

Version Control, Git, and GitHub

Ryan M. Richard

Ames National Laboratory and Iowa State University

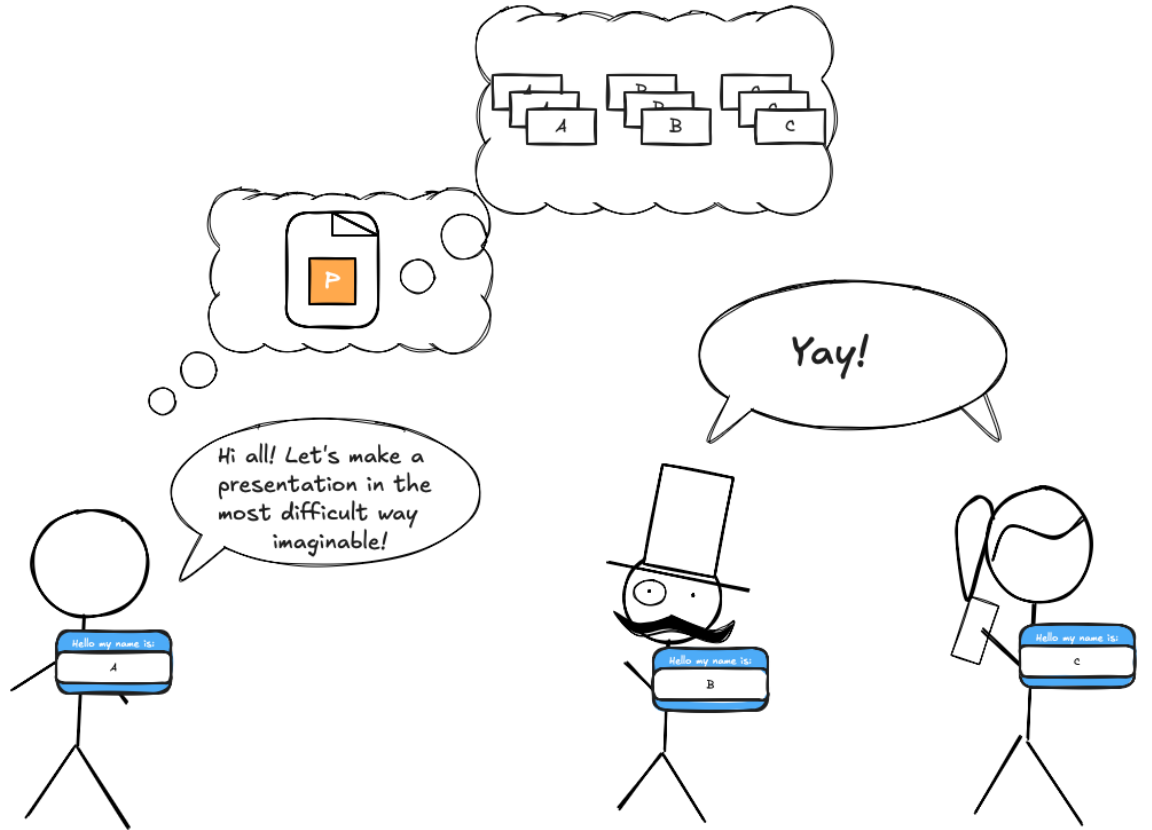
Topics

- Why do we need version control (VC)?
- What is the relationship among VC, Git, and GitHub?
- What do all the Git/GitHub terms mean?
- What is the typical Git/GitHub Workflow?

Why Do We Need Version Control?

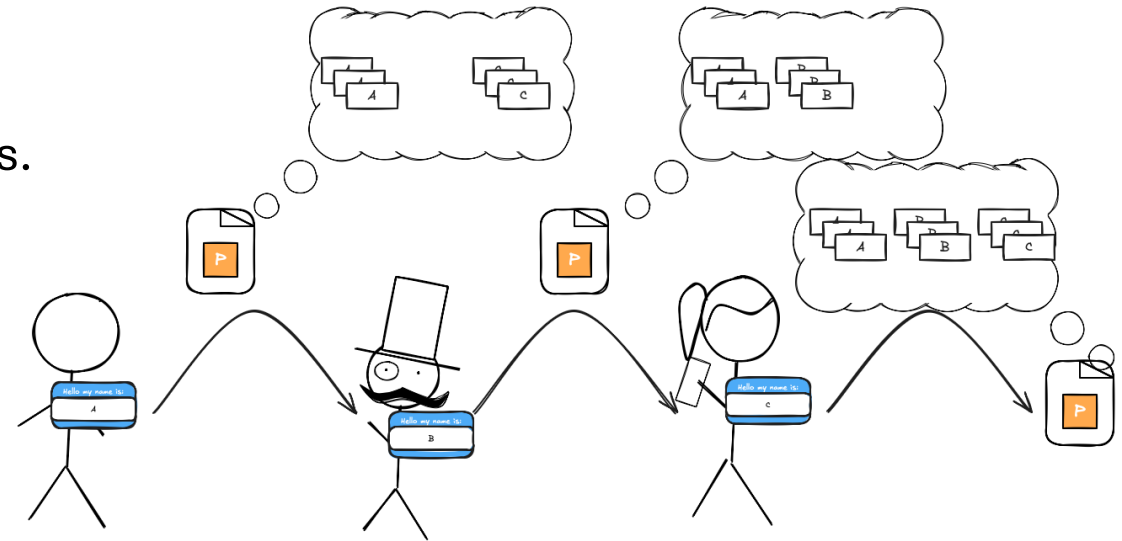
Motivation

- Imagine three people are trying to write this presentation.
- We'll assume their parents were unoriginal and named them A, B, and C.
- We'll also assume these people haven't heard of Google Slides or Office 365 and are trying to do this via email.
- We'll assume the slides are supposed to be ordered so that A's slides come first, B's come second, and C's third.
- Let's work through some scenarios in which the presentation could end up coming together.



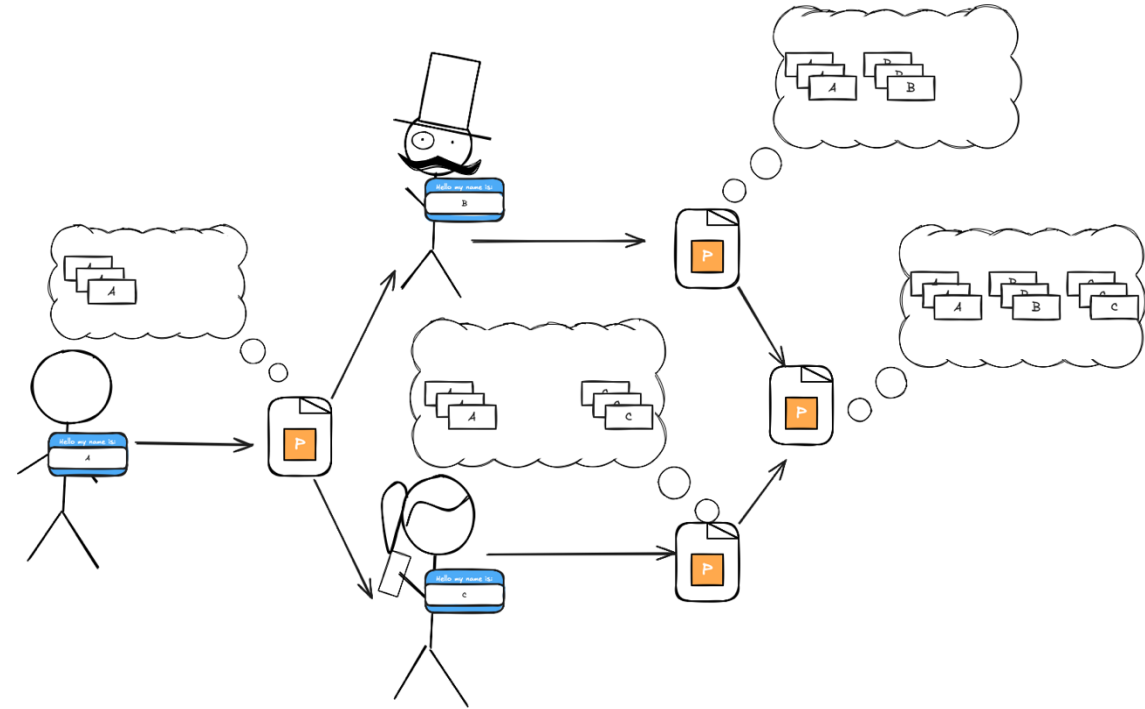
Scenario 1: Linear History (i.e., the history no time travel movie ever has)

1. Person A makes their slides and emails the presentation to person B.
 2. Person B adds their slides to the presentation A sends.
 3. Person B emails the presentation to person C.
 4. C adds their slides.
 5. Presentation is done.
- Emailing the document works fine for this scenario.
 - Called a “linear history” because the timeline has no “branches.”
 - “Branches” are easier to define when we have one (see Scenario 2).



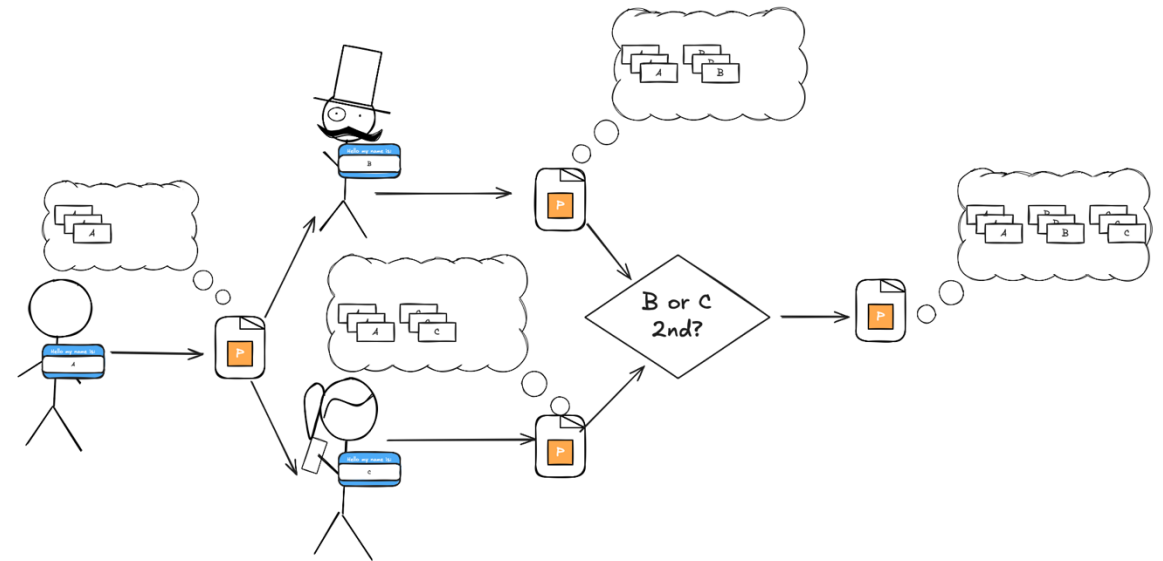
Scenario 2: Non-conflicting branched history (i.e., the timeline every time travel movie thinks it has)

1. Person A makes their slides.
 2. Person A sends the slides to both B and C.
 3. Simultaneously:
 1. Person B adds their slides in the correct place.
 2. Person C adds their slides in the correct place.
 4. The two presentations are magically merged.
 5. We get the final presentation.
- Extra steps but works.
 - “Branched history” because different changes happen simultaneously.



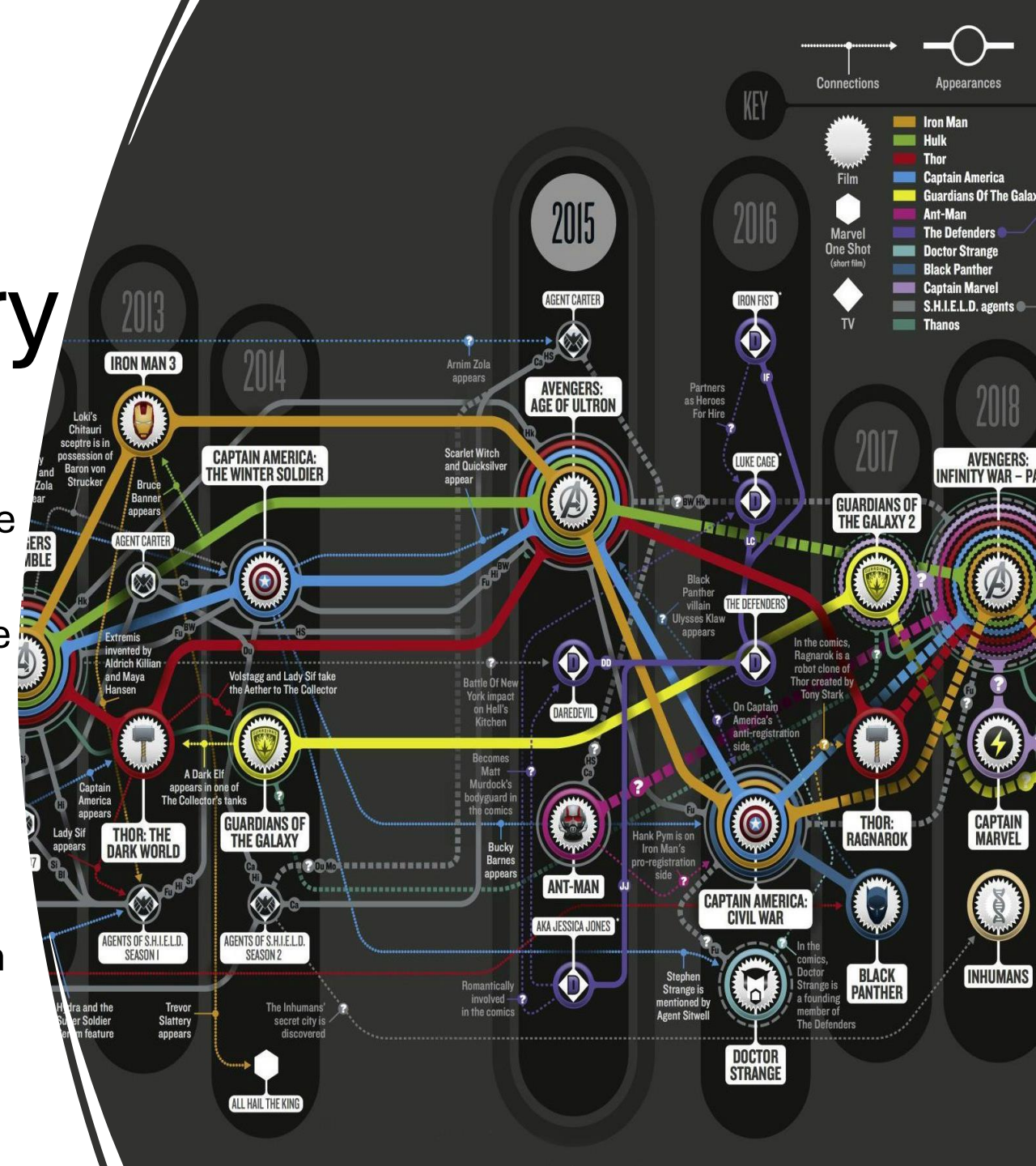
Scenario 3: The conflicting branched timeline (i.e., the timeline every time travel movie actually has)

1. Person A makes their slides.
2. Person A sends the slides to both B and C.
3. Simultaneously:
 1. Person B appends their slides.
 2. Person C appends their slides.
4. Problem, both B and C think their slides come directly after A.
5. The magic merging the presentations moves C's slides to the end.
6. We get the final presentation.
 - Even more extra steps but works.
 - “Branched history” because different changes happen simultaneously.
 - “Conflicting” because the histories are not compatible.



Motivation Summary

- Having multiple people work on the same file can be difficult without coordination.
- Guess what happens when we have multiple people work on multiple files without coordination?
 - Joke answer: the Avengers timeline!!!
 - Serious answer: same as joke answer, but with less A-list celebrities.
- Modern software development almost always involves multiple people working on multiple files in an uncoordinated (or loosely coordinated) manner.

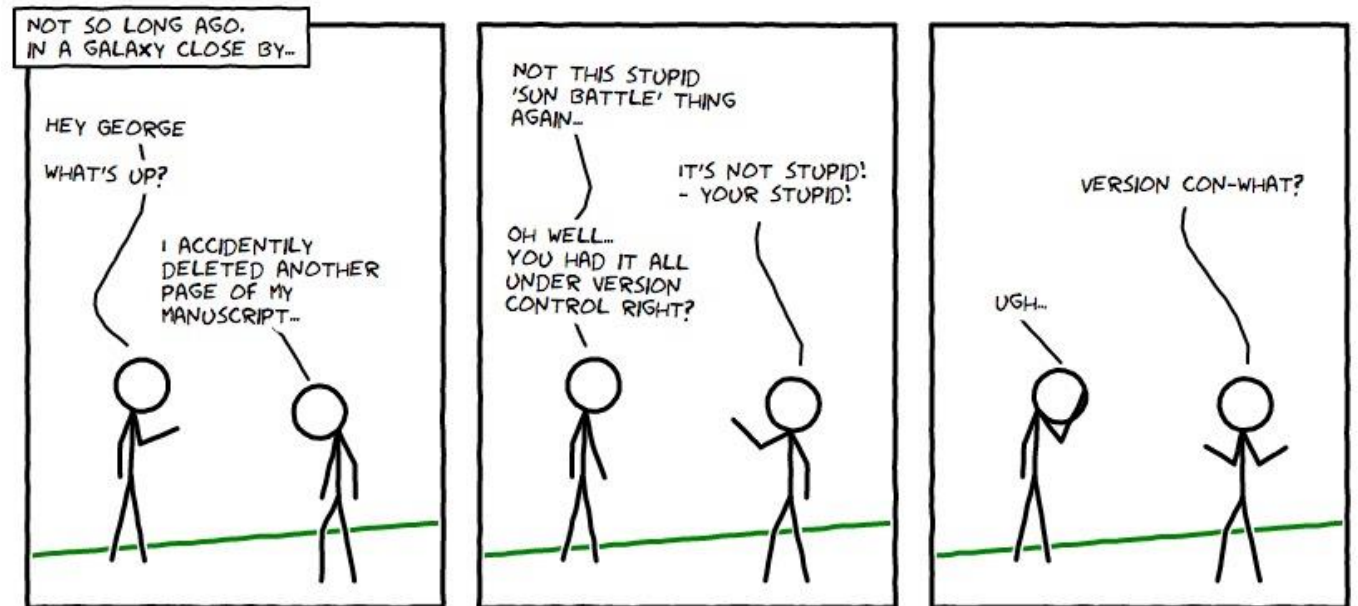


What is the Difference Among Version Control, Git, and GitHub?



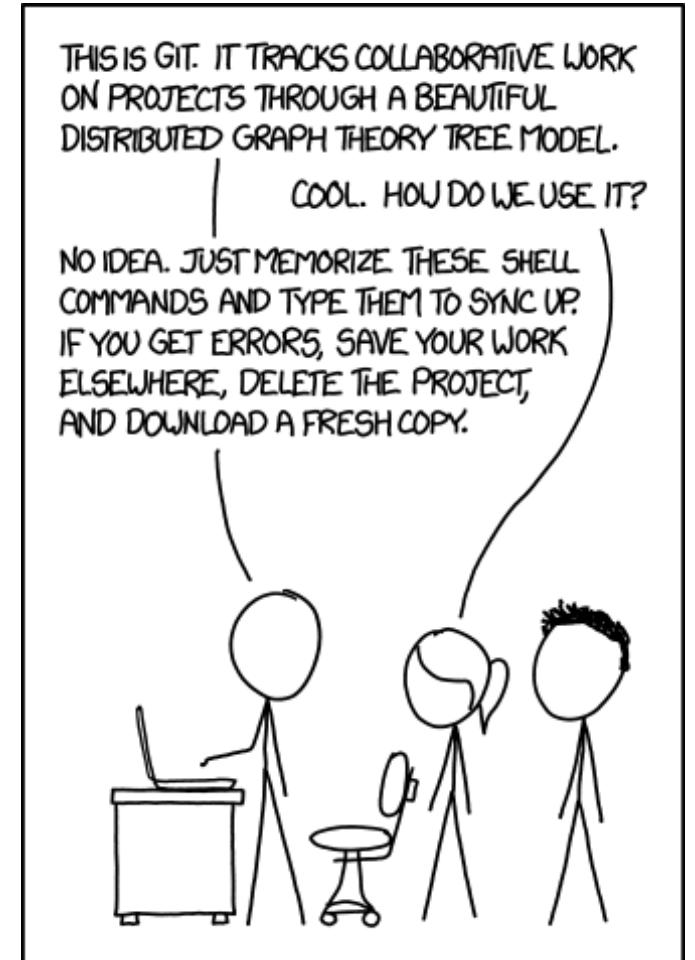
What is Version Control (VC)?

- VC software is used to manage the history (i.e., versions) of a software project.
- Superficially, VC gives you save/load, undo, redo for a set of files.
- Automates (to the extent possible) the steps we termed “magic.”
- Historically, there have been many different VC software packages.
- Now, most developers (95% according to Wikipedia) have coalesced around Git.



What is Git?

- Developed by Linus Torvalds (creator of Linux) for version controlling the Linux kernel.
 - Written because Linux community lost their free license to a proprietary VC software.
 - Amusingly written in about two weeks and was already more performant than all other existing VC software in week three.
- According to Linus Torvalds, the name's meaning depends on your mood:
 - Random three-letter combination that is a mispronunciation of "get".
 - "Global information tracker" if you're in a good mood.
 - "Goddamn idiotic truckload of sh*t" when it breaks.
 - Linus joked he names all his projects after himself ("git" means unpleasant person in British English).



What is GitHub?

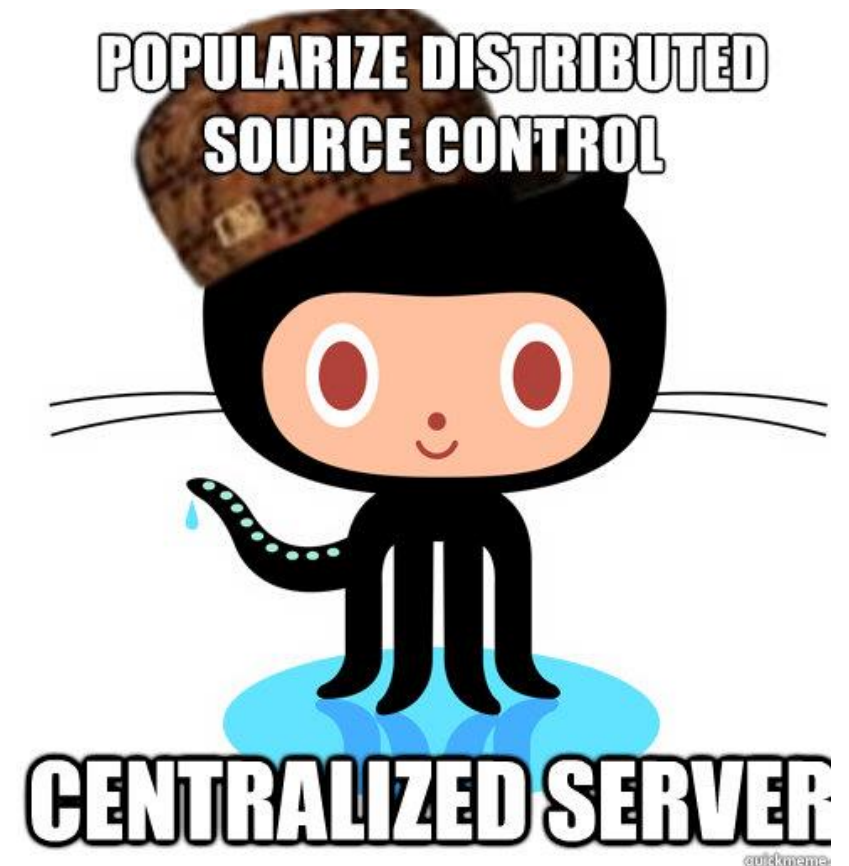
- For SIMCODES we are using Git in the context of GitHub.
- Git is the software responsible for VC-ing our software project.
- GitHub is “just” a storage solution.
 - Conceptually like Google Drive or DropBox.
- You use Git to interact with GitHub.
 - While you can also manually upload/download files this is cumbersome when you are working with multiple files.



Git and GitHub Terminology

Disclaimer: Simplifications Ahead

- Git is an incredibly powerful VC solution.
- Git's power comes from not assuming any “copy” of the project is special.
 - This requires additional levels of abstraction that complicate things.
- Often one copy is actually special, kinda like the “sacred timeline”.
 - Usually the copy on GitHub.
- We introduce concepts assuming there exists a special “copy”.
- The git Jedis in the room know that there's exceptions to pretty much everything I'm about to tell you.



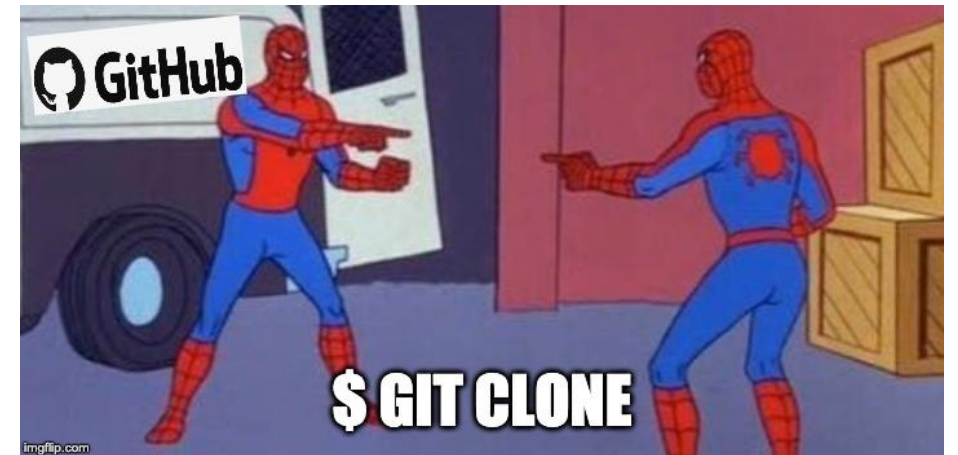
Git Terminology: Repository

- A **repository** is a directory containing the source files and the history of changes to those files.
- A **local** repository is a repository that lives on the computer you are currently using.
 - You will develop software in a local repository.
- A **remote** repository is a repository that lives on a different computer.
 - For now, "remote" repository is synonymous with "the repository that lives on GitHub".
- Repository is usually shortened to just "repo."



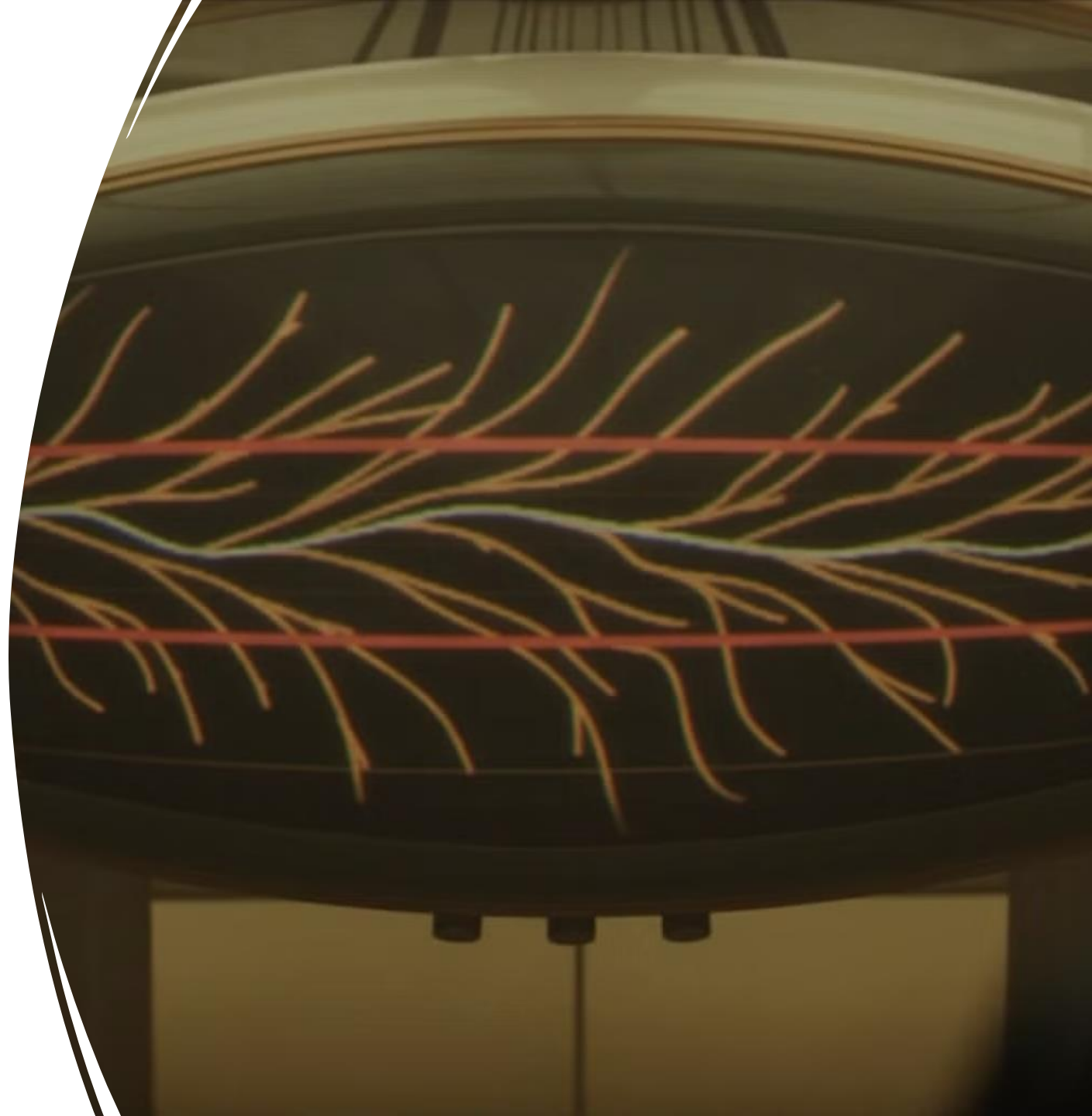
Git Terminology: Clone

- The first step in developing software is to get a copy of the existing source code.
- Usually, the repo you want to develop for already lives on GitHub.
 - If you are starting a brand-new repo, first create the repo on GitHub.
 - Then treat the newly created repo like an existing repo.
- To get a local copy of a remote repo you **clone** it.
- Unless you delete your local repo, you typically only clone a repo once.



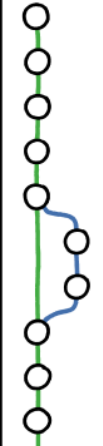
Git Terminology: Branch

- A **branch** is a (linear) history of the repo.
 - It's one of the many timelines.
- Usually, one branch is special.
 - Often called the **main** branch (previously **master**).
 - The special branch is the "final" version of the project.
- Other branches are sometimes called **development** branches.
 - You should always be working on a development branch.



Git Terminology: Commit

- Git doesn't come with "auto-save".
- A **commit** is the set of changes you made to the branch since the last commit.
 - Conceptually, a commit is Git's equivalent of "saving".
 - In the timeline analogy, commits are events.
- Commits can add content and/or remove content.
- Commits save your work on the current branch. Other branches remain unchanged.

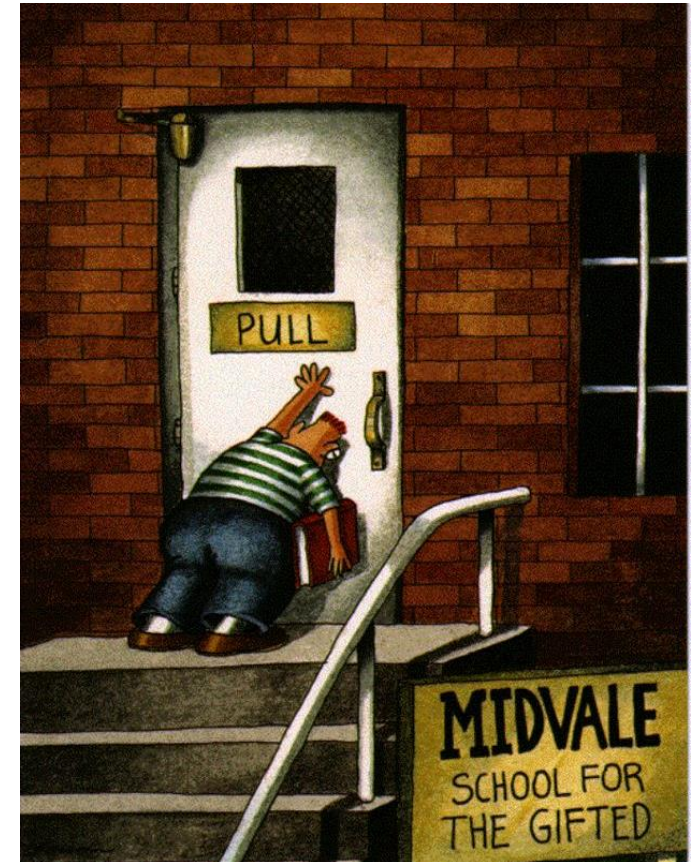


	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

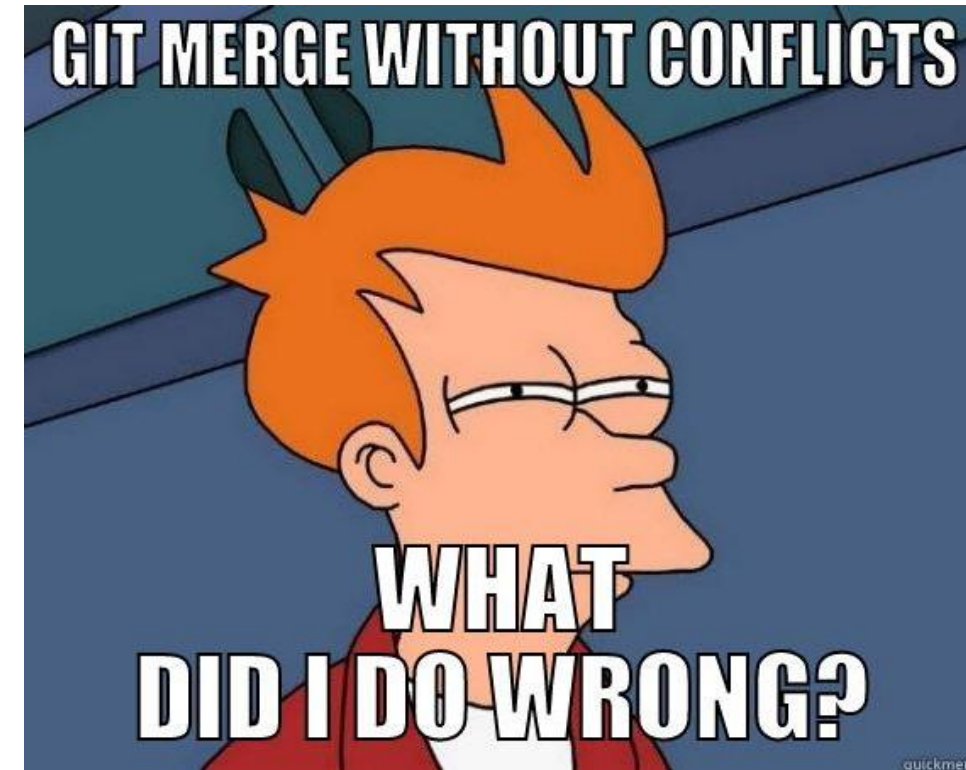
Git Terminology: Push, Pull, Merge

- When you are done developing a feature locally you need to add the feature to the remote repo.
- You **push** your changes from your local branch to the remote branch.
- You **pull** changes from a remote branch to your local branch.
- If you are moving changes from one local branch to another local branch you **merge** the branches.
- As a note, pushing and pulling are implemented in terms of merge, so “merge” is usually used as a catchall.



Git Terminology: Conflict

- Git is pretty good about merging changes automatically.
- However, when the same lines of code have been modified, Git usually needs help merging.
- When Git cannot automatically merge two branches, the branches are said to be in **conflict**.
- If a conflict occurs Git will show you the two changes and you will have to decide how to reconcile them.

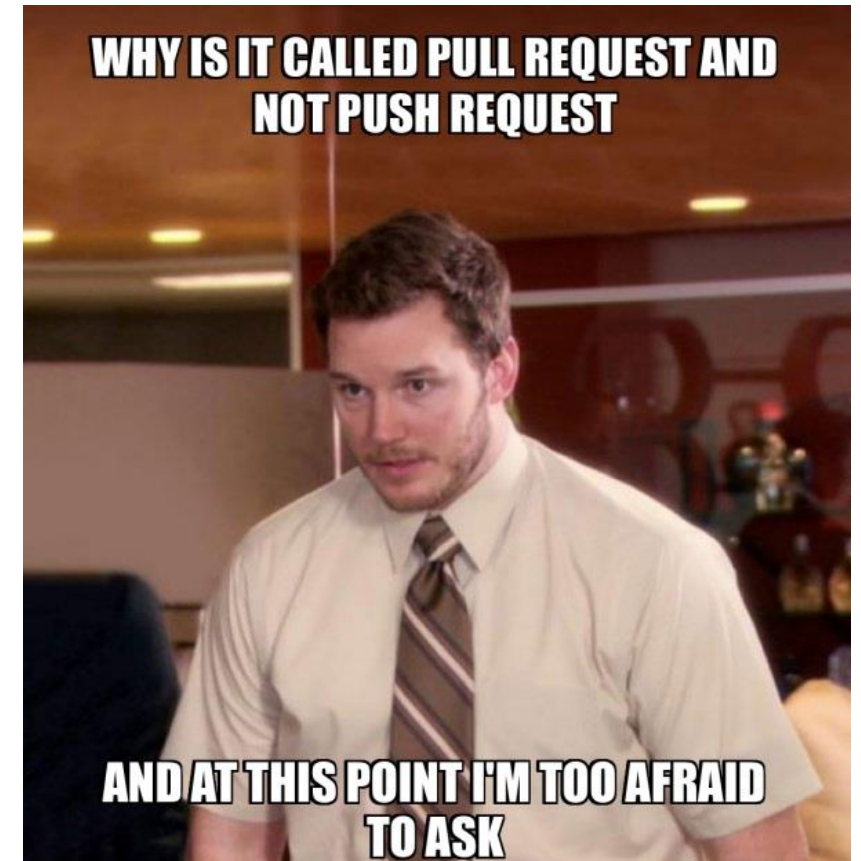


GitHub Terminology: Organization

- Developer teams usually work on a series of related projects.
 - E.g., the SIMCODES team has one project for the website, one project for training materials, and several projects for research.
- Git provides no mechanism for grouping repositories.
 - We don't speak of git submodule...
- A GitHub **organization** allows a developer team to group their repos together.
- SIMCODES has the GitHub organization SIMCODES-ISU.
 - SIMCODES was taken...

GitHub Terminology: Pull Request

- A **pull request** (or PR for short) tells the maintainers of a GitHub repo that you want them to pull changes into their repo.
 - The repo owners can then decide if they want those changes or not.
- Originally, PRs were needed because we didn't want just anyone making changes to our code.
- Most repos now require PRs, regardless of who is making changes.
 - PRs are used to automatically test changes before merging them.



GitHub Terminology: Fork

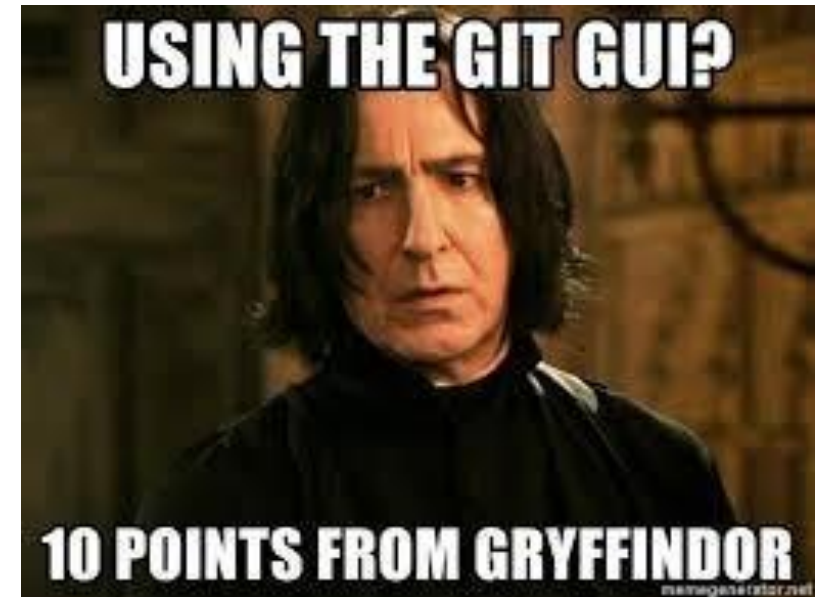
- PRs must be between branches that live on GitHub.
- Say you are trying to change the SIMCODES-ISU/training_materials repo.
 - If you have sufficient privileges, then create a branch and make a PR.
 - But what if you don't have sufficient privileges?
- A GitHub ***fork*** is a clone of a GitHub repository that you own.
 - Can make the PR from a branch of your fork.
- Even if you have permission to create a branch in a repo, you can always fork the repo and make a PR from the fork.



What is a typical Git/GitHub Workflow?

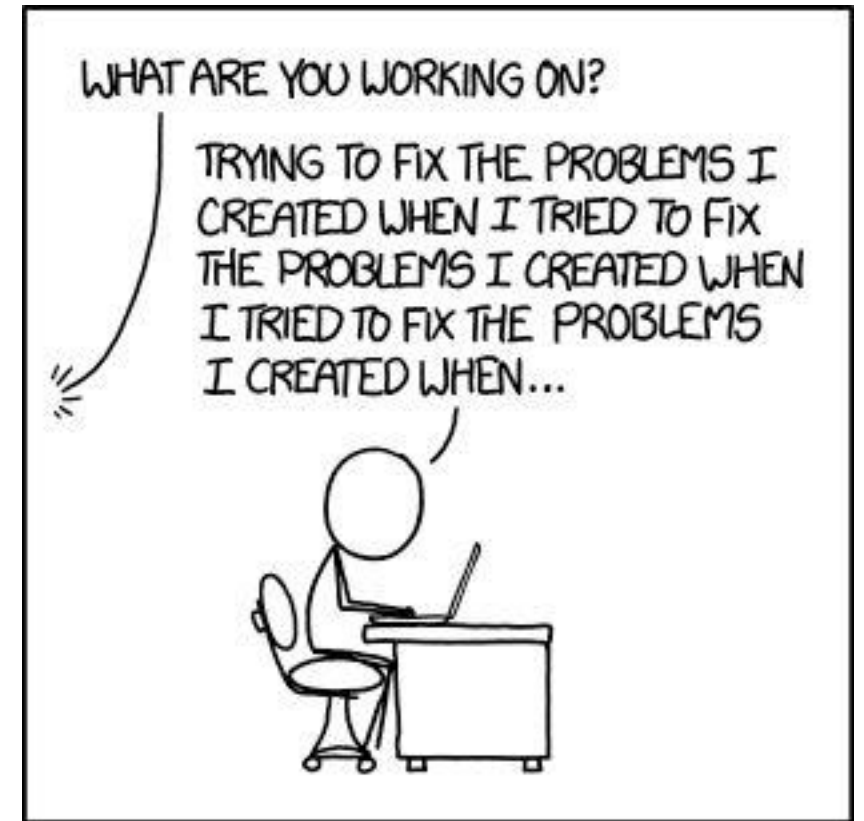
How to get Git?

- Should already have it, if not:
 - Windows: Open a terminal, type `sudo apt-get install git`
 - MacOS: It comes pre-installed.
- This tutorial goes over the basics of Git using a terminal.
- GUIs for Git are plentiful.
- If you use an integrated development environment (like VSCode) Git is usually built in.
- IMHO it's good to use Git from terminal a couple times to understand what is going on. So that's what this tutorial does.

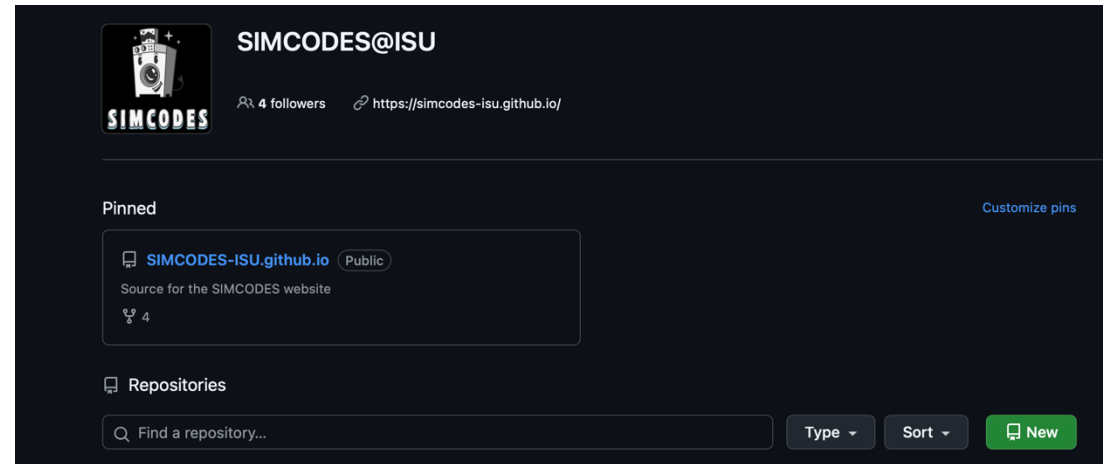


Public Service Announcement

- If you follow these steps, every single time you develop software you will rarely (if ever) encounter a Git/GitHub problem.
 - Disclaimer: problem is not the same as “conflict”, though this workflow will greatly reduce conflicts too!
- If you don't follow these steps, you can still accomplish your goals, but you will make life 10x harder for yourself.
- In my experience, 99% of Git problems come from skipping one of these steps.



Step 0: Create the GitHub Repo



- If the repo doesn't exist yet you need to create it.
- Click the green button, then fill out the fields.
- For this tutorial pick:
 - Name the repo (you can change it later)
 - Select “public” repo.
 - Leave the rest unchecked and click “Create repository”.
- Since y'all will need a repo for this summer anyways, go ahead and do that now.

A screenshot of the 'Create a new repository' form on GitHub. The form includes fields for 'Owner' (SIMCODES-ISU) and 'Repository name'. It has radio buttons for 'Public' (selected) and 'Private'. There are checkboxes for 'Add a README file' and 'Add .gitignore'. At the bottom, there is a 'Choose a license' section and a green 'Create repository' button. A note at the bottom states: 'You are creating a private repository in the SIMCODES-ISU organization.'

Slight Aside: Get Cookiecutter

- Setting up a software project from scratch takes a long time.
- Most of the setup work is creating configuration files.
 - This is often called “boiler plate.”
- Cookiecutter is a tool that allows you to create template software projects roject makes it easy to create a repo with default options.
- MolSSI has prepared a CookieCutter project aimed at computational chemistry. We’ll use that for this tutorial.
- Recommend you us a virtual environment.
 - `python -m venv name_of_venv && source name_of_venv/bin/activate`
- Install Cookiecutter: `pip install cookiecutter`



Slight Aside: Initialize a Repo

- Run Cookiecutter: `cookiecutter gh:molssi/cookiecutter-cms`
- To initialize the repo Cookiecutter will ask you some questions. Recommended answers are:
 - Project name: Up to you, but should probably be related to, if not the same as, the name of the GitHub project.
 - Repo name: whatever the repo you created on GitHub is called.
 - First module name: default is fine.
 - Author name: Mark Twain (kidding, put YOUR name)
 - Author email: I think you can figure this one out
 - Description: A SHORT phrase describing the project.
 - Open-source license: pick 4 (we want to use the Apache 2.0 License)
 - Dependency source: 3 (pip only)
 - Select include ReadTheDocs: n



Step 0: Populate the Repo

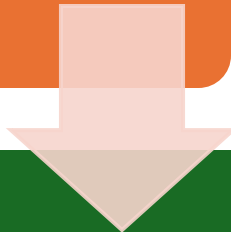
- When you made a repo, GitHub should have told you how to populate it.

```
cd your_repo_name  
git remote add origin https://github.com/SIMCODES-ISU/<your_repo_name>.git  
git branch -M main  
git push -u origin main
```

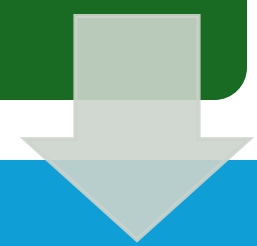
- Don't worry too much about these commands. You don't initialize repos often and when you do, GitHub will remind you what commands to run...

Alternative Step 0: Get a Local Copy of an existing Repo

We just initialized the remote repo with a copy of our local repo.



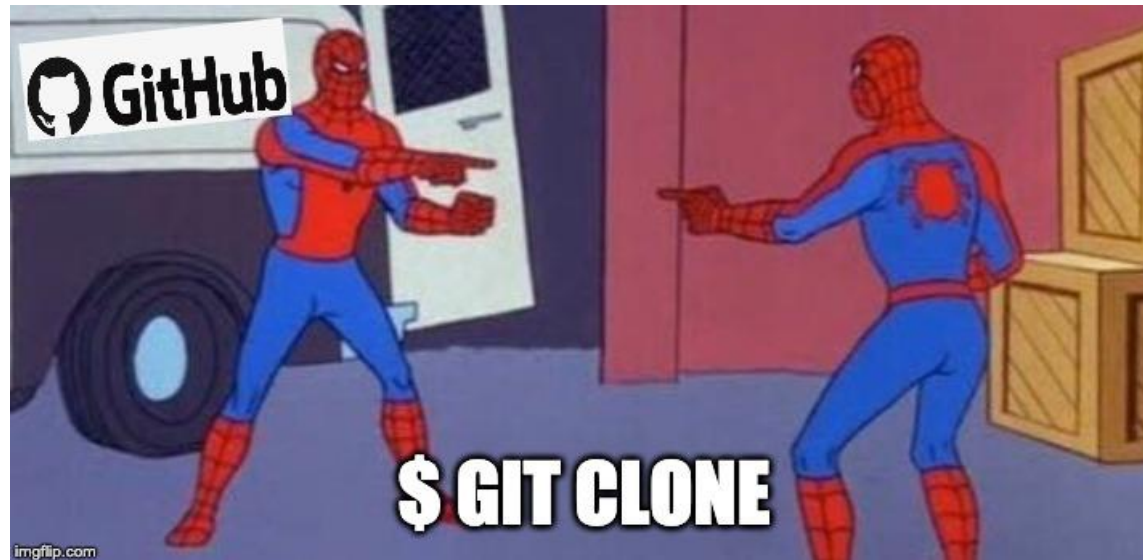
Usually, the remote repo exists, and you want to initialize a local repo with a copy of the remote repo.



Pop Quiz: What was the Git term introduced for the action of getting a local copy of a repo?

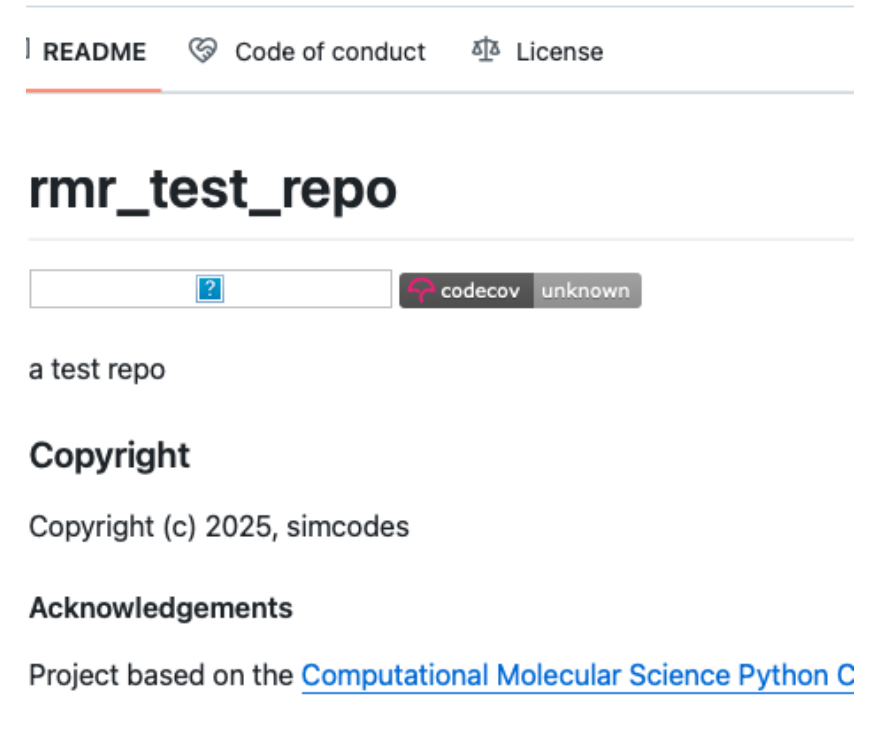
Alternative Step 0: Clone the Repo

- Answer: Clone.
- If we didn't already have the repo, we could get it by running `git clone <url_of_your_repo>`



First Task: Fix the README!

- If we look at our repo on GitHub we see one of the badges is broken.
 - Refresh the page if it still shows the commands to upload it.
- Fixing the badge is easy (and a great first tutorial).
- The following slides will walk you through the typical Git/GitHub workflow to fix the badge.
- The usual Git/GitHub workflow starts at this point, i.e. once you have a local copy of the remote repo.



Step 1: Make a Feature Branch

- You should NEVER make changes to the main/master branch!
 - Exception: None. There is no exception!
 - But what about if...No! Don't do it!
- In my experience, making changes directly to the main/master branch is the primary way Git newbies get in a pickle.
- To make a branch: `git checkout -b fix_badge`
 - “checkout” is the git sub-command used to switch branches.
 - “-b” flag is used to additionally create the bbranch
 - “fix_badge” is the name of the branch.
- Pro tip: `git status` is used to see what branch you are on (amongst other things).

When your coworker asks you which git branch you're currently working on



Step 2: Make Some Changes

- Now open README.md with a text editor.
- Replace “REPLACE_WITH_OWNER_ACCOUNT” with “SIMCODES-ISU”
 - Don’t forget to save the file.
- Note that saving the file, saves it locally. It does NOT save it with Git.
- Pop quiz: What is the term for saving a file with Git?



I Am Devloper
@iamdevloper

Remember, a few hours of trial and error can save you several minutes of looking at the README.

2:11 AM · 07 Nov 18

Step 3: Commit the Changes

- To save the changes we made to VC, we need to commit them.
- To commit: `git commit -a -m "fixed badges"`
 - “commit” is the sub-command for deleting all your files (kidding)
 - “-a” flag tells git commit you want to commit all the changes.
 - “-m” tells git commit that the next argument is your commit message.
 - “fixed badges” is the commit message, i.e., a very short (think like 10 words max) description of what you did.

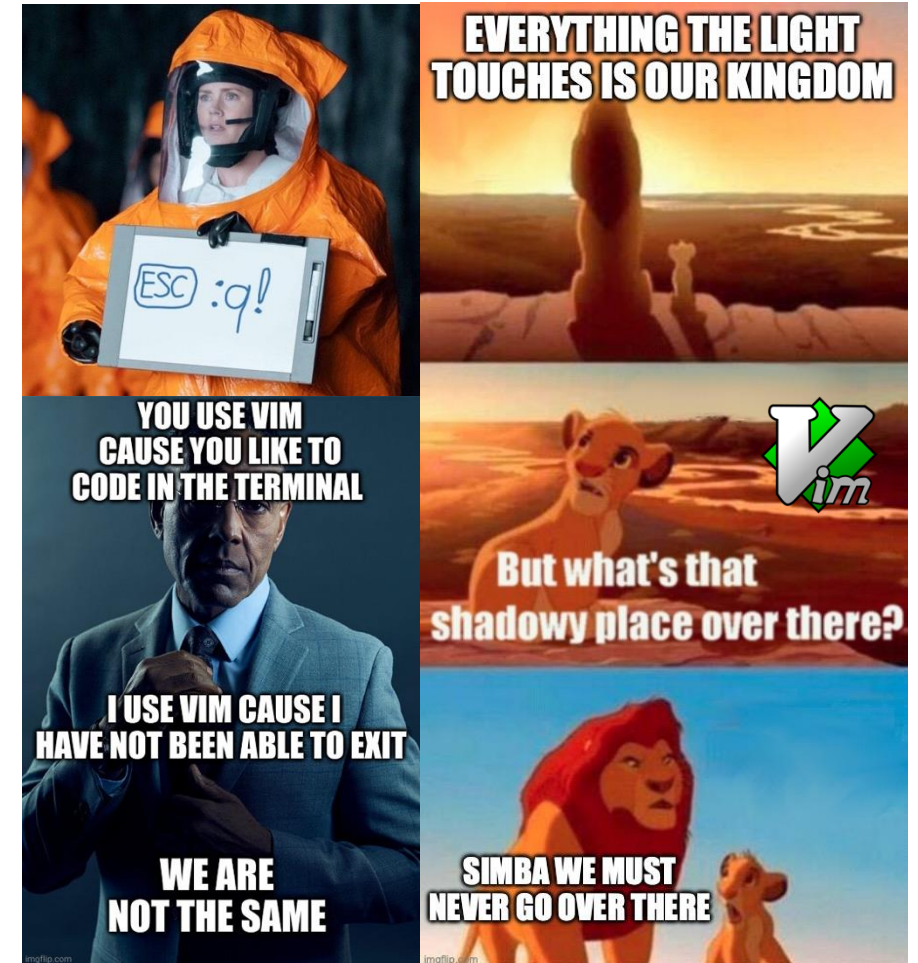


GIT COMMIT -M
"T#JIRA-123|SPRINT2|
V1.2|FIXED
BUG PREVENTING
USER SIGNING IN"

GIT COMMIT
-M
"FIXES"

Notes on Committing

- Git requires you provide a commit message with each commit.
 - If you leave off the “-m” flag it will open your terminal’s text editor of choice so you can write the message.
 - Many of these editors are highly non-intuitive.
 - Case and point, the default editor is most likely “vim”. If you open vim the way to close it is to type the sequence (no commas): esc, :, q, !, enter.
 - I won’t even begin to explain to you how to type text in vim...
- Our example assumed you wanted to commit all changes. Sometimes that’s not the case.
 - `git add <name_of_file1> <name_of_file2> ...`
 - Then run git commit without the “-a” flag.



Step 4: Push the Changes

- Now the changes are saved locally, but not remotely.
- To push the changes: `git push origin fix_badge`
 - “push” is the git sub-command for (you can figure this out).
 - “origin” is the name given to the remote repo (so you don’t have to type out the URL).
 - “fix_badge” is the name of the branch on the remote repo.
 - Will generally be the same as the name of your local branch.

IN CASE OF FIRE 



1. git commit

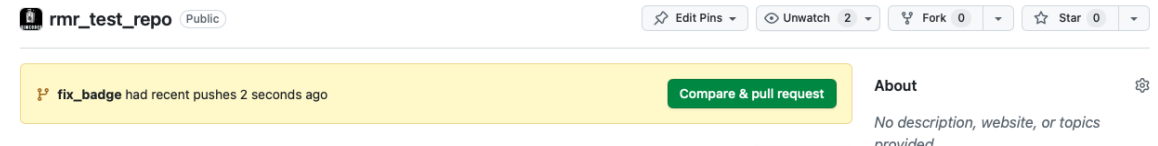


2. git push

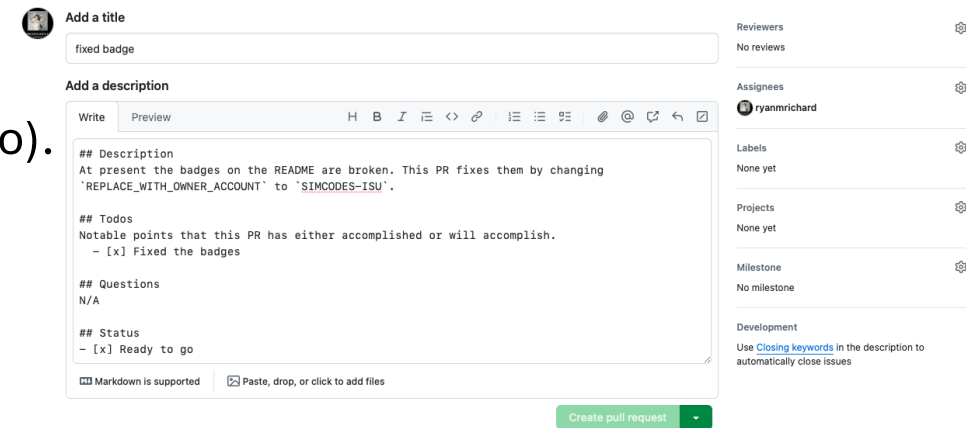


3. git out!

Step 5: Make a PR

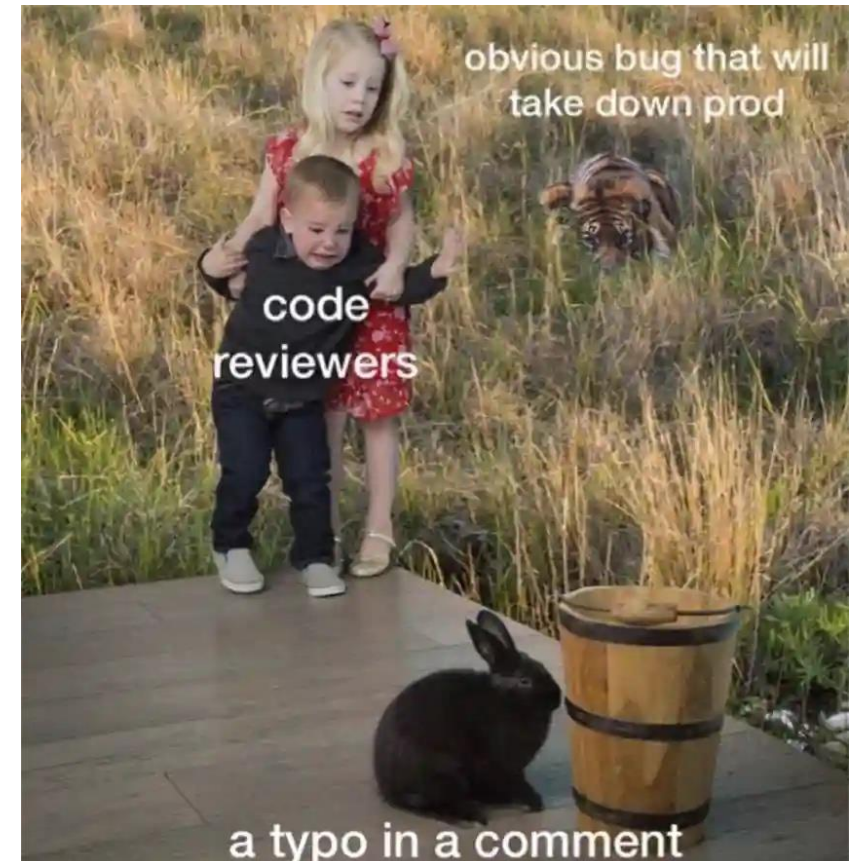


- Your GitHub repo should display a “Compare & pull request” button. Click it.
- You now need to fill out the next page.
 - The description tells people what you did (and plan to do).
 - Reviewers is used to pick who will review your PR (pick ryanmr Richard)
 - Assignees is yourself (click “assign yourself”).
 - Ignore the rest for now and click “Create pull request”.
- This is only done once per branch.
- If you have more changes, go back to step 2.



Step 6: Code Review

- The goal is to get our changes into the main branch of the remote repo. Merging the PR accomplishes this.
- During a code review the maintainers of the remote repo's main branch decide if your changes can/should be merged or not.
- Code review usually involves automated and human checks.
 - Automated checks run your changes and see if they work.
 - Reviewers will also look over your changes and may request changes.
- If you pass code review, the maintainers will merge your PR.
- If you do not pass code review, you go back to Step 2 to make the requested changes.



Step 7: Resetting

- When your PR is merged the main branch of the remote repo is updated.
- Before you can make additional changes, we need to reset.
 - You should NOT continue to keep making changes to the branch that was merged!
- Resetting is another often skipped step.
- First, we switch back to main: `git checkout main`
- Pop quiz: How can we determine what branch we're on?



Origin



Master

```
>git merge
```



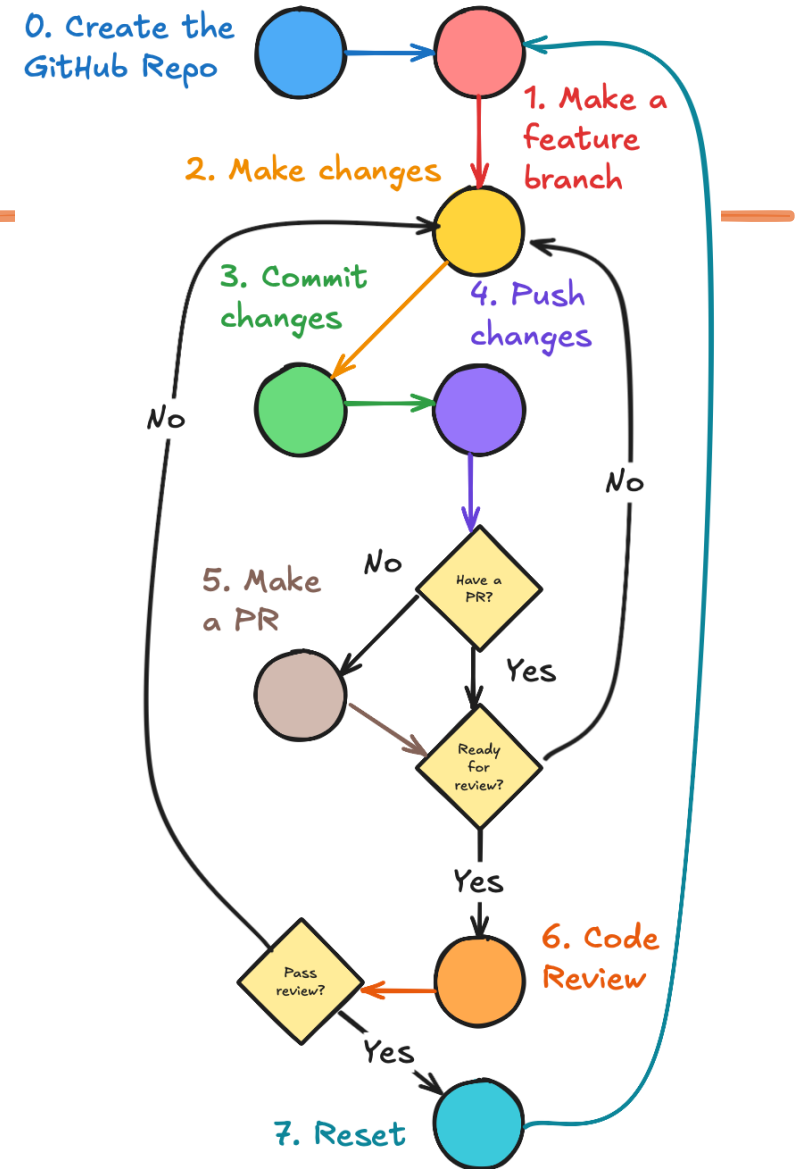
Step 7: Resetting Cont.

- Answer: “git status”.
- At this point, the local main branch is behind the remote main branch.
 - “Behind” means the remote branch has all the commits your local branch has and more.
 - Your local main branch is missing the commits from your PR.
 - If you run “git status”, ignore “Your branch is up to date with ‘origin/main’”, believe it or not, that doesn’t mean what you think it means...
- To update local/main run: `git pull origin main`
 - If you encounter an error in this process, you probably ignored me and made changes directly to the main branch...
 - Running this command when you are up to date with the remote main branch is fine and will do nothing.
- Return to Step 1.



Summary

- I strongly recommend following the suggested workflow every single time you make changes!
- While intimidating at first, once it's second nature this workflow adds very little time.
- This workflow can scale to simultaneous development of multiple features.
 - I strongly suggest sticking with a single feature when you first start out.
- I would only start to deviate from this workflow when you are comfortable managing multiple branches across multiple remote repositories.
 - Even then, NEVER skip steps 1 and 7!!!



Bonus practice: License File

- We mentioned that our repos will be licensed under the Apache 2.0 license, but it wasn't a choice.
- Practice what you learned by creating a file "LICENSE" that lives in the root of your repository.
 - (In file system lingo "root" refers to the top directory).
- The contents of the Apache 2.0 LICENSE can be obtained from: <https://www.apache.org/licenses/LICENSE-2.0>
 - Copy/paste the contents of the box into the file.
- Create a PR with me tagged as the reviewer when you're ready.

APACHE LICENSE, VERSION 2.0

- Text version: <https://www.apache.org/licenses/LICENSE-2.0.txt>
- SPDX short identifier: [Apache-2.0](#)
- OSI Approved License: <https://opensource.org/licenses/Apache-2.0>

The 2.0 version of the Apache License, approved by the ASF in 2004, helps us achieve our goal of providing products through collaborative, open-source software development.

All packages produced by the ASF are implicitly licensed under the Apache License, Version 2.0, unless otherwise stated.

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Section 1 of the document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to direct the management or administration of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to source code, documentation source, and configuration files.

FAQs

- How often should I commit?
 - This is personal preference. I'd argue best practice is once per "task." If you do one commit per task (and your commit messages describe those tasks well), then your PR writes itself in that it's your commit log.
- When should I make a PR?
 - I recommend making a PR as soon as you push to GitHub. When you're first starting to develop this makes it easier for others to monitor your progress. When you're more experienced, it helps others know what you're working on.
- What's the other stuff on GitHub we skipped over?
 - GitHub is designed to be a one-stop-shop for software development. It has a lot of features. Most of the other features fall under software engineering and DevOps. Other tutorials will cover some software engineering, but DevOps is beyond this bootcamp.

Summary

- Collaborating on multiple files with multiple people is difficult.
- Version control (VC) makes the process much less painful.
 - We use Git for VC and GitHub for storing our repos.
- Software development follows the same workflow every time.
 - Deviating from the workflow will increase the likelihood of conflicts, or other problems, arising from when you try to pull or push.



Acknowledgements

- MolSSI's git tutorials
<https://education.molssi.org/python-package-best-practices/02-git.html>
- MolSSI's GitHub tutorials.
- NSF for funding SIMCODES.

