# Zentrale und verteilte Versionskontrolle

## Lernziele

Nach diesem Abschnitt kannst du:

- zentrale und verteilte Versionskontrollsysteme unterscheiden,
- den grundlegenden Ablauf beider Modelle erklären,
- Vorteile, Grenzen und Risiken beider Ansätze bewerten,
- einordnen, warum Git als *verteiltes* Versionskontrollsystem konzipiert ist,
- verstehen, weshalb GitHub trotz seiner zentralen Rolle nicht aus Git ein zentrales System macht.

---

## Zwei grundlegende Modelle

Versionskontrollsysteme organisieren Änderungen an Dateien und machen die Entwicklungsgeschichte nachvollziehbar. Dabei haben sich zwei grundlegende Architekturmodelle etabliert:

1. **Zentrale Versionskontrolle**: Es gibt ein maßgebliches Repository auf einem Server.
2. **Verteilte Versionskontrolle**: Jede lokale Kopie enthält grundsätzlich die vollständige Repository-Historie.

Der entscheidende Unterschied betrifft nicht nur den Speicherort der Daten. Er verändert auch, **wann** Teams miteinander kommunizieren müssen, wie sie offline arbeiten können, wie sicher ihre Historie ist und welche Arbeitsabläufe sinnvoll sind.

---

## Zentrale Versionskontrolle

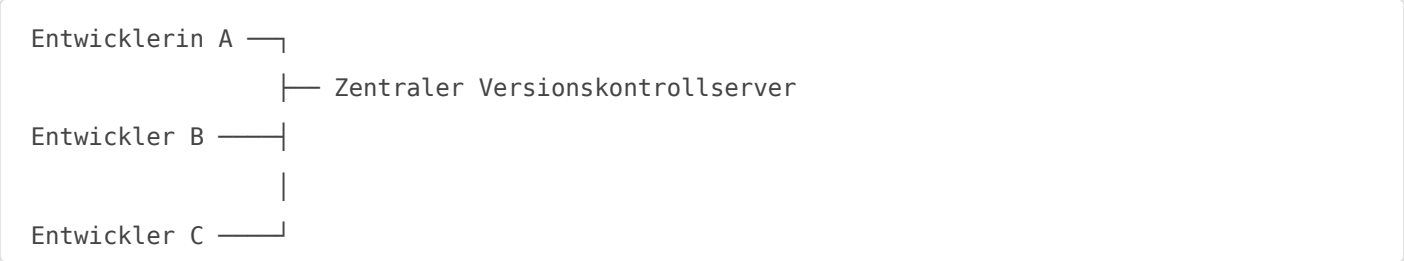
Bei einem zentralen Versionskontrollsystem liegt die vollständige Projektgeschichte auf einem zentralen Server. Entwicklerinnen und Entwickler laden Dateien daraus herunter, bearbeiten sie lokal und übertragen ihre Änderungen wieder zurück auf den Server.

Typische zentrale Systeme sind:

- Subversion, meist über den Befehl `svn`
- Perforce
- Team Foundation Version Control, kurz TFVC
- ältere Systeme wie CVS

# Grundprinzip

Die Arbeitskopie auf dem lokalen Computer enthält in der Regel die aktuelle Version der Dateien, aber nicht zwingend die vollständige Historie aller Versionen. Der Server ist die zentrale Wahrheit über den Projektzustand.




Ein typischer Ablauf sieht so aus:

1. Änderungen vom Server abrufen.
2. Dateien lokal bearbeiten.
3. Vor dem Hochladen prüfen, ob andere Personen Änderungen veröffentlicht haben.
4. Eigene Änderungen an den zentralen Server senden.
5. Der Server speichert die neue offizielle Revision.

# Typischer zentraler Workflow


In Subversion wären die zentralen Aktionen beispielsweise:

```
svn update
svn commit -m "Validierung für E-Mail-Adresse ergänzt"
```

Die exakte Syntax ist hier weniger wichtig als das Denkmodell: Ohne Verbindung zum Server sind viele zentrale Aktionen nicht möglich.

---

# Vorteile zentraler Systeme

Zentrale Versionskontrolle kann gerade in klar geregelten Umgebungen sinnvoll sein.

## Ein eindeutig zentraler Projektstand

Es gibt einen klaren Ort, an dem sich die offizielle Historie befindet. Teams müssen nicht entscheiden, welches Repository maßgeblich ist: Der zentrale Server ist es per Definition.

Das vereinfacht manche organisatorischen Prozesse:

- Berechtigungen werden zentral verwaltet.
- Backups konzentrieren sich auf einen Server.
- Regeln für Änderungen lassen sich zentral erzwingen.
- Der aktuelle Projektstand ist an einer Stelle sichtbar.

## Einfaches mentales Modell

Für Einsteigerinnen und Einsteiger wirkt das Modell häufig intuitiv:

> „Ich lade die aktuelle Version herunter, ändere etwas und speichere es wieder auf dem Server.“

Die Abläufe ähneln der Arbeit mit einer gemeinsam genutzten Dateiablage, auch wenn Versionskontrolle deutlich leistungsfähiger ist.

## Serverseitige Kontrolle

Unternehmen mit strengen Vorgaben können zentrale Serverregeln nutzen, etwa:

- verpflichtende Prüfungen vor einem Commit,

- zentrale Zugriffsrechte,
- Audit-Protokolle,
- verbindliche Dateisperren für bestimmte Dateitypen.

Besonders bei großen Binärdateien oder Dateien, die sich kaum sinnvoll zusammenführen lassen, können Sperrmechanismen nützlich sein. Beispielsweise soll nicht gleichzeitig an derselben Photoshop-Datei gearbeitet werden.

---

# Grenzen zentraler Systeme

Die Einfachheit des Modells bringt Abhängigkeiten mit sich.

## Abhängigkeit vom Server

Ist der zentrale Server nicht erreichbar, sind zentrale Funktionen eingeschränkt oder unmöglich. Das kann passieren durch:

- Netzwerkprobleme,
- Wartungsarbeiten,
- VPN-Ausfälle,
- einen Serverausfall,
- fehlende Internetverbindung auf Reisen.

Lokale Änderungen sind zwar häufig weiterhin möglich, aber das Übertragen, Abrufen und oft auch das komfortable Anzeigen der Historie hängt vom Server ab.

## Eingeschränkte Offline-Arbeit

In einem zentralen System kann ein Commit direkt eine Serveroperation sein. Ohne Netzwerk lässt sich die Änderung daher nicht als vollwertiger, gemeinsamer Versionsstand speichern.

Das führt oft dazu, dass Entwicklerinnen und Entwickler größere Änderungspakete sammeln und später gesammelt übertragen. Große Pakete sind aber schwieriger zu prüfen, zu testen und bei Problemen zurückzunehmen.

## Zentraler Ausfallpunkt

Der Server ist ein sogenannter *Single Point of Failure*: Fällt er aus und existiert kein funktionierendes Backup, ist die Projektgeschichte gefährdet.

Ein zentraler Server kann selbstverständlich professionell gesichert werden. Dennoch bleibt seine Verfügbarkeit für den täglichen Arbeitsablauf entscheidend.

# Branches können teuer sein

Bei einigen älteren zentralen Systemen sind Branches technisch oder organisatorisch schwergewichtiger. Teams vermeiden sie dann häufig und arbeiten lange auf gemeinsamen Entwicklungszweigen.

Das erhöht die Wahrscheinlichkeit, dass parallele Änderungen miteinander kollidieren.

> **Merksatz:** In zentralen Systemen ist Zusammenarbeit oft stärker an den gemeinsamen Server und dessen Verfügbarkeit gebunden.

---

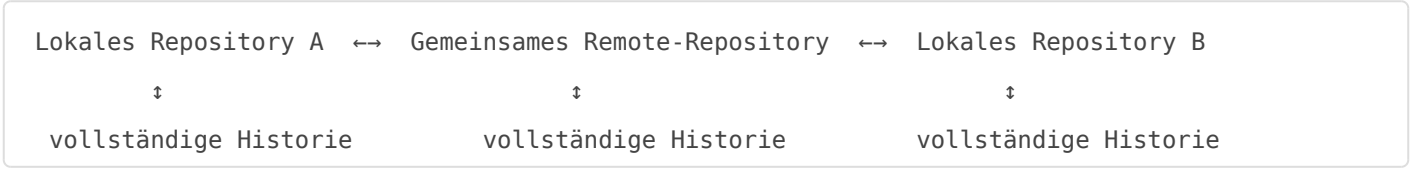
# Verteilte Versionskontrolle

Ein verteiltes Versionskontrollsystem speichert die vollständige Projektgeschichte nicht nur auf einem Server, sondern auch lokal bei den Beteiligten.

Git ist das bekannteste verteilte Versionskontrollsystem. Weitere Beispiele sind Mercurial und Fossil.

Wenn du ein Git-Repository klonst, erhältst du nicht bloß die aktuell sichtbaren Dateien. Du erhältst normalerweise auch:

- die Commit-Historie,
- Branches und Tags,
- die Git-Objekte,
- lokale Referenzen,
- die Informationen, die Git für Vergleiche, Branches und viele Wiederherstellungen benötigt.




Das Wort *verteilt* bedeutet nicht, dass es keinen zentralen Server geben darf. Es bedeutet vielmehr:

>

> Jedes vollständige Git-Repository ist grundsätzlich eine eigenständige Kopie der Projektgeschichte.

---

# Das lokale Repository als vollwertige Kopie

Ein Git-Repository besteht aus mehr als dem Projektordner mit den sichtbaren Dateien. Im versteckten Verzeichnis `.git` verwaltet Git unter anderem:

- Commits,
- Branch-Referenzen,
- Tags,
- die Staging Area,
- Konfiguration,
- lokale Bewegungsprotokolle, die Reflogs.

Beim Klonen eines Repositorys entsteht daher eine lokale, funktionsfähige Datenbasis.

```
Projektordner
├── src
├── tests
├── composer.json
└── .git
    ├── objects
    ├── refs
    ├── HEAD
    ├── index
    └── config
```

Die Details dieser Dateien und Verzeichnisse werden in späteren Kapiteln vertieft. Für den Moment genügt dieses Verständnis:

- Die sichtbaren Projektdateien bilden dein **Arbeitsverzeichnis**.
- Das `.git`-Verzeichnis enthält die **lokale Versionsdatenbank**.

---

# Lokale Commits ohne Netzwerk

Ein entscheidender Vorteil von Git ist, dass ein Commit lokal erstellt wird. Dafür ist weder GitHub noch ein Unternehmensserver erforderlich.

```
git add src/EmailValidator.php
git commit -m "Validierung für E-Mail-Adresse ergänzen"
```

Dieser Commit wird zunächst nur im lokalen Repository gespeichert.

Erst mit einem Push überträgst du ihn in ein anderes Repository, beispielsweise zu GitHub:

```
git push origin main
```

Die Begriffe beschreiben unterschiedliche Vorgänge:

| Aktion | Bedeutung |
| --- | --- |
| `git commit` | Speichert einen neuen Versionsstand **lokal** |
| `git push` | Überträgt lokale Commits in ein entferntes Repository |
| `git fetch` | Lädt Informationen und Commits aus einem entfernten Repository herunter |
| `git pull` | Ruft Änderungen ab und integriert sie in den aktuellen lokalen Branch |

Diese Trennung ist ein Kernmerkmal verteilter Versionskontrolle.

---

# Typischer Git-Workflow

Ein einfacher Git-Workflow kann so aussehen:

```
Dateien lokal ändern
        ↓
Änderungen auswählen und vormerken
        ↓
Lokalen Commit erstellen
        ↓
Tests lokal ausführen
        ↓
```

```
Commit zu GitHub oder einem anderen Remote pushen
```

Als Befehlsfolge:

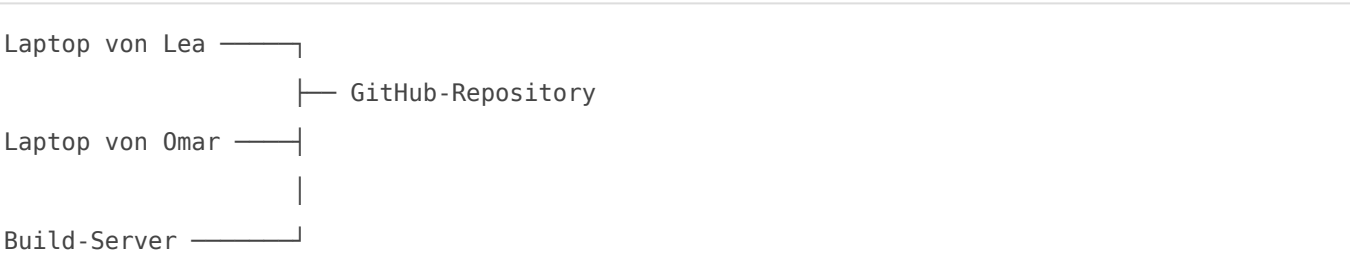
```
git status
git add src/EmailValidator.php
git commit -m "Validierung für E-Mail-Adresse ergänzen"
git push
```

Zwischen `git commit` und `git push` können Sekunden, Stunden oder auch Tage liegen. Ob das sinnvoll ist, hängt vom Teamworkflow ab. Technisch ist es möglich, weil der Commit bereits lokal existiert.

---

# Remote-Repositorys sind nicht „der Git-Server“

In der Praxis verwenden Teams oft ein gemeinsames Remote-Repository als Koordinationspunkt. Das kann bei GitHub, GitLab, Bitbucket oder auf einem eigenen Server liegen.




Dieses GitHub-Repository ist für das Team häufig die **gemeinsame Referenz**. Es ist aber technisch nicht das einzige Repository, das die Historie enthält.

Auch Leas und Omars lokale Repositories enthalten eine vollständige Historie. GitHub ergänzt Git um Funktionen für Zusammenarbeit und Plattformbetrieb, zum Beispiel:

- Pull Requests,
- Code Reviews,
- Issues,
- Rechteverwaltung,
- GitHub Actions,
- Releases,
- Sicherheitsprüfungen.

GitHub ist damit für viele Teams organisatorisch zentral, Git selbst bleibt jedoch ein verteiltes System.

> **Wichtig:** „Wir arbeiten mit GitHub“ bedeutet nicht, dass Git zentralisiert wäre. GitHub ist ein Remote und eine Kollaborationsplattform für Git-Repositories.

---

# Mehrere Remotes verwenden

Ein Git-Repository kann mit mehreren entfernten Repositories verbunden sein. Das ist ein deutlicher Ausdruck der verteilten Architektur.

Ein Open-Source-Beitrag kann beispielsweise so organisiert sein:

```
Lokales Repository
       │
       ├── origin   → eigener Fork auf GitHub
       │
       └── upstream → ursprüngliches Projekt auf GitHub
```

Die zugehörigen Remote-Namen sind frei wählbar. `origin` ist nur eine Konvention, die Git beim Klonen automatisch verwendet.

```
git remote -v
```

Eine mögliche Ausgabe:

```
origin    git@github.com:beispielkonto/mein-fork.git (fetch)
origin    git@github.com:beispielkonto/mein-fork.git (push)
upstream  git@github.com:organisation/originalprojekt.git (fetch)
upstream  git@github.com:organisation/originalprojekt.git (push)
```

Du kannst also Änderungen aus einem Repository abrufen und in ein anderes veröffentlichen. Git schreibt keine einzige zentrale Instanz vor.

---

# Vergleich der Modelle

| Aspekt | Zentrale Versionskontrolle | Verteilte Versionskontrolle mit Git |
|---|---|---|
| Maßgebliche Historie | Liegt primär auf einem zentralen Server | Liegt in vollständigen lokalen Repository-Kopien und optionalen Remotes |
| Commit | Häufig direkt auf dem Server | Zunächst lokal |
| Offline-Arbeit | Oft eingeschränkt | Für Commits, Branches, Logs und Diffs weitgehend möglich |
| Serverausfall | Kann den gesamten Workflow blockieren | Lokale Arbeit und Historienanalyse bleiben möglich |
| Branches | Je nach System eher aufwendig | Sehr leichtgewichtig und schnell |
| Zusammenarbeit | Stark auf zentralen Server ausgerichtet | Über Push, Fetch und frei wählbare Remotes |
| Datensicherung | Zentraler Server ist besonders kritisch | Mehrere vollständige Kopien können zusätzliche Redundanz schaffen |
| Zugriffssteuerung | Typischerweise zentral | Meist über Remotes, Plattformen und Serverregeln organisiert |

Die Tabelle beschreibt Tendenzen. Moderne zentrale Systeme können Offline-Funktionen anbieten, und ein Git-Team kann seinen Workflow stark um ein einziges GitHub-Repository herum organisieren. Die grundlegende Architektur bleibt dennoch verschieden.

---

# Verteilung bedeutet nicht automatisch Sicherheit

Es wäre falsch zu behaupten, dass Git automatisch ein Backup-Konzept ersetzt. Mehrere lokale Kopien können zwar helfen, doch sie sind nicht automatisch zuverlässig gesichert.

Beispiele für Risiken:

- Niemand außer einer Person hat die neuesten lokalen Commits.
- Ein Laptop wird beschädigt oder geht verloren.
- Ein lokaler Branch wird gelöscht.
- Ein Commit wird nie zu einem Remote gepusht.
- Ein zentraler GitHub-Account wird falsch konfiguriert oder kompromittiert.

Ein professionelles Team sorgt deshalb dafür, dass wichtige Arbeit regelmäßig auf ein geeignetes Remote übertragen wird und dass dieses Remote zusätzlich abgesichert ist.

> **Praxisregel:** Ein lokaler Commit schützt vor vielen Fehlern im Arbeitsverzeichnis. Ein gepushter Commit schützt zusätzlich vor dem Verlust eines einzelnen Computers. Ein getestetes Backup schützt vor weitergehenden Ausfällen.

---

# Warum Git trotzdem oft „zentral“ wirkt

Ein GitHub-Workflow fühlt sich für viele Teams zentral an:

1. Ein Repository auf GitHub wird erstellt.
2. Alle Teammitglieder klonen dieses Repository.
3. Feature-Branches werden dorthin gepusht.
4. Pull Requests werden dort geprüft.
5. Der Hauptbranch wird dort geschützt.
6. CI-Prüfungen laufen dort.

Das ist keine Schwäche oder ein Widerspruch. Git erlaubt eine zentrale *Zusammenarbeitsstruktur*, ohne seine verteilte Natur aufzugeben.

Man kann es so unterscheiden:

| Ebene | Zentral oder verteilt? |
|---|---|
| Git-Datenmodell | Verteilt |
| Lokale Git-Repositories | Eigenständig und vollständig |
| GitHub als Teamplattform | Häufig zentraler Koordinationspunkt |
| Teamregeln und Branch-Schutz | Zentral organisiert |
| Commits, Branches und Diffs auf dem Laptop | Lokal verfügbar |

Diese Kombination ist einer der Gründe für Gits Erfolg: Teams erhalten lokale Geschwindigkeit und Flexibilität, können ihre Zusammenarbeit aber trotzdem klar über GitHub organisieren.

---

# Ein konkretes Alltagsszenario

Stell dir vor, du arbeitest im Zug ohne Internetverbindung an einem PHP-Projekt.

Mit Git kannst du weiterhin:

```
git status
git diff
git switch feature/email-validation
git add src/EmailValidator.php
git commit -m "Fehlermeldung für ungültige Domain verbessern"
git log --oneline
```

Du kannst also Änderungen prüfen, Branches wechseln, Commits erstellen und die Historie untersuchen.

Sobald wieder eine Verbindung besteht, veröffentlichst du die bereits vorhandenen Commits:

```
git push origin feature/email-validation
```

Danach kann GitHub einen Pull Request anbieten oder du erstellst ihn selbst über die Website beziehungsweise PhpStorm.

In einem strikt zentralen System wäre der lokale Arbeitsschritt vergleichbar möglich, aber die vollständige Versionsverwaltung und insbesondere der Commit könnten stärker von der Erreichbarkeit des Servers abhängen.

---

# Auswirkungen auf Branching und Experimente

Weil lokale Branches in Git sehr günstig erzeugt werden können, eignet sich Git gut für kurze, isolierte Experimente.

```
git switch -c experiment/neue-validierungsregel
```

Du kannst dort eine Idee umsetzen, testen und später entscheiden:

- Die Änderungen sind sinnvoll: Branch weiterentwickeln oder zusammenführen.
- Die Änderungen sind nicht sinnvoll: Branch löschen.
- Die Idee ist noch offen: Branch lokal behalten.

```
git switch main
git branch -D experiment/neue-validierungsregel
```

Solange du keinen Push ausführst, bleibt dieser Branch vollständig lokal. Er beeinflusst weder GitHub noch andere Teammitglieder.

Dieses Verhalten fördert kleine, risikoarme Experimente und klar abgegrenzte Änderungen.

---

# Häufige Missverständnisse

## „GitHub speichert meine Commits automatisch“

Nein. Ein lokaler Commit wird nicht automatisch zu GitHub übertragen.

Nach diesem Befehl:

```
git commit -m "Formularvalidierung verbessern"
```

existiert der Commit zunächst nur lokal. Erst ein Push veröffentlicht ihn im Remote:

```
git push
```

## „Ein Git-Repository braucht immer GitHub“

Nein. Git funktioniert vollständig ohne GitHub.

Du kannst ein lokales Repository anlegen und verwenden:

```
git init
```

GitHub wird erst relevant, wenn du ein Remote für Zusammenarbeit, Sicherung, Veröffentlichung oder Automatisierung verwenden möchtest.

## „Verteilt heißt, dass jede Person alles veröffentlichen darf“

Nein. Die technische Architektur und die Berechtigungen sind verschiedene Dinge.

Git erlaubt lokale Commits und lokale Branches ohne zentrale Freigabe. Ein GitHub-Repository kann aber genau festlegen:

- Wer pushen darf,
- welche Branches geschützt sind,
- ob Pull-Request-Reviews nötig sind,
- welche Prüfungen erfolgreich sein müssen,
- wer Releases erstellen darf.

## „Lokale Commits sind unwichtig, weil nur GitHub zählt“

Lokale Commits sind sehr wichtig. Sie strukturieren deine Arbeit, ermöglichen sichere Experimente und bilden die Grundlage für jeden späteren Push. GitHub erhält keine magischen Änderungen, sondern die Commits, die du lokal erstellt hast.

---

# Entscheidungshilfe: Welches Modell passt wann?

In der Praxis entscheidet man selten ausschließlich aufgrund der Architektur. Anforderungen an Teams, Infrastruktur, Sicherheit und bestehende Werkzeuge spielen ebenfalls eine Rolle.

**Zentrale Versionskontrolle** kann passend sein, wenn:

- ein bestehendes Unternehmen stark auf ein zentrales System ausgerichtet ist,
- sehr spezielle Sperr- und Dateiverwaltungsprozesse benötigt werden,
- große Binärdateien und exklusive Bearbeitung im Vordergrund stehen,
- ein vorhandenes System nicht kurzfristig migriert werden kann.

**Verteilte Versionskontrolle mit Git** ist besonders geeignet, wenn:

- Teams häufig mit Branches arbeiten,
- Entwicklerinnen und Entwickler auch offline produktiv sein sollen,
- kleine, lokale Commits erwünscht sind,
- Pull Requests und automatisierte Prüfungen etabliert werden sollen,
- Open-Source-Zusammenarbeit oder Forks relevant sind,
- moderne Plattformen wie GitHub, GitLab oder Bitbucket genutzt werden.

Für die meisten Softwareprojekte ist Git heute der verbreitete Standard. Das liegt nicht nur an GitHub, sondern vor allem an Gits leistungsfähigem lokalen Datenmodell, seinen günstigen Branches und seiner flexiblen Zusammenarbeit über Remotes.

---

# Merkpunkte

- Ein **zentrales** Versionskontrollsystem speichert die maßgebliche Historie primär auf einem Server.
- Ein **verteiltes** System wie Git gibt jeder vollständigen lokalen Repository-Kopie die Projektgeschichte mit.
- In Git ist ein Commit zunächst **lokal**; `git push` veröffentlicht ihn in einem Remote.
- GitHub ist ein wichtiger zentraler Koordinationspunkt, macht Git aber nicht zu einem zentralen Versionskontrollsystem.
- Verteilte Repositories bieten Redundanz, ersetzen aber kein bewusstes Backup- und Push-Konzept.
- Die Fähigkeit, lokal zu committen, zu vergleichen, zu verzweigen und Historie zu lesen, ist ein zentraler Vorteil von Git.

---

# Selbsttest

1. Worin besteht der wichtigste architektonische Unterschied zwischen zentraler und verteilter Versionskontrolle?
2. Warum kannst du mit Git ohne Internet einen Commit erstellen?
3. Was ist der Unterschied zwischen `git commit` und `git push`?
4. Weshalb bleibt Git ein verteiltes System, obwohl ein Team GitHub als gemeinsamen Mittelpunkt verwendet?
5. Warum ist ein lokaler Commit allein noch keine ausreichende Datensicherung?
6. Nenne eine Situation, in der lokale Branches besonders hilfreich sind.

---

Revision #1
Created 2026-07-25 11:13:39 UTC by art10m
Updated 2026-07-25 11:13:39 UTC by art10m