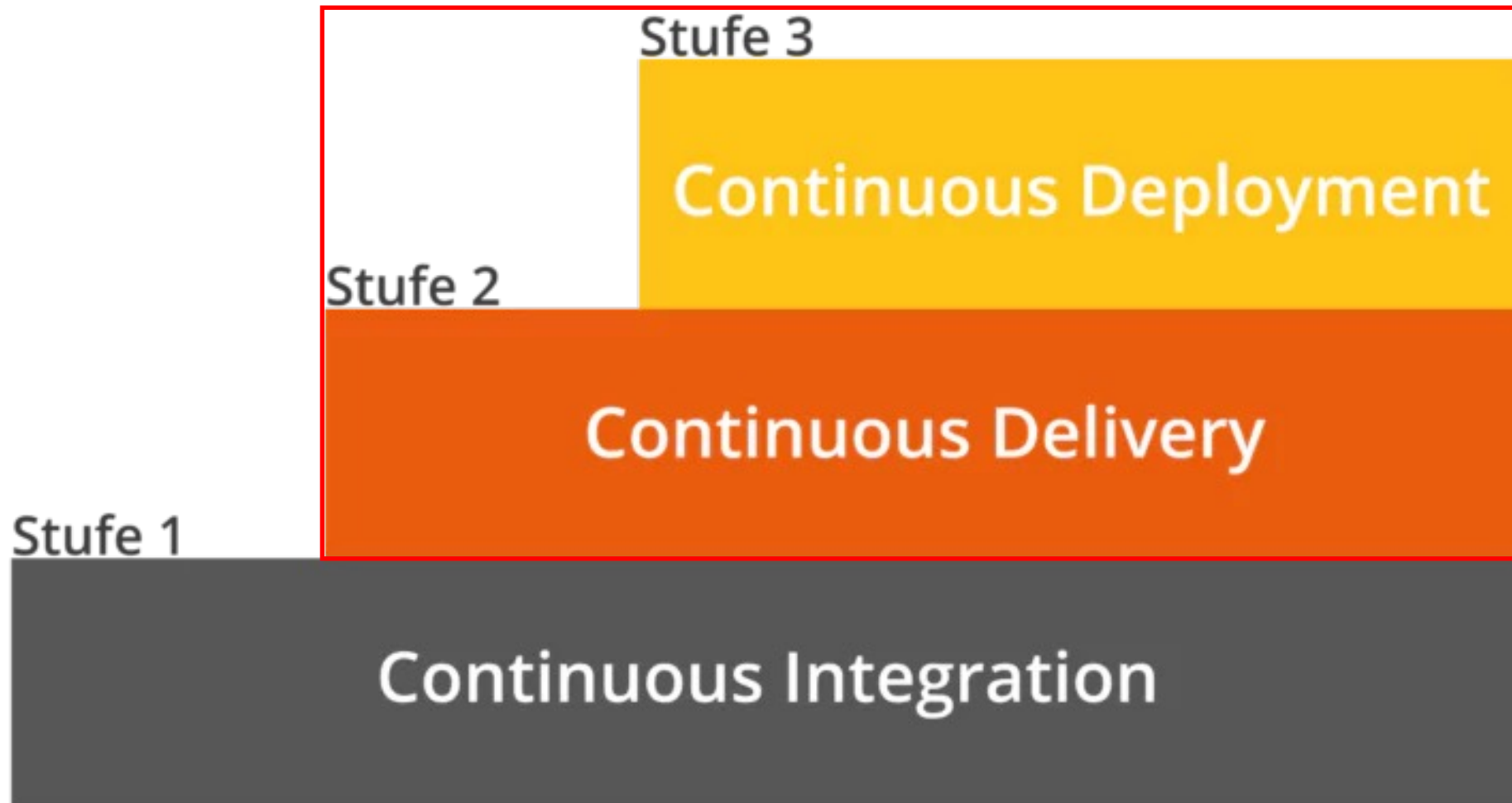
Jederzeit ein  
baufähiger  
Zustand



Schnelle  
Feedbackloop

[6]

# Continuous Delivery/Deployment



# Übersicht CI/CD

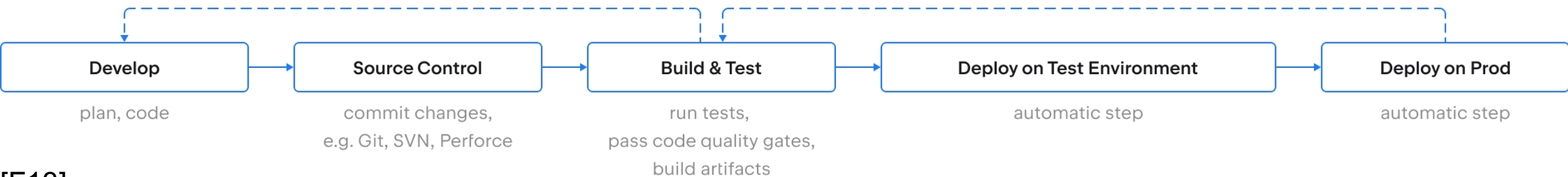
## Continuous Integration:



## Continuous Delivery:



## Continuous Deployment:





# Einführung in Continuous Delivery und Deployment

## Continuous Delivery

- Automatisiert manuelle Schritte zur Kompilierung und Veröffentlichung von Software
- Ziel: Code immer in veröffentlichungsbereitem Zustand halten
- Automatisierte Tests als Teil des CD-Workflows
- Beschleunigung des Software-Lieferprozesses durch Automatisierung
- Verkürzung der Feedbackschleife und schnellere Bereitstellung
- Robuster und reproduzierbarer Releaseprozess ermöglicht häufigere Durchführung von Verbesserungen

# Einführung in Continuous Delivery und Deployment

## Continuous Deployment

- Automatisiert Build-, Test- und Deployment-Schritte bis zur automatischen Übernahme in die Produktion
- Ziel: Neue Features schnellstmöglich bereitstellen, ohne Kompromisse bei der Qualität einzugehen
- Voraussetzung: Ausgereiftes Continuous-Integration-System und mehrere Phasen umfassender Continuous-Delivery-Prozess
- Kleine Codeänderungen in den Master-Branch übernommen, durchlaufen automatisierten Build- und Testprozess
- Robuste und zuverlässige automatisierte Deployment-Pipeline erlauben Releases mehrmals täglich
- Automatisierung des endgültigen Rollouts nicht für jedes Softwareprojekt geeignet, aber einzelne Elemente können trotzdem profitabel sein

# Einführung in Continuous Delivery und Deployment

Continuous Delivery und Continuous Deployment werden oft synonym verwendet

**Continuous Delivery** bezieht sich auf die kontinuierliche Auslieferung von Software, während **Continuous Deployment** die automatische Auslieferung von Software in die Produktionsumgebung beschreibt.

# Übersicht CI/CD

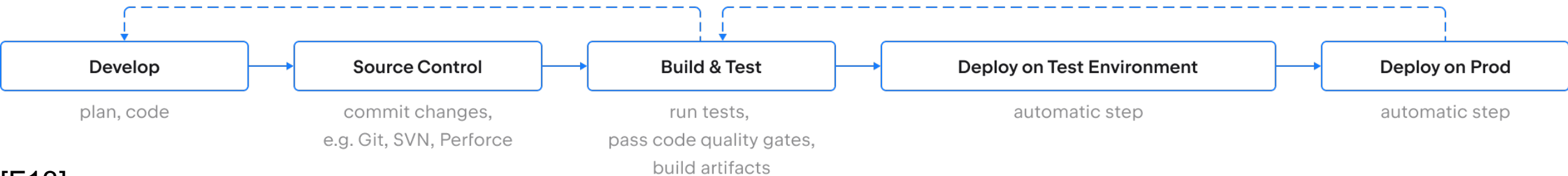
## Continuous Integration:



## Continuous Delivery:



## Continuous Deployment:



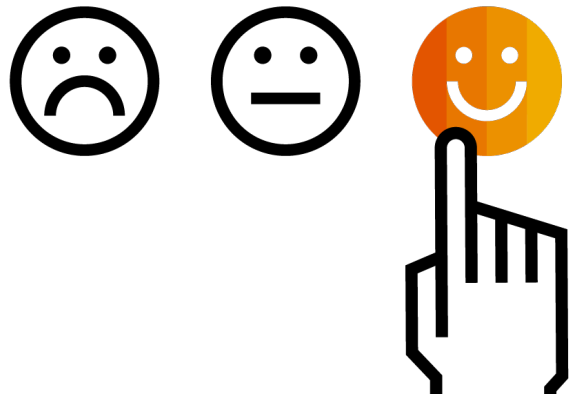
# Vorteile von Continuous Delivery und Deployment

1. Schnelle Markteinführung
  - Wettbewerbsvorteil durch rasche Bereitstellung neuer Funktionen
2. Höhere Softwarequalität
  - Kontinuierliche Integration und Tests reduzieren Fehler und Ausfallzeiten
3. Automatisierung
  - Effizienzsteigerung und Fehlerreduktion durch automatisierte Prozesse
4. Kundenfeedback
  - Schnellere Iterationen ermöglichen schnelleres Lernen und Anpassungen

# Vorteile von Continuous Delivery und Deployment

5. Reduzierte Rollback-Kosten
  - Kleinere Änderungen erleichtern Fehlerbehebung und Rollbacks
6. Effektive Zusammenarbeit
  - Bessere Kommunikation zwischen Entwicklung, Test und Betrieb
7. Skalierbarkeit
  - Einfacheres Management und Bereitstellung von Anwendungen in großen Umgebungen
8. Risikominderung
  - Häufigere, kleinere Releases reduzieren das Risiko großer Fehlschläge

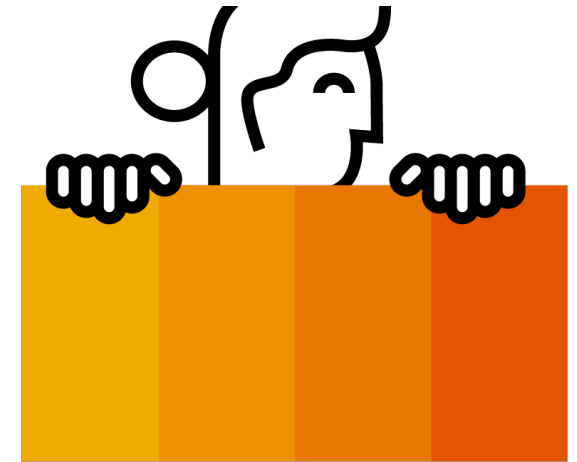
# Best Practices der Continuous Delivery und Deployment



Das gesamte Team einbinden



Keine andere Deployment-  
Methoden zulassen



Pipeline-Daten überwachen  
und Rollout kontrollieren

# CI/CD Zusammengefasst

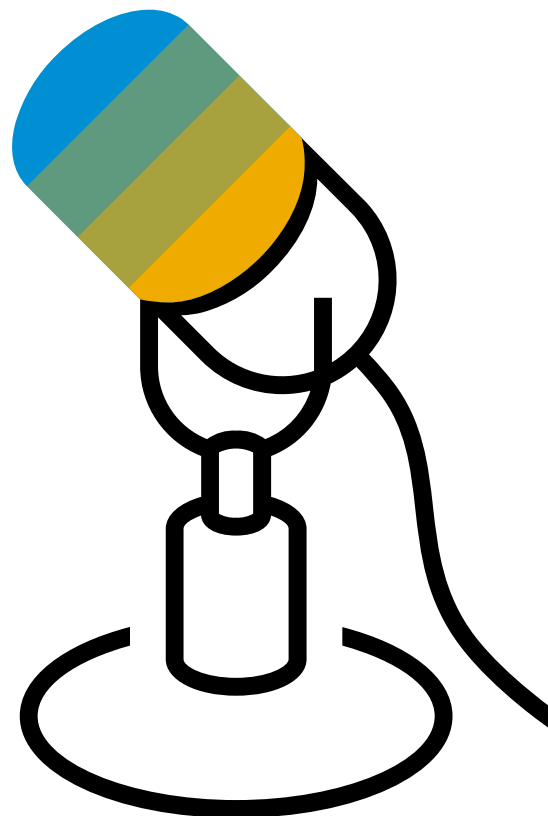
- Continuous Integration, Delivery und Deployment sind aufeinander folgende Phasen des Softwarebereitstellungsprozesses
- Helfen Teams, Software schneller zu veröffentlichen, die Feedback-Schleife zu verkürzen
- Bei **Continuous Integration** werden alle neuen Codeänderungen in den Hauptbranch integriert
- **Continuous Delivery** automatisiert manuelle Aufgaben, die zum Erstellen und Testen von Software erforderlich sind
- **Continuous Deployment** ist die logische Fortsetzung der Automatisierung von Build- und Testschritten → sobald alle erforderlichen Prüfpunkte durchlaufen



# Beispiel

**Vielen Dank für Eure Aufmerksamkeit**

**Q & A**



# Quellen

- [1] <https://github.com/about>
- [2] <https://git-scm.com/book/en/v2>
- [3] <https://www.jetbrains.com/de-de/teamcity/ci-cd-guide/>
- [4] <https://aws.amazon.com/de/devops/continuous-delivery/>
- [5] [Fowler, Martin, and Matthew Foemmel. "Continuous integration." \(2006\)](#)
- [6] <https://mindsquare.de/>
- [7] <https://www.jetbrains.com/de-de/teamcity/ci-cd-guide/benefits-of-ci-cd/>
- [8] <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>
- [9] <https://fhlug.at/~dkf/CV/ba-dvcs-analysis.pdf>

# Quellen

[10] <https://www.atlassian.com/de/git/tutorials/what-is-git>

# BildQuellen

[F1]: <https://mindsquare.de/files/Stufendiagramm-e1634130504602-730x401.png.webp>

[F2]: [https://www.jetbrains.com/teamcity/ci-cd-guide/continuous-integration-vs-delivery-vs-deployment/img/Key\\_differences\\_desktop.png](https://www.jetbrains.com/teamcity/ci-cd-guide/continuous-integration-vs-delivery-vs-deployment/img/Key_differences_desktop.png)

[F3]: [https://learn.microsoft.com/de-de/devops/\\_img/git\\_repositories.png](https://learn.microsoft.com/de-de/devops/_img/git_repositories.png)

[F4]: <https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell>

[F5]: [https://www.alibabacloud.com/blog/how-to-select-a-git-branch-mode\\_597255](https://www.alibabacloud.com/blog/how-to-select-a-git-branch-mode_597255)

[F6]: <https://about.gitlab.com/blog/2016/09/06/resolving-merge-conflicts-from-the-gitlab-ui/>

[F7]: [www.jetbrains.com/conflict-resolving](http://www.jetbrains.com/conflict-resolving)

[F8]: [https://alex106.github.io/intro2R/images/Messy\\_folder.PNG](https://alex106.github.io/intro2R/images/Messy_folder.PNG)

[F9]: <https://git-scm.com/book/en/v2/images/local.png>

[F10]: <https://git-scm.com/book/en/v2/images/centralized.png>

[F11]: <https://git-scm.com/book/en/v2/images/distributed.png>

# BildQuellen

[F12]: [https://www.flaticon.com/de/kostenloses-icon/git\\_4494740](https://www.flaticon.com/de/kostenloses-icon/git_4494740)

[F13]: [https://www.jetbrains.com/teamcity/ci-cd-guide/continuous-integration-vs-delivery-vs-deployment/img/Key\\_differences\\_desktop.png](https://www.jetbrains.com/teamcity/ci-cd-guide/continuous-integration-vs-delivery-vs-deployment/img/Key_differences_desktop.png)

[F14]: <https://mindsquare.de/files/Stufendiagramm-e1634130504602-730x401.png.webp>