



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kapitel 02

FEATURE-ENTWICKLUNG

Das Problem: direkt auf main pushen

```
$ git add .  
$ git commit -m "neues Feature hinzugefügt"  
$ git push origin main  
# Hoffen, dass nichts bricht 🙌
```

Probleme mit diesem Workflow:

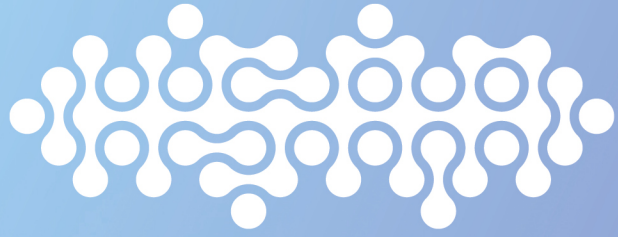
1. **Kein Review** — Code geht direkt in die Produktion
2. **Kein Sicherheitsnetz** — Einmal gepusht, ist live
3. **Bricht das Team** — Bugs blockieren alle
4. **Keine Kollaboration** — Kein Feedback möglich
5. **Schwer nachvollziehbar** — Alle Commits landen auf `main`

Reale Folge: Sie pushen um 23 Uhr fehlerhaften Code. Am nächsten Morgen ist die Umgebung Ihres Teamkollegen kaputt. Er verliert 2 Stunden beim Debuggen **Ihres** Codes.

Lernziele

Am Ende dieser Vorlesung können Sie:

1. **Git-Grundlagen verstehen** — Arbeitsverzeichnis, Staging, Commits, Branches
2. **Feature-Branch-Workflows beherrschen** — Erstellen, Arbeiten, Mergen
3. **Pull Requests nutzen** — Review vor dem Merge einfordern
4. **Branch-Schutz anwenden** — Direkte Pushes auf `main` verhindern
5. **CI/CD vorbereiten** — Grundlage für automatische Qualitätsprüfungen
6. **Professionell kollaborieren** — So arbeiten echte Software-Teams



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Git-Grundlagen

1

Die drei Zustände von Git

1. Arbeitsverzeichnis (Modified)

- Dateien, die Sie aktiv bearbeiten
- Änderungen noch nicht in Git gespeichert

2. Staging-Bereich (Staged)

- Änderungen, die für den nächsten Commit vorgemerkt sind
- Eine Vorschau dieses Commits

3. Repository (Committed)

- Permanenter Snapshot in Gits Historie

```
Arbeitsverzeichnis  
  ↓ git add  
Staging-Bereich  
  ↓ git commit  
Repository
```

Jede Änderung fließt **durch** den Staging-Bereich, bevor sie ein Commit wird.

Die drei Zustände visualisieren

Szenario: Einen Import in `main.py` reparieren.

```
# 1. ARBEITSVERZEICHNIS – Datei bearbeiten
$ vim main.py                                # import sys,os → import sys\nimport os
$ git status
Changes not staged for commit:
  modified:   main.py

# 2. STAGING-BEREICH – für Commit vormerken
$ git add main.py
$ git status
Changes to be committed:
  modified:   main.py

# 3. REPOSITORY – Snapshot speichern
$ git commit -m "Fix PEP8: imports auf verschiedene Zeilen aufteilen"
$ git status
nothing to commit, working tree clean
```

Warum der Staging-Bereich existiert

Er gibt Ihnen Kontrolle darüber, was in jeden Commit kommt.

Statt eines riesigen Commits bauen Sie aus einem unaufgeräumten Arbeitsverzeichnis logische Commits:

```
$ git add -p main.py          # Nur Import-Änderungen vormerken
$ git commit -m "Fix PEP8: imports trennen, unused entfernen"

$ git add -p main.py          # Nur Spacing-Änderungen vormerken
$ git commit -m "Fix PEP8: Spacing um Operatoren"

$ git add main.py             # Rest (Namensänderung) vormerken
$ git commit -m "Fix PEP8: HelperFunction → helper_function"
```

Jeder Commit hat genau einen klaren Zweck — leichter zu reviewen, leichter zurückzunehmen.

`git status` — Ihr bester Freund

```
$ git status
On branch feature/fix-code-quality
Changes to be committed:
  modified:   main.py          ← STAGED (Import-Fixes)

Changes not staged for commit:
  modified:   main.py          ← MODIFIED (Spacing-Fixes)

Untracked files:
  .ruff_cache/                ← UNTRACKED (in .gitignore aufnehmen)
```

- **Staged** → kommt in den nächsten Commit
- **Modified** → geändert, aber *nicht* im nächsten Commit
- **Untracked** → Git hat die Datei noch nie gesehen

Tipp: Sie sollten `git status` ständig ausführen. Es ist Ihre Landkarte.

Anatomie eines Commits

Ein Commit ist ein **Snapshot** Ihres Projekts zu einem bestimmten Zeitpunkt.

```
Commit: a1b2c3d4
Author: Student Name <student@example.com>
Date:   2025-10-16 10:30:00 +0200

    Fix PEP8 violations: imports and spacing

Changed files:
main.py | 8 ++++----
```

1. **Hash** (a1b2c3d4) — eindeutiger Identifier
2. **Author / Date** — wer und wann
3. **Message** — was sich geändert hat und *warum*
4. **Diff** — die tatsächliche Code-Änderung

Einmal erstellt ist ein Commit **permanent** — dieser Snapshot lebt für immer weiter.

Git-Historie anschauen

```
$ git log --oneline --graph --all
```

```
* d4e5f6g (HEAD -> feature/fix-code-quality) Fix PEP8: Rename helper_function
* a1b2c3d Fix PEP8: Add proper spacing around operators
* 7h8i9j0 Fix PEP8: Separate imports and remove unused ones
* k1l2m3n (origin/main, main) Add Road Profile Viewer application
* f9e8d7c Initial commit
```

- * — ein Commit
- HEAD -> — wo Sie sich gerade befinden
- (origin/main, main) — Branch-Zeiger
- d4e5f6g — Kurzform des Commit-Hashes
- Die Nachricht rechts erzählt die Geschichte

Häufige `git status` Szenarien

Sauberes Arbeitsverzeichnis

```
nothing to commit, working tree clean
```

✓ Branch-Wechsel ist sicher.

Uncommittete Änderungen

```
Changes not staged for commit:  
  modified:   main.py
```

! Vor dem Wechsel committen oder stashen.

Staged, aber nicht committet

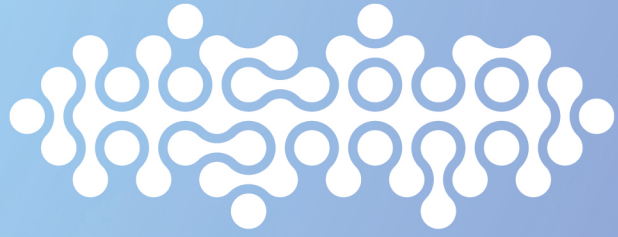
```
Changes to be committed:  
  modified:   main.py
```

! `git commit` ausführen.

Staged UND unstaged

```
Changes to be committed:  
  modified:   main.py  
Changes not staged for commit:  
  modified:   main.py
```

! Nach dem Staging bearbeitet — bei Bedarf erneut `add`.



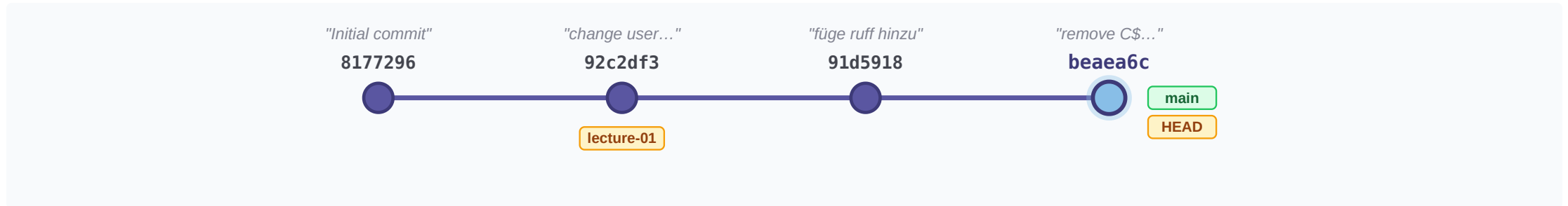
Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Branches

2

Beispiel 1: Vor dem Erstellen des Branches

Zustand am 21.10.2025 um 16:26:11 (auf main)



- `beaea6c` ist der tatsächliche Commit (Snapshot: "remove C\$ from ruff settings")
- `main` ist ein Branch-Zeiger, der auf Commit `beaea6c` zeigt
- `HEAD` zeigt auf `main` (Sie sind gerade auf dem main-Branch)
- `lecture-01` Tag zeigt auf Commit `92c2df3`

Beispiel 1: Erstellen des Branches

Befehl:

```
$ git checkout -b feauture/fix_code_quality  
# (Hinweis: hatte ursprünglich den Tippfehler!)
```

Nach dem Erstellen des Branches:



Beispiel 1: Was ist gerade passiert?

1. Git hat einen **NEUEN Zeiger** namens `feature/fix_code_quality` erstellt
2. Dieser Zeiger zeigt auf den **GLEICHEN Commit** wie `main` (`beaea6c`)
3. Git hat **HEAD** verschoben, um auf den neuen Branch zu zeigen
4. **Keine Commits wurden erstellt oder kopiert** - nur ein neuer Zeiger!

Wichtige Erkenntnis: Das Erstellen eines Branches ist günstig - es ist nur das Erstellen eines Zeigers (ein paar Bytes)!

Beispiel 2: Erster Commit auf dem Branch

Den ersten Commit erstellen:

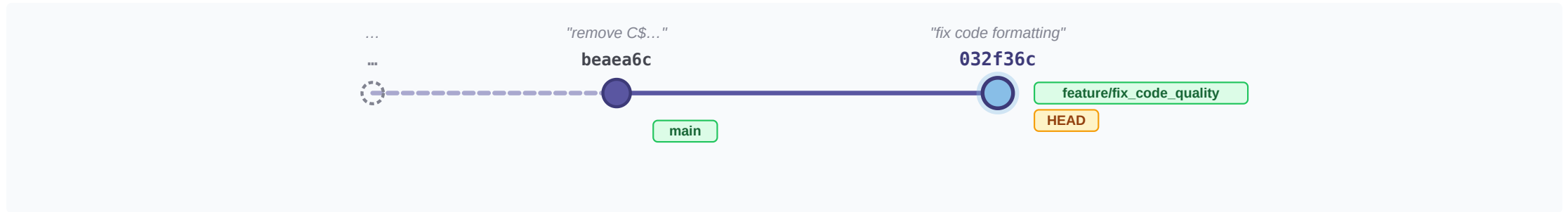
```
$ vim main.py # Code-Formatierung reparieren  
$ git add main.py  
$ git commit -m "fix code formatting using ruff format"
```

Vor dem Commit:



Beispiel 2: Nach dem ersten Commit

Nach dem Commit (032f36c):



Was ist passiert:

1. Git hat einen **NEUEN Commit** `032f36c` erstellt
2. Git hat den `feature/fix_code_quality` -Zeiger vorwärts zu `032f36c` bewegt
3. Der `main` -Zeiger **blieb bei** `beaea6c` (unverändert!)
4. HEAD zeigt immer noch auf `feature/fix_code_quality`

Beispiel 2: Wichtige Erkenntnis

Nur der aktuelle Branch-Zeiger bewegt sich, wenn Sie commiten.

Andere Branches bleiben, wo sie sind!

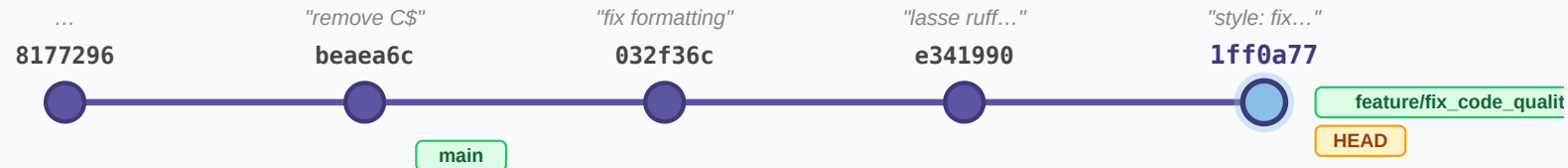
So bieten Feature-Branches **Isolation** - Ihre Arbeit auf dem Feature-Branch beeinflusst `main` nicht.

Beispiel 3: Nach mehreren Commits

Weitere Commits erstellen:

```
$ git commit -m "lasse ruff check laufen..." # e341990  
$ git commit -m "style: fix code quality..." # 1ff0a77
```

Finaler Zustand nach 3 Commits auf dem Branch:



Beispiel 3: Die Divergenz visualisieren



Was Sie sehen:

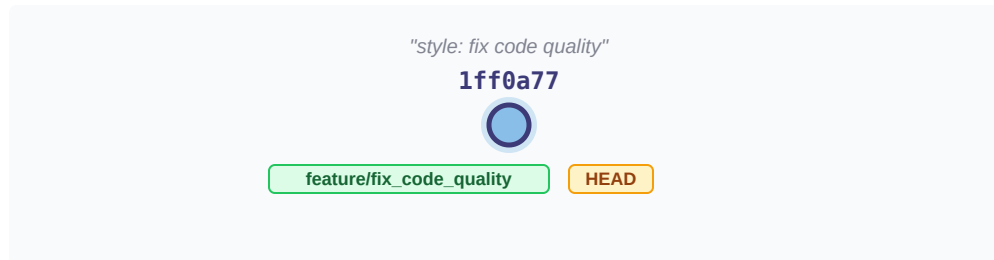
- `main` ist immer noch bei `beaea6c` (wo Sie abgezweigt sind)
- `feature/fix_code_quality` ist durch 3 Commits vorwärts gegangen
- Die Arbeit am Feature-Branch ist **isoliert** von main
- HEAD zeigt auf den Feature-Branch (Sie arbeiten daran)

Beispiel 4: Zwischen Branches wechseln

Zurück zu main wechseln:

```
$ git checkout main
```

Vor dem Checkout:



Nach dem Checkout:



Beispiel 4: Was hat sich geändert?

Nur HEAD hat sich bewegt!

- HEAD ist vom Feature-Branch-Zeiger zum Main-Zeiger gewechselt
- Ihr Arbeitsverzeichnis hat sich geändert, um Commit `beaea6c` zu zeigen
- Die Branch-Zeiger selbst **haben sich nicht bewegt**
- `feature/fix_code_quality` zeigt immer noch auf `1ff0a77`

Das ist alles, was `git checkout` **macht** - es bewegt den HEAD-Zeiger!

Beispiel 5: Feature-Branch in Main mergen

Wenn Sie den Feature-Branch mergen:

```
$ git checkout main  
$ git merge feauture/fix_code_quality  
# (Oder merge über GitHub PR)
```

Ergebnis: Fast-Forward-Merge



Beispiel 5: Nach dem Merge

Nach dem Merge (Fast-Forward):

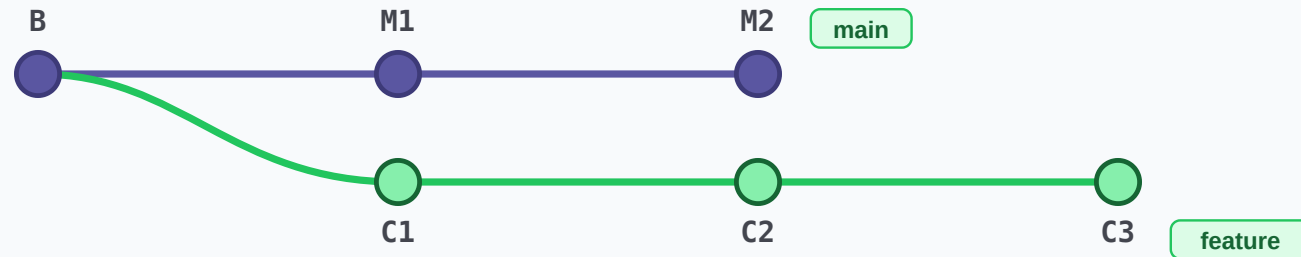


Was ist passiert:

- Git hat den `main`-Zeiger vorwärts zu `1ff0a77` bewegt
- Das ist ein "Fast-Forward"-Merge (main holt nur auf)
- Sowohl `main` als auch `feature/fix_code_quality` zeigen jetzt auf denselben Commit
- Kein Merge-Commit war nötig (lineare Historie)

Aber was, wenn `main` schon weiter ist?

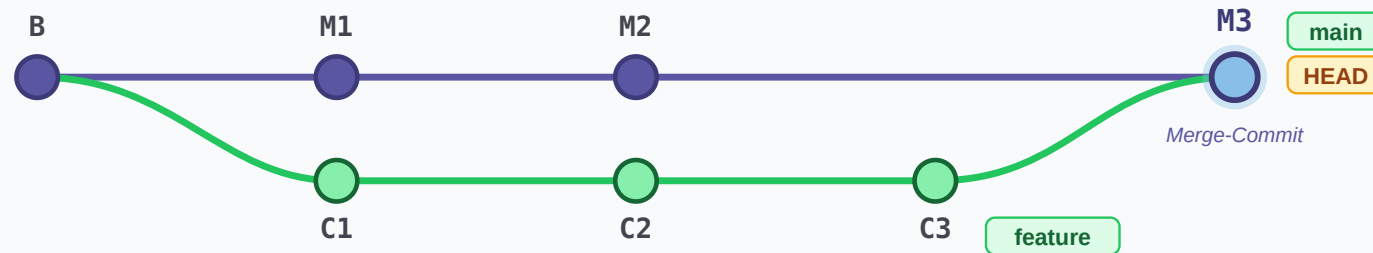
Im Beispiel oben war es einfach — `main` hatte sich nicht bewegt. In echten Teams arbeiten mehrere Entwickler parallel: während Sie am Feature waren, hat jemand anderes seinen PR gemergt.



`main` und Ihr Feature sind **divergiert** — ein Fast-Forward ist nicht mehr möglich. **Frage an Sie:** Wie bringen Sie die drei Commits jetzt sauber zurück in `main` ?

Strategie 1: Merge-Commit (`--no-ff`)

```
$ git checkout main  
$ git merge --no-ff feature/fix_code_quality  
# Erstellt einen Merge-Commit M3 mit zwei Eltern: M2 und C3
```



M3 ist ein **neuer Commit** mit zwei Eltern: M2 (von main) und C3 (vom feature-Branch). Die Feature-Commits behalten ihre **Original-Hashes** — sie werden nicht kopiert, sondern sind über M3 jetzt auch von main aus erreichbar.

Strategie 1: Merge-Commit — Vor- & Nachteile

✓ Vorteile

- Vollständige Historie bleibt erhalten
- Sichtbar, wann ein Feature integriert wurde
- Originale Commit-Hashes unverändert
- Branch-Lineage in `git log --graph` lesbar

⚠ Nachteile

- "Diamantförmige" Historie kann unübersichtlich werden
- Viele "Merge branch X into main"-Commits
- Bei vielen kleinen PRs entsteht Lärm

Typischer Einsatz: Teams, die Feature-Lineage bewusst sichtbar halten wollen — jeder Merge ist ein erkennbarer Integrationspunkt im `git log --graph`.

Nach dem Merge: Cleanup & Commit-Message

Was steht im Merge-Commit?

Git generiert eine Default-Message und öffnet Ihren Editor:

```
Merge branch 'feature/fix_code_quality' into main
```

Direkt setzen oder anpassen:

```
$ git merge --no-ff feature/... \  
-m "Merge: code quality (PR #42)"
```

Auf GitHub via PR — automatisch:

```
Merge pull request #42 from user/feature/fix_code_quality  
  
<PR-Beschreibung als Body>
```

Was passiert mit dem Feature-Branch?

In der Praxis nach dem Merge **löschen**:

```
$ git branch -d feature/fix_code_quality  
$ git push origin --delete feature/fix_code_quality
```

Sind Ihre Commits damit weg? Nein.

C1 , C2 , C3 bleiben über M3

erreichbar:

- `git log` zeigt sie weiterhin
- `git blame` findet ihre Autoren
- Hashes funktionieren in Issues, PRs, Slack
- `git checkout C2` springt direkt dorthin

Strategie 2: Squash & Merge

```
$ git checkout main  
$ git merge --squash feature/fix_code_quality  
$ git commit -m "Fix code quality (PR #42)"  
# Auf GitHub: der "Squash and merge"-Button
```



S1 ist ein **neuer Commit**, der alle Änderungen aus **C1**, **C2**, **C3** als einen Diff zusammenfasst. **C1**, **C2**, **C3** selbst sind nicht in der Historie von **main** — der feature-Branch wird typischerweise gelöscht.

Strategie 2: Squash & Merge — Vor- & Nachteile

✓ Vorteile

- Lineare, saubere Historie auf `main`
- Ein Commit pro Feature/PR
- Einfaches Revert eines ganzen Features
- Weniger Lärm bei vielen "WIP"-Commits

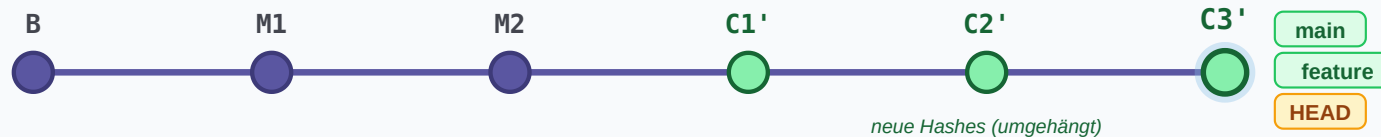
⚠ Nachteile

- Einzelne Commits gehen auf `main` verloren
- `git bisect` weniger granular
- Co-Authoring schwerer sichtbar
- Refactoring + Bugfix landen oft im selben Commit

Typischer Einsatz: GitHub-Repos mit "Squash and merge"-Default — Feature-Autor macht beliebig viele kleine Commits, die beim Merge zu einem sauberen `main`-Commit verschmelzen.

Strategie 3: Rebase + Merge

```
$ git checkout feature/fix_code_quality  
$ git rebase main          # C1, C2, C3 werden auf M2 "umgehängt"  
$ git push --force-with-lease  
$ git checkout main  
$ git merge feature/fix_code_quality  # Jetzt Fast-Forward möglich!
```



`C1'`, `C2'`, `C3'` haben **neue Hashes** — sie sind umgehängte Kopien der ursprünglichen `C1`, `C2`, `C3`. Die Originale existieren nur noch im Reflog.

Strategie 3: Rebase + Merge — Vor- & Nachteile

✓ Vorteile

- Lineare Historie ohne Merge-Commits
- Alle einzelnen Commits bleiben sichtbar
- `git log` liest sich wie eine Geschichte

✗ Nachteile

- **Schreibt Historie um** — neue Hashes
- **Niemals auf geteilten Branches** rebasen
- Konflikte oft pro Commit lösen
- Force-Push nötig

Goldene Regel: Nur Branches rebasen, an denen sonst niemand arbeitet — sonst zerstören Sie die Historie der Kolleg:innen.

Welche Strategie wann?

Strategie	Historie auf <code>main</code>	Wann sinnvoll
Fast-Forward	Linear, keine Merge-Spuren	Solo-Projekte, <code>main</code> hat sich nicht bewegt
Merge-Commit (<code>--no-ff</code>)	Verzweigt, vollständig	Teams, die Feature-Lineage sehen wollen
Squash & Merge	Linear, ein Commit pro PR	Viele kleine "WIP"-Commits; saubere <code>main</code>
Rebase + FF	Linear, alle Commits	Feature-Branch lokal aktualisieren vor PR

In der Praxis: Teams entscheiden sich pro Repo, nicht pro PR. GitHub-Repos werden oft auf **Squash-only** oder **Merge-Commit-only** konfiguriert. Rebase bleibt meist ein lokales Werkzeug, um den eigenen Feature-Branch sauber zu halten.

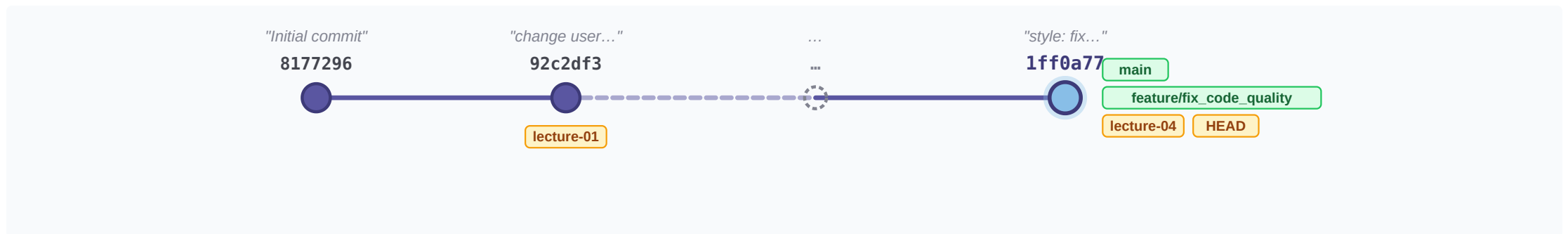
Wichtig: Nicht die "perfekte" Strategie wählen — sondern eine, die zum Team passt und konsequent durchhalten.

Beispiel 6: Einen Tag hinzufügen

Den lecture-04 Tag erstellen:

```
$ git tag -a lecture-04 -m "End of Chapter 02 (Refactoring): Code quality fixes"
```

Aktueller Zustand mit Tag:



Alle vier Zeiger zeigen auf den GLEICHEN Commit `1ff0a77` !

Beispiel 6: Tags sind auch Zeiger

Tags sind genau wie Branches - sie sind Zeiger auf Commits!

Der Unterschied:

- **Branch-Zeiger** bewegen sich vorwärts, wenn Sie commiten
- **Tag-Zeiger** bewegen sich nie (permanente Markierungen)

Beide sind nur Referenzen auf Commits - keine Kopien von Commits.

Beispiel 7: Den Branch umbenennen

Den Tippfehler korrigieren:

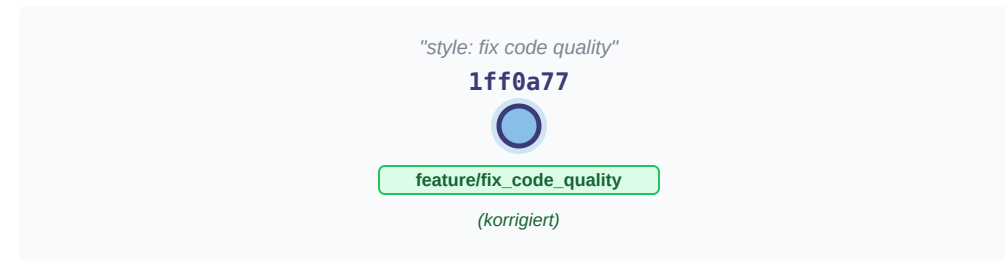
```
$ git branch -m feauture/fix_code_quality feature/fix_code_quality
```

Was ist passiert:

Vor dem Umbenennen:



Nach dem Umbenennen:



Beispiel 7: Der Commit hat sich nie geändert

Der Commit `1ff0a77` hat sich nie geändert!

Sie haben nur:

1. Einen neuen Zeiger mit der korrekten Schreibweise erstellt
2. Den alten Zeiger mit dem Tippfehler gelöscht

Deshalb:

- Der Tag `lecture-04` funktioniert immer noch (zeigt auf Commit, nicht auf Branch-Namen)
- Jeder, der den Tag ausgecheckt hat, hat den Tippfehler nie gesehen
- Branches sind nur Labels, nicht der eigentliche Code

Warum Branches verwenden?

Ohne Branches (direkt auf main):

main: A → B → C → D(kaputt) → E(fix) → F(fix) → G(ok)

↑

Alle sehen kaputten Code
Andere Devs blockiert
Rückgängig machen schwer

Mit Feature-Branches:

main: A → B → C —————→ M (merge)



feature:

D → E → F → G (getestet)



↑

Nur Sie sehen das
Sicher zu experimentieren
Einfach zu löschen

5 Vorteile: Isolation · Sicherheit · Experimentieren · Review · leichtes Rollback

Branch-Operationen — Erstellen & Wechseln

Branches anzeigen:

```
$ git branch
main
* feature/fix-code-quality    ← Sie sind hier (*)
feature/improve-road-curve
```

Branch erstellen:

```
$ git branch feature/new-feature    # Erstellen, auf aktuellem Branch bleiben
$ git checkout -b feature/new-feature # Erstellen UND wechseln (klassisch)
$ git switch -c feature/new-feature  # Erstellen UND wechseln (modern)
```

Zwischen Branches wechseln:

```
$ git checkout feature/fix-code-quality # klassisch
$ git switch feature/fix-code-quality   # modern
```

Branch-Operationen — Löschen & Inspizieren

```
# Branch löschen
$ git branch -d feature/old-feature    # Sicher – nur wenn gemerged
$ git branch -D feature/old-feature    # Erzwingen – auch wenn nicht gemerged

# Branches inspizieren
$ git branch                          # Lokale Branches
$ git branch -a                       # Alle Branches (inkl. Remote)
$ git branch --merged main            # In main gemergte Branches
$ git branch --no-merged              # Noch nicht gemergte Branches
```

Nach dem Mergen löschen hält das Repo sauber. Branches sind günstig — erstelle einen frischen für das nächste Feature.

Der `main`-Branch ist besonders

- **Standard-Branch** — wird beim Initialisieren des Repos angelegt
- **Produktions-Branch** — muss immer stabil und deploybar sein
- **Geschützt** — in professionellen Teams können Sie *nicht* direkt auf `main` pushen
- **Merge-Ziel** — Feature-Banches werden beim Fertigstellen in `main` gemerged

Goldene Regel: Niemals direkt auf `main` pushen. Immer Feature-Banches verwenden.

(Das erzwingen Sie gleich mit Branch-Schutz.)

Namenskonventionen für Feature-Branches

✓ Gut — beschreibend, mit Präfix

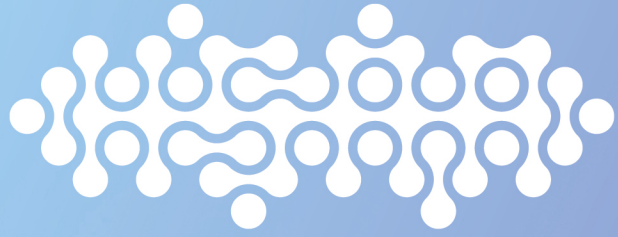
```
feature/improve-road-curve  
fix/intersection-calculation  
refactor/ray-calculation  
docs/add-usage-instructions  
fix/42-intersection-bug
```

✗ Schlecht — nichtssagend, vage

```
branch1  
test  
dominik-stuff  
fix  
new
```

Best Practices:

- Kleinbuchstaben mit Bindestrich (kebab-case)
- Präfix: feature/, fix/, refactor/, docs/
- Beschreibend, aber kurz
- Ticket-Nummer einbauen, wenn passend



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Feature-Branch-Workflow

3

Der 10-Schritte-Lebenszyklus

1. Feature-Branch von `main` erstellen
2. Feature implementieren (mehrere Commits)
3. Branch zu GitHub pushen
4. Pull Request (PR) öffnen
5. Automatische Checks laufen
6. Code-Review durch das Team
7. Feedback einarbeiten (weitere Commits)
8. Approval + Checks grün
9. In `main` mergen
10. Feature-Branch löschen

Schritte 1–3: lokal · **Schritte 4–9:** kollaborativ · **Schritt 10:** Aufräumen

Schritte 1–3: Erstellen · Implementieren · Pushen

```
# 1. Feature-Branch vom aktualisierten main erstellen
$ git checkout main
$ git pull origin main
$ git checkout -b feature/fix-code-quality

# 2. Implementieren – kleine, logische Commits
$ vim main.py
$ git add main.py
$ git commit -m "Fix PEP8: imports auf verschiedene Zeilen aufteilen"
$ vim main.py
$ git add main.py
$ git commit -m "Fix PEP8: Spacing um Operatoren"

# 3. Zu GitHub pushen (erstes Mal: -u verknüpft lokal ↔ remote)
$ git push -u origin feature/fix-code-quality
```

Gute Commits: klein, fokussiert, beschreibend · referenziere den Standard (PEP8) · Checks vorher laufen lassen.

Schritt 4: Pull Request öffnen

Ein **Pull Request** bittet: *"Bitte review meine Änderungen und merge sie."*

Auf GitHub:

1. **Pull requests** → **New pull request** klicken
2. **Base:** `main` · **Compare:** `feature/fix-code-quality`
3. Diff prüfen — alle PEP8-Fixes hervorgehoben
4. **Titel** (prägnant) und **Beschreibung** (was / warum / wie / Testing) ausfüllen
5. **Create pull request** klicken

Der PR ist jetzt für das ganze Team sichtbar — sie können kommentieren, reviewen und approve.

Schritte 5–6: Automatische Checks & Code-Review

5. Automatische Checks (GitHub Actions)

```
✓ Ruff check passed
✓ Ruff format check passed
✓ Pyright type check passed
✓ Tests pass
Alle Checks grün – bereit zum Merge!
```

Check rot → lokal fixen → pushen → PR läuft automatisch neu.

6. Code-Review — Reviewer achten auf:

1. **Korrektheit** — tut es, was es behauptet?
2. **Qualität** — lesbar, wartbar, getestet?
3. **Design** — richtiger Ansatz?
4. **Style** — hält sich an Konventionen?
5. **Randfälle** — was kann schiefgehen?

Schritte 7–9: Feedback · Mergen · Löschen

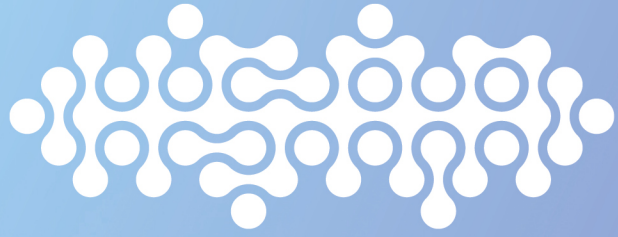
7. Feedback einarbeiten — weitere Commits pushen, PR aktualisiert sich automatisch.

8. Mergen (auf GitHub, drei Strategien):

Strategie	Was sie macht
Merge commit	Alle Commits + neuer Merge-Commit (Standard)
Squash and merge	Alle Commits zu einem kombiniert
Rebase and merge	Commits linear auf <code>main</code> replayed

9. Feature-Branch löschen — auf GitHub "Delete branch" klicken, dann lokal:

```
$ git checkout main
$ git pull origin main
$ git branch -d feature/fix-code-quality
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Pull Requests

4

Was ist ein Pull Request?

Trotz des Namens ist ein **Pull Request** eigentlich ein **Merge Request**.

Sie bitten die Maintainer, Ihre Änderungen in ihren Branch zu ziehen. Historischer Begriff; moderne Bedeutung:

"Bitte review und merge meine Änderungen."

Warum PRs wichtig sind:

- Review-Gate, bevor Code `main` erreicht
- Ort für Design-Diskussion und frühes Erkennen von Bugs
- Audit-Trail: *wer* hat *was* und *wann* approved
- Natürlicher Anknüpfungspunkt für automatische CI-Checks

Anatomie eines Pull Requests

1. **Titel & Beschreibung** — was er tut, warum er nötig ist, Besonderheiten
2. **Commits** — alle Commits vom Feature-Branch; aktualisiert sich beim Pushen
3. **Files changed (Diff)** — zeilenweise Gegenüberstellung, einfach zu reviewen
4. **Conversation** — Review-Kommentare, Approvals, Rejections
5. **Checks** — automatische Tests / Linting / Type-Checking / Build-Status

Jeder Teil hat einen eigenen Tab in der GitHub-UI — Reviewer arbeiten sie der Reihe nach durch.

PR Best Practices

Größe zählt

✓ **Guter PR** — 50–300 Zeilen

- Einfach zu reviewen
- Klarer Zweck
- Versteckt weniger Bugs

✗ **Schlechter PR** — 2.000+ Zeilen

- Überwältigend
- Wird nur abgenickt
- Versteckt Probleme

Beschreibungs-Template

```
## Was  
Ein-Satz-Zusammenfassung.  
  
## Warum  
Motivation, Link zum Issue.  
  
## Wie  
Wichtige Entwurfsentscheidungen.  
  
## Testing  
Welche Befehle, welches Ergebnis.  
  
## Checklist  
- [ ] Tests aktualisiert  
- [ ] Docs aktualisiert
```

Wenn der PR zu groß ist — in mehrere kleinere PRs aufteilen.

Einen PR reviewen — Beispiel

```
# Code im PR
def generate_road_profile(num_points=100, x_max=80):
    x = np.linspace(0, x_max, num_points)
    x_norm = x / x_max
    y = 0.015 * x_norm**3 * x_max + 0.3 * np.sin(2 * np.pi * x_norm) + 0.035 * x_norm * x_max
    y = y - y[0]
    return x, y
```

Reviewer-Kommentare:

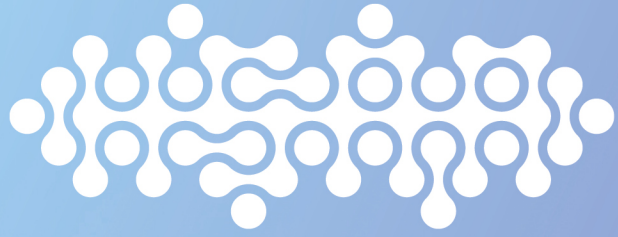
- Type Hints hinzufügen: `-> tuple[np.ndarray, np.ndarray]`
- Validieren, dass `num_points > 0` und `x_max > 0` ?
- Magic Numbers (`0.015`, `0.3`, `0.035`) — als benannte Konstanten extrahieren?

Autor antwortet: einen Folge-Commit pushen; der PR aktualisiert sich automatisch.

PR-Zustände

Zustand	Bedeutung
Open	Unter Review, kann noch geändert werden, noch nicht gemerged
Merged	Akzeptiert und in den Ziel-Branch gemerged — Branch kann gelöscht werden
Closed	Abgelehnt oder aufgegeben — kann bei Bedarf wieder geöffnet werden
Draft	Work in progress, noch nicht reif für Review — gut für frühes Feedback

Tipp: Sie öffnen Ihren PR früh als **Draft**, wenn Sie Feedback zur Richtung wollen, bevor Sie mehr Zeit investieren.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Branch-Schutz

5

Branch-Protection-Regeln

Das Problem: technisch hindert Sie nichts daran, das hier zu tun —

```
$ git checkout main  
$ git commit -m "Quick fix"  
$ git push origin main    # Umgeht den PR-Prozess!
```

Die Lösung: Branch Protection auf GitHub konfigurieren (Settings → Branches):

-  **Require a pull request before merging**
-  **Require approvals** (mindestens 1 Review)
-  **Require status checks to pass** (CI-Checks)
-  **Do not allow bypassing these settings** (auch Admins)

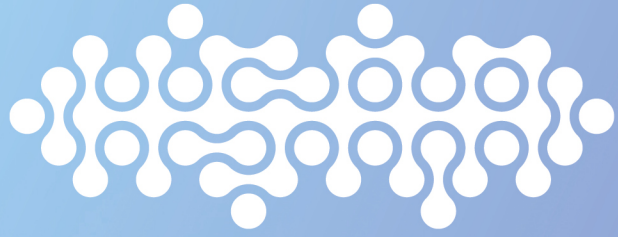
Was passiert beim Umgehen

```
$ git push origin main

remote: error: GH006: Protected branch update failed for refs/heads/main.
remote: error: Cannot push directly to main. Use a pull request.
To github.com:yourname/yourrepo.git
 ! [remote rejected] main -> main (protected branch hook declined)
error: failed to push some refs to 'github.com:yourname/yourrepo.git'
```

Vorteile:

1. **Erzwingt Code-Review** — kein Code erreicht `main` ohne Review
2. **Erzwingt Qualitätsprüfungen** — Tests, Linting, Type-Checking müssen grün sein
3. **Verhindert Unfälle** — kein versehentliches Pushen auf den falschen Branch
4. **Erzeugt Audit-Trail** — jede Änderung hat einen PR mit Diskussion



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

CI/CD-Vorschau

6

Als nächstes: Automatische Qualitätsprüfungen

In Teil 2 fügen Sie **GitHub Actions** hinzu, die bei jedem PR Checks laufen lassen:

```
# .github/workflows/quality.yml
name: Code Quality

on:
  pull_request:          # Trigger bei PRs (nicht bei direkten Pushes)
    branches: [main]

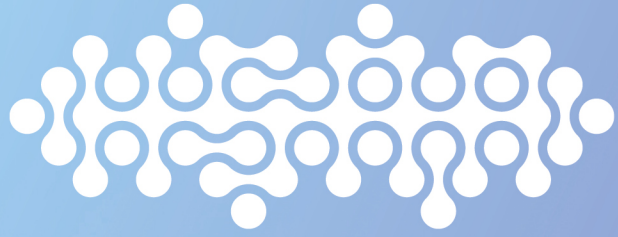
jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: astral-sh/setup-uv@v4
      - run: uv sync --dev
      - run: uv run ruff check .
```

Beim Öffnen eines PR läuft GitHub automatisch alles durch und blockiert den Merge bei Fehlern.

Der komplette professionelle Workflow

1. Feature-Branch von main erstellen
2. Feature implementieren (kleine Commits)
3. Branch zu GitHub pushen
4. Pull Request öffnen
- ↓
5. Automatische Checks laufen (GitHub Actions)
✓ Ruff linter · ✓ Ruff format · ✓ Pyright · ✓ Tests
- ↓
6. Checks rot → fixen → pushen → Checks laufen erneut
- ↓
7. Review anfragen → Feedback einarbeiten → Approval holen
- ↓
8. In main mergen (nur möglich, wenn Checks grün + approved)
9. Feature-Branch löschen

So liefern professionelle Teams Code in hoher Qualität aus.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Ihr erster Feature-Branch

7

Aufgabenstellung

Ziel: Ein echtes Code-Quality-Problem im Road Profile Viewer mit dem vollständigen Feature-Branch-Workflow fixen.

Das Ziel: die ungenutzte `HelperFunction` in `main.py` entfernen — einer der 14 Ruff-Fehler aus Kapitel 02.

```
# Unused function entfernen oder auskommentieren:  
# def HelperFunction(val):  
#     """Unused helper function that violates naming convention"""  
#     result=val*2  
#     return result
```

Sie erstellen einen Branch, machen den Fix, öffnen einen PR, mergen und räumen auf.

Durchlauf: Schritte 1–4

```
# 1. Von aktualisiertem main starten
$ git checkout main
$ git pull origin main

# 2. Feature-Branch erstellen
$ git checkout -b feature/remove-unused-helper

# 3. Änderung machen
$ vim main.py    # HelperFunction auskommentieren oder löschen

# 4. Fix lokal verifizieren
$ uv run ruff check main.py
# → Sollte EINEN Fehler weniger zeigen
```

Durchlauf: Schritte 5–8

```
# 5. Committed
$ git status
$ git add main.py
$ git commit -m "Fix PEP8: unused HelperFunction entfernen"

# 6. Zu GitHub pushen (erstes Mal: -u)
$ git push -u origin feature/remove-unused-helper
```

7. Auf GitHub → Pull Request öffnen

- Titel: `Fix PEP8: unused HelperFunction entfernen`
- Beschreibung: was / warum / Testing / vorher/nachher

8. Eigenen PR reviewen im "Files changed"-Tab

- Löschung rot hervorgehoben
- Sicherstellen, dass sonst nichts versehentlich geändert wurde

Durchlauf: Schritte 9–11

```
# 9. Auf GitHub mergen – "Merge pull request" → "Confirm merge" → "Delete branch"

# 10. Lokal aufräumen
$ git checkout main
$ git pull origin main
$ git branch -d feature/remove-unused-helper

# 11. Verifizieren
$ git log --oneline
# → Sie sollten sehen: "Fix PEP8: unused HelperFunction entfernen"
$ uv run ruff check main.py
# → Ein Fehler weniger! 📉
```

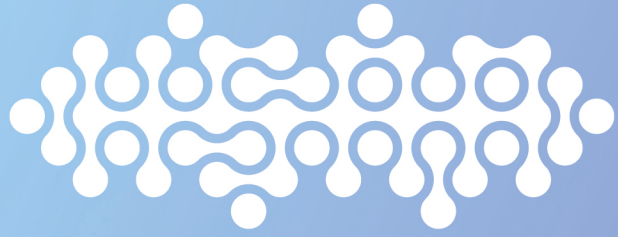
Challenge: wiederhole diesen Prozess für jeden der verbleibenden 13 Ruff-Fehler — ein Branch, ein PR pro Fix.

Checkliste zur Aufgabe

Haben Sie:

- ☒ Von aktualisiertem `main` gestartet?
- ☒ Einen beschreibenden Feature-Branch-Namen verwendet?
- ☒ Eine fokussierte Änderung gemacht (nur den einen Fix)?
- ☒ Eine klare Commit-Message mit Bezug auf den Standard (PEP8) geschrieben?
- ☒ Zu GitHub gepusht?
- ☒ Einen Pull Request mit echter Beschreibung geöffnet?
- ☒ Erst nach eigenem Review gemerged?
- ☒ Den Feature-Branch lokal nach dem Merge gelöscht?

Bei „Ja“ zu allem → Sie denken wie professionelle Entwickler.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Täglicher Workflow & Zusammenfassung

8

Spickzettel: täglicher Workflow

Feature starten

```
git checkout main
git pull origin main
git checkout -b feature/my-feature
```

Während der Arbeit

```
git status
git diff
git add <file>
git commit -m "Message"
```

Pushen

```
git push -u origin feature/my-feature # erstes Mal
git push                             # danach
```

Abschluss (nach PR-Merge)

```
git checkout main
git pull origin main
git branch -d feature/my-feature
```

Nützliche Git-Befehle

Information

```
git status
git log --oneline --graph --all
git branch
git diff
git diff --staged
git show <commit-hash>
```

Rückgängig machen

```
git restore <file>           # Arbeitsverzeichnis-Änderung verwerfen
git restore --staged <file>  # Unstagen (Änderung behalten)
git reset HEAD~1             # Letzten Commit zurücknehmen (Dateien bleiben)
git commit --amend           # Letzte Commit-Message ändern
```

Häufige Szenarien

Auf den falschen Branch committet (*noch nicht gepusht*)

```
git reset HEAD~1  
git checkout correct-branch  
git add .  
git commit -m "Message"
```

Feature-Branch mit neuestem **main** aktualisieren

```
git checkout feature/my-feature  
git pull origin main # Konflikte ggf. auflösen  
git push
```

Vergessen, einen Feature-Branch zu erstellen

```
git stash  
git checkout -b feature/new-feature  
git stash pop  
git add . && git commit -m "Message"
```



Vom Cowboy-Coding zum professionellen Workflow

Vorher — Cowboy-Coding

```
$ git add .  
$ git commit -m "stuff"  
$ git push origin main  
# 🔥 Hoffen, dass nichts bricht!
```

Nachher — professioneller Workflow

```
$ git checkout main  
$ git pull origin main  
$ git checkout -b feature/fix-code-quality  
# ... arbeiten ...  
$ git add main.py  
$ git commit -m "Fix PEP8: imports trennen..."  
$ git push -u origin feature/fix-code-quality  
# PR öffnen · Review · Merge  
$ git checkout main && git pull  
$ git branch -d feature/fix-code-quality
```

Wichtigste Erkenntnisse

1. **Niemals direkt auf `main` pushen** — immer Feature-Banches verwenden
2. **Branches sind günstig** — einer pro Feature oder Fix
3. **Commits sind permanent** — klare Messages schreiben
4. **PRs ermöglichen Kollaboration** — auch wenn Sie allein arbeiten
5. **Kleine PRs sind besser** — 50–300 Zeilen ist ideal
6. **Branch-Schutz ist essenziell** — Workflow durch Regeln erzwingen
7. **Von aktualisiertem `main` starten** — immer `pull` vor dem Branch
8. **Gemergte Branches löschen** — Repo sauber halten
9. **`git status` ist Ihr Freund** — ständig ausführen
10. **Dieser Workflow ist Industriestandard** — jetzt lernen, für immer nutzen



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

WEITER: REMOTE-TRACKING
BRANCHES