Lecture 2: Project Workflows and Version Control

EC7422: Data Science for Economic Analysis

Adam Altmejd Selder

adam.altmejd@su.se

The Institute for Evaluation of Labour Market and Education Policy (IFAU)

March 30, 2026

Software install check (Problem set 0)

Did you…

- Install R?
- Install VS Code and the vscode-R extension?
- Install and configure Git?
- Create an account on GitHub and GitHub Education?

Problem Set 0 is a setup verification step rather than a formality. The course depends on having a working editor, R interpreter, Git installation, and GitHub account early enough that workflow problems do not keep compounding later.

If so...

- You have already edited a file, made a commit, and pushed it
- Today is about understanding that workflow

Why this matters

- Reproducible work is easier to debug
- Clean projects are easier to share
- `Git` gives you history with more than just timestamps

Workflow discipline is mainly about reducing avoidable confusion. When projects are structured clearly and changes are recorded intentionally, mistakes are easier to diagnose and collaboration becomes much less fragile.

Warning signs

- You can't find the file you are looking for
- You're not sure which data is raw and which one is cleaned
  - → Have outliers been removed?
- You have data/script files in `Downloads`, `Desktop`
  - → What project is `summarize_earnings.R` for?
- You have scripts called `final.R`, `final_v2.R`, and `final_final_REALLY.R` - but which is the latest?
- You and your colleagues are working in different versions

These are not cosmetic annoyances. They are early signals that the project lacks a clear source of truth about files, data versions, and analysis steps.

- Project Hygiene
- Short shell intro
- Version Control in Git
- Git Branches
- Git Merge conflicts
- Git Failure Modes
- Extras

Self-contained projects

- Give each project its own folder
- Keep code, documentation, and project-specific outputs together
- Avoid reaching out to random files elsewhere on your computer
- Portable projects are easier to rerun, archive, and share

A self-contained project should be movable as one folder. If the project breaks when copied to another machine or another location on the same machine, the structure is usually too dependent on local accidents.

A sensible project tree

```
municipality-project/
├── README.md
├── .gitignore
├── data/
│   ├── raw/
│   └── processed/
├── src/
└── output/
    ├── figures/
    └── tables/
```

This is a convention rather than a law. The important part is not the exact folder names but the separation of source files, code, and generated outputs.

What goes where?

- `data/raw/`: untouched source files
- `data/processed/`: cleaned or reshaped data created by your code
- `src/`: code/scripts that do the work
- `output/`: things your code produces for humans to inspect or submit
- `README.md`: what the project is and how to run it

The structure makes the project easier to inspect because each folder answers a different question: where the source data came from, what code transforms it, and which outputs are meant for humans.

`src/` is for your code

```
src/
├── 00_run_pipeline.R   | runs the project in the intended order
├── 01_clean_data.R     | cleans and standardizes inputs
├── 02_merge_data.R     | joins tables and checks keys
├── 03_make_figures.R   | produces human-facing output
└── utils.R             | reusable helper functions, if you really need them
```

- Put code in `src/`, not in random top-level files
- Numbers are fine for ordering, but never let them replace descriptive names
- A good test: could another person (or agent) guess where to look for a specific step?

Good script names reduce search costs. A filename such as `02_merge_data.R` communicates both order and purpose, while names like `02.R` or `new_final.R` force the reader to open files just to guess what they do.

Split code by job, not by mood

- Better: one (or several) script for cleaning, merging, figures
- Worse: `final.R`, `misc.R`, `functions.R`, `new_final.R`
- Split by function

Separating data cleaning, merging, analysis, and output generation is a basic form of modularity. When a project is split by job, it is much easier to isolate which step failed and rerun only the relevant part. This is also best practice when working with AI agents, as it allows agents to read only the relevant files for a specific task.

Use relative paths

Many do-files start like this:

```
clear all
cd "C:/my_project/analysis"
use "mydata.dta"
```

or even worse:

```
clear all
use "C:/my_project/analysis/mydata.dta"
```

Don’t do this!

- Absolute paths hard-code your own machine
- Use **relative paths** that travel with the project

Absolute paths hard-code one machine and one folder layout. Relative paths make the project portable because they describe where files are located inside the project rather than on a specific computer.

Use relative paths (cont.)

- The **working directory** is the folder your code is running from
- `fread("mydata.csv")` looks for `mydata.csv` in the working directory

Many path errors are really working-directory errors. If code cannot find a file that clearly exists, the first question is often whether the session is running from the expected project root.

To set working directory: open the folder, not the file

- In `VS Code`, open the project root folder
- Then the file tree, working directory, terminal, and `Git` repo all line up
- You should see your folder and files in the sidebar

Opening the root folder keeps the editor, terminal, file explorer, and Git repository synchronized. Opening a single file in isolation often breaks that alignment and leads to confusing path behavior.

Use relative paths (cont.)

To reach a subfolder, navigate with `/`:

- `source("src/script.R")` runs `script.R` in the `src` folder

Forward slashes are worth using consistently because they work across operating systems in most course examples. That removes one unnecessary source of cross-platform friction.

Use relative paths (cont.)

To reach a parent folder, navigate with `..`:

- `fread("../../data_outside_project.csv")`
- Each `..` navigating one level “up”

But you should never need to do this if you keep your projects self-contained! Why?

Using `..` is sometimes necessary, but frequent parent-directory navigation usually signals that the project is pulling in files from outside its own folder. That makes the workflow harder to move, archive, and reproduce.

Start new sessions

- Don’t put `clear all` or `rm(list = ls())` at the top of scripts
- Instead, make it a habit to always start new sessions!
  - → More robust/reproducible
  - → Forces self-contained code
  - → Avoid bugs from leftover objects and settings

Relying on leftover objects in memory makes scripts fragile because success depends on invisible session state. Starting fresh is a simple way to test whether the code really creates everything it needs.

Raw is raw

- Put untouched source data in `data/raw`
- Do not edit raw data by hand
- Write cleaned outputs to new files
- If you overwrite raw data, you have deleted evidence

For extra points: **write-protect** your raw data. To write-protect in Windows you can right-click on the folder,

Raw data should be treated as evidence, not as a working draft. If a cleaning decision is important, it belongs in code that writes a new processed file rather than in manual edits to the original source.

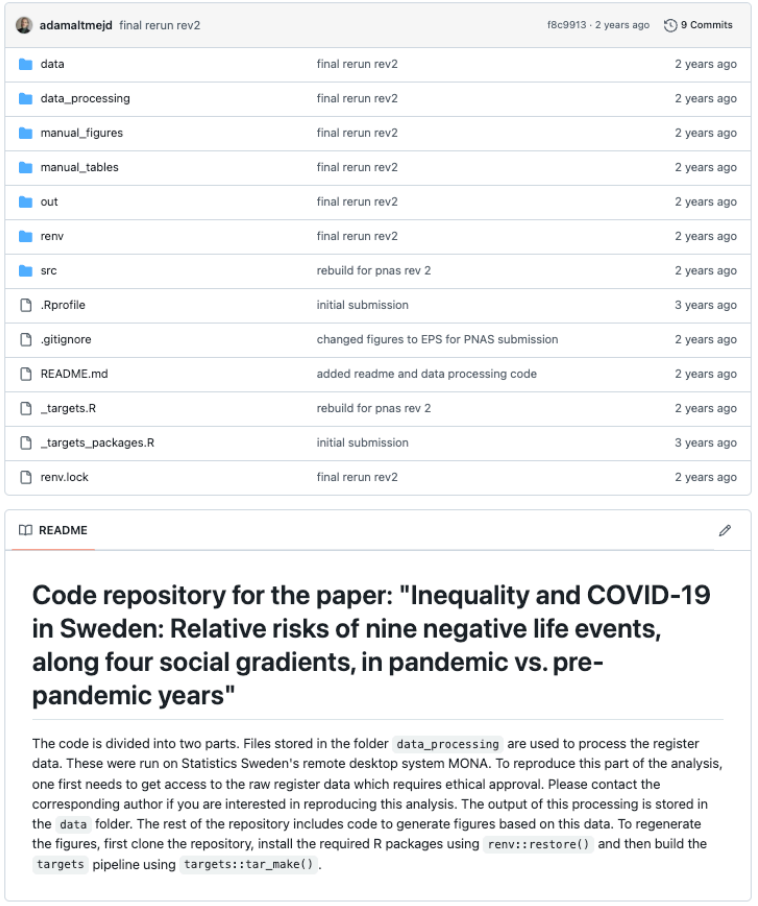
Naming is part of analysis

- Name folders, files, variables so anyone can understand
- Long names are better than abbreviations you will forget
- Figures: `scatter_income_by_gender.png` not `figure_3.png`
- Variables: `log_family_income` not `inc4`

Naming is not cosmetic. Clear names preserve meaning across files, scripts, tables, and figures, which makes later checking and collaboration much easier.

README.md

- Every project root needs one
- What is this project?
- What data does it use?
- How do I run it?
- What output does it create?

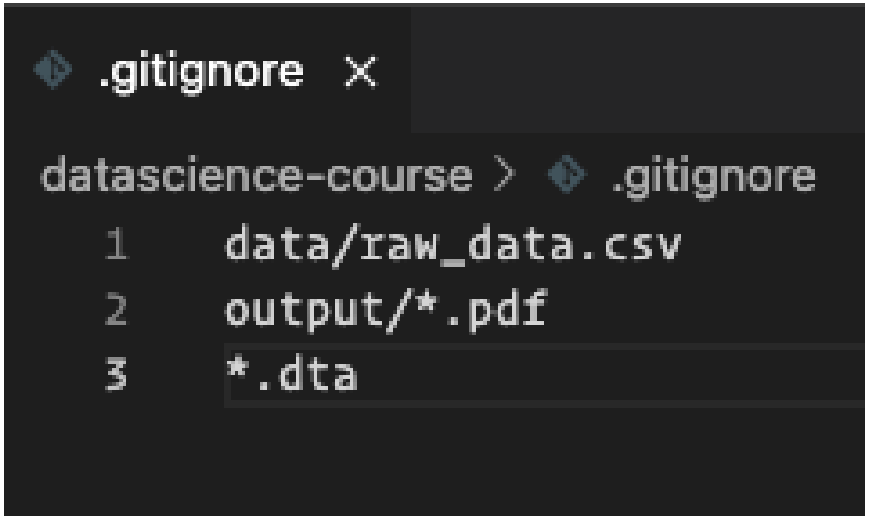


Repo for my paper

A short `README` lowers the entry cost for anyone reopening the project later. It should explain what the project is, what data it uses, and which script or command reproduces the main outputs.

`.gitignore` tells `Git` what to ignore

- `Git` should mainly track code and small text files
- Usually ignore:
  - → Data (especially proprietary/sensitive)
  - → large files
  - → generated output / process artifacts
  - → secrets and `.env` files



Example `.gitignore`

`.gitignore` exists to keep repositories focused on source and logic. Large raw files, generated outputs, credentials, and disposable artifacts usually belong outside version control.

Example `.gitignore` lines

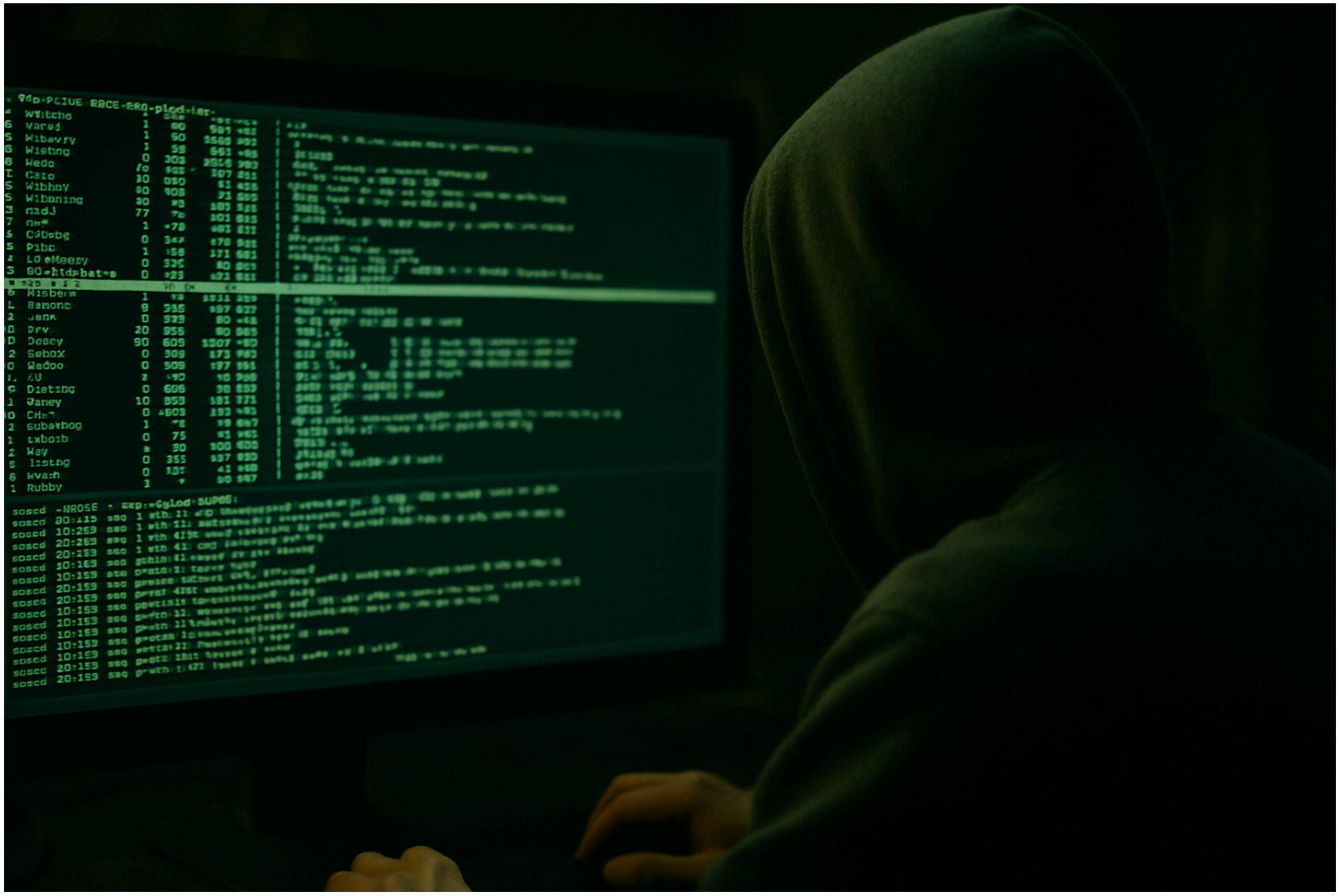
```
data/raw/**
output/**
.env
*.zip
```

- Ignore patterns are just text lines
- Prefer ignoring folders or generated artifacts you can recreate
- Be careful with broad patterns like `*.csv`
- The point is to track source and logic, not every disposable by-product

Ignore rules need some care because an overly broad pattern can hide files that should be tracked. A good rule is to ignore outputs and machine-specific artifacts, not inputs or code that define the analysis.

- Project Hygiene
- **Short shell intro**
- Version Control in Git
- Git Branches
- Git Merge conflicts
- Git Failure Modes
- Extras

The Unix shell



The Unix shell (cont.)

Hackers in movies are often portrayed working in terminals.

In reality, developers use shell commands because they offer efficient ways to interact with computers and files. We will work in a shell originating in the **Unix** family of operating systems.

The **Unix philosophy** is for tools to be "minimalist and modular" to:

> **Do One Thing And Do It Well**

Terminology

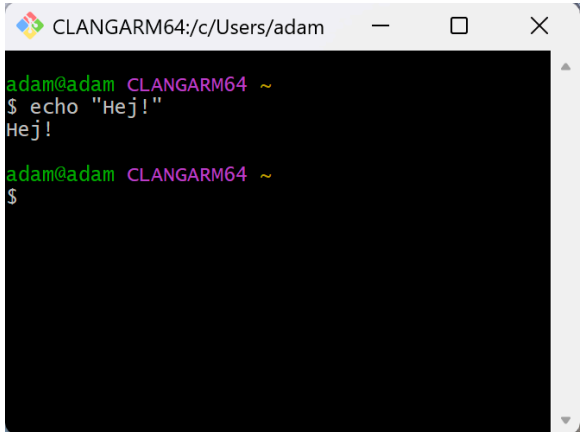
- *Terminal, command prompt, tty*: wrapper that runs the shell
- *Shell, Bash, zsh*: programs that run commands and return output
- *Console*: your computer

We will use **Bash** which is a *shell variant*

- Included by default on Linux and MacOS[1]
- Windows users should install Git Bash

People often use these words interchangeably, which causes confusion. The key distinction is that the terminal is the window or interface, while the shell is the program inside it that reads and runs commands.

The Terminal



Git Bash on Windows

Terminal on Mac

In VS Code:

- Open it from `Terminal -> New Terminal`
- Same project, same folder tree, same repo

Running the terminal inside VS Code keeps command-line work inside the same project context as the files and Git repository. That reduces the chance of working in the wrong folder by accident.

The `shell`

- Mac and Linux are Unix-like OS with Unix-style shells
- Git for Windows provides Bash emulation
- Why?
  - → Almost all servers speak shell
  - → To know what goes on “under the hood”
  - → Automation
  - → Reproducibility (again!)

The shell is useful well beyond programming classes. It is a standard interface for servers, automation, version control, and many command-line tools used in data work.

Things I use the shell for

- Git
- Renaming and moving files
- Installing and updating programs
- Interacting with servers
- Scheduling tasks
- Manipulating large sets of files:

```
find "raw_data" -name "*.csv" -type f -exec \
  perl -i -0pe 's/Vet ej\/\nVill ej svara/Vet ej\/Vill ej svara/g' {} \;
```

The long example is not something to memorize. It an example of how a quick shell command can clean up survey data that R refused to read because of problematic *newline* (`\n`) characters inside the cells. The shell is often the best tool for quick fixes to large numbers of files.

First look

```
username@hostname:~$
```

- `username`: who you are
- `hostname`: which computer you are on
- `~`: your home folder
- `$`: the prompt where you type commands

The prompt gives context before any command is run. Reading it helps answer three useful questions immediately: who is running the command, on which machine, and from which folder.

Path symbols

- `.` = current directory
- `..` = parent directory
- `~` = home directory
- `/` = sub folder
- Relative paths use these symbols instead of hard-coding your whole machine

These symbols are a compact way to describe location without typing long absolute paths. Once they become familiar, navigation and file references get much faster.

Commands have a simple structure

```
command flags argument
```

- command = what to do
- flag = how to do it
- argument = what to do it to

Most shell commands follow this pattern even when the details vary. Reading commands in terms of action, flags/options, and target makes unfamiliar commands easier to parse.

Commands and flags

- Flags always start with one or two dashes
- Example command `ls`:
  - → `-l` = output a list
  - → `-a` = show all, also hidden
  - → `-h` = human readable file sizes
  - → `../` = look in parent folder (`argument`, not a flag)

```
1 ls -lah ../
```

```
total 32K
drwxr-xr-x  6 runner runner 4.0K Apr  1 10:26 .
drwxr-xr-x 12 runner runner 4.0K Apr  1 10:28 ..
-rw-r--r--  1 runner runner  306 Apr  1 10:26 _metadata.yml
-rw-r--r--  1 runner runner  636 Apr  1 10:26 custom.scss
drwxr-xr-x  2 runner runner 4.0K Apr  1 10:28 lecture_1
drwxr-xr-x  2 runner runner 4.0K Apr  1 10:28 lecture_2
drwxr-xr-x  2 runner runner 4.0K Apr  1 10:26 lecture_3
drwxr-xr-x  2 runner runner 4.0K Apr  1 10:26 lecture_4
```

The key reading habit is to separate the command from its modifiers. In `ls -lah ../`, `ls` is the program, `-lah` changes the output format, and `../` tells it where to look.

Seven commands are enough for now

```
pwd
ls
cd path/to/project
find . -name "*.R"
mkdir output/figures
mv notes.txt old_notes.txt
git status
```

- `pwd`: where am I?
- `ls`: what is here?
- `cd`: go somewhere else
- `find`: locate files
- `mkdir`: create a directory
- `mv`: move or rename a file
- `git status`: what changed?

This list is intentionally small. A few navigation and file-management commands cover a large share of beginner shell use, especially when combined with Git.

A tiny navigation example

```
pwd
ls
cd data
ls
cd ..
```

The example shows that shell work is usually incremental: check where the session is, inspect contents, move one step, and inspect again. That rhythm prevents many avoidable mistakes.

Small file tasks

```
mkdir output/figures
cp README.md README_backup.md
mv notes.txt old_notes.txt
```

- `mkdir`: create a directory
- `cp`: copy a file
- `mv`: move or rename a file

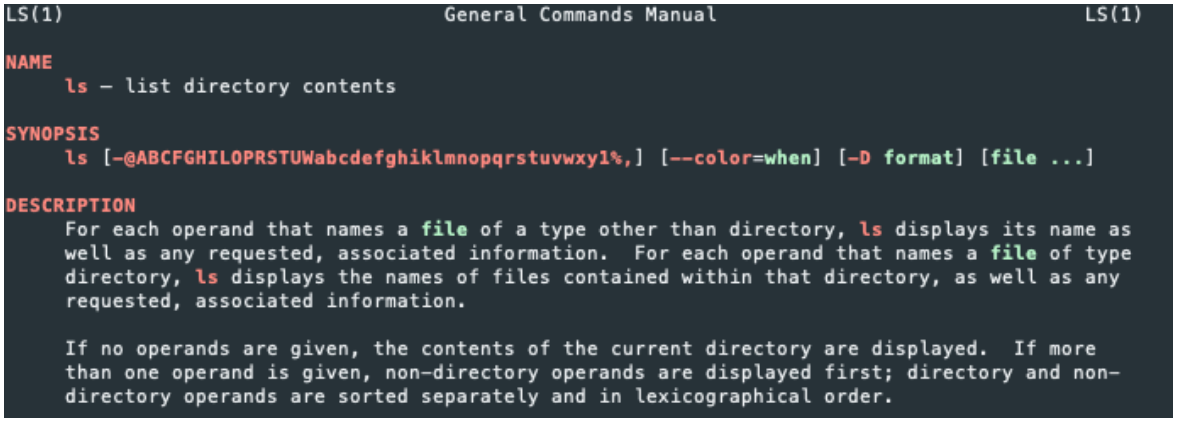
Tab completion is free speed

- Start typing a path, then hit `Tab`
- Use the up-arrow to repeat old commands
- This reduces typos and makes long paths tolerable

Tab completion is one of the fastest ways to reduce command-line mistakes. It also reveals available filenames and folders, which makes it useful as a discovery tool rather than just a typing shortcut.

`man` for manual

```
1 man ls
```



Use `space` to browse, press `h` for help and `q` to quit.

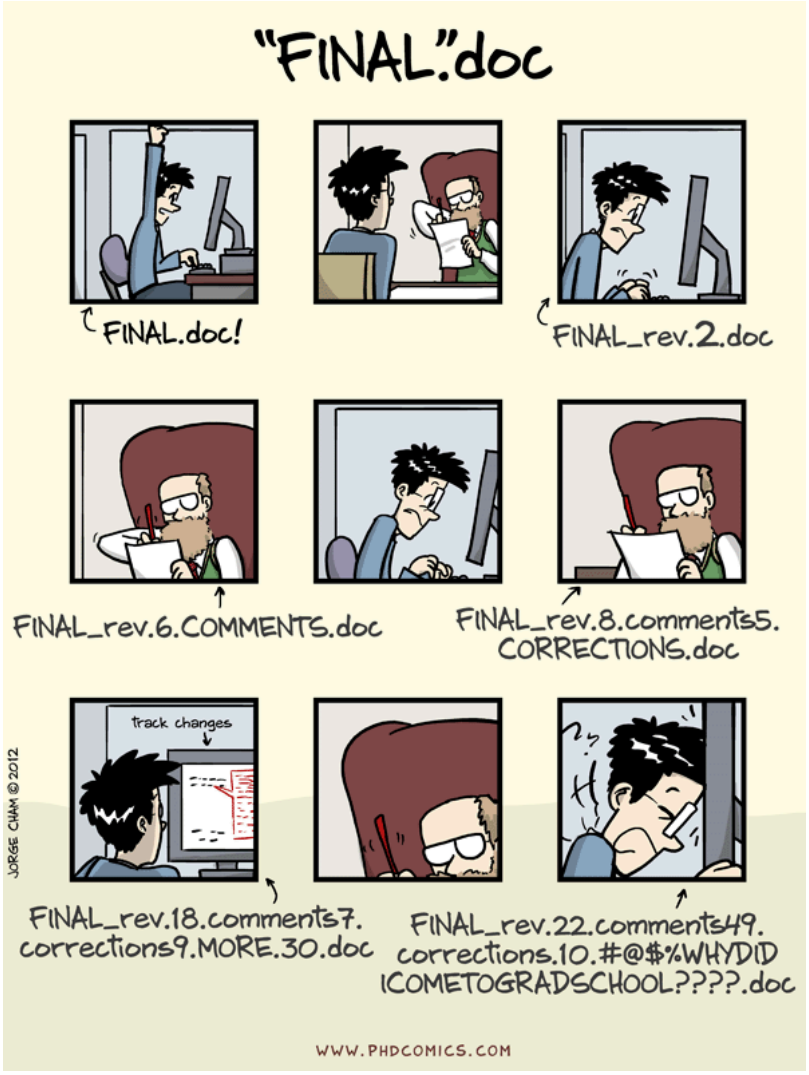
Manual pages are terse, but they are the canonical reference for many shell commands. Knowing that `man` exists is often more important than memorizing specific flags. Another option is cheat which provides more concise examples for common commands but needs to be installed manually.

Short shell intro

- This was just a very basic introduction
- Hopefully the sight of a command line will be a bit less scary
- Resources at the end for those who want more!

- Project Hygiene
- Short shell intro
- **Version Control in Git**
- Git Branches
- Git Merge conflicts
- Git Failure Modes
- Extras

Why version control?



- It records what changed
- It lets you go back
- It makes collaboration less chaotic
- It is much better than emailing `.zip` files around like it is 2007

Version control creates a shared history of what changed, when it changed, and why. That is useful for solo work as well as collaboration because it provides an audit trail and a recovery mechanism.

`Git`, `GitHub`, `VS Code`

- `Git`: the version-control system on your computer
- `GitHub`: a remote home for repositories—uses `Git`
- `VS Code`: the editor that lets you work with files and `Git` in one place
- Keep the roles separate in your head and life gets easier

`Git`, GitHub, and VS Code solve different problems. `Git` manages local history, GitHub hosts shared repositories and collaboration workflows, and VS Code is just the editing environment that exposes those tools in one interface.

How does Git work?

- A project = a Git repository
- Each user works in their own *local* copy
- Labelled history with **commits**
- **Branches** enable multi-feature development
- Changes synchronized to GitHub on demand (unlike Dropbox)
- Facilitates collaboration with branching, conflict management, pull requests

Any ordinary folder can become a Git repository. Git then starts tracking changes to files in that folder, provided those changes are staged and committed.

A repository is just a folder with memory

- Your working tree = files as they look right now
- The staging area = what will go into the next commit
- The commit history = saved snapshots with messages
- The remote = the GitHub copy

The staging area is the part people often miss at first. It exists so that the next commit can be assembled intentionally rather than automatically including every changed file.

The four operations to remember

- `git add`: stage changes for the next snapshot
- `git commit`: record staged changes in your local history
- `git pull`: bring remote changes down to your machine
- `git push`: send your local commits up to GitHub

These four commands cover the minimum viable Git workflow for the course. Most later commands are refinements or extensions of these same basic ideas.

The basic cycle

1. Edit files
2. Check `git status`
3. Stage what belongs in the next commit
4. Commit with a useful message
5. Pull if the remote may have changed
6. Push to GitHub

The order matters because Git separates editing, staging, committing, and synchronization. Thinking in that sequence reduces confusion when the repository state does not match expectations.

The core commands

```
git status
git add README.md src/01_build_panel.R
git commit -m "Add first cleaning step"
git pull
git push
```

The commit message is part of the record. A useful message explains the change in a way that helps later readers understand what was done.

Stage and commit in two steps, why?

- `git commit` just commits staged files
- We use `git add <file>` to chose what to commit
  - → Can even `git add --patch <file>` to stage parts
- This allows to divide up changes into multiple commits based on features/components

Separating staging from committing makes the history cleaner. Instead of recording one large mixed snapshot, changes can be grouped into smaller commits that each represent one coherent idea.

Pull and push are not the same

- `git pull` brings remote changes to you
- `git push` sends your committed changes to the remote

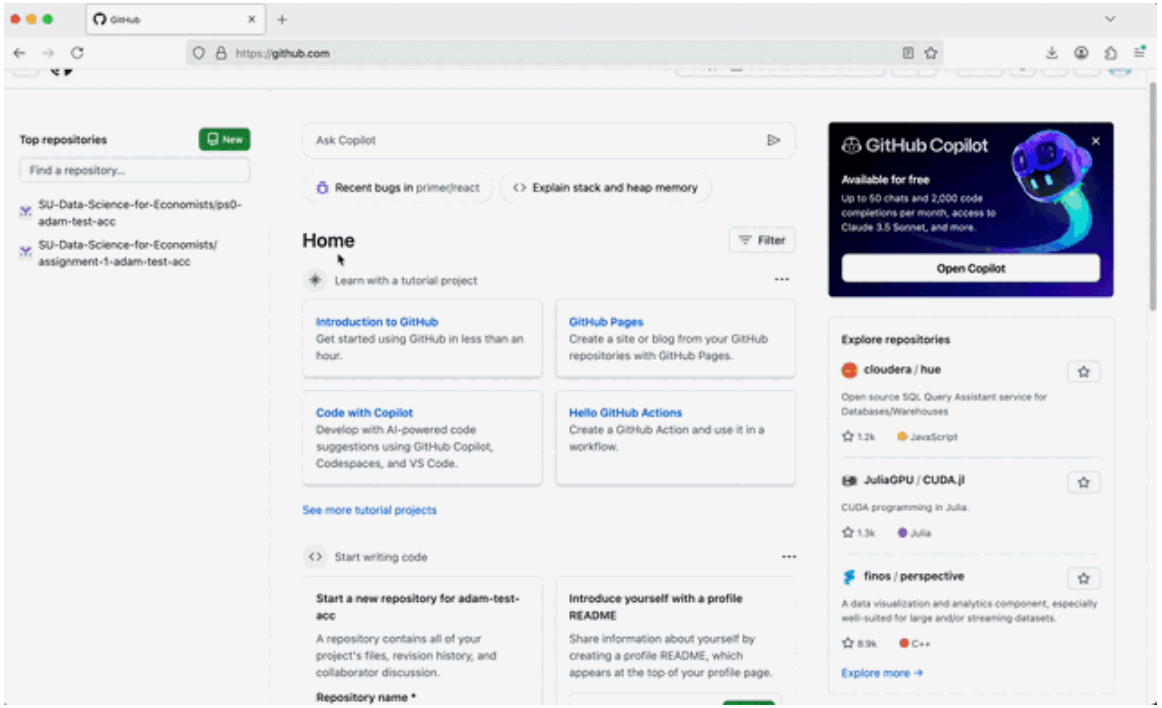
If you work with others, or across multiple computers, pulling before pushing is a good habit. On a quiet solo

`git status` is your friend

- Am I in a repo?
- What changed?
- What is staged?
- Am I ahead or behind?
- When confused, start here

`git status` is often the best first diagnostic because it shows the current state without changing anything. Many Git problems become easier once the repository state is visible.

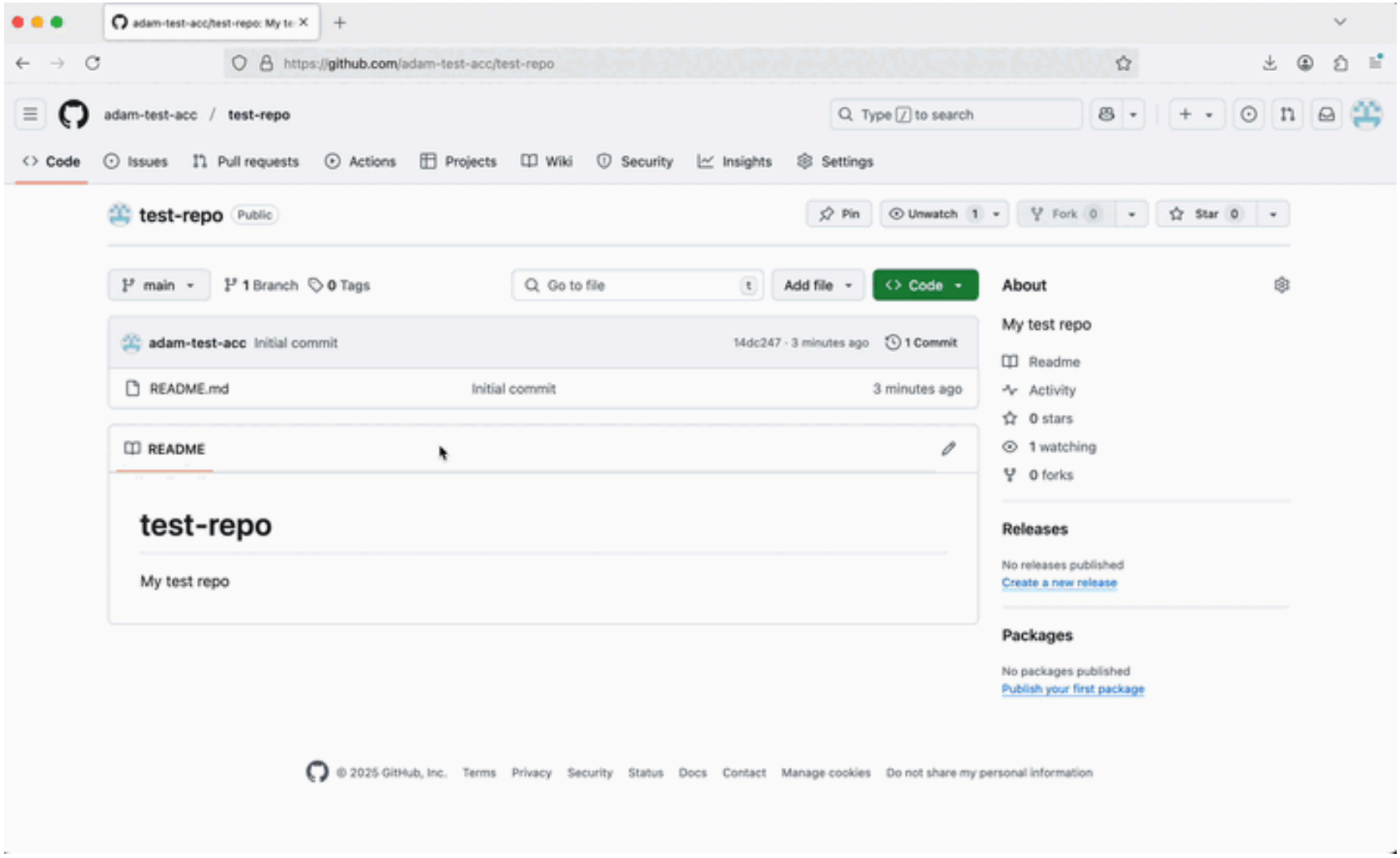
Initialize a repository (on GitHub)




```
1 git init test-repo
```

Starting from GitHub is often simpler for beginners because the remote is configured from the start. That avoids an extra round of setup around linking a local repository to its remote counterpart.

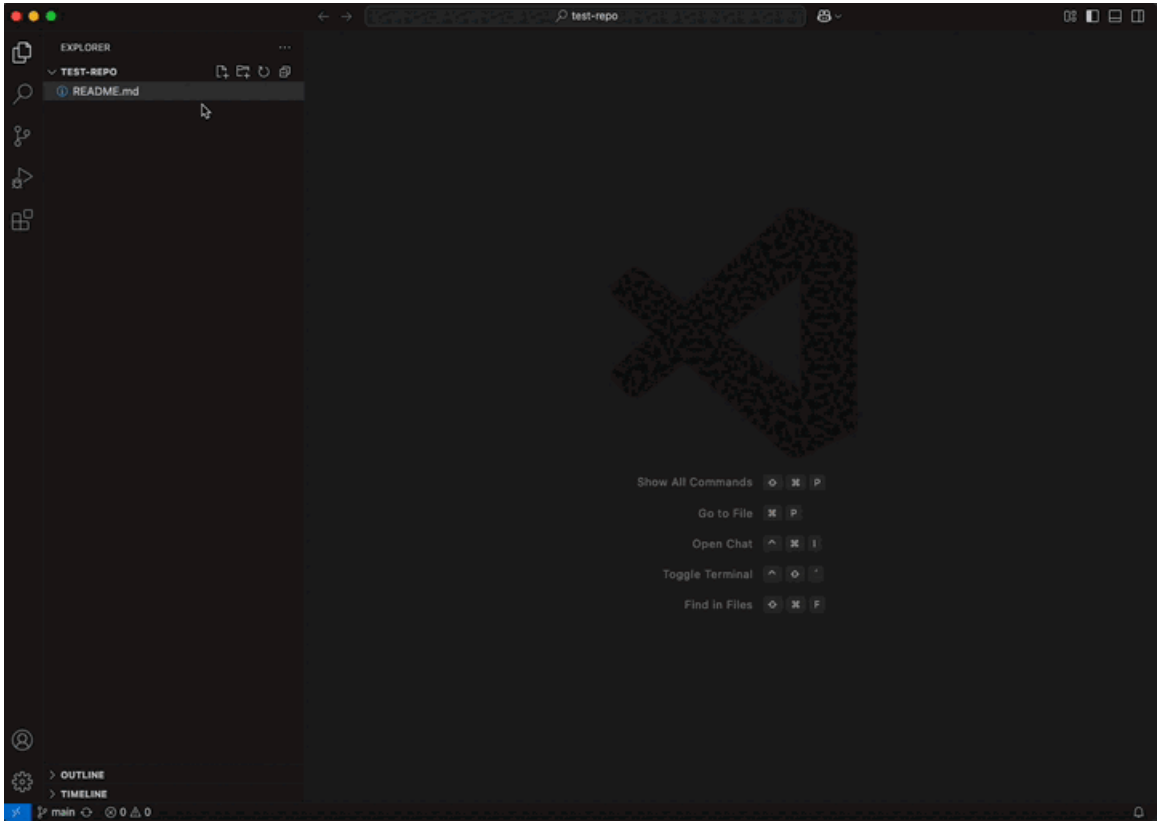
Clone a repo in VS Code



Clone or open an existing repo

Cloning creates a complete local copy of the repository, including its history. After that, edits happen locally and are synchronized with GitHub through pull and push.

Stage, commit, and push changes




```
git add README.md
git commit -m "Readme change"
git push
```

Follow the instructions in Problem set 0 to set up your Git `user.name` and

Git records author identity inside each commit. That is why `user.name` and `user.email` must be configured before the first real commit can be created cleanly.

Good commits are small and legible

- Bad: `update`, `changes`, `stuff`
- Better: `Add README setup section`
- Better: `Clean municipality names before merge`
- Commit early and often
- Commit when one small idea is done

Small commits are easier to review, understand, and reverse. A commit history full of vague labels such as `update` or `stuff` quickly loses its value as documentation.

`Git` is **not backup**

- `Git` tracks changes
- It does not protect you from deleting the wrong folder
- It does not make giant raw files pleasant to manage
- Keep a real backup too (Box/Dropbox, external drive, etc.)

Git is optimized for versioned source files, not for general disaster recovery. Repositories still need ordinary backups, especially when projects depend on large raw data or local machine state.

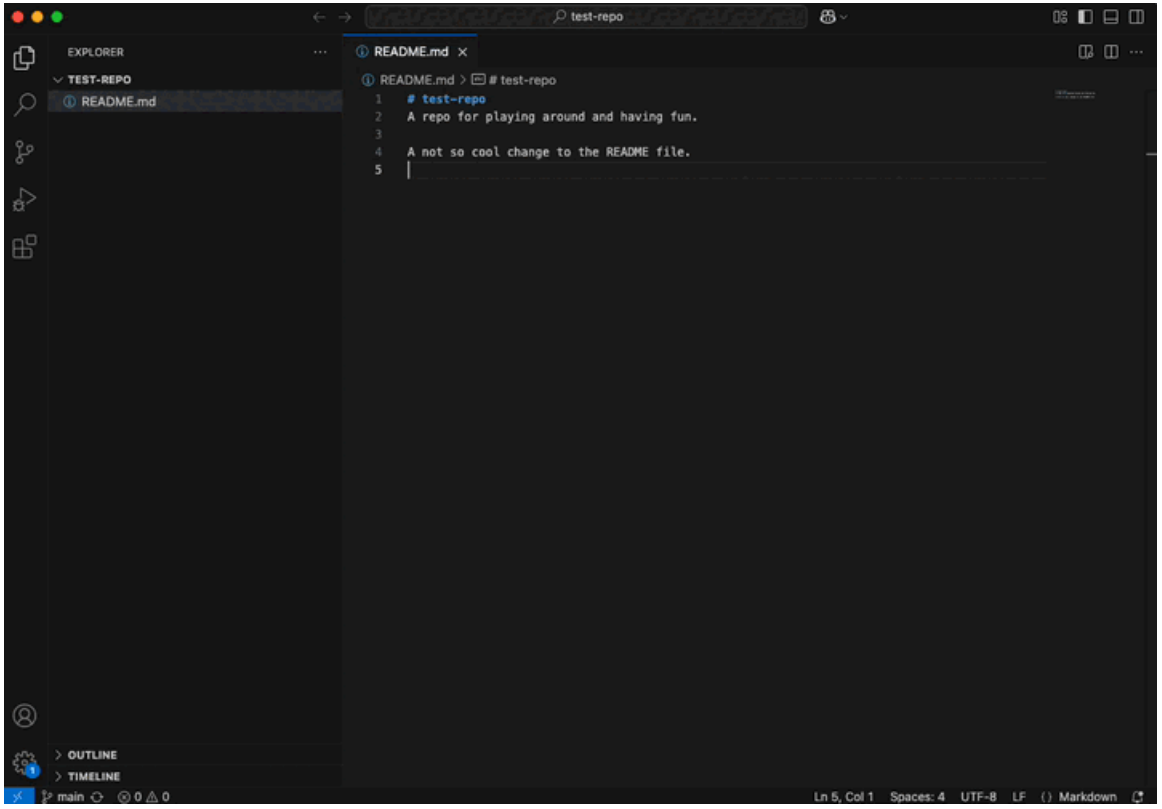
- Project Hygiene
- Short shell intro
- Version Control in Git
- **Git Branches**
- Git Merge conflicts
- Git Failure Modes
- Extras

Branches

- Want to test a large change, but unsure it will work?
- Create a new branch to try it out, then just revert to main if it fails.
- Keep track of your changes (with commits) without disturbing your collaborators with unfinished code.
- If you are happy with your changes, merge them to the main branch.
- Branching is awesome, use it!

Branches create a safe place for experiments, revisions, or features that are not ready for the main line of work. They are useful in research as well as software development because they keep unfinished ideas separate from the stable version.

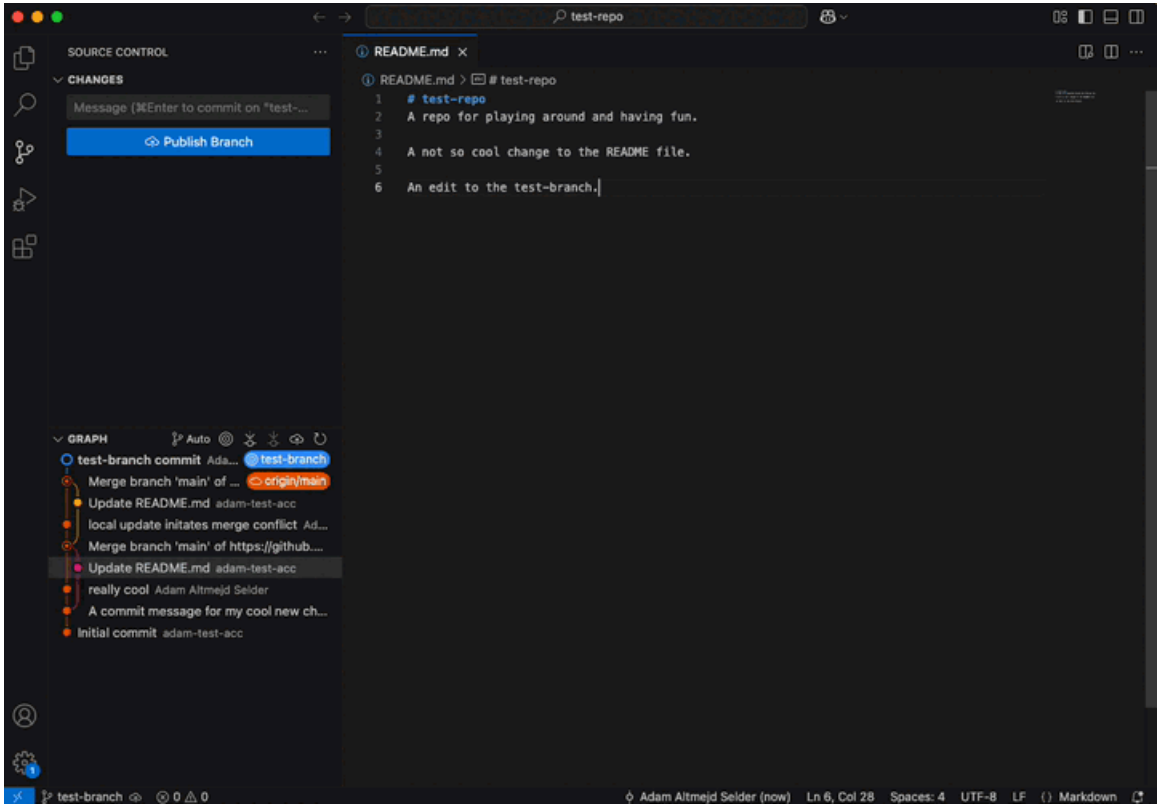
Creating a new branch in VS Code




```
git switch --create test-branch
```

Creating a branch copies the current state into a new line of development. From that point on, commits on the branch do not affect `main` until the branch is merged.

Merging branches locally

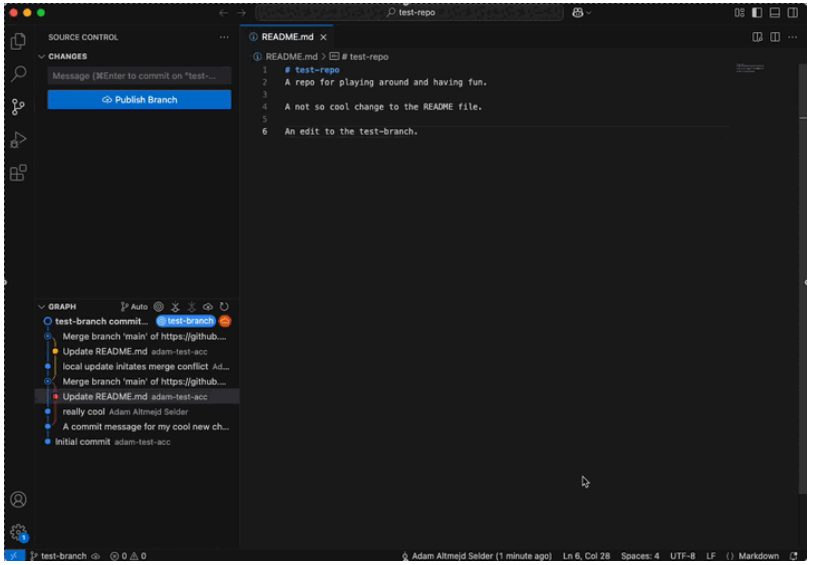



```
1 git switch main
2 git merge test-branch
```

Local merging is the direct way to combine branches when no review step is needed. In collaborative work, the same merge is often done through a pull request instead.

Pull requests (GitHub feature)

- A pull request is a proposal to merge one branch into another
- Useful in team projects (code review), overkill for solo work



Pull request flow

A pull request is mainly a review and discussion tool built around a proposed merge. It should really be called a `merge request`. It adds context, comments, and checks before the branch is integrated into the target branch.

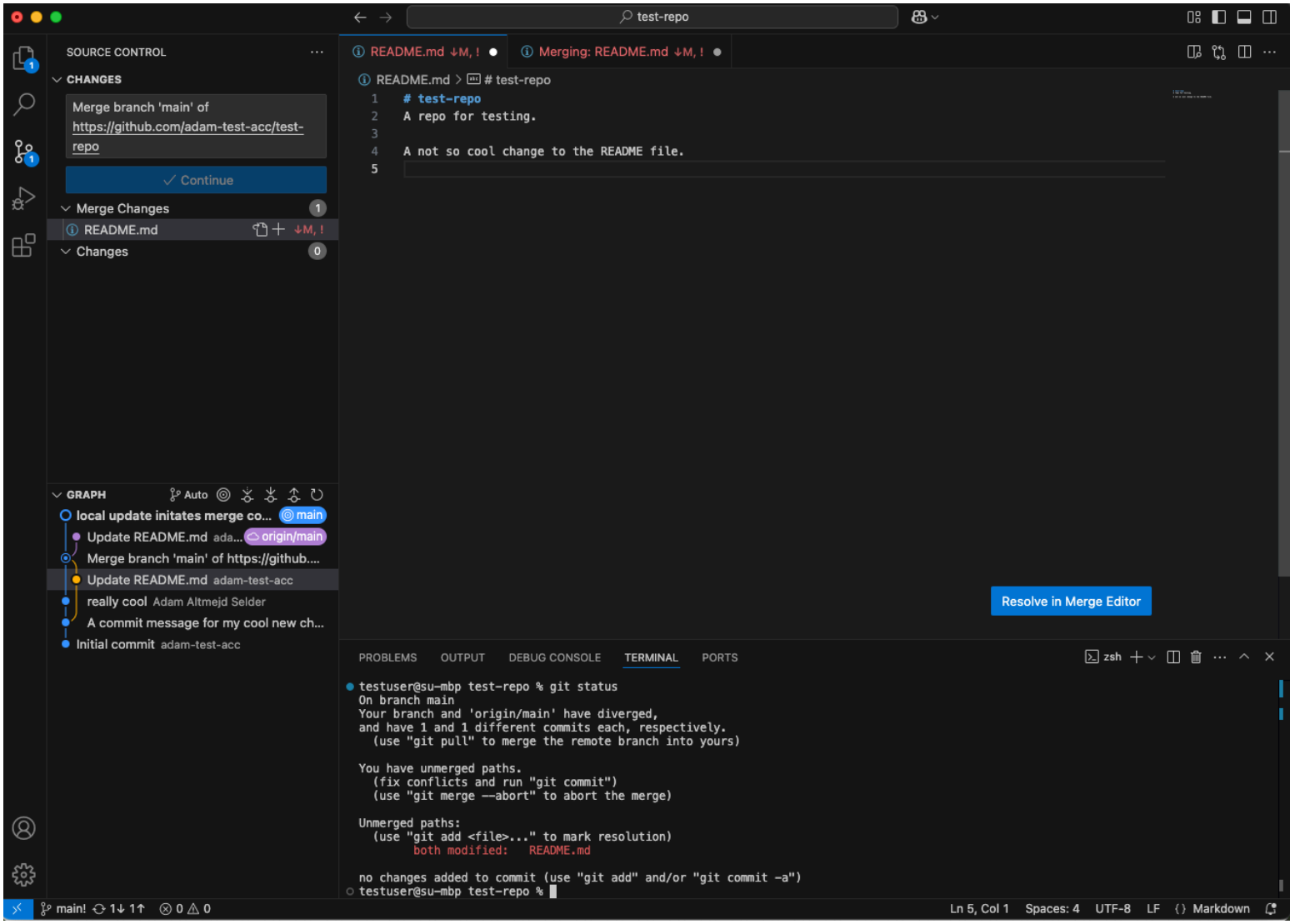
- Project Hygiene
- Short shell intro
- Version Control in Git
- Git Branches
- **Git Merge conflicts**
- Git Failure Modes
- Extras

Merge conflicts

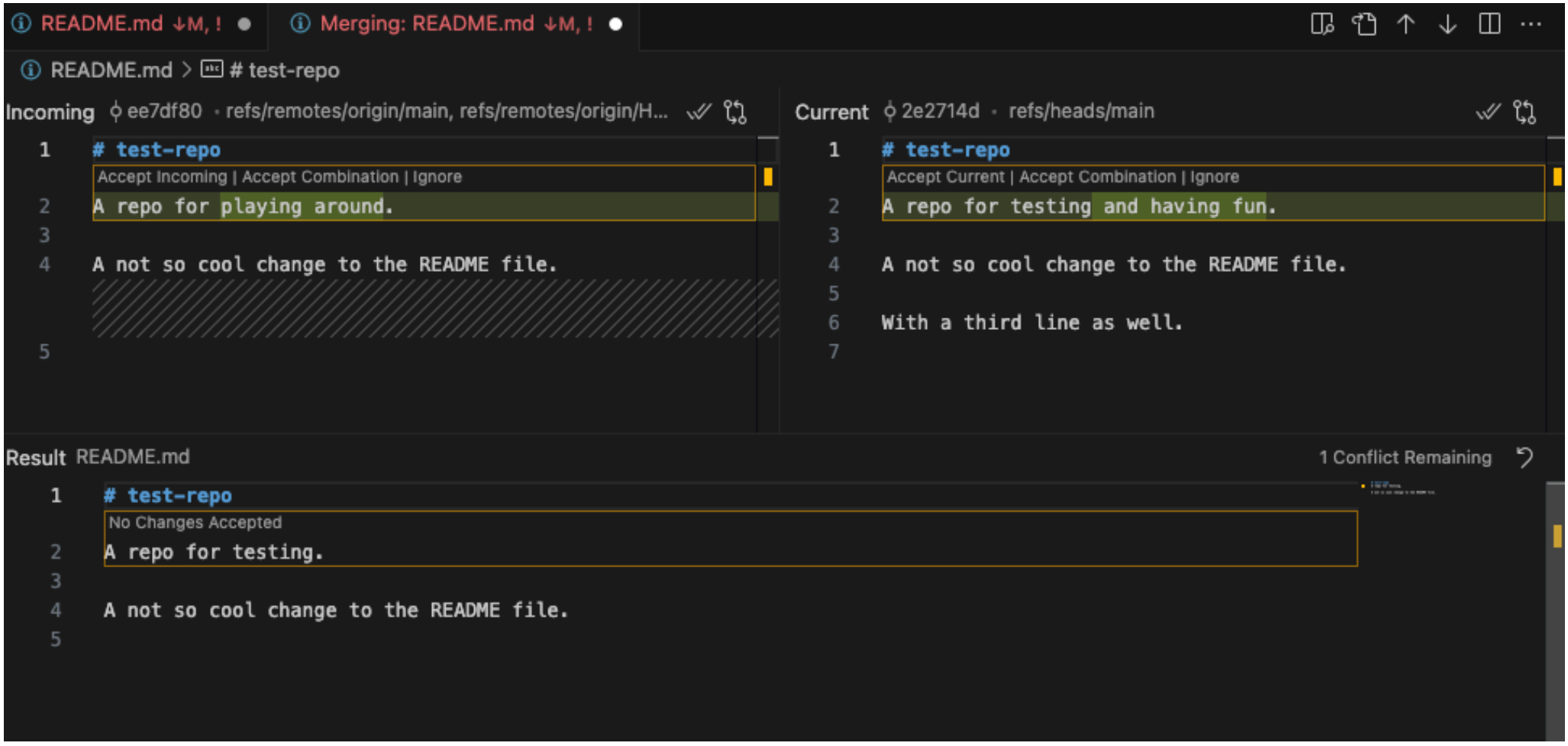
- A conflict happens when `Git` sees competing edits
- `Git` stops because it does not want to guess
- Annoying, yes
- Still better than silent overwriting

Conflicts happen when two lines of development change the same part of a file in incompatible ways. Git pauses because choosing automatically could destroy one side of the work.

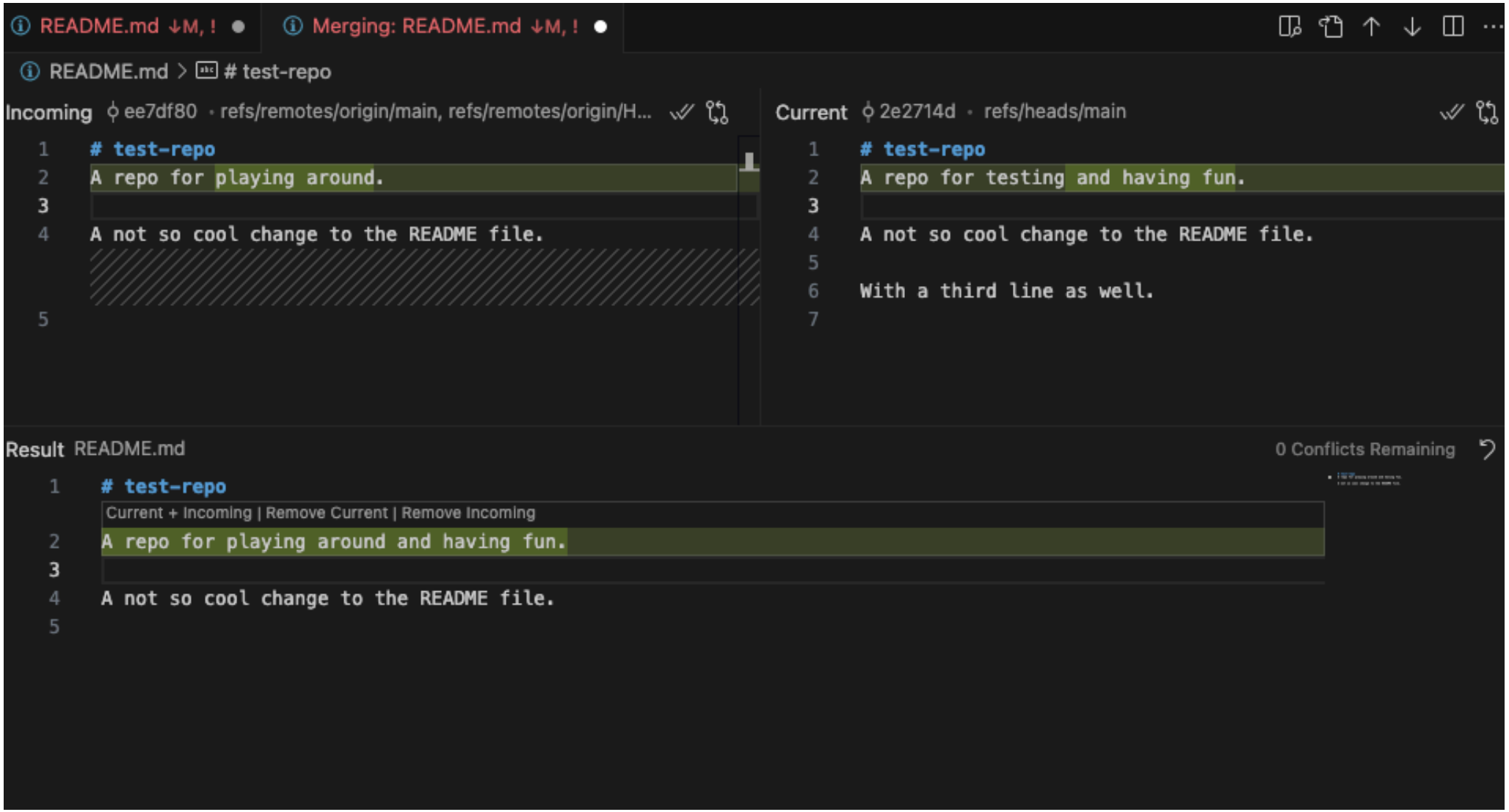
Merge conflicts in VS Code



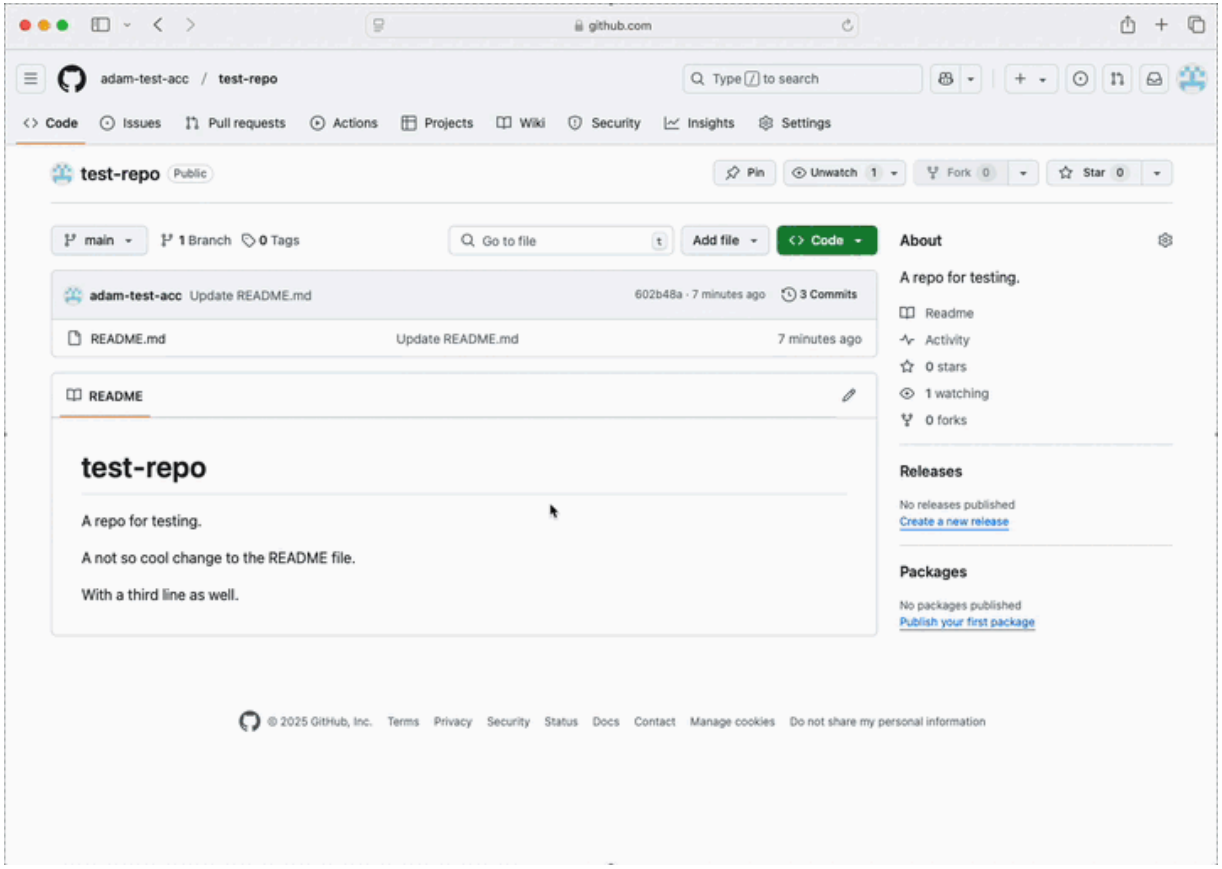
Resolving conflicts using the merge editor



Resolving conflicts using the merge editor



Example: Code change by multiple sources



Two contributors have made changes to the same file. Git stops merge to

Conflict resolution is a judgment call about which version, or which combination of versions, should survive. Smaller commits and clearer review workflows usually make that judgment easier.

Manual edits

VS Code provides a great UI for resolving merge conflicts, but you can also do it manually by editing the conflicting files. If you open the file you will see something like this. The strange characters are Git's way of highlighting the merge conflict.

```
# test-repo
<<<<<<< HEAD
A repo for testing and having fun.
=======
A repo for playing around.
>>>>>>> 814e09178910383c128045ce67a58c9c1df3f558.
A not so cool change to the README file.
```

You fix the conflict manually by simply removing the characters and chosing the text you prefer.

The conflict markers show the competing versions inline: the current branch above the separator and the incoming branch below it. Resolving the conflict means editing the file into the exact final text and then removing the markers.

You do not need all of `Git` today

- Need now: repo, `status`, `add`, `commit`, `push`
- Useful soon: `pull`, branches, pull requests, conflict awareness
- Useful later: `SSH`, rebasing, tags, more careful collaboration workflows

- Project Hygiene
- Short shell intro
- Version Control in Git
- Git Branches
- Git Merge conflicts
- **Git Failure Modes**
- Extras

Boring failure modes

- Opened the wrong folder
- Edited a file outside the repo
- Forgot to save before committing
- Tried to commit generated junk
- Saw an error and started guessing

First recovery habits

- Read the error message
- Check what folder you are in
- Run `git status`
- Look at the file tree
- Change one thing at a time

If `Git` says “nothing to commit”

- Maybe you changed nothing
- Maybe you changed a file outside the repo
- Maybe the file is ignored by `.gitignore`
- Maybe you forgot to save

This message usually means the edit never reached the repository in a way Git can see, either because the wrong file changed or because the change was not saved or is ignored.

If push is rejected

- Usually authentication failed, or the remote moved
- Read the message before clicking random buttons
- Do not delete the `.git` folder
- Do not start over unless you understand why

Push rejection usually means either a permissions problem or that the remote branch has new commits that the local branch does not yet include. The error message normally says which of those cases it is.

If you get stuck

- Stop making random changes
- Figure out what state the project is in
- Ask for help before you dig deeper
- Most workflow problems are fixable if you stop early enough

Version-control problems get worse when several speculative fixes are stacked on top of each other. Freezing the state and inspecting it carefully is usually the fastest route back.

What I want you to be able to do after today

- Organize a project sanely
- Use relative paths
- Open a repo in `VS Code`
- Make and explain a commit
- Push work without panic
- Understand what branches, pull requests, merge conflicts are for

Git advice

- Commit often, work in small features
  - → Don't: "New data processing script"
  - → Do: "Removed duplicates from survey data"
- Push changes when you want your collaborators to see
- Git is not backup!
- Branches are useful even when solo

When all else fails... 🤷

Next Time: R Basics

- Running code
- Objects and vectors
- Missing values
- Functions
- Reading simple errors without immediately spiraling

- Project Hygiene
- Short shell intro
- Version Control in Git
- Git Branches
- Git Merge conflicts
- Git Failure Modes
- **Extras**

SSH=Secure shell

- Logs in to server over remote channel
- Really useful but setup requires some work
- Uses *public key cryptography*
  - → Private key - stored on your computer
  - → Public key - stored on the server
  - → When connecting the private key encrypts a message that can only be validated by the corresponding public key

SSH and GitHub

- SSH is another way to authenticate with GitHub
- HTTPS is fine for beginners
- SSH becomes convenient if you use Git a lot

Generating a key pair

To generate an SSH key pair and register the public key with GitHub, follow these steps. For detailed instructions, see GitHub Docs.

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

Press Enter to accept the default file location. When prompted, enter a secure passphrase (your terminal will not show characters as you type).

Storing the passphrase using ssh-agent

You should secure your keys with a passphrase. But it might become annoying having to type the passphrase each time you use the SSH key. Instead, you can use a `ssh-agent` to store the passphrase for you. On Mac/Linux this is easy:

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

On Windows, it's more complicated. I've included the instructions in Problem Set 0 but let's go through them together now.

Connecting to Github over SSH

To use SSH with Github you first need to add your public key to your **GitHub profile**. Go to settings and “SSH and GPG keys” to add it.

Afterwards, run:

```
#| error: TRUE
ssh -T git@github.com
```

Now you can clone repositories using SSH rather than https.

We will get back to how to use SSH to connect to servers later in the course.

Resources

- When Git is not working: https://ohshitgit.com
- Practicing shell commands: https://cmdchallenge.com
- Everything you need about Git+R: https://happygitwithr.com
- Basic shell https://swcarpentry.github.io/shell-novice
- Intermediate shell https://seankross.com/the-unix-workbench
- Advanced shell https://jeroenjanssens.com/dsatcl