



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Chapter 02

FEATURE DEVELOPMENT

The Problem with Pushing to Main

```
$ git add .  
$ git commit -m "Added new feature"  
$ git push origin main  
# Hope for the best 🙌
```

Serious problems with this workflow:

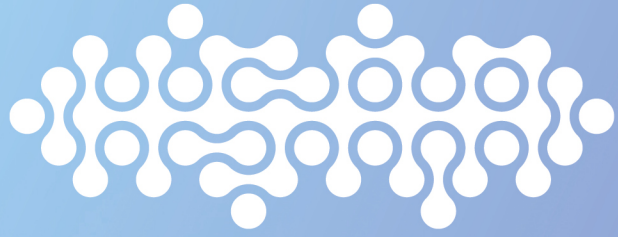
1. **No review** — Code goes straight to production
2. **No safety net** — Once pushed, it's live
3. **Breaks the team** — Bugs block everyone
4. **No collaboration** — No chance for feedback
5. **Hard to track** — All commits mixed on `main`

Real consequence: You push broken code at 11 PM. Next morning, your teammate's environment is broken. They lose 2 hours debugging **your** code.

Learning Objectives

By the end of this lecture you will:

1. **Understand Git's fundamentals** — working directory, staging, commits, branches
2. **Master feature branch workflows** — create, work, merge
3. **Use pull requests** — request review before merging
4. **Apply branch protection** — prevent direct pushes to `main`
5. **Prepare for CI/CD** — foundation for automated quality checks
6. **Collaborate professionally** — how real software teams work



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Git Fundamentals

1

The Three States of Git

1. Working Directory (Modified)

- Files you're actively editing
- Changes not yet saved to Git

2. Staging Area (Staged)

- Changes marked for the next commit
- A preview of that commit

3. Repository (Committed)

- Permanent snapshot in Git's history

```
Working Directory  
  ↓ git add  
Staging Area  
  ↓ git commit  
Repository
```

Every change flows **through** the staging area on its way to being a commit.

Visualizing the Three States

Scenario: Fix an import in `main.py`.

```
# 1. WORKING DIRECTORY – edit the file
$ vim main.py                # import sys,os → import sys\nimport os
$ git status
Changes not staged for commit:
  modified:   main.py

# 2. STAGING AREA – mark it for commit
$ git add main.py
$ git status
Changes to be committed:
  modified:   main.py

# 3. REPOSITORY – save the snapshot
$ git commit -m "Fix PEP8: separate imports onto different lines"
$ git status
nothing to commit, working tree clean
```

Why the Staging Area Exists

It gives you control over what goes into each commit.

Instead of one giant commit, build logical commits from a messy working directory:

```
$ git add -p main.py                # Stage only import-related changes
$ git commit -m "Fix PEP8: separate imports, remove unused"

$ git add -p main.py                # Stage only spacing changes
$ git commit -m "Fix PEP8: add spacing around operators"

$ git add main.py                  # Stage the remaining naming fix
$ git commit -m "Fix PEP8: rename HelperFunction to helper_function"
```

Each commit now has a single clear purpose — easier to review, easier to revert.

`git status` — Your Best Friend

```
$ 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 (add to .gitignore)
```

- **Staged** → will be in the next commit
- **Modified** → changed but *not* in the next commit
- **Untracked** → Git has never seen this

Pro tip: Run `git status` constantly. It's your map.

Anatomy of a Commit

A commit is a **snapshot** of your project at a point in time.

```
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) — unique identifier
2. **Author / Date** — who and when
3. **Message** — what changed and *why*
4. **Diff** — the actual code change

Once created, a commit is **permanent** — that snapshot lives on forever.

Viewing Git History

```
$ 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
```

- * — a commit
- HEAD -> — where you currently are
- (origin/main, main) — branch pointers
- d4e5f6g — short commit hash
- The message on the right tells the story

Common `git status` Scenarios

Clean working tree

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

✓ Safe to switch branches.

Uncommitted changes

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

⚠ Commit or stash before switching.

Staged but not committed

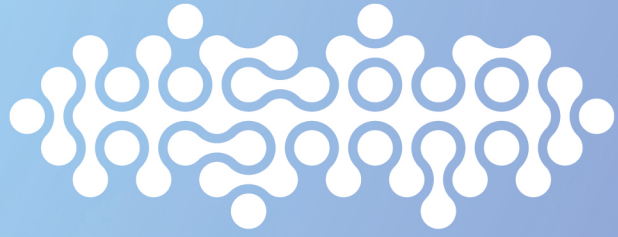
```
Changes to be committed:  
  modified:   main.py
```

⚠ Run `git commit`.

Staged AND unstaged

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

⚠ Edited after staging — re-add if needed.



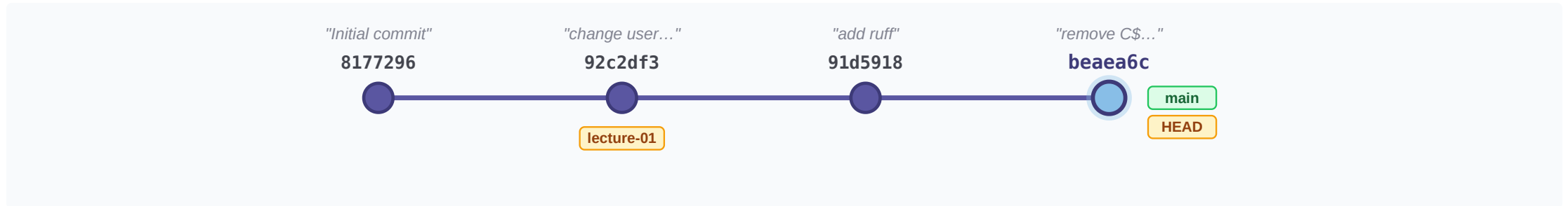
Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Branches

2

Example 1: Before Creating the Branch

State on 2025-10-21 at 16:26:11 (on main)



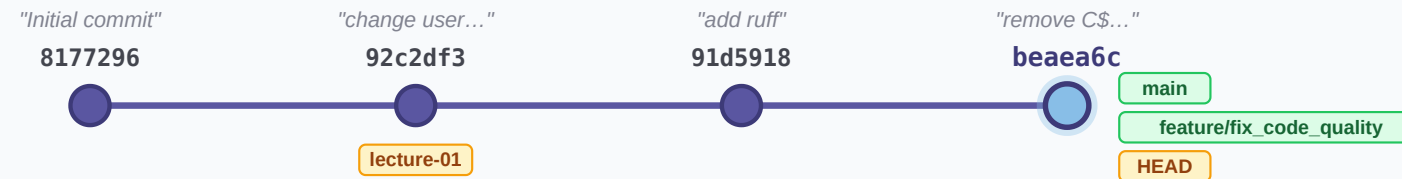
- `beaea6c` is the actual commit (snapshot: "remove C\$ from ruff settings")
- `main` is a branch pointer pointing to commit `beaea6c`
- `HEAD` points to `main` (you're currently on the main branch)
- `lecture-01` tag points to commit `92c2df3`

Example 1: Creating the Branch

Command:

```
$ git checkout -b feauture/fix_code_quality  
# (note: had the typo originally!)
```

After creating the branch:



Example 1: What Just Happened?

1. Git created a **NEW pointer** called `feature/fix_code_quality`
2. This pointer points to the **SAME commit** as `main` (`beaea6c`)
3. Git moved **HEAD** to point to the new branch
4. **No commits were created or copied** - just a new pointer!

Key insight: Creating a branch is cheap - it's just creating a pointer (a few bytes)!

Example 2: First Commit on Branch

Making the first commit:

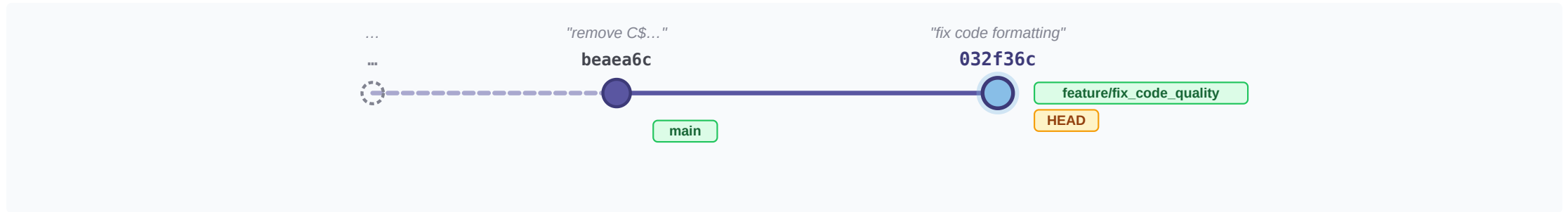
```
$ vim main.py # Fix code formatting
$ git add main.py
$ git commit -m "fix code formatting using ruff format"
```

Before the commit:



Example 2: After First Commit

After the commit (032f36c):



What happened:

1. Git created **NEW commit** 032f36c
2. Git moved the feature/fix_code_quality pointer forward to 032f36c
3. The main pointer **stayed at** beaea6c (unchanged!)
4. HEAD still points to feature/fix_code_quality

Example 2: Key Insight

Only the current branch pointer moves when you commit.

Other branches stay where they are!

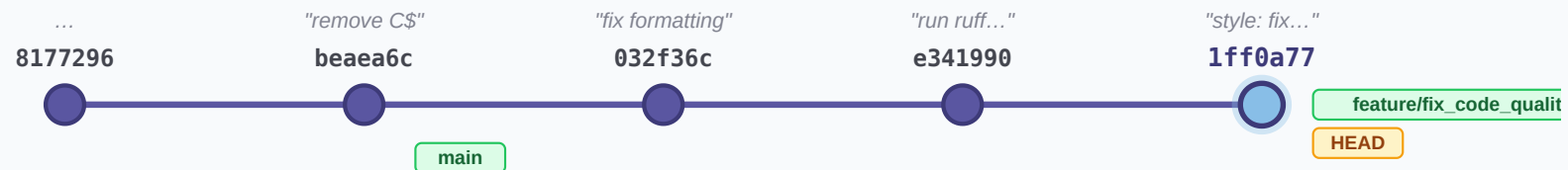
This is how feature branches provide **isolation** - your work on the feature branch doesn't affect `main`.

Example 3: After Multiple Commits

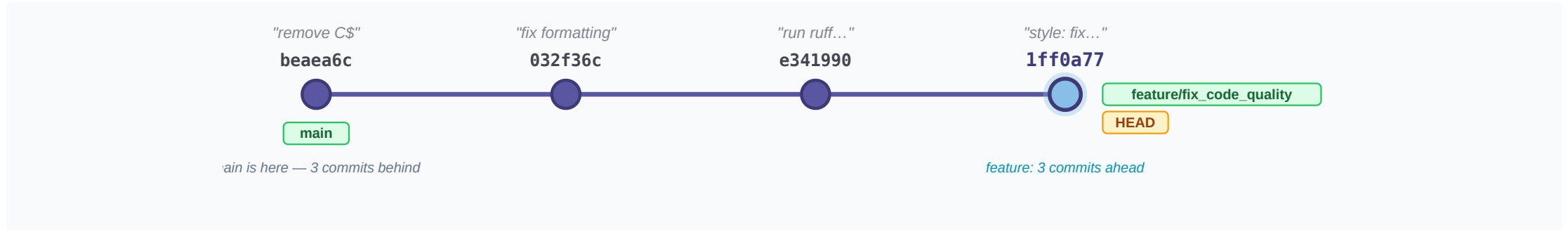
Making more commits:

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

Final state after 3 commits on branch:



Example 3: Visualizing the Divergence



What we see:

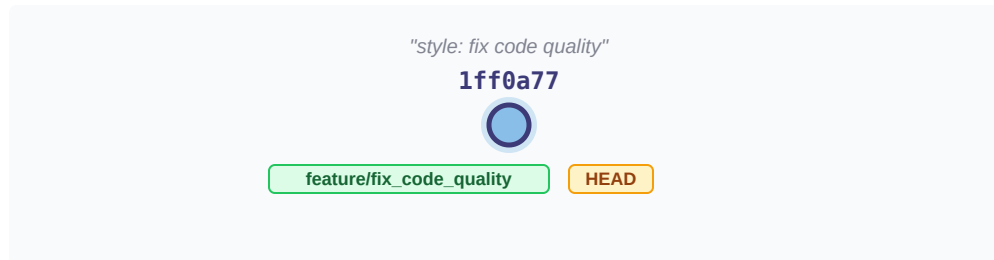
- `main` is still at `beaea6c` (where we branched from)
- `feature/fix_code_quality` has moved forward through 3 commits
- The feature branch work is **isolated** from `main`
- `HEAD` points to the feature branch (we're working on it)

Example 4: Switching Branches

Switching back to main:

```
$ git checkout main
```

Before the checkout:



After the checkout:



Example 4: What Changed?

Only HEAD moved!

- HEAD moved from feature branch pointer to main pointer
- Your working directory changed to show commit `beaea6c`
- The branch pointers themselves **didn't move**
- `feature/fix_code_quality` still points to `1ff0a77`

This is all that `git checkout` **does** - it moves the HEAD pointer!

Example 5: Merging Feature Branch

When you merge the feature branch:

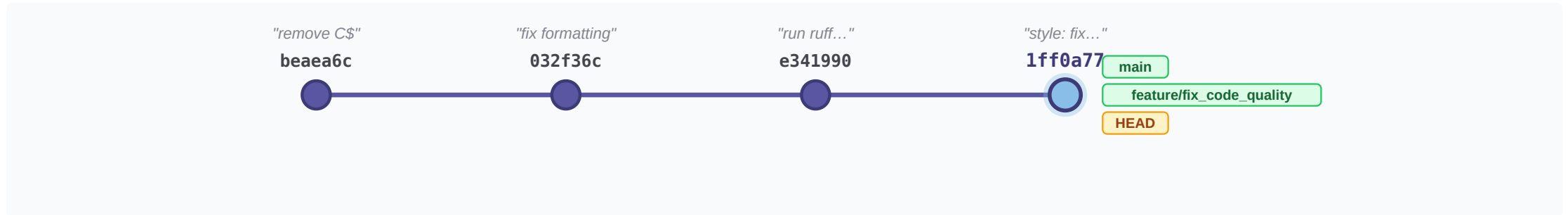
```
$ git checkout main  
$ git merge feauture/fix_code_quality  
# (Or merge via GitHub PR)
```

Result: Fast-forward merge



Example 5: After the Merge

After the merge (fast-forward):

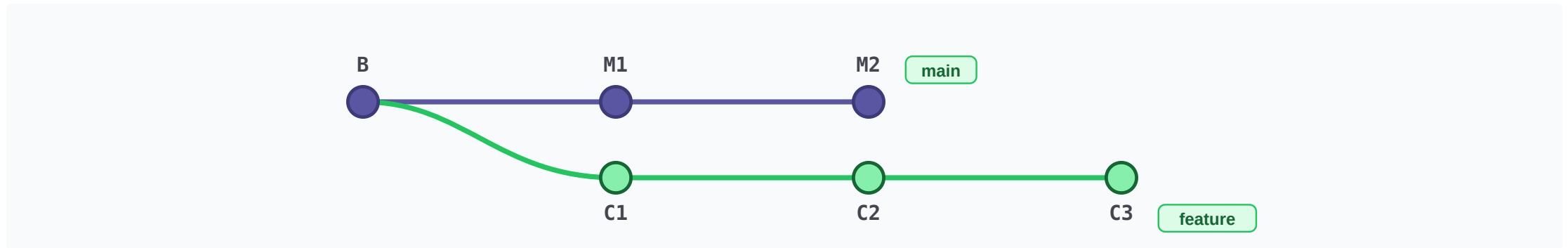


What happened:

- Git moved the `main` pointer forward to `1ff0a77`
- This is a "fast-forward" merge (main just catches up)
- Both `main` and `feature/fix_code_quality` now point to the same commit
- No merge commit was needed (linear history)

But what if `main` has moved on?

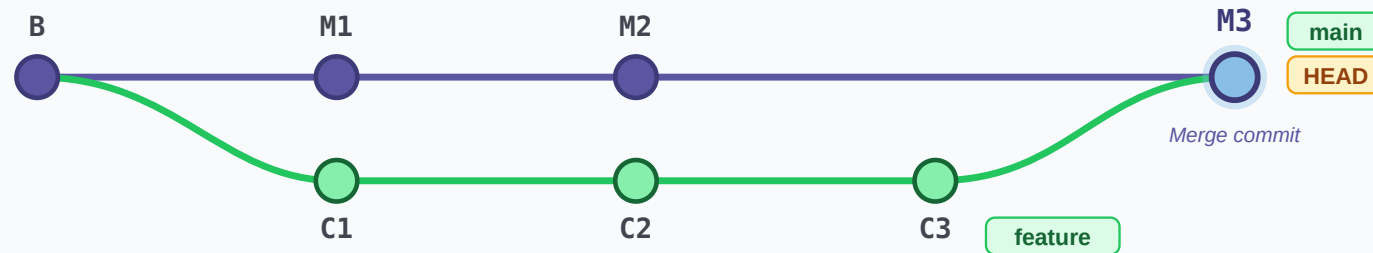
In the example above, things were easy — `main` hadn't moved. In real teams, multiple developers work in parallel: while we were on the feature branch, someone else merged their PR.



`main` and our feature have **diverged** — a fast-forward is no longer possible. **The question:** how do we get our three commits back into `main` cleanly?

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

```
$ git checkout main  
$ git merge --no-ff feature/fix_code_quality  
# Creates a merge commit M3 with two parents: M2 and C3
```



M3 is a **new commit** with two parents: M2 (from `main`) and C3 (from the `feature` branch). The feature commits keep their **original hashes** — they are not copied; they are simply now reachable from `main` via M3.

Strategy 1: Merge Commit — Pros & Cons

✓ Pros

- Full history is preserved
- Visible when a feature was integrated
- Original commit hashes unchanged
- Branch lineage readable in `git log -graph`

⚠ Cons

- "Diamond" history can get cluttered
- Many "Merge branch X into main" commits
- With many small PRs, becomes noisy

Typical use: teams that want feature lineage to remain visible — every merge is an explicit integration point in `git log --graph`.

After the Merge: Cleanup & Commit Message

What's in the merge commit?

Git generates a default message and opens our editor:

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

Set or override directly:

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

On GitHub via PR — auto-generated:

```
Merge pull request #42 from user/feature/fix_code_quality  
  
<PR description as body>
```

What happens to the feature branch?

In practice, **delete** it after the merge:

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

Are our commits gone? No. **C1**, **C2**, **C3** remain reachable via **M3**:

- `git log` still shows them
- `git blame` still finds their authors
- Hashes still work in issues, PRs, Slack
- `git checkout C2` jumps straight to it

A branch is just a pointer — deleting the pointer does not delete the commits.

Strategy 2: Squash & Merge

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



S1 is a **new commit** that combines all changes from **C1**, **C2**, **C3** into a single diff. **C1**, **C2**, **C3** themselves are not in **main**'s history — the feature branch is typically deleted afterwards.

Strategy 2: Squash & Merge — Pros & Cons

✓ Pros

- Linear, clean history on `main`
- One commit per feature/PR
- Easy revert of an entire feature
- Less noise from many small "WIP" commits

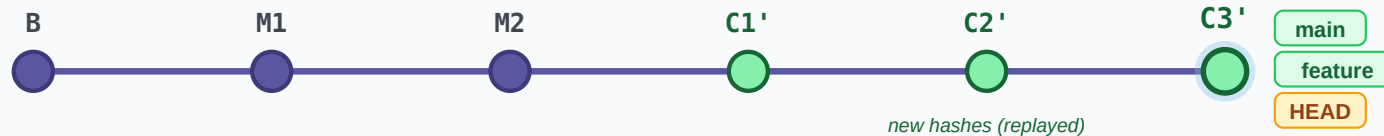
⚠ Cons

- Individual commits are lost on `main`
- `git bisect` is less granular
- Co-authoring harder to see
- Refactor + bug fix often land in the same commit

Typical use: GitHub repos configured with "Squash and merge" as default — feature authors make as many small commits as they want, and the merge collapses them into one clean `main` commit.

Strategy 3: Rebase + Merge

```
$ git checkout feature/fix_code_quality
$ git rebase main          # C1, C2, C3 are "replayed" onto M2
$ git push --force-with-lease
$ git checkout main
$ git merge feature/fix_code_quality  # Now a fast-forward is possible!
```



C1', C2', C3' have **new hashes** — they are replayed copies of the original C1, C2, C3. The originals only exist in the reflog.

Strategy 3: Rebase + Merge — Pros & Cons

✓ Pros

- Linear history without merge commits
- All individual commits remain visible
- `git log` reads like a story

✗ Cons

- **Rewrites history** — new hashes
- **Never rebase shared branches**
- Conflicts often need to be resolved per commit
- Force-push required

Golden rule: only rebase branches that nobody else is working on — otherwise you destroy your colleagues' history.

Which Strategy When?

Strategy	History on <code>main</code>	When it fits
Fast-Forward	Linear, no merge traces	Solo projects, <code>main</code> hasn't moved
Merge Commit (<code>--no-ff</code>)	Branched, complete	Teams that want to see feature lineage
Squash & Merge	Linear, one commit per PR	Many small "WIP" commits; clean <code>main</code>
Rebase + FF	Linear, all commits	Updating a feature branch locally before PR

In practice: teams decide per repo, not per PR. GitHub repos are often configured to **Squash-only** or **Merge-Commit-only**. Rebase usually stays a local tool to keep your own feature branch clean.

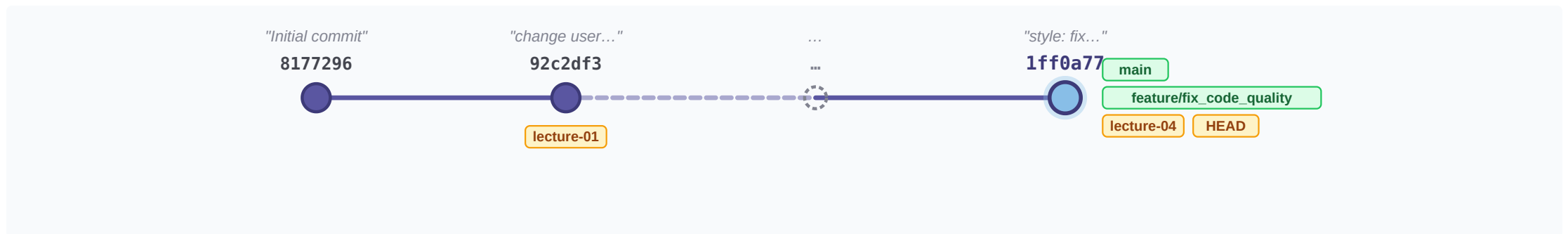
Important: don't pick the "perfect" strategy — pick one that fits the team and apply it consistently.

Example 6: Adding a Tag

Creating the lecture-04 tag:

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

Current state with tag:



All four pointers point to the SAME commit `1ff0a77` !

Example 6: Tags are Also Pointers

Tags are just like branches - they're pointers to commits!

The difference:

- **Branch pointers** move forward when you commit
- **Tag pointers** never move (permanent markers)

Both are just references to commits - not copies of commits.

Example 7: Renaming the Branch

Fixing the typo:

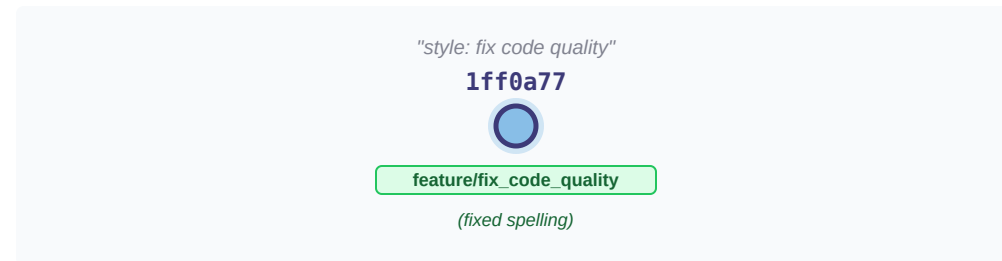
```
$ git branch -m feauture/fix_code_quality feature/fix_code_quality
```

What happened:

Before rename:



After rename:



Example 7: The Commit Never Changed

The commit `1ff0a77` never changed!

We just:

1. Created a new pointer with the correct spelling
2. Deleted the old pointer with the typo

This is why:

- The tag `lecture-04` still works (points to commit, not branch name)
- Anyone who checked out the tag never saw the typo
- Branches are just labels, not the actual code

Why Use Branches?

Without branches (pushing to main):

```
main: A → B → C → D(broken) → E(fix) → F(fix) → G(works)
      ↑
      Everyone sees broken code
      Other devs blocked
      Hard to revert
```

With feature branches:

```
main:  A → B → C —————→ M (merge)
      ↗           ↘
feature: D → E → F → G (tested)
      ↑
      Only you see this
      Safe to experiment
      Easy to delete
```

5 benefits: isolation · safety · experimentation · review · easy rollback

Branch Operations — Create & Switch

View branches:

```
$ git branch
main
* feature/fix-code-quality    ← You're here (*)
feature/improve-road-curve
```

Create a branch:

```
$ git branch feature/new-feature      # Create, stay on current branch
$ git checkout -b feature/new-feature  # Create AND switch (classic)
$ git switch -c feature/new-feature    # Create AND switch (modern)
```

Switch between branches:

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

Branch Operations — Delete & Inspect

```
# Delete a branch
$ git branch -d feature/old-feature    # Safe — only if merged
$ git branch -D feature/old-feature    # Force — even if not merged

# Inspect branches
$ git branch                          # Local branches
$ git branch -a                       # All branches (including remotes)
$ git branch --merged main            # Branches merged into main
$ git branch --no-merged               # Branches not yet merged
```

Delete after merging keeps the repo clean. Branches are cheap — create a fresh one for the next feature.

The `main` Branch is Special

- **Default branch** — created when you initialize a repo
- **Production branch** — should always be stable and deployable
- **Protected** — in professional teams, you *cannot* push directly to `main`
- **Merge target** — feature branches merge into `main` when complete

Golden rule: Never push directly to `main`. Always use feature branches.

(We'll enforce this with branch protection in a few slides.)

Feature Branch Naming Conventions

✓ Good — descriptive, prefixed

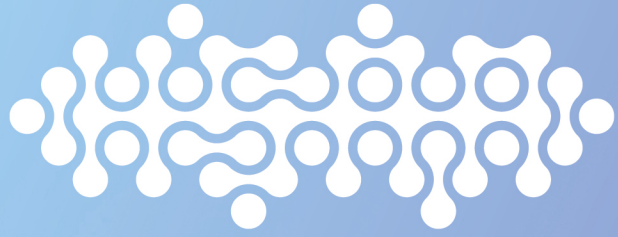
```
feature/improve-road-curve  
fix/intersection-calculation  
refactor/ray-calculation  
docs/add-usage-instructions  
fix/42-intersection-bug
```

✗ Bad — meaningless, vague

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

Best practices:

- Lowercase with hyphens (kebab-case)
- Prefix: feature/, fix/, refactor/, docs/
- Descriptive but concise
- Include issue number if applicable



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Feature Branch Workflow

3

The 10-Step Lifecycle

1. Create feature branch from `main`
2. Implement feature (multiple commits)
3. Push branch to GitHub
4. Open Pull Request (PR)
5. Automated checks run
6. Code review by team
7. Address feedback (more commits)
8. Approval + checks pass
9. Merge to `main`
10. Delete feature branch

Steps 1–3: local · **Steps 4–9:** collaborative · **Step 10:** cleanup

Steps 1–3: Create · Implement · Push

```
# 1. Create feature branch from updated main
$ git checkout main
$ git pull origin main
$ git checkout -b feature/fix-code-quality

# 2. Implement – small, logical commits
$ vim main.py
$ git add main.py
$ git commit -m "Fix PEP8: separate imports onto different lines"
$ vim main.py
$ git add main.py
$ git commit -m "Fix PEP8: add spacing around operators"

# 3. Push to GitHub (first time: -u links local ↔ remote)
$ git push -u origin feature/fix-code-quality
```

Good commits: small, focused, descriptive · reference the standard (PEP8) · run checks first.

Step 4: Open a Pull Request

A **Pull Request** asks: *"Please review and merge my changes."*

On GitHub:

1. Click **Pull requests** → **New pull request**
2. **Base:** `main` · **Compare:** `feature/fix-code-quality`
3. Review the diff — all your PEP8 fixes highlighted
4. Fill in **Title** (concise) and **Description** (what / why / how / testing)
5. Click **Create pull request**

The PR is now visible to the whole team — they can comment, review, and approve.

Steps 5–6: Automated Checks & Code Review

5. Automated checks (GitHub Actions)

✓ Ruff check passed
✓ Ruff format check passed
✓ Pyright type check passed
✓ Tests pass
All checks passed – Ready to merge!

If a check fails, fix locally → push → PR
re-runs automatically.

6. Code review — reviewers look for:

1. **Correctness** — does it do what it claims?
2. **Quality** — readable, maintainable, tested?
3. **Design** — right approach?
4. **Style** — follows conventions?
5. **Edge cases** — what could go wrong?

Steps 7–9: Feedback · Merge · Delete

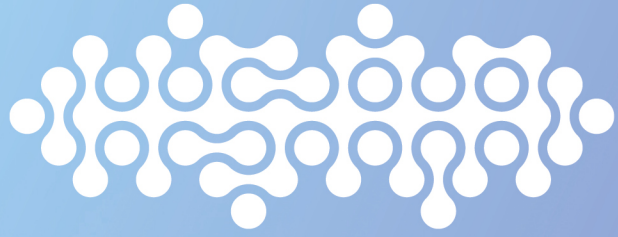
7. Address feedback — push more commits, PR updates automatically.

8. Merge (on GitHub, three strategies):

Strategy	What it does
Merge commit	Keeps all commits + creates merge commit (default)
Squash and merge	Combines all commits into one
Rebase and merge	Replays commits linearly on top of <code>main</code>

9. Delete feature branch — click "Delete branch" on GitHub, then locally:

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



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Pull Requests

4

What is a Pull Request?

Despite the name, a **pull request** is really a **merge request**.

You're asking maintainers to pull your changes into their branch. Historical term; modern meaning is:

"Please review and merge my changes."

Why PRs matter:

- A review gate before code reaches `main`
- A place to discuss design and catch bugs early
- An audit trail of *who* approved *what* and *when*
- A natural target for automated CI checks

Anatomy of a Pull Request

1. **Title & description** — what it does, why it's needed, caveats
2. **Commits** — all commits from your feature branch; updates when you push more
3. **Files changed (diff)** — line-by-line comparison, easy to review
4. **Conversation** — review comments, approvals, rejections
5. **Checks** — automated tests / linting / type checking / build status

Each part has a dedicated tab in the GitHub UI — reviewers move through them in order.

PR Best Practices

Size matters

✓ **Good PR** — 50–300 lines

- Easy to review
- Clear purpose
- Less likely to hide bugs

✗ **Bad PR** — 2,000+ lines

- Overwhelming
- Gets rubber-stamped
- Hides problems

Description template

```
## What
One-sentence summary.

## Why
Motivation, link to issue.

## How
Key implementation decisions.

## Testing
Commands run, results observed.

## Checklist
- [ ] Tests updated
- [ ] Docs updated
```

If the PR is too large — split it into multiple smaller PRs.

Reviewing a PR — Example

```
# Code in the 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's comments:

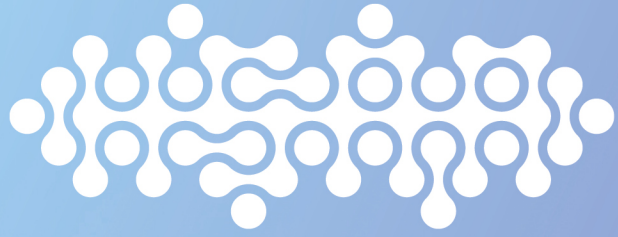
- Consider adding type hints: `-> tuple[np.ndarray, np.ndarray]`
- Validate `num_points > 0` and `x_max > 0`?
- Magic numbers (`0.015`, `0.3`, `0.035`) — extract as named constants?

Author responds: push a follow-up commit; the PR updates automatically.

PR States

State	Meaning
Open	Under review, can still be modified, not yet merged
Merged	Accepted and merged into target branch — branch can be deleted
Closed	Rejected or abandoned — can be reopened if needed
Draft	Work in progress, not ready for review — useful for early feedback

Tip: Open your PR as a **draft** early if you want feedback on direction before investing more time.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Branch Protection

5

Branch Protection Rules

The problem: nothing technically stops you from doing this —

```
$ git checkout main  
$ git commit -m "Quick fix"  
$ git push origin main    # Bypasses the PR process!
```

The fix: configure **branch protection** on GitHub (Settings → Branches):

-  **Require a pull request before merging**
-  **Require approvals** (at least 1 review)
-  **Require status checks to pass** (CI checks)
-  **Do not allow bypassing these settings** (even admins)

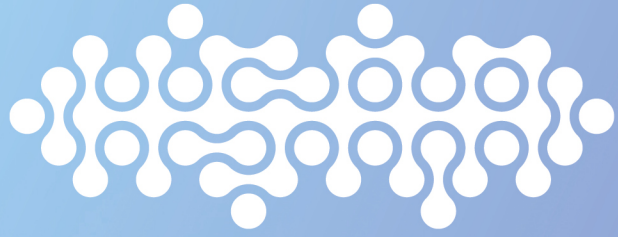
What Happens When You Bypass

```
$ 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'
```

Benefits:

1. **Forces code review** — no code reaches `main` without a review
2. **Enforces quality checks** — tests, linting, type checking must pass
3. **Prevents accidents** — can't accidentally push to the wrong branch
4. **Creates an audit trail** — every change has a PR with discussion



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

CI/CD Preview

6

What's Next: Automated Quality Checks

In Part 2 we add **GitHub Actions** to run checks on every PR:

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

on:
  pull_request:      # Trigger on PRs (not direct pushes)
    branches: [main]

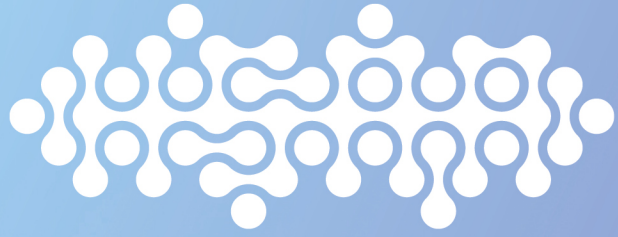
jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: astral-sh/setup-uv@v4
      - run: uv sync --dev
      - run: uv run ruff check .
```

When you open a PR, GitHub runs all of this automatically and blocks merge on failure.

The Complete Professional Workflow

1. Create feature branch from main
2. Implement feature (small commits)
3. Push branch to GitHub
4. Open Pull Request
- ↓
5. Automated checks run (GitHub Actions)
✓ Ruff linter · ✓ Ruff format · ✓ Pyright · ✓ Tests
- ↓
6. If checks fail → fix → push → checks run again
- ↓
7. Request review → address feedback → get approval
- ↓
8. Merge to main (only possible if checks pass + approved)
9. Delete feature branch

This is how professional teams ship high-quality code.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Your First Feature Branch

7

Exercise Scenario

Goal: Fix one real code quality issue in the Road Profile Viewer using the full feature branch workflow.

The target: remove the unused `HelperFunction` in `main.py` — one of the 14 Ruff errors from Chapter 02.

```
# Remove or comment out the unused function:  
# def HelperFunction(val):  
#     """Unused helper function that violates naming convention"""  
#     result=val*2  
#     return result
```

You'll create a branch, make the fix, open a PR, merge, and clean up.

Walkthrough: Steps 1–4

```
# 1. Start from an updated main
$ git checkout main
$ git pull origin main

# 2. Create the feature branch
$ git checkout -b feature/remove-unused-helper

# 3. Make your change
$ vim main.py    # Comment out or delete HelperFunction

# 4. Verify the fix locally
$ uv run ruff check main.py
# → Should show ONE less error than before
```

Walkthrough: Steps 5–8

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

# 6. Push to GitHub (first time: -u)
$ git push -u origin feature/remove-unused-helper
```

7. On GitHub → open Pull Request

- Title: `Fix PEP8: Remove unused HelperFunction`
- Description: what / why / testing / before/after

8. Review your own PR on the "Files changed" tab

- Deletion highlighted in red
- Verify nothing else changed accidentally

Walkthrough: Steps 9–11

```
# 9. Merge on GitHub – click "Merge pull request" → "Confirm merge" → "Delete branch"

# 10. Clean up locally
$ git checkout main
$ git pull origin main
$ git branch -d feature/remove-unused-helper

# 11. Verify
$ git log --oneline
# → You should see: "Fix PEP8: Remove unused HelperFunction"
$ uv run ruff check main.py
# → One less error! 📄
```

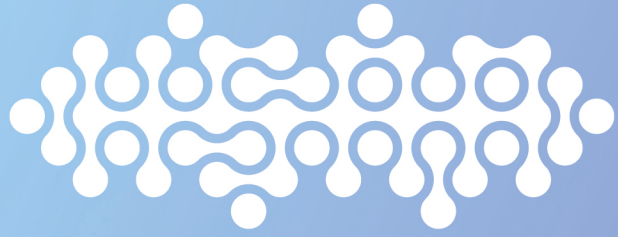
Challenge: repeat this process for each of the remaining 13 Ruff errors — one branch, one PR per fix.

Exercise Checklist

Did you:

- ☒ Start from an up-to-date `main` branch?
- ☒ Use a descriptive feature branch name?
- ☒ Make a focused change (only the one fix)?
- ☒ Write a clear commit message referencing the standard (PEP8)?
- ☒ Push to GitHub?
- ☒ Open a pull request with a real description?
- ☒ Merge only after reviewing your own PR?
- ☒ Delete the feature branch locally after merge?

If yes to all → you're thinking like a professional developer.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Daily Workflow & Summary

8

Daily Workflow Cheat Sheet

Starting a feature

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

While working

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

Pushing

```
git push -u origin feature/my-feature # first time  
git push                             # after
```

Finishing (after PR merged)

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

Useful Git Commands

Information

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

Undoing

```
git restore <file>           # discard working-dir change
git restore --staged <file>  # unstage (keep change)
git reset HEAD~1             # undo last commit (keep files)
git commit --amend           # edit last commit message
```

Common Scenarios

Committed to the wrong branch (*not pushed yet*)

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

Update feature branch with latest **main**

```
git checkout feature/my-feature  
git pull origin main # resolve conflicts if any  
git push
```

Forgot to create a feature branch

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



From Cowboy Coding to Professional

Before — Cowboy coding

```
$ git add .  
$ git commit -m "stuff"  
$ git push origin main  
# 🔥 Hope nothing breaks!
```

After — Professional workflow

```
$ git checkout main  
$ git pull origin main  
$ git checkout -b feature/fix-code-quality  
# ... work ...  
$ git add main.py  
$ git commit -m "Fix PEP8: separate imports..."  
$ git push -u origin feature/fix-code-quality  
# open PR · review · merge  
$ git checkout main && git pull  
$ git branch -d feature/fix-code-quality
```

Key Takeaways

1. **Never push directly to `main`** — always use feature branches
2. **Branches are cheap** — one per feature or fix
3. **Commits are permanent** — write clear messages
4. **PRs enable collaboration** — use them even when working alone
5. **Small PRs are better** — 50–300 lines is ideal
6. **Branch protection is essential** — enforce the workflow with rules
7. **Start from updated `main`** — always `pull` before branching
8. **Delete merged branches** — keep the repo clean
9. **`git status` is your friend** — run it constantly
10. **This workflow is industry standard** — learn it now, use it forever



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

NEXT: REMOTE-TRACKING
BRANCHES