cscc01 (summer 2021)
introduction to software engineering

tutorial 1

git flow and more!

tutorial outline

01 git

02 merging and rebasing

03 git flow

04 intro to ci/cd

05 pull requests

info

attendance

tutorials will rotate between workshop/project every other week, attendance is mandatory for both

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.

timeslot signups

signups are made through a google form on the course website.

acknowledgement: material for this lesson is inspired by Krish Chowdhary's Advanced Git Workshop!

git is a 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 B07 and its basic commands (add, commit, push, pull, clone)

typical git usage

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

1. **branch off main and create your feature branch**
2. implement your feature
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

rebasing

modify, mutate and move commits on your branch

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

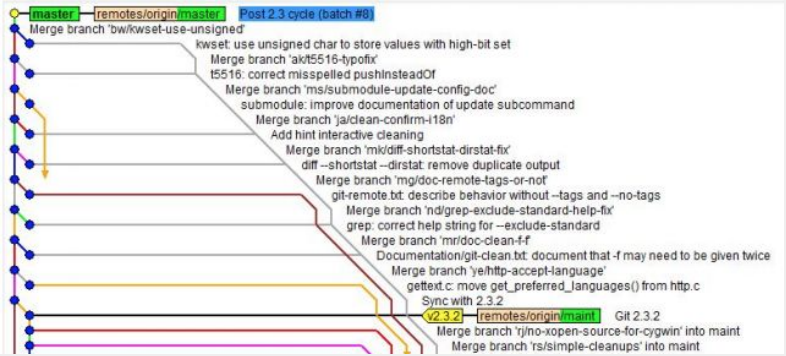
non-destructive, doesn't alter any existing branches ✓

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

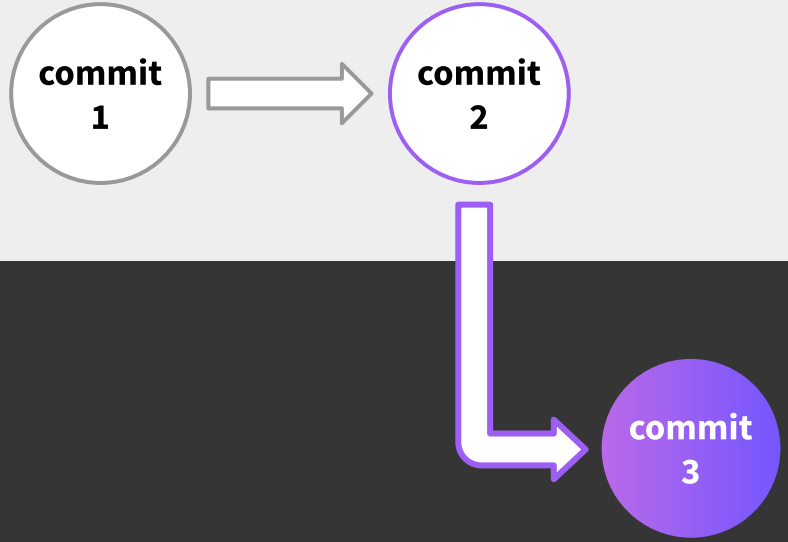


git merge

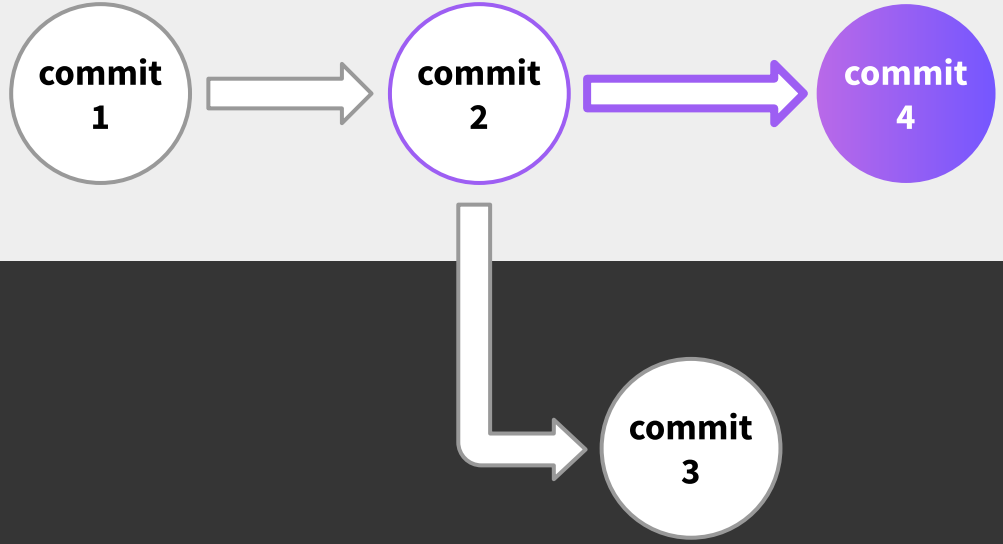
master

new branch

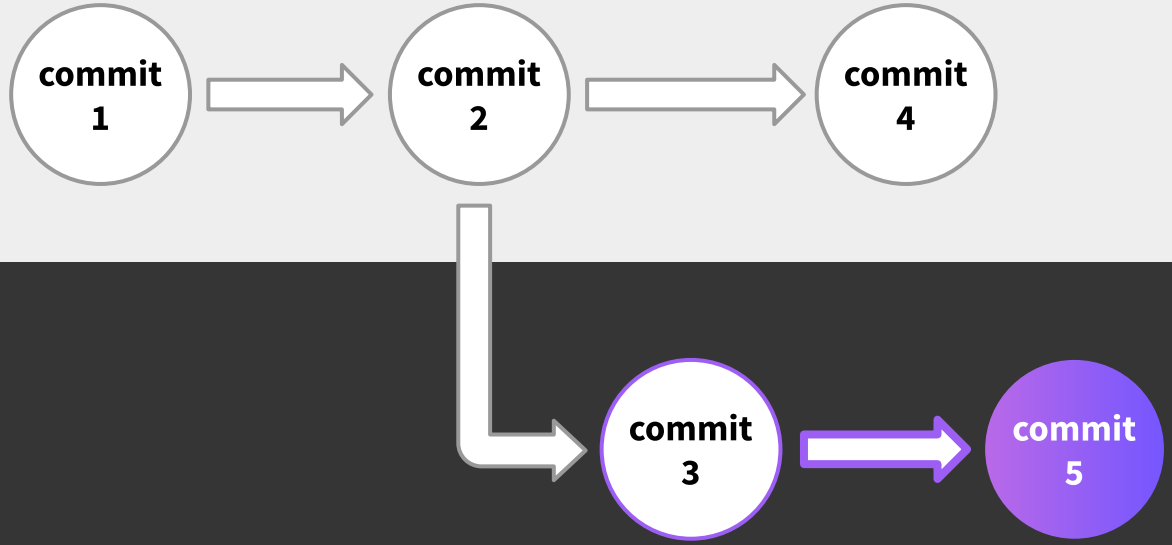
git merge



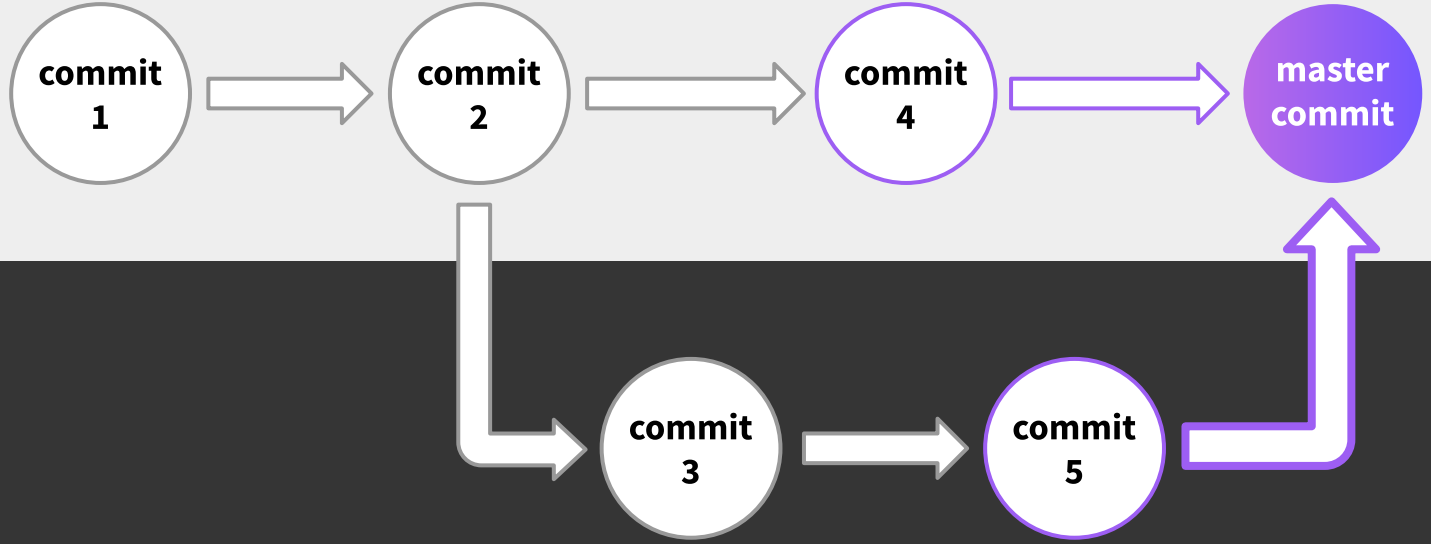
git merge



git merge


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

git merge


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

rebasing

```
> 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)

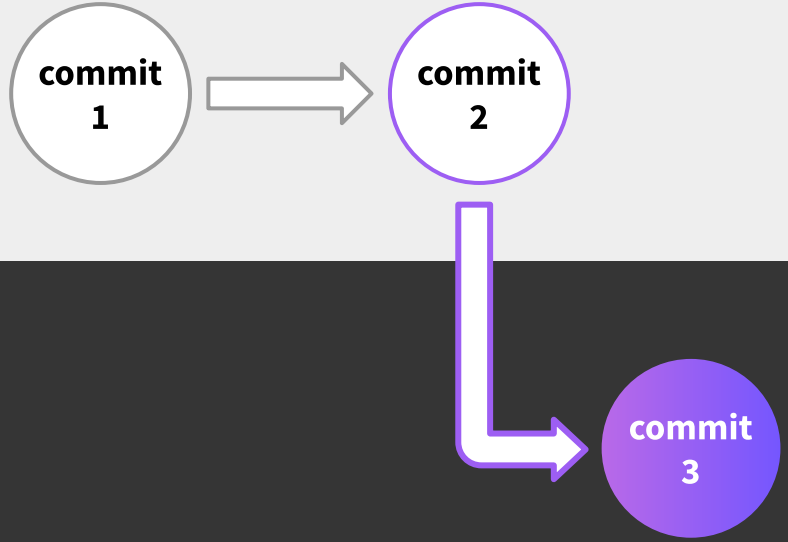
cons

✗ **requires caution**

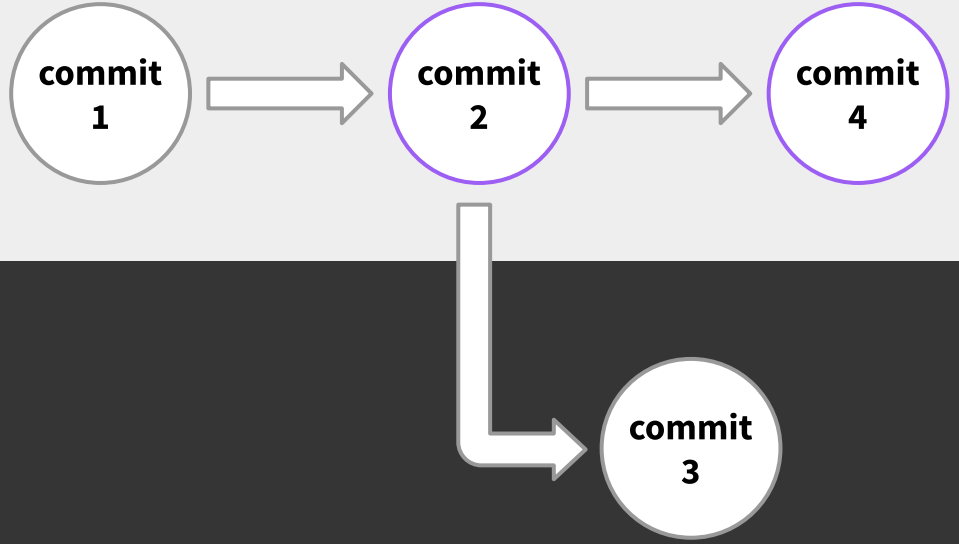
- Never rebase a public branch onto your feature branch
- This will result in two different versions of the master branch, which will need to be merged back together

</>

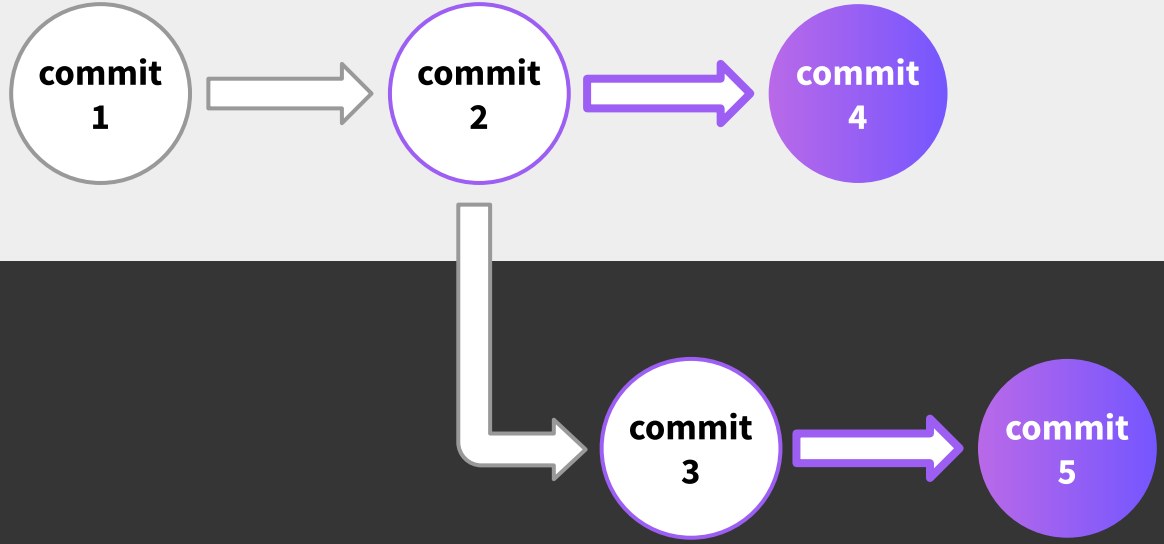
git rebase



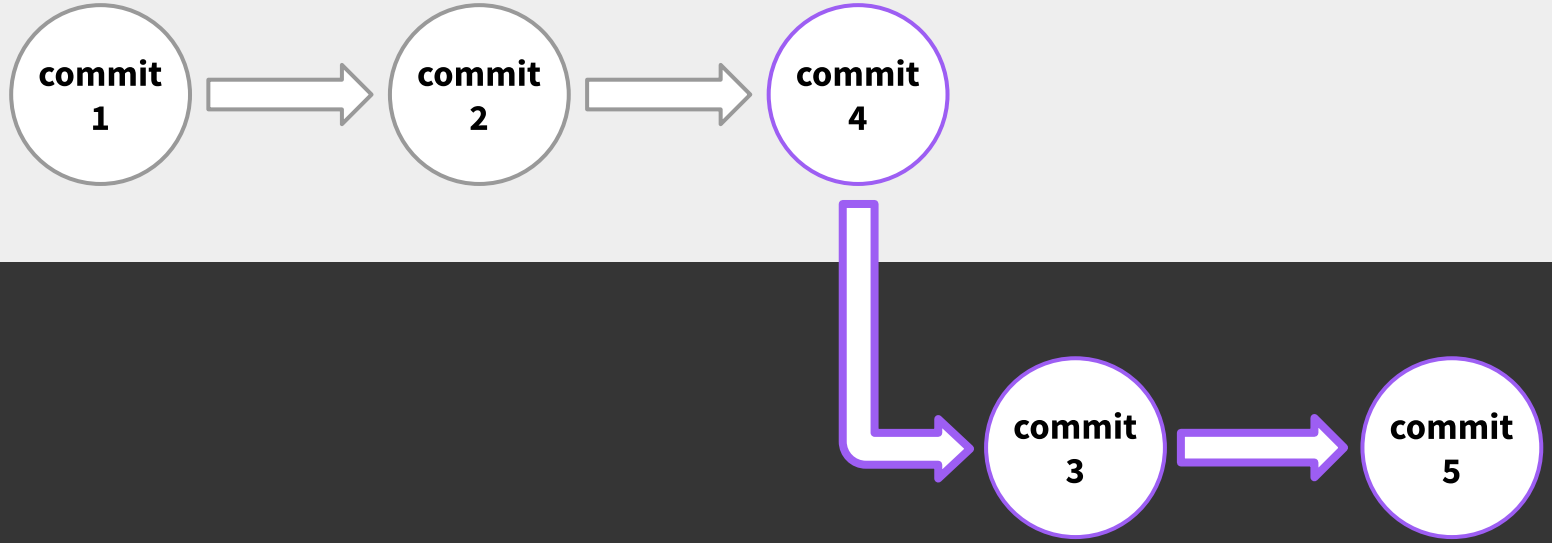
git rebase



git rebase


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

git rebase


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

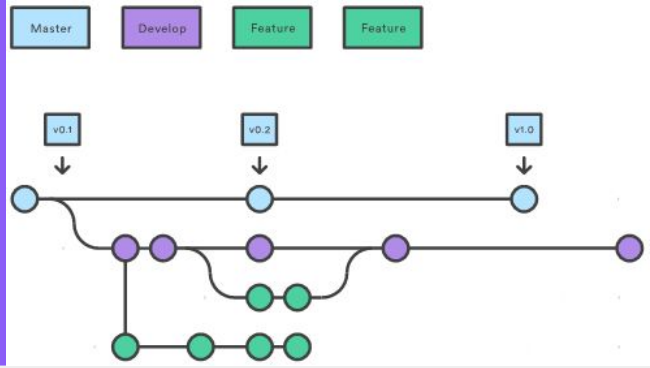
- Run git status to see the modified files
- `git restore [files]`

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!

ci/cd

continuous integration/continuous delivery (ci/cd) is a software engineering philosophy where building, packaging, testing, and deploying applications is automated consistently.

example: a frontend site automatically deploys to a hosting server after every push on the main branch

(we will go further into detail about ci/cd in a later tutorial)

git flow + ci/cd

because of how our project is setup, git flow enables engineers to:

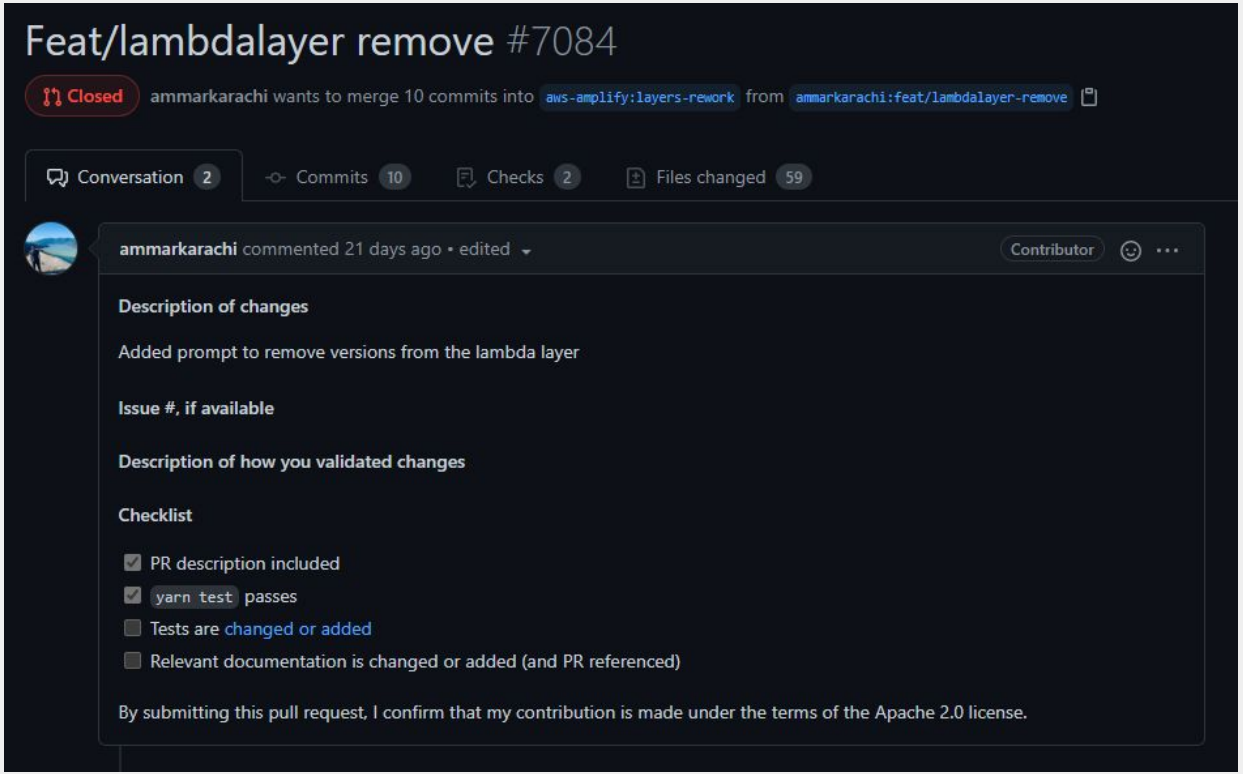
- automatically run the codebase’s test suite for all changes introduced
- automatically deploy software in main branch to production
- deploy the version in develop branch to staging servers, where developers can integrate and test features
- implement small changes frequently

pull requests

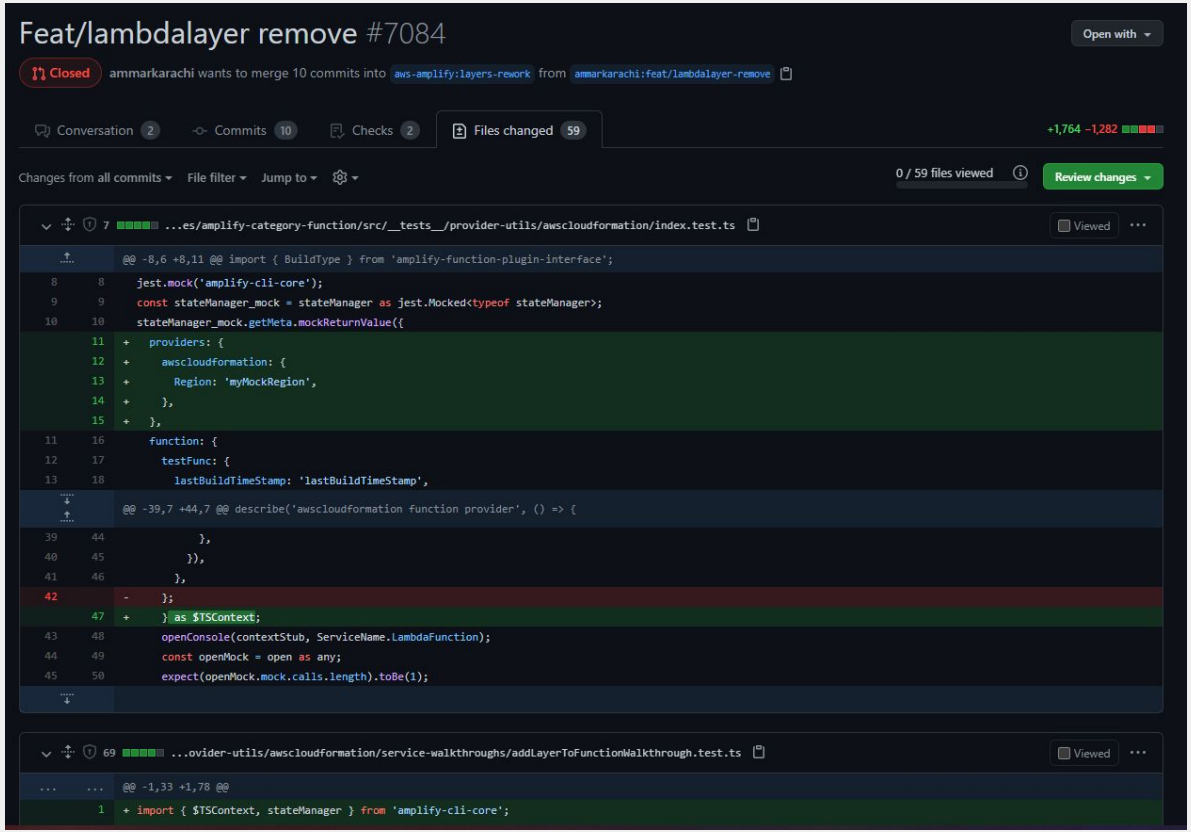
the crux of the git workflow

- 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



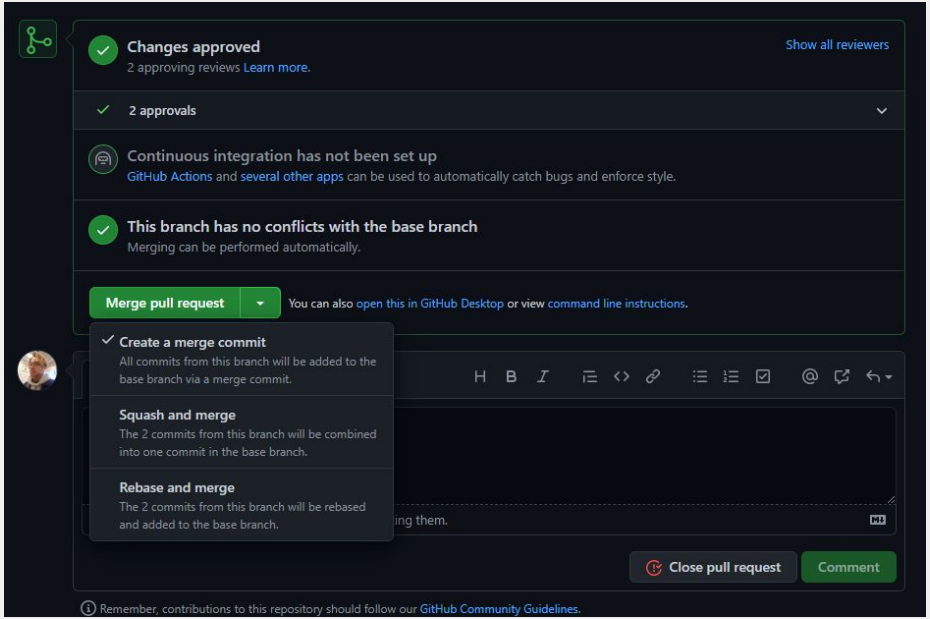
pull requests



merging in pull requests

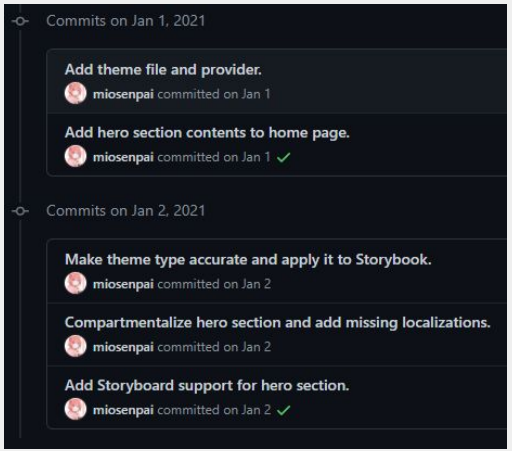
after being approved, use the github UI to merge your pull request! we recommend squashing and merging your commits so that all your changes are added to develop as one commit.

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**



- **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