

CS 45, Lecture 8

Version Control

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Winter 2023

Outline

Contents

1	Review	1
2	Version Control	2
2.1	Version Control Systems	2
2.2	Comparison of VCSs	3
3	Git	4
3.1	Linear History	4
3.2	Branching Workflow	5
3.3	Combining Branches	5
3.4	When/How to Commit	11
4	GitHub	13

1 Review

Computer Networks

Last lecture, we saw:

- How computers can use a network to talk to each other
- How information gets sent from one place to another over the internet

In this lecture, we will see:

- How to safely store your files (code or text)
- How to collaboratively on files with others over the internet
- *How to avoid losing all your homework!*

Files

- Many of the files you work with will be text:
 - Source Code
 - Documentation

– Markup Files

- As you change these files over time, you’ll eventually want some way to keep track of different “versions” of the file.
- What we need is a “version control system”.

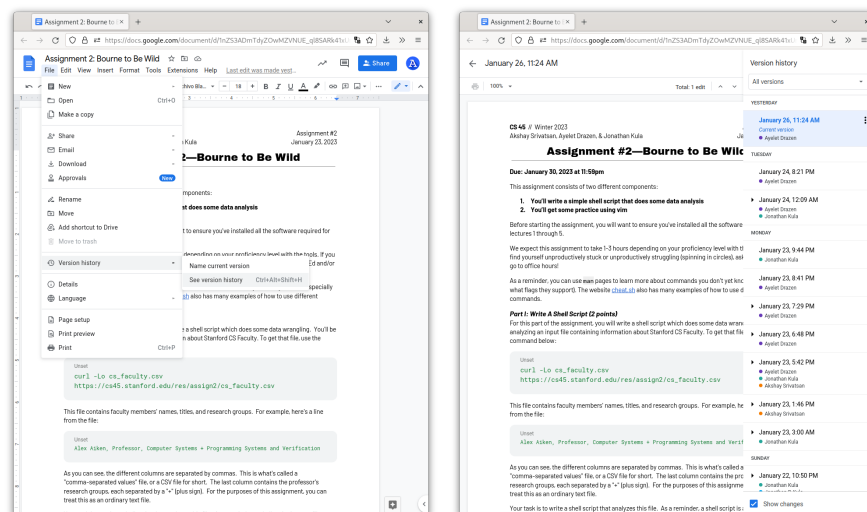
2 Version Control

2.1 Version Control Systems

Version Control Systems

- A VERSION CONTROL SYSTEM (VCS) is a piece of software which manages different versions of your files and folders for you.
- A good VCS will let you look at old versions of files and restore files (or information) which you might have accidentally deleted.
- You’ve seen these before!

Version Control Systems



Version Control Systems

A good version control system:

- Will store many versions of your files
- Will let you “revert” a file (or a part of a file) to an older version
- Will track the order of different versions
- Will ensure each “version” is neither too big nor too small

A great version control system:

- Will let you collaborate on files with other people
- Will combine the different versions of the files produced by different people sanely

2.2 Comparison of VCSs

There are many different ways to set up a version control system. Let's see the pros and cons of some of the more common ones.

Google Docs

Google Docs automatically keeps track of file history in a basic VCS.

Pros:

- Great for rich text
- Allows real-time collaboration
- Saved on the cloud automatically¹

Cons:

- Bad for plain text (especially code)
- Requires an internet connection
- Only supports a single “current” version of a single file

You might wonder why you'd ever want multiple “current” versions of a file. One example is when we are planning out this class—we have multiple ideas for each part of an assignment, and try to flesh them out enough to see what works and what doesn't. It would be nice if we could have multiple, equally-valid, versions of the assignment at the same time, but right now the only way to do that is to copy the file.

Copying Files

You can make a bunch of copies of files or folders with `cp` as a simple form of version control. You can compare versions with `diff`.

Pros:

- Works on either rich or plain text (or anything else)
- It's simple and makes it easy to move data between versions

Cons:

- It's messy and a lot of manual work
- It's hard to tell what the relationship between different versions is
- It takes a lot of hard drive space

Zip Files

Instead of just `cp` ing folders, we could bundle them up into a Zip file (a single file which can be “unzipped” into a folder).

Pros:

- Tracks versions for an entire folder at once
- Easy to share a version with someone else (email)

Cons:

¹Stay tuned for our lecture on the cloud!

- It's still a lot of manual work
- It's hard to tell what the relationship between different versions is
- It's hard to extract a single file from an old version

Zip Files++

- What if we had a tool which did all this zip file stuff automatically?
- We could tell it to take a “snapshot” of a directory, and it would save all the changes in it.²
- We could ask it to recover an old version of a specific file, or to reset everything to an old version to “undo” our work.
- The tool could track the relationships between different versions, so we can have multiple “current” versions at the same time.
- If we want to combine different versions, the tool can automatically do it for us (instead of us copying and pasting the parts together).

Git

`git` is a version control system which tracks “commits” (snapshots) of files in a REPOSITORY.

- Git stores old versions of files in a hidden folder (`.git`), and automatically manages them.
- We can tell Git to keep track of certain files, and tell it when to take a snapshot.
- We can ask Git to go back to an old snapshot (even for a single file).
- We can ask Git to keep track of who's working on what, so multiple people can work on different things without conflicting.
- If we want to combine multiple people's work, we can ask Git to automatically merge them together. If it can't for some reason, it'll ask us to manually merge them.

There are several other version control systems in use, but Git is by far the most popular. Different companies sometimes decide to use different VCSes for various reasons; for example, Facebook/Meta uses a version of Mercurial (another VCS) they call “Sapling”. However, these different VCSes are conceptually very similar, so moving from one to another is pretty easy.

3 Git

3.1 Linear History

Basic Workflow

The simplest way to use git is the “linear” workflow, which is the same way you'd use Google Docs:

1. `git init` to enable Git in a certain directory
2. `git add` any files you want Git to “track”
3. `git commit` the currently “staged” changes to save a snapshot
4. make changes to your files
5. `git add` the changed files to “stage” them again

²Or, even better, we could tell it *which* files to save in the snapshot. Everything else stays as it was in the previous snapshot.

6. Repeat from 3

You can use `git log` to see your commit history, and use `git status` to see the current state of staged/unstaged/untracked changes.

Before you do anything in Git, you should tell Git who you are:

```
git config --global user.name "<YOUR FULL NAME>"
git config --global user.email "<YOUR PUBLIC EMAIL ADDRESS>"
```

Note that these configuration values will be permanently baked into any commits you make, so other people may be able to see them. Use a pseudonym or a pseudonymous email address if you don't want to publish your information; see the GitHub Docs for more info.

Basic Workflow

Demo

Let's practice how to:

- Create a new Git repository
- Commit a new file
- Commit changes to files
- Revert commits
- Look at an old version of a file
- Compare two versions of files
- See your commit history

3.2 Branching Workflow

Branching Workflow

We can also split our “repo” into multiple BRANCHES, which are like alternate versions of a folder. This means different people can work on different things without interfering with one another.

1. Make sure your repository is “clean” (i.e., you have no uncommitted changes).
2. `git checkout -b <branch>` to create a new branch and move to it; at this point, the new branch will be identical to the old one.
3. Make changes, `git add`, `git commit` as usual
4. `git checkout` to switch between branches

3.3 Combining Branches

Branching Workflow

Combining Branches

Now that we have multiple branches, we probably want to join them back together at some point.

There are several ways to do this:

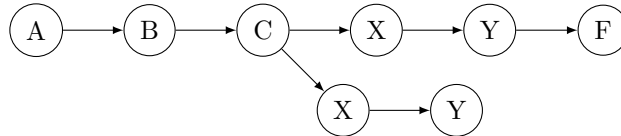
- `git merge` two branches into one
- `git merge --fast-forward` a long branch onto a shorter version of itself

- `git rebase` one branch onto another branch
- `git cherry-pick` a specific commit from one branch to another

Branching Workflow

Fast Forwarding

The simplest case of MERGING is called FAST-FORWARDING.

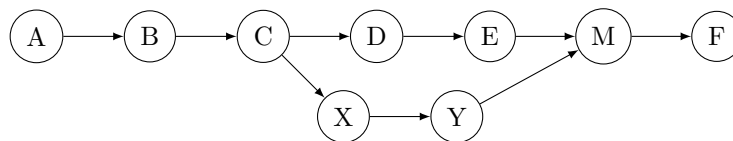


Assume there was a main branch containing A and B. We then created a new branch based on B, containing X and Y. Merging the main branch and our new branch is easy—our new branch is just a longer version of the main branch, so we can “fast-forward” the main branch to catch up. We can then commit F on the main branch, based on Y.

Branching Workflow

Merging

MERGING (in general) creates a MERGE COMMIT to join the two branches.

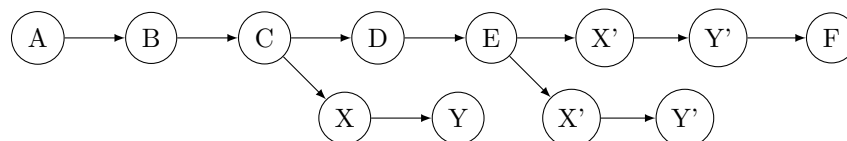


In this case, assume as before that we had a main branch containing A and B. We created a new branch and added X and Y. However, while we were doing that, someone else added D and E to the main branch. Our branch has now DIVERGED from the main branch; we both have different, possibly contradictory ideas of what the “real” state is. We fix this by creating a MERGE COMMIT M, which contains a combination of E and Y; if E and Y were contradictory, Git will ask the user to resolve the MERGE CONFLICT by picking which version to keep. Now we’re back to a single “real” version M, so we can add F on the end.

Branching Workflow

Rebasing

REBASING moves the “base” of a branch to be a different commit. *REBASING edits Git’s history to make FAST-FORWARDING possible.*



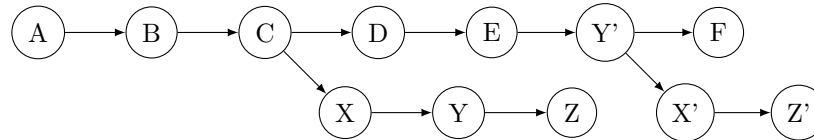
Again assume that we had a main branch containing A and B, from which we created a new branch and added X and Y. Once again, someone else added D and E to the main branch while we weren’t looking. However, this time, we want to maintain the illusion of “linear history”; the idea that every version has exactly one predecessor and one successor. This is obviously not true in this case—the commit C has two successors—so we “rebase” our branch. This creates a new commit X’ which combines X and E, and a new commit Y’ which combines Y and X’. However, if you look at the Git history, you never see a reference to the original X or Y; for all practical purposes, X and Y never existed. Now we’re back to the fast-forward case; our branch is a longer version of `main`, so we can fast-forward `main` to include X’ and Y’. Now we can

commit a new F based on Y'; if anyone in the future ever looks at the repository, it'll look like everything up to F was a linear, non-branching, sequence of changes.

Branching Workflow

Cherry-Picking

CHERRY-PICKING copies a *single commit* from one branch to another branch. *CHERRY-PICKING and rebasing is a good way to move a single commit from one branch to another.*



Sometimes we want to grab a specific change from another branch, without merging or rebasing the entire branch. This is a use case for “cherry-picking”, selecting a specific commit to copy into another branch. In this case, assume a similar setup to the previous cases—a main branch containing A, B, C, D, and E, and a new branch containing A, B, X, and Y, and Z. We’re on the main branch at E, and we realize we need something that was added in commit Y, but we **don’t** want to include commit X or commit Z. We can’t literally just copy Y onto our main branch, since it depends on X and doesn’t include anything from D or E, so we create a new commit Y’. Y’ is like a mashup of Y and E, but without any of the stuff from X. Now we can commit F based on Y’, and F will contain no references to X or Z.

Note that this can get weird if you later do a merge—Y and Y’ will both be in your history! If you cherry-pick commits like this, you probably want to do a rebase to make your history less confusing; in this case we could rebase the new branch onto Y’, creating a new X’ and Z’, which would effectively erase the original X, Y, and Z from the new branch.

Branching Workflow

When to merge/rebase/cherry-pick?

- **fast-forward** when possible (`git merge --ff-only`).
- **rebase and then fast-forward** if possible, i.e., if you’re the only one working on the branch; **never** rebase a branch other people are using (`git rebase` and `git merge --ff-only`).
- **merge** if neither of the above are possible (`git merge`).
- **cherry-pick** if you want to copy a specific commit to another branch (`git cherry-pick`)³.

Some projects insist upon having linear history in `main`, which means any merge commits will be rejected. In this case, you should first `rebase` your changes onto the most recent `main`, then `merge --ff-only` to fast-forward `main` to include your changes. Most projects are okay with merge commits, so you can just use `merge` and forget that `rebase` ever existed.

Branching Workflow

Branching Demo

Let’s practice how to:

- Split our repository into two branches
- Switch between branches
- Make commits on either branch

³This is pretty rare, I’ve only used it a handful of times.

- Merge two branches together

Merge Conflicts

Definition 3.1 (merge conflict). A merge conflict is what happens when you try to combine two contradictory branches. Git can't always figure out how to resolve the contradiction, so it'll ask the user (you).

- Git normally resolves merge conflicts automatically.
- Some conflicts have multiple valid resolutions (e.g., what if one person edited a file that another person deleted?).
- If Git doesn't know what to do, it'll ask you to resolve the conflict.

Merge Conflicts

Git will tell you which files conflicted, and tell you to resolve the commits and commit the results:

```
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Merge Conflicts

Conflict Markers

Git will also add conflict markers to the files:

```
Hello, my name is Akshay Srivatsan.
<<<<<<< HEAD
I'm doing my PhD in the Stanford CS department.
=====
I am a PhD student studying CS at Stanford.
>>>>>>> add-major
I'm currently co-teaching CS45 and doing research.
```

This might look scary, but it's not that bad!

Merge Conflicts

Conflict Markers: The Base Branch

The top part (labeled `HEAD`) are the changes in the base branch (the branch you're currently on):

```
Hello, my name is Akshay Srivatsan.
<<<<<<< HEAD
I'm doing my PhD in the Stanford CS department.
=====
I am a PhD student studying CS at Stanford.
>>>>>>> add-major
I'm currently co-teaching CS45 and doing research.
```


Merge Conflicts

Conflict Markers: The Incoming Branch

The top part (labeled with a branch name) are the changes in the other (incoming) branch:

```
Hello, my name is Akshay Srivatsan.
<<<<<<< HEAD
I'm doing my PhD in the Stanford CS department.
=====
I am a PhD student studying CS at Stanford.
>>>>>>> add-major
I'm currently co-teaching CS45 and doing research.
```

When you see these conflict markers, all you have to do is make the files look the way you want them to look at the end. In this case, I added the text “I’m doing my PhD in the Stanford CS department.” on `main`, but I added the text “I am a PhD student studying CS at Stanford.” on the branch `add-major`. When I tried to merge `add-major` into `main`, Git didn’t know what to do, so it’s asking me. Now I can choose either of the two sentences to keep, and delete the other (or I could keep both of them, if I wanted).

As an aside, you might see the name `HEAD` pop up in Git. This basically just means “what commit you’re currently looking at”.

Merge Conflicts

Resolving a Conflict

Pick how you want to resolve the conflict (i.e., decide what the “correct” result of the merge is), and make the file look that way!

```
Hello, my name is Akshay Srivatsan.
I'm a PhD student in the Stanford CS department.
I'm currently co-teaching CS45 and doing research.
```

In this case, I mixed together both versions. The “correct” answer depends on what you’re doing, which is why Git can’t figure it out for you.

Merge Conflicts

Committing the Merge

Resolve all the conflicts in all the files however you want, then:

1. `git add` your changes to track them
2. `git commit` the changes (with no message)

Git will auto-generate a message, and open your `$EDITOR` to have you confirm it:

```
Merge branch 'add-major'
```

Save the file in your editor and close it (`:wq` in Vim), and Git will save the merge commit. That’s it—the merge conflict is gone!

That’s all you have to do—make the files look “correct”, then commit! A merge conflict really isn’t as bad as people sometimes make it sound; all it means is that there are multiple ways to merge the two branches, and Git wants you to pick one.

If you decide to use `rebase` instead, the process is pretty much the same—just run `git rebase --continue` instead of `git commit` at the end.

Merge Conflicts

Rebase Conflicts

Resolve all the conflicts in all the files however you want, then:

1. `git add` your changes to tell Git you fixed them
2. `git rebase --continue`

Since rebasing doesn't create a merge commit, you don't have to run `git commit`; use `git rebase --continue` instead!

Resolving Merge Conflicts

To resolve a merge conflict:

1. Don't panic!
2. Look at the files in conflict (run `git status` to see what's going on).
3. Fix each conflict, one-by-one.
4. When you're done, `git add` all the fixed files and `git commit`.

Let's practice!

Merge conflicts usually happen in shared repos, so let's CLONE one of my repos onto your computer:

```
git clone https://github.com/Akshay-Srivatsan/cs45-23win-demo-repo.git
```

We'll go into more detail about how shared repositories work in the last section of this lecture, but for now:

- You can “clone” a shared repository using `git clone`, which makes a local copy.
- You can “fetch” commits from the shared repository into yours using `git fetch`. The commits will go into a separate branch so they don't conflict with yours; by convention, the branch names have the prefix “origin/” prepended to them, so `main` goes into `origin/main`.

Pulling Changes

You might have seen references to the `git pull` command before. This is a combination of two commands, but the exact two depends on your Git version and configuration:

```
git pull --ff-only: git fetch and git merge --ff-only (Default)
```

```
git pull --no-rebase: git fetch and git merge (Old Default)
```

```
git pull --rebase git fetch and git rebase
```

Depending on your preferences, you can configure `git pull` to do any of these.

I personally use `git pull --rebase` the most often, since I don't like having merge commits in my repo history.

3.4 When/How to Commit

Commit Messages

Git only saves work that we've committed, so we want to commit as often as possible, but...

If you run `git log`, you can see the commit history of your repo.

Your commit messages in the history should be short and specific, but descriptive enough that someone new can understand what they do.

Similarly, each of your commits should do a single thing, so a single message can describe it easily.

Good commits are BISECTABLE; you should be able to checkout any commit in `main` and get a valid state of your repo.

Writing good commit messages is part of being a good programmer!

Squashing Commits

We can commit often locally but still have meaningful commits in the end by SQUASHING commits together with INTERACTIVE REBASE.

Editing History

Interactive rebasing edits history! Don't do this on a branch you share with other people (like `main`). In general, only do this on commits you **have not** pushed. Otherwise, you'll have to FORCE-PUSH (`git push --force`) your changes, which will **destroy** everyone else's changes.

You can start an interactive rebase using the command `git rebase --interactive <base>`; for example, `git rebase --interactive main` will let you edit every commit that's in your branch but not in `main`.

Interactive Rebasing

Git will open \$EDITOR with a list of actions (which you can edit!).

```
pick 0cd3296 start working on new file
pick 594a80c continue working
pick 162392b almost done
pick bf45520 done
pick c545ae9 oops, had a bug
pick 9b3d056 fix the bug for real this time
```

Each line represents one commit. The first word is a “command”; `pick` keeps the commit, `reword` lets you edit the commit message, `edit` lets you change the commit contents, `squash` and `fixup` both squash the commit into the previous one, and `drop` removes the commit completely.

Squash and Fixup Commits

Squash commits let you specify that two commits are closely related, so they should be combined into a single commit with both messages.

Fixup commits let you specify that a particular commit just “fixes” a previous one, and therefore should be absorbed into the previous commit.

```
reword 0cd3296 start working on new file
squash 594a80c continue working
squash 162392b almost done
squash bf45520 done
squash c545ae9 oops, had a bug
squash 9b3d056 fix the bug for real this time
```

Note that we could also use `fixup` here, the only difference is whether the original message gets saved or thrown away. In this case, we're using `reword` on the first commit anyway, so it's a moot point.

Rewording Commits

When you want to `reword` a commit, Git will open `$EDITOR` and ask you for a new commit message. Enter the message you want, save, and quit.

```
Add a file providing more information about the project

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Fri Feb 3 22:34:02 2023 -0800
```

Amending Commits

If you want to edit the commit you just made, you can skip the rebase and just AMEND it, using `git commit --amend`. This will also let you edit the commit message of the last commit.

If you want to amend an earlier commit but don't want to do the full interactive rebase yet, you can use `git commit --fixup <hash>` to mark a commit as being a fixup commit of an earlier commit.

You can then use `git rebase --interactive --autosquash <base>` to automatically absorb your fixup commits into the original commits.

Again, only do this if **no one else** is using your branch.

Good Commit Messages

From Stanford Stagecast: Proleptic:

```
5a64e14    2022-12-05  Add jaccard similarity index [Akshay Srivatsan]
565f686    2022-11-30  Add constant metronome to piano roll split-ear program [Akshay
↪  Srivatsan]
f4554ad    2022-11-30  Add full-piano split-ear program [Akshay Srivatsan]
75dd1f7    2022-11-30  Add MIDI-to-piano program [Keith Winstein]
62ba612    2022-11-30  Initial synth benchmark [yasmitch]
42df5fd    2022-11-30  midi-recorder.cc: print time in decimal [Keith Winstein]
d8478b0    2022-11-30  midi-recorder.cc: pop event each time (and flush output each
↪  line) [Keith Winstein]
```

These commit messages are descriptive and make it clear what each commit does. They're also bisectable; that means, if I notice a bug, I can *binary search* to figure out which commit introduced the bug. Git actually has a tool for this built-in, called `git bisect`—you give it a start and end commit (the start

commit definitely doesn't have the bug, and the end commit definitely does), and it'll checkout commits in between to help you figure out where the bug was introduced.

Bad Commit Messages

From CS 140E, Winter 2023:

```
bd8771f    2023-02-02  minor [engler]
88452fa    2023-02-02  minor [engler]
504175b    2023-02-02  minor [engler]
d190a50    2023-02-02  Merge branch 'main' of github.com:dddrreee/cs140e-23win into
↳ main more [engler]
f59cf98    2023-02-02  fixed [engler]
1552052    2023-02-02  minor [Akshay Srivatsan]
5860c92    2023-02-02  minor [engler]
f269ea4    2023-02-02  Merge branch 'main' of github.com:dddrreee/cs140e-23win into
↳ main dd [engler]
89d3dc9    2023-02-02  minor [engler]
63b1f52    2023-02-02  minor [Akshay Srivatsan]
```

These commit messages are, honestly, useless. What does “minor” mean? What was “fixed”? If we find a bug, how are we supposed to figure out where it came from? Why are there random merge commits?

4 GitHub

GitHub

- One of the most common reasons to use Git is to be able to collaborate.
- Git has built-in support for REMOTE repos, which exist on the internet somewhere.
- You CLONE a remote repo to get a local copy. You can then make commits on the local repo. The remote repo is conventionally named `origin`.
- You FETCH while inside a clone, which copies the remote `main` branch into a branch called `origin/main`.
- You MERGE or REBASE your changes to `main` into `origin/main`.
- You PUSH your new `origin/main` back to the remote, which updates its `main`.

You can actually have multiple remote repos for a single local repo. For example, you might have `origin` as your copy of the repo on GitHub, and `upstream` as someone else's copy of the same repo from which you want to cherry-pick changes.

GitHub Demo

Let's create a new repository on GitHub!

You'll need the `git` command and the GitHub CLI.

1. Go to <https://github.com/new> and pick a name.
2. Click “Create repository” to continue.
3. Run `git clone` with the URL of your new repo.

4. Run `gh auth login` from inside your new clone. Tell `gh` that you want to use it to authenticate with `git`.
5. Make some changes (add a file), and run `git push` to upload them!

GitHub Demo

Let's start collaborating!

1. On the GitHub website for your repo, go to "Settings" and click on "Collaborators".
2. Add the person sitting next to you as a collaborator!
3. Make a clone of their repo, make some changes, then commit and push them. Use `git fetch` or `git pull` to download their changes to your repo.
4. What happens if you both try to edit the same file at the same time?
5. Can you push a new branch to your partner's repo?⁴

Pull Requests

- It's dangerous to give access to the `main` branch on your repo to everyone; someone might start messing with it!
- In "Settings/Branches", you can enable BRANCH PROTECTION for `main`. Specifically, you can enable "Require a pull request before merging".
- A PULL REQUEST⁵ is a way to review a change before merging it. You (the repo owner/maintainer) can choose whether to approve or reject the request.
- To create a pull request: create a new branch, make your changes, push your new branch, then run `gh pr create`.

When to use Git

- When you want to look at past versions of a folder.
- When you want to be safe from accidentally overwriting your work.
- When you want to collaborate with other people asynchronously (use GitHub!).
- When you want to keep a backup copy of a folder with full history (use GitHub!).
- You want to "fork" a project already using Git/GitHub and contribute back to it.

I personally use Git pretty much whenever I write code, and even sometimes when I'm writing prose. Even my lecture slides for this class are tracked in Git... and lecture 6 only happened because Git saved my slides from accidental deletion.

I actually use Git so often that I have a bunch of aliases, both in my shell and in Git itself, to make using it faster. From my `.zshrc`:

```
alias ga="git add"
alias gc="git commit"
alias gc!="git commit --amend"
alias gcmgs="git commit --message"
```

⁴Hint: you might have to use the `--set-upstream` flag; Git will tell you exactly what to do.

⁵This is misleadingly named, it's really a "merge request"

```
alias gca="git commit --all"
alias gcam="git commit --all --message"
alias gca!="git commit --all --amend"
alias gp="git push"
alias gf="git fetch"
alias gfm="git pull --no-rebase"
alias gfr="git pull --rebase"
alias gff="git merge --ff-only"
alias gst="git status"
alias gr="git rebase"
alias gri="git rebase --interactive"
alias glog="git log"
alias gco="git checkout"
alias gb="git branch"
```

And from my `.gitconfig`:

```
[alias]
  co = checkout
  c = commit
  st = status
  b = branch
  hist = log --pretty=format:@"%Cred%h%x09%Cgreen%cs%x09%Creset%s%x20%Cblue[%an]%Creset"
  uncommit = reset --soft HEAD~
  amend = commit --amend
  histedit = rebase -i origin/main
  unstash = stash pop
  unadd = restore --staged
  skip = update-index --skip-worktree
  unskip = update-index --no-skip-worktree
  skipped = ! git ls-files -v | grep '^S' | cut -d' ' -f2
  list = ls-files -v
  ff = merge --ff-only
  delete-remote-branch = push origin --delete
```

I wouldn't recommend copying all of these, since some of them are particular to my use case, but they might give you ideas of ways you can make your own Git usage more convenient. A git alias can be run as a git subcommand, so I can run `git unadd hello.txt` instead of `git restore --staged hello.txt`.