CSC301 (winter 2022)
introduction to software engineering

Tutorial 1

Git Flow

tutorial outline

01 git

02 merging and rebasing

03 git flow

info

5% of your final grade in this course will come from tutorial participation
There will be a total of **7 (regular non-project related) tutorials** where we will be introducing/teaching new content to you, you will earn 1% per tutorial up to a max of 5% earned.

-> This means that you can attend 5 out of the 7 regular non-project related) tutorials and still get full marks for this portion of your final course grade.

What counts as participating in a tutorial?

-> Just attend the tutorial, pay attention, take part in the tutorial activities, and make sure I record your utorId before you leave the room. (and no, unfortunately you can't show up more than halfway through the tutorial or leave very early and expect to still earn the 1%)

info

you'll be expected to form teams on your own. **please** put in effort to create or join a team, if you do not, you will be assigned to a random team that has space, but generally that's more complicated for all of us.

all of the sprint 0 deliverables will be due at 11:59pm, January 27th, but you'll have a chance to submit **proposal.md** on January 20th to us for feedback before the final submission and grading on January 27th. more information is on the sprint 0 handout.

project tutorials

your team will be required to sign up for a timeslot that your entire team can attend so that you can demo your sprint work to the TAs. signups are made through a google form on the course website (will come out next week).

- There will **4** tutorials where you will be presenting your sprint demos to the TAs!
- These will take place **ONLINE ON ZOOM** (link TBA)

Your **sprint 1 demos** will take place on **February 10th** (please see the other demo dates on the course website).

git is a distributed version control system that enables teams of software engineers to collaboratively work on repositories of code

an open source project started by Linus Torvalds and the most popular version control system used by software engineers currently.

modifications to the codebase are represented as “commit” objects, and the git history is just an immutable, linked list of commits

no doubt you have been exposed to it in previous courses such as csc207 and its basic commands (add, commit, push, pull, clone)

</>

typical git usage

what is the typical command flow for contributing to a software project?

1. **branch off main and create your feature branch**
2. implement your feature on your feature branch
3. **add the untracked files to your next commit**
4. **create your commit**
5. **push the commit**
6. merge feature branch into main

```
> git checkout -b DEV-101
```

```
> git add --all
```

```
> git commit -m “msg”
```

```
> git push
```

typical git usage

easy enough, right? unfortunately with multiple contributors, things can get complicated

Q. what if someone pushes code to the main branch that I need for my feature branch?

Q. what if I have to undo a commit?

Q. what if I want to reset local changes?

what if someone pushes code to the main branch that I need for my feature branch?

there are **two methods** for bringing code in from other branches into your own.

merging

combines branches together into a commit

Rebase + merging

modify, mutate and move commits on your branch with rebase and **then** merge

merge

```
> git merge <branch>
```

merging creates a commit which combines the tip of the master branch (HEAD) and the tip of the feature branch into one commit

- this commit is referred to as a “merge commit”
- the merge commit becomes the new HEAD after the merge is complete

merge

pros

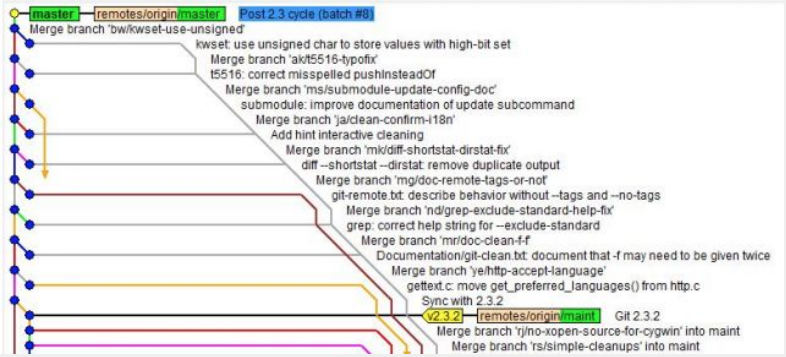
non-destructive, doesn't alter any existing branches ✓

simpler to manage, not as complicated as rebase ✓

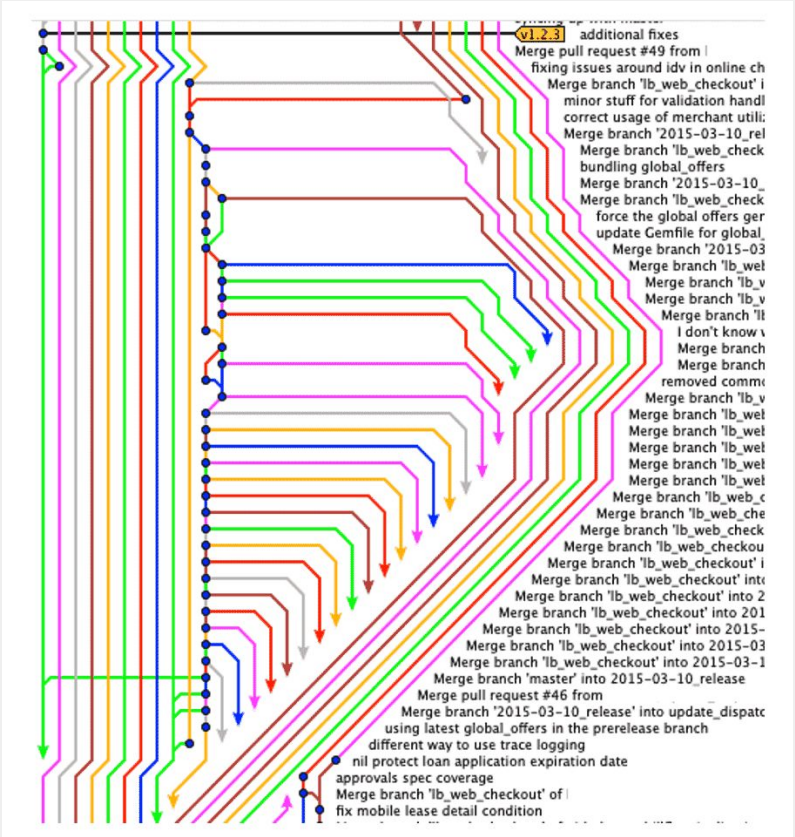
cons

✗ tends to **pollute the master branch** with extra merge commits

- Becomes an issue in high traffic projects with many contributors



merge

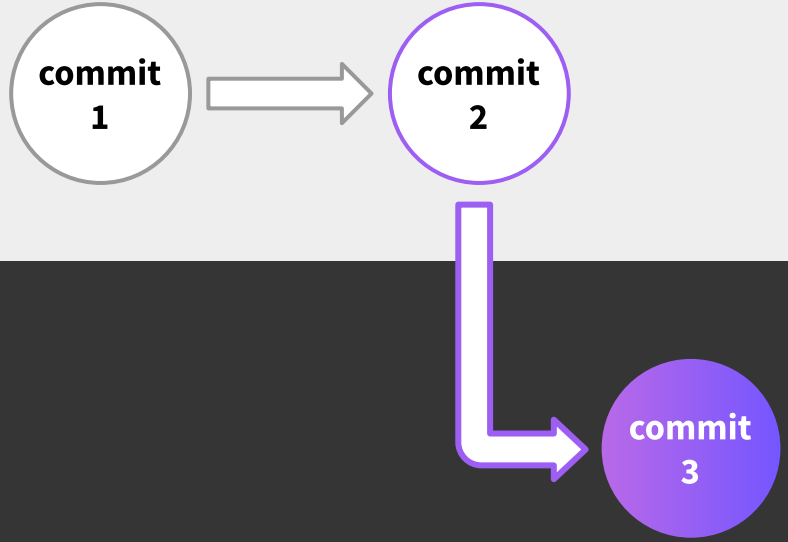

v1.2.3 additional fixes
Merge pull request #49 from
fixing issues around idv in online ch
Merge branch 'lb_web_checkout' i
minor stuff for validation handl
correct usage of merchant utili
Merge branch '2015-03-10_rel
Merge branch 'lb_web_check
bundling global_offers
Merge branch '2015-03-10_
Merge branch 'lb_web_check
force the global offers ger
update Gemfile for global_
Merge branch '2015-03
Merge branch 'lb_wel
Merge branch 'lb_v
Merge branch 'lb_v
Merge branch 'lb
I don't know
Merge branch
Merge branch
removed comm
Merge branch 'lb_v
Merge branch 'lb_wel
Merge branch 'lb_wel
Merge branch 'lb_wel
Merge branch 'lb_wel
Merge branch 'lb_wel
Merge branch 'lb_web_c
Merge branch 'lb_web_che
Merge branch 'lb_web_check
Merge branch 'lb_web_checkou
Merge branch 'lb_web_checkout' i
Merge branch 'lb_web_checkout' int
Merge branch 'lb_web_checkout' into 2
Merge branch 'lb_web_checkout' into 201
Merge branch 'lb_web_checkout' into 2015-
Merge branch 'lb_web_checkout' into 2015-03
Merge branch 'lb_web_checkout' into 2015-03-1
Merge branch 'master' into 2015-03-10_release
Merge pull request #46 from
Merge branch '2015-03-10_release' into update_dispatc
using latest global_offers in the prerelease branch
different way to use trace logging
nil protect loan application expiration date
approvals spec coverage
Merge branch 'lb_web_checkout' of
fix mobile lease detail condition

git merge

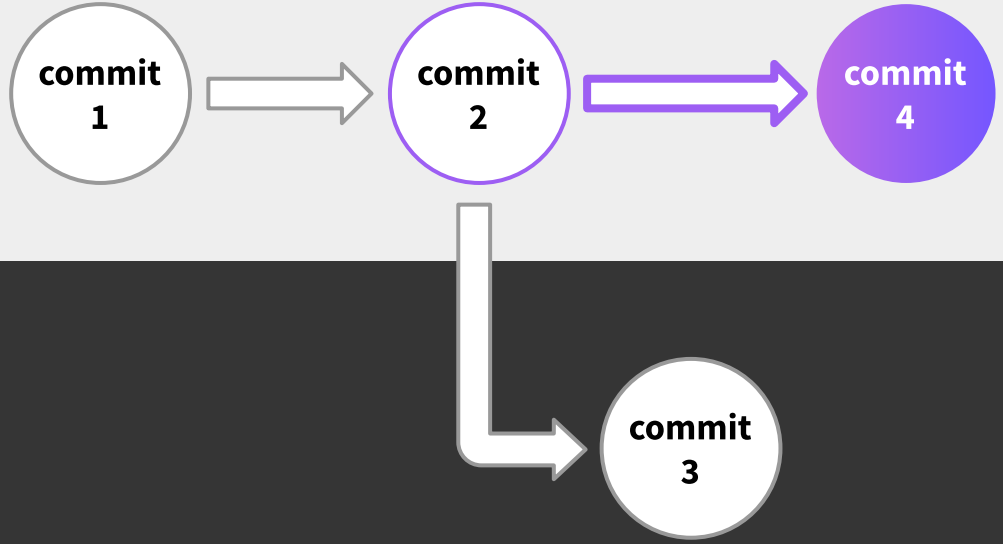
master

new branch

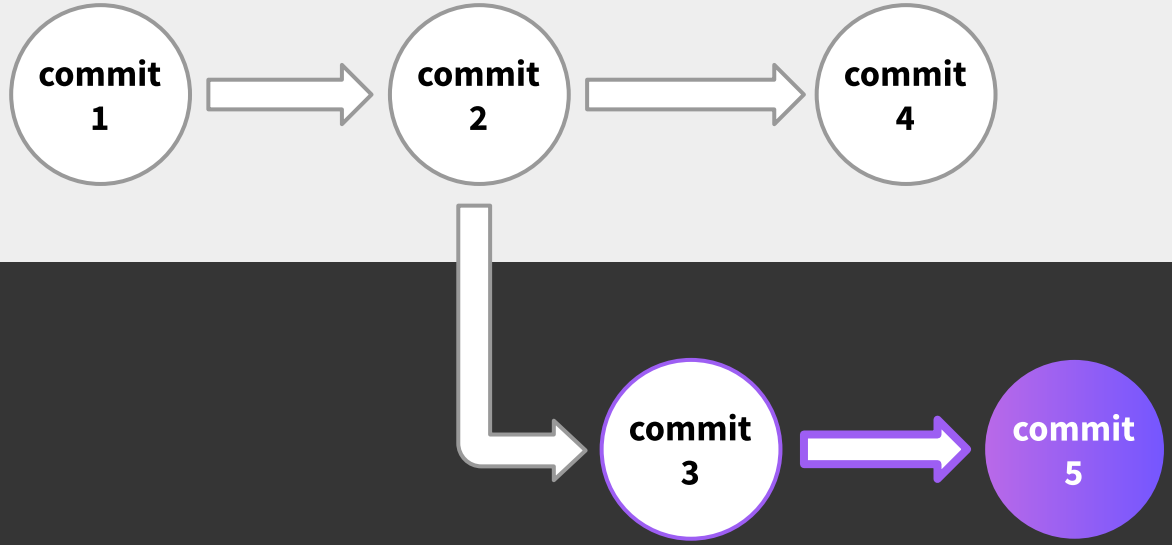
git merge



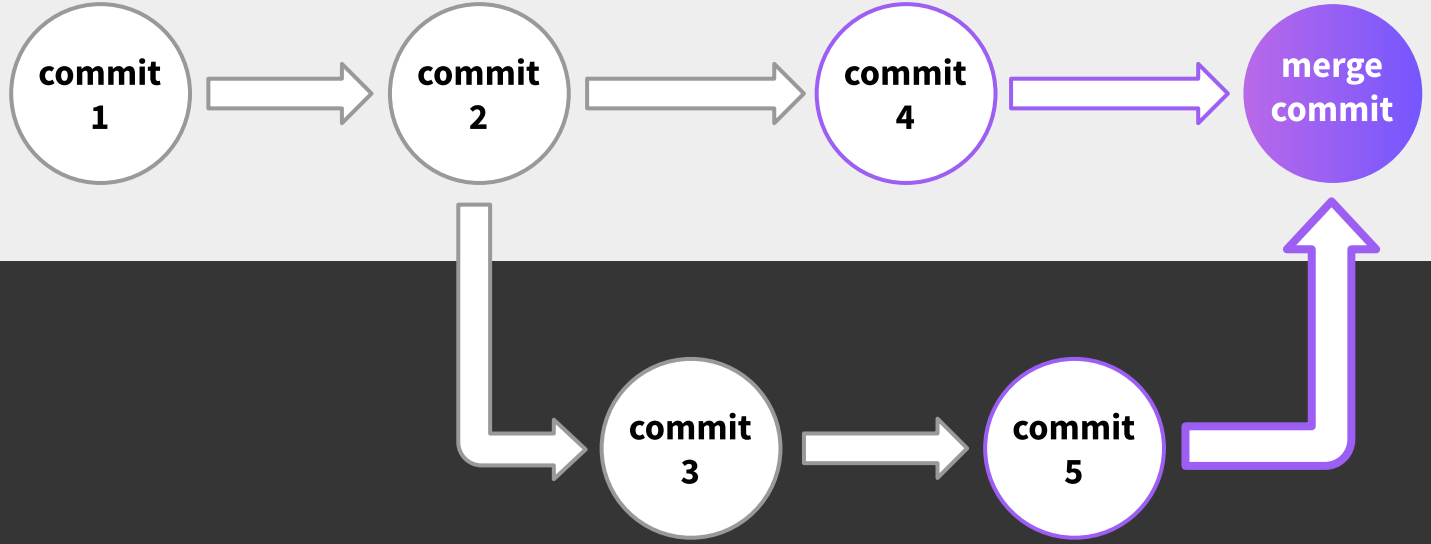
git merge



git merge


commit 1
commit 2
commit 4
master
commit 3
commit 5
new branch

git merge


commit 1
commit 2
commit 4
merge commit
master
commit 3
commit 5
new branch

rebasing + merge

`> git rebase <branch>`

rebasing moves things around, the first commit of feature branch is placed sequentially after the tip of the branch you're rebasing onto

- no extra commit merging the two
- partially rewrites the git history by creating brand new commits for each commit in the master branch
- requires extra step to merge: *rebase feature branch on master, then merge*

rebasing

pros

cleaner project history, no unnecessary merge commits ✓

linear history is maintained ✓

(Can follow the entire history of the project from the tip of the feature branch back all the way to where master was beforehand)

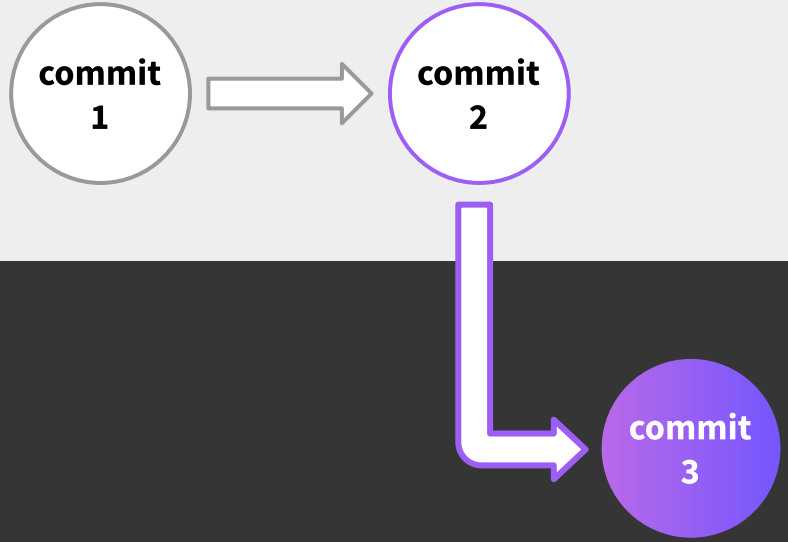
cons

✗ **requires caution**

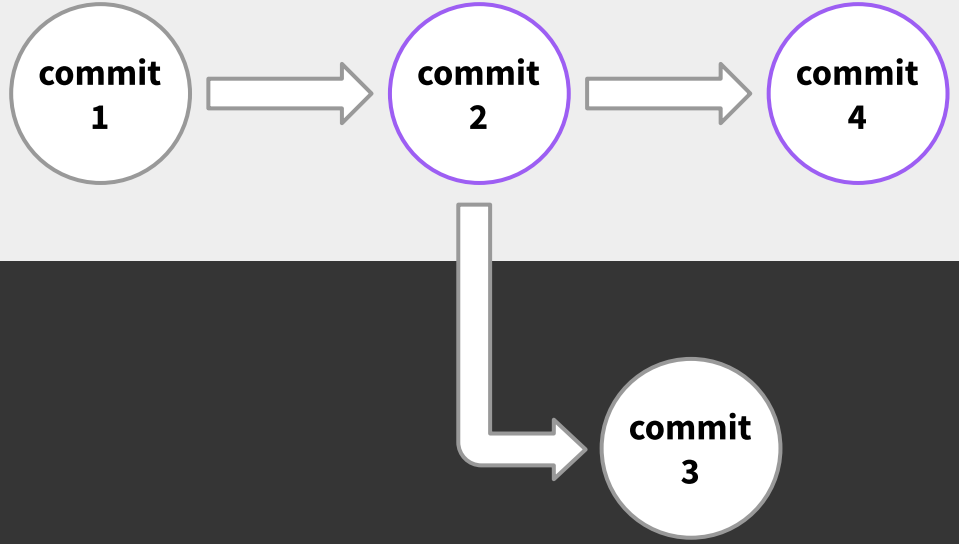
- Never rebase a public branch onto your feature branch
- This will result in two different versions of the master branch, which will need to be merged back together

</>

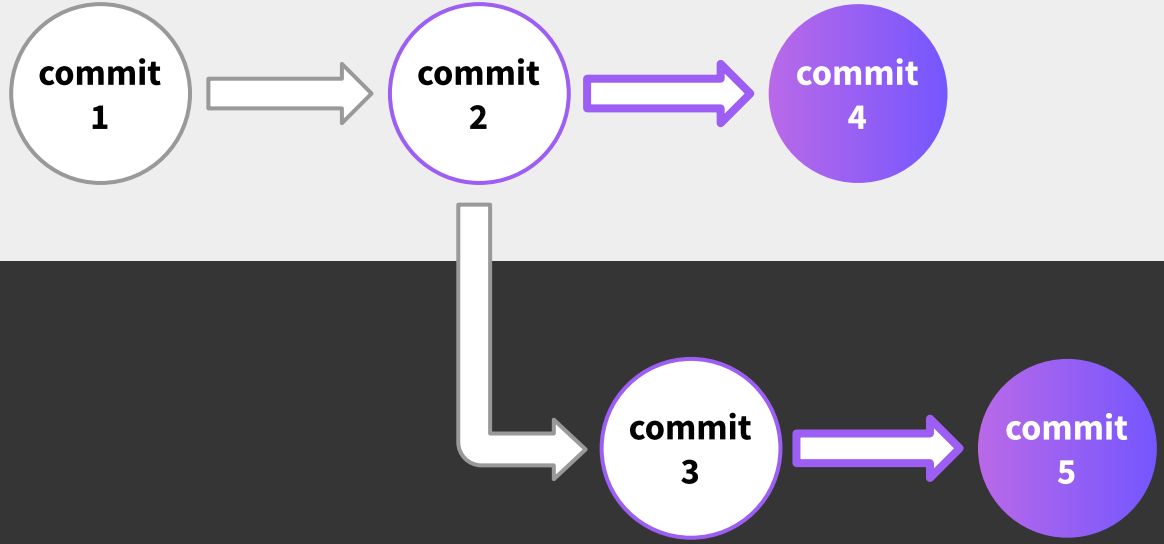
git rebase



git rebase

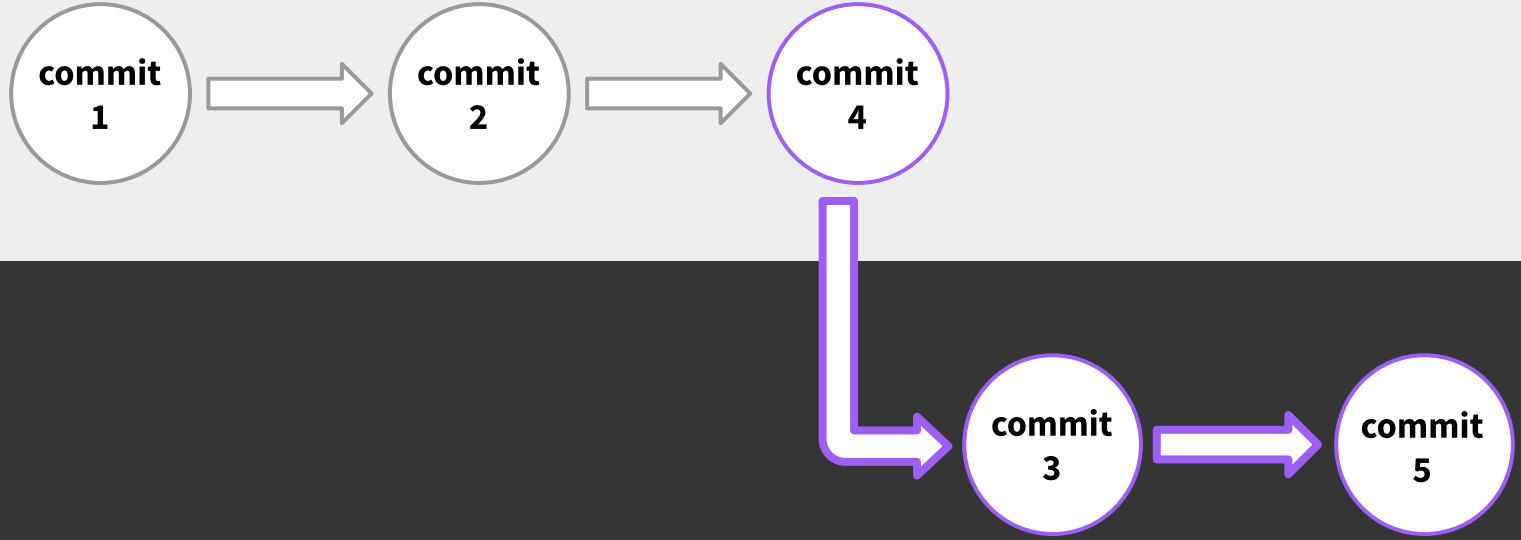


git rebase


commit 1
commit 2
commit 4
master
commit 3
commit 5
new branch

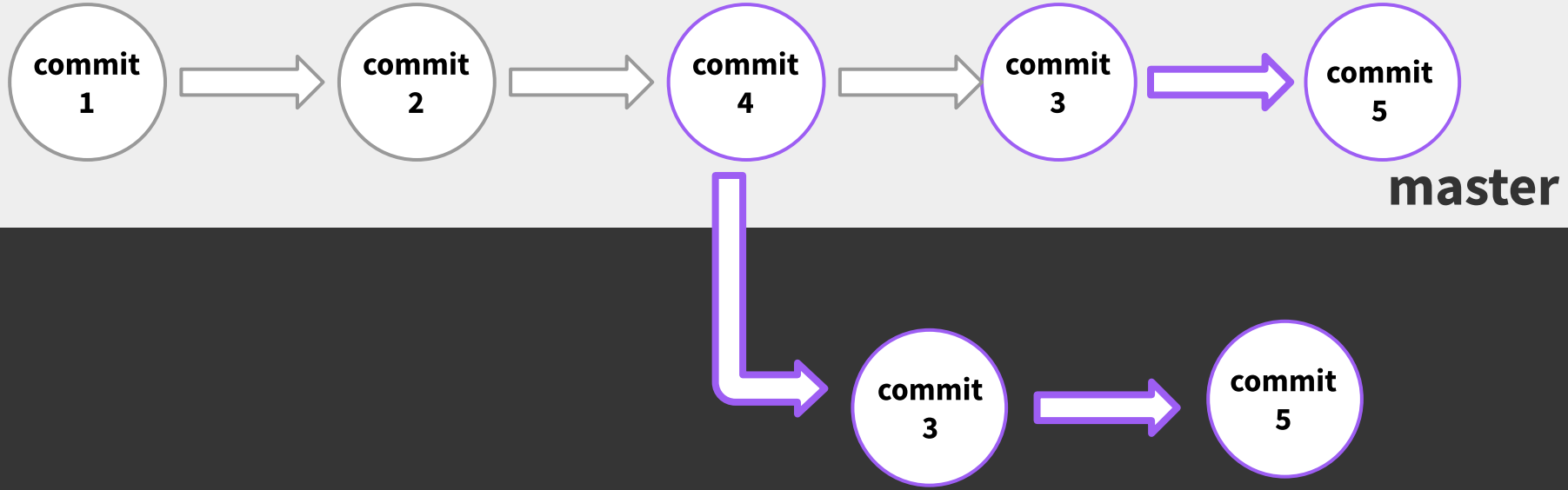
git rebase

Rebasing new branch ONTO master.

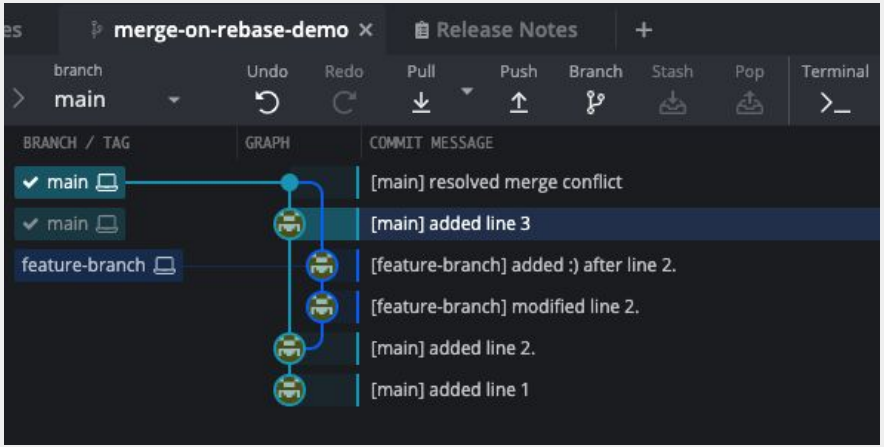


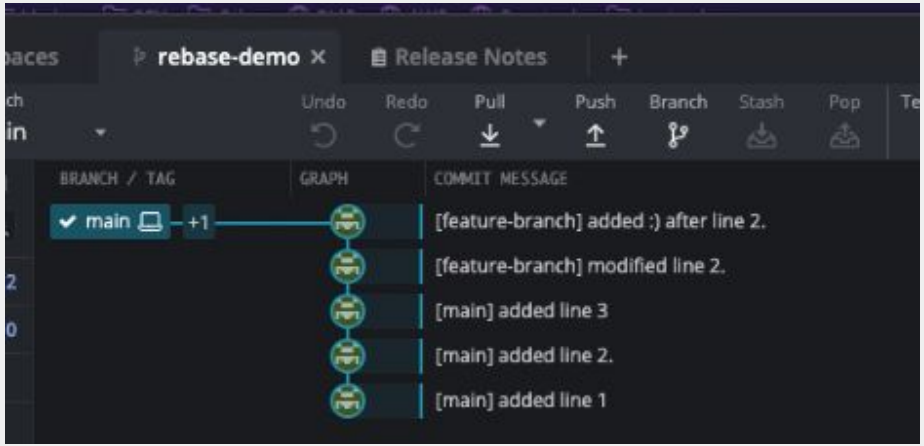
git rebase

After rebasing we merge master with the new branch (will result in git just “fast forwarding” the master branch for us).



A better visualization of the benefits of rebasing.





I ran the exact same commands in both of these scenarios. Except that in the first image I ran “git merge feature-branch” (while on main). In the second image, instead I first **rebased the feature branch onto** main (“git rebase main” while on feature-branch) and then switched to main and ran “git merge feature-branch”.

what if I have to undo a commit?

git is designed to be immutable, so **undoing things can get dicey**.
You have to be careful and use the right tools!

reverting
undoing your changes

resetting
changes the state of head

restore
restore your changes

stash
store your changes for later

undoing things in git

reverting

things break, commits introduce bugs, etc. -- this is perfectly normal.

The most simple form of “undo” in git is a revert

`git revert <ID/Ref>` - Creates a new commit that simply un-does the commit which was specified

reverting is not ideal if you want to undo something locally without an extra commit

undoing things in git

resetting

resets the state of the repository back to a certain state in the past, in various ways

- **soft**: modifies where HEAD points, staged/unstaged changes are not touched; previous commits become staged files
- **mixed**: modifies where HEAD points, wipes the index clean (staged files), but doesn't touch unstaged files; previous commits become unstaged files
- **hard**: "nukes" everything, be careful when using. Staged and unstaged files reset to the specific commit, HEAD updated, previous commits are gone

undoing things in git

restore / stash

Q. what if I want to reset local changes?

restore

resets the state of the repository to the latest commit. Can be used to restore one or more files based on the provided file path.

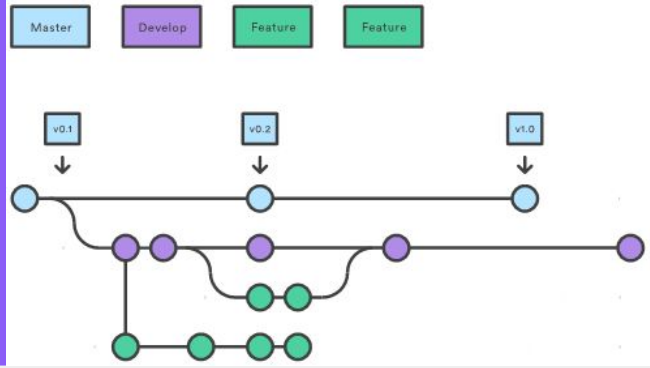
- Run git status to see the modified files
- `git restore [files]`

stash

stores the locally modified files into a stash, which can be popped from to restore changes. good for storing changes before merging/rebasing

- `git stash`
- `git stash pop`
- `git stash clear`

git flow



git flow is a commonly used git branch structure that promotes agile software development & continuous integration/continuous deployment.

the master branch stores the official release history, and the develop branch is used to integrate features.

git flow

feature branches are merged into develop

once develop has been thoroughly tested and contains all the features/fixes for a release, you merge develop into master!

if there are issues with a feature, we can revert the feature branch in develop
if there are issues with a release, we can revert the merge commit in master

result: isolating our environments prevents issues from becoming a problem!

pull requests

the crux of the git workflow

As mentioned here, *“pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.”*

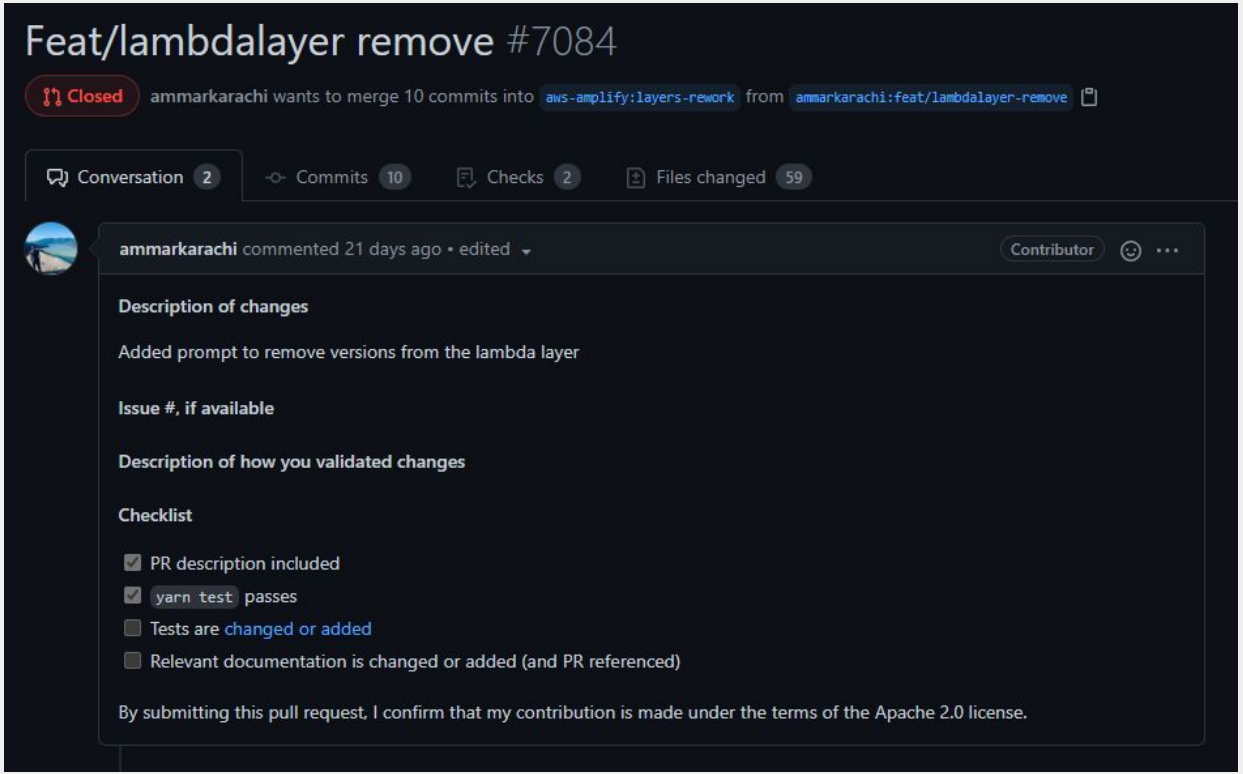
Pull requests are extremely useful when you may want others to review your before merging into other branches!

pull requests

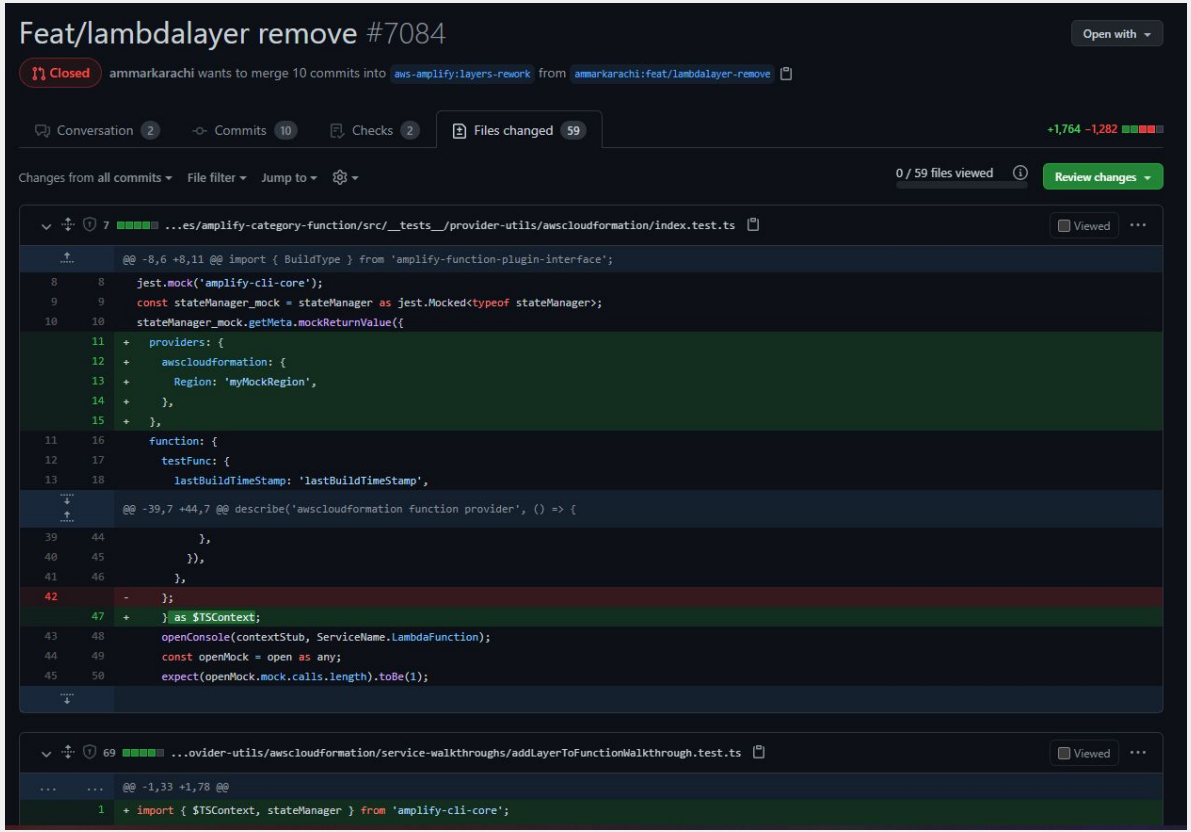
the crux of the git workflow

- after pushing your changes into a branch, you create a pull request from your branch into the develop branch.
- automated checks will run against it informing you of any bugs/errors.
- it can be then be reviewed by one or more engineers, where they can provide comments on how to improve it.

pull requests

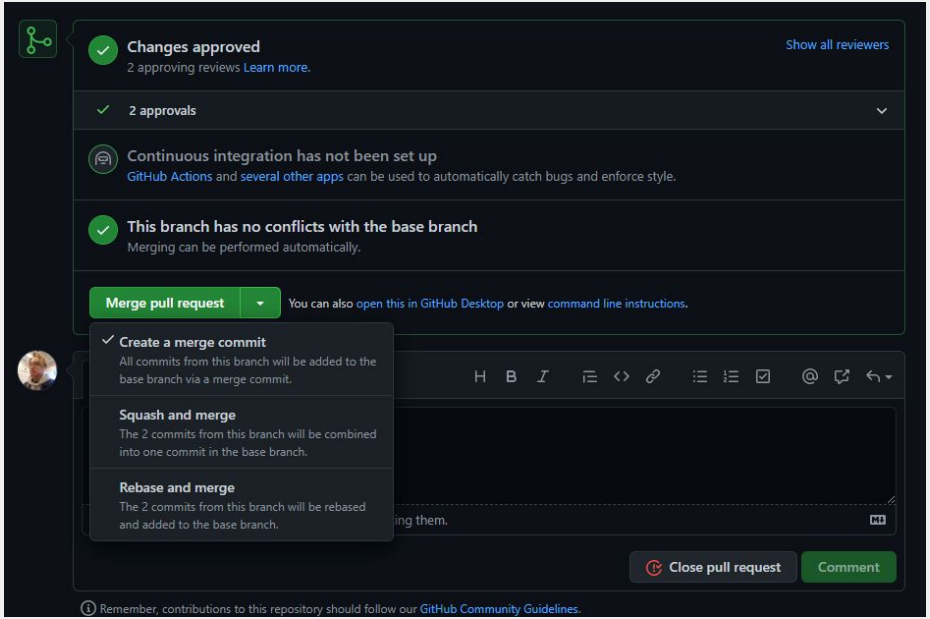


pull requests



merging in pull requests

after being approved, use the github UI to merge your pull request! we recommend squashing and merging your commits so that all your changes are added to develop as one commit.



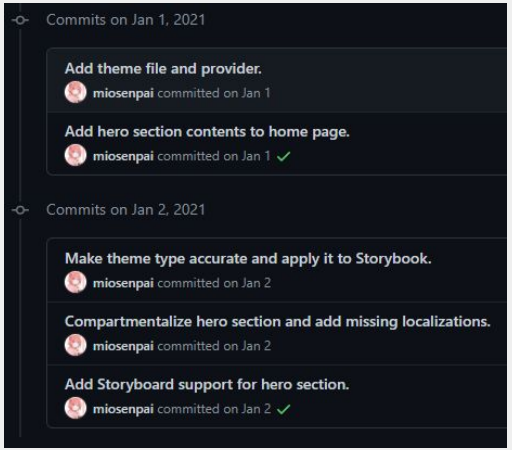
it’s a nice way to encapsulate the changes and ensure the commit history of develop is clean.

project advice

- **use git flow!** It's industry practice for a reason
- **commit messages should be descriptive and useful**

Bad

Good



- **automated testing** is worth the effort
- use **pull request** templates (https://gist.github.com/jcserv/33f19818fde83c18e755b1c138eeac49)
- have **one or two people thoroughly review** each pull request

Additional Resources

Here are some additional resources on git that I co-created when I TA-ed CSC207 last semester.

"CSC207: Using Git and GitHub to Collaborate on a Project":
https://docs.google.com/document/d/1Q__9_O-hfYV4Roqy2xW7XWfvhlmPNbeTGbb05H6nFSs/edit?usp=sharing

"CSC207: Merging and Pull Requests":
https://docs.google.com/document/d/1HdASRicsjV16Nl9aZDsAZ92d8kwJOYiO1ln72vLCz48/edit?usp=sharing

Quick pull request activity

https://github.com/Daniel-Laufer/csc301-w23-tutorial-1