DudeWheresMyBike:  
Branching Strategy

Prepared: 13/02/2022, TK

Revisions:

# Branching strategy based on “Git Flow”

* The *main* branch always reflects the current production state (the main branch is “live”).
* There is a second long-running branch we’re calling *dev*.
* All feature branches start from *dev* and will be merged into *dev*.
* If we had the time to do it separately, *dev* would be the starting point for new releases: we would open a new *release* branch, work on that, test it, and commit all bug fixes on such a release branch. Once everything works and we’re confident that it’s ready for production, we merge it back into *main*.
* As the last step, you add a **tag** for the release commit on *main* and delete the *release* branch.

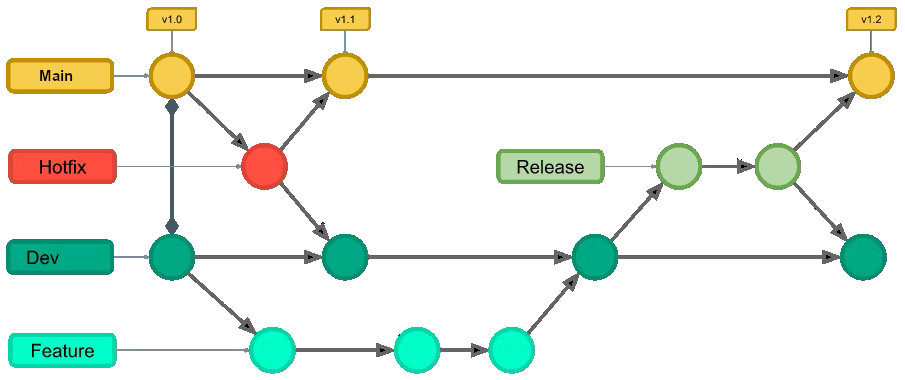


Fig. 1: Git Flow Branching Strategy

Team Motto: branching early, branching often, committing early, and committing often.

## Wait… why is there a branch “origin/Head”?

## We’ve just said that we’re going to use branches called “main”, “dev”, “feature/nnnn”, etc.. So why - when I am managing my branches in VS Code - do I see “origin/HEAD”?

## 

## It turns out “HEAD” has a few additional uses. HEAD is a pointer to the "current" commit, it is also the commit you get by default when cloning that repository. In *remote* repositories, even though they don't have a worktree\*, HEAD can still exist as a pointer to the "default" branch – the one that clones will check out automatically. For example, some projects point HEAD of the "master" repo to a 'dev' or 'stable' branch instead of master, so that people get something more useful by default.

## Git also keeps the <remote>/HEAD ref around because it has *yet another* use: it defines the branch Git tools will try if only the remote name is specified. So if you go to clone a repository and don’t specify “what branch” then Git automatically looks for origin/HEAD and gives you whatever branch *that* points to. This is settable using git remote set-head origin ... to whatever branch you find convenient.

Long story short - you can ignore it. Just be aware it’s there ‘cause… reasons.

## \* A Git worktree is **a linked copy of your Git repository**, allowing you to have multiple branches checked out at a time

## Naming Conventions:

main - Main branch

dev - Development branch

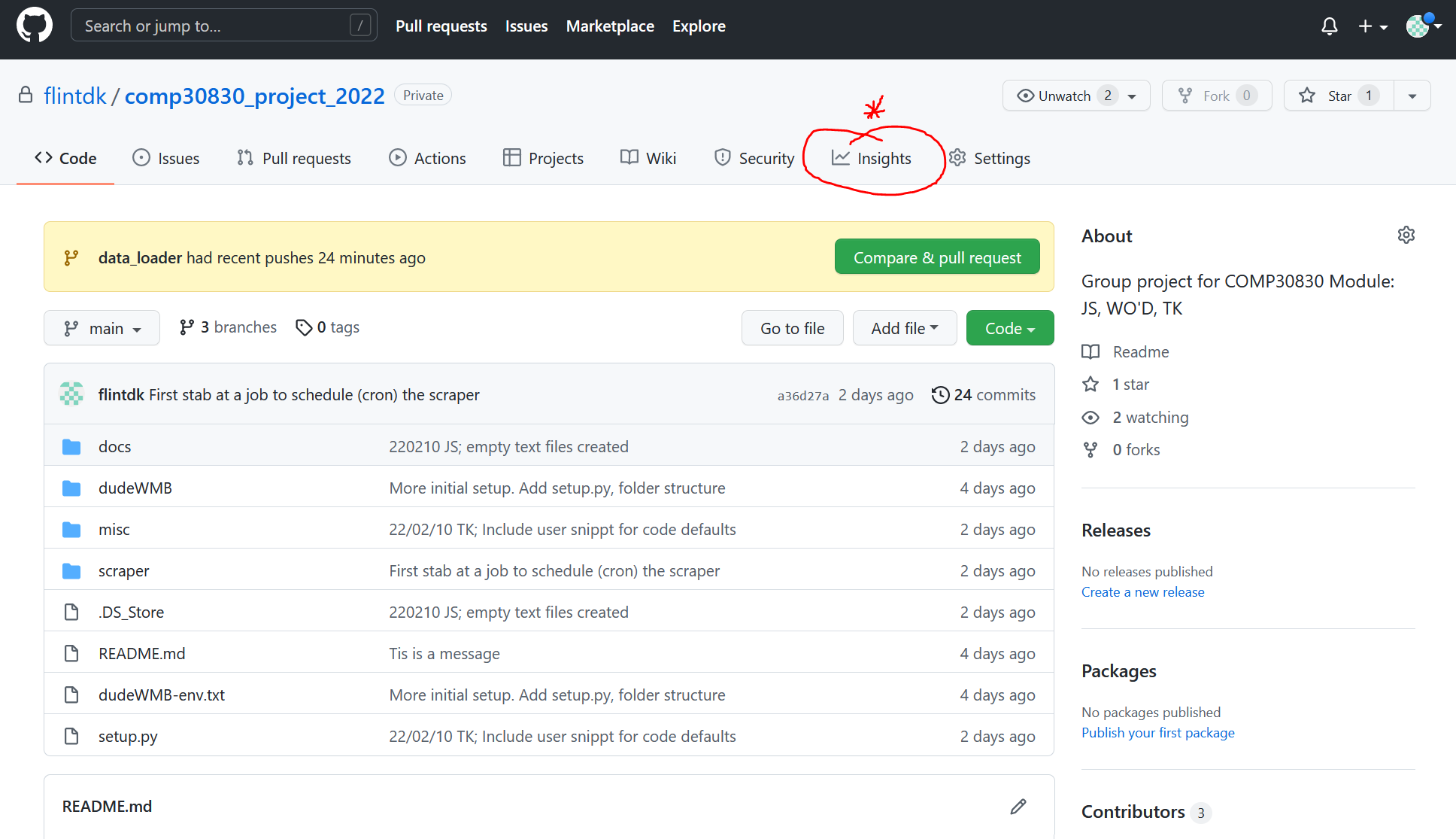
feature/*<feature-name>* - Branch for development of feature called “*<feature-name>*”

hotfix/*<issue-number>* - Branch for emergency repair of bug “*<issue-number>*”

## 

## Looking at current state:

Understanding where your project is at can be really helpful when trying to work out where to branch, tag, commit etc.. You can see a visual representation of the state of your project by looking at the “Insights” section (see below).

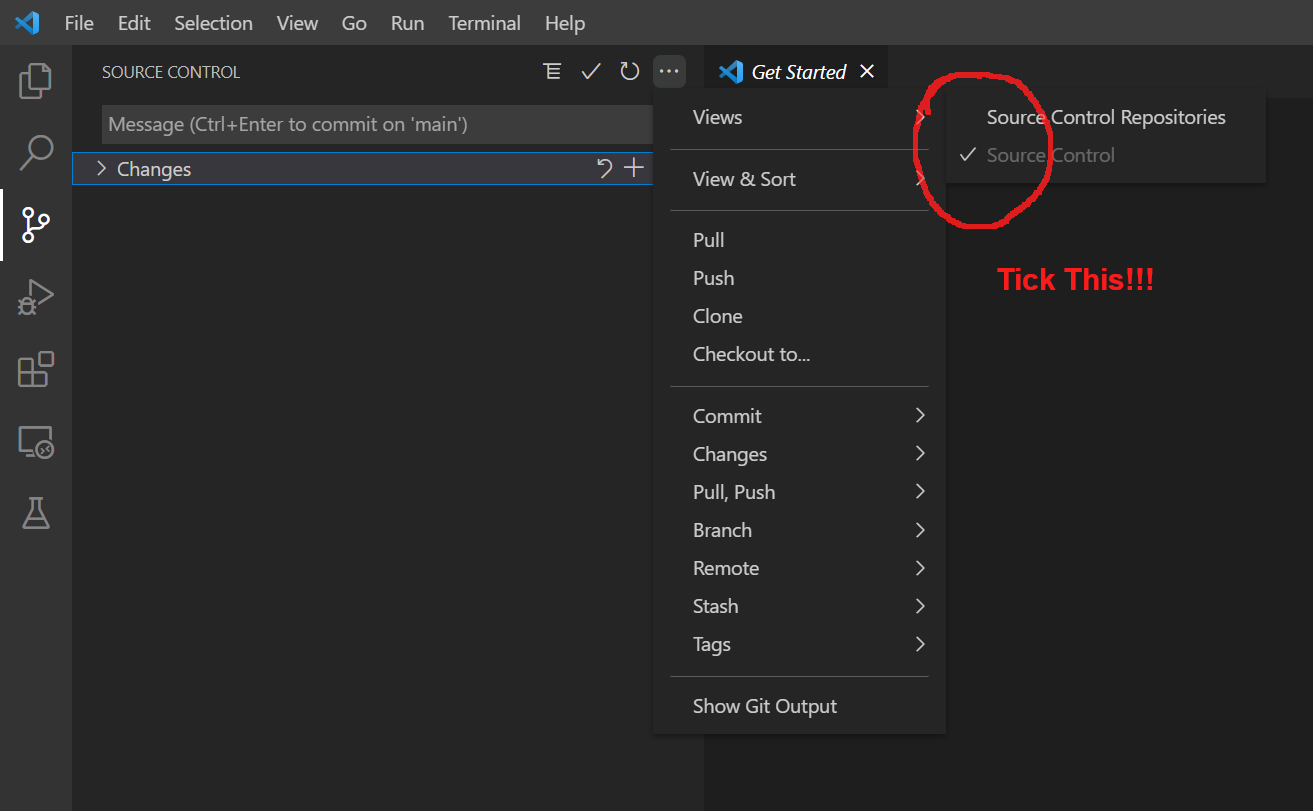


The Insights Button

Then in the left sidebar click “Networks” to get a graphical representation of what the project looks like. I don’t have great screenshots yet - as our project lookslike a flat line. We need some actual work committed before it starts looking like the pretty pictures on the branching strategy page. TODO: Add a picture…

## VS Code: First Steps

Make sure that View: Source Control Repositories is ticked in Source Control View. It’s not by default. It’s confusing without it.



## Warnings:

As with all source control that I’ve used so far:

DO NOT delete one branch, then rename another branch to be the same as the deleted branch. I did this on purpose to see what happens. What happened was not good. GIT got completely lost and my local repository became detached in some way from the remote repository with some changes happening locally, some remote. It was a mess. I had to delete my local repository and clone it again to sort it out. Be warned. Don’t do mad stuff with names.

## Cloning a Repository / Starting From Scratch:

NOTE: I’m already authenticated to check out the repository in this example. If you are not - and you’re cloning a repository you’re not yet authenticated for - there may be some additional steps.

With no folder open in VS Code, select the Source Control View on the left and then click “clone repository”.

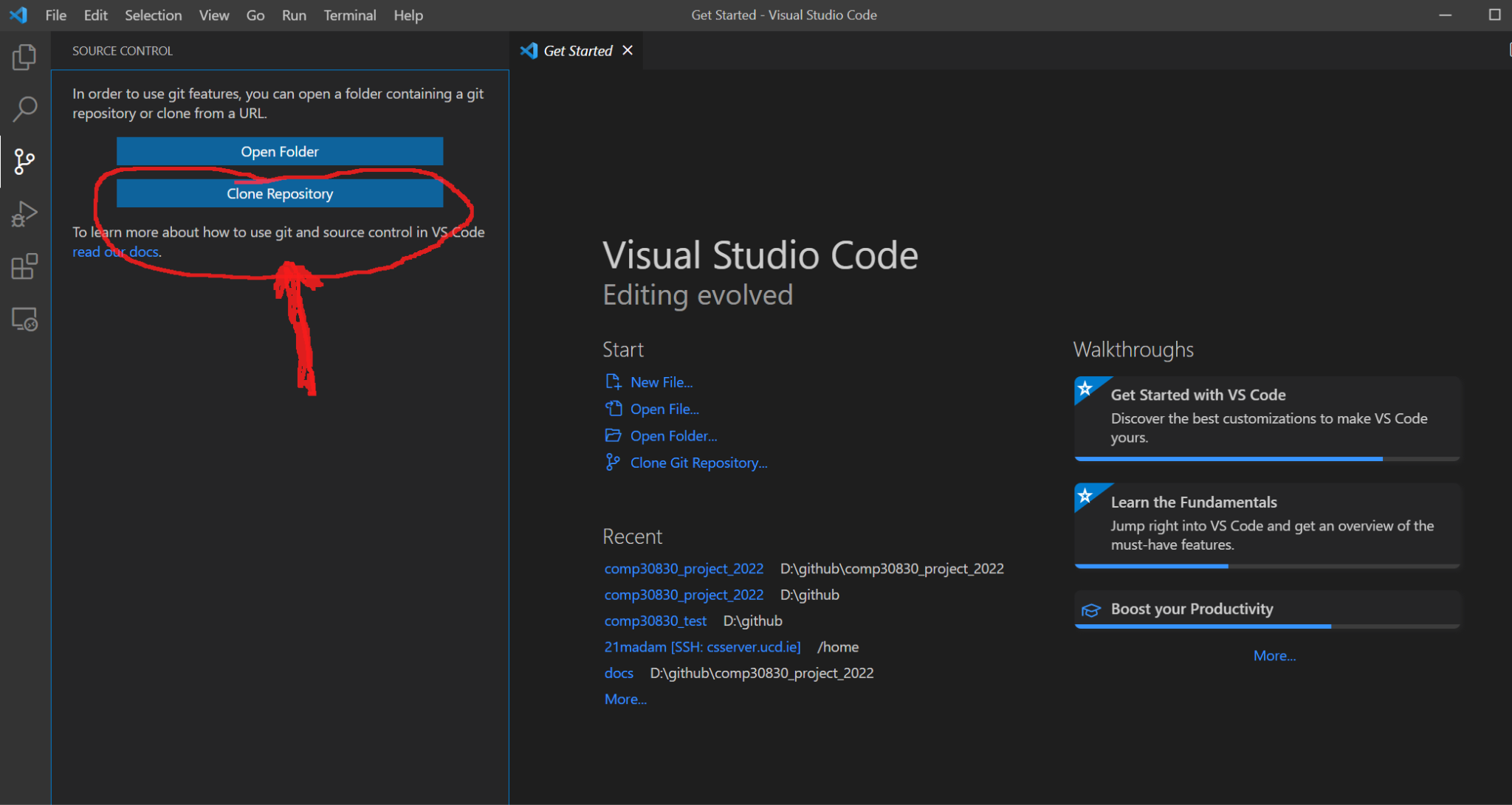


Fig. 2: Clicking ‘Clone Repository’

A small dialog will appear that says “Clone from Gitgub”. Don’t click that…

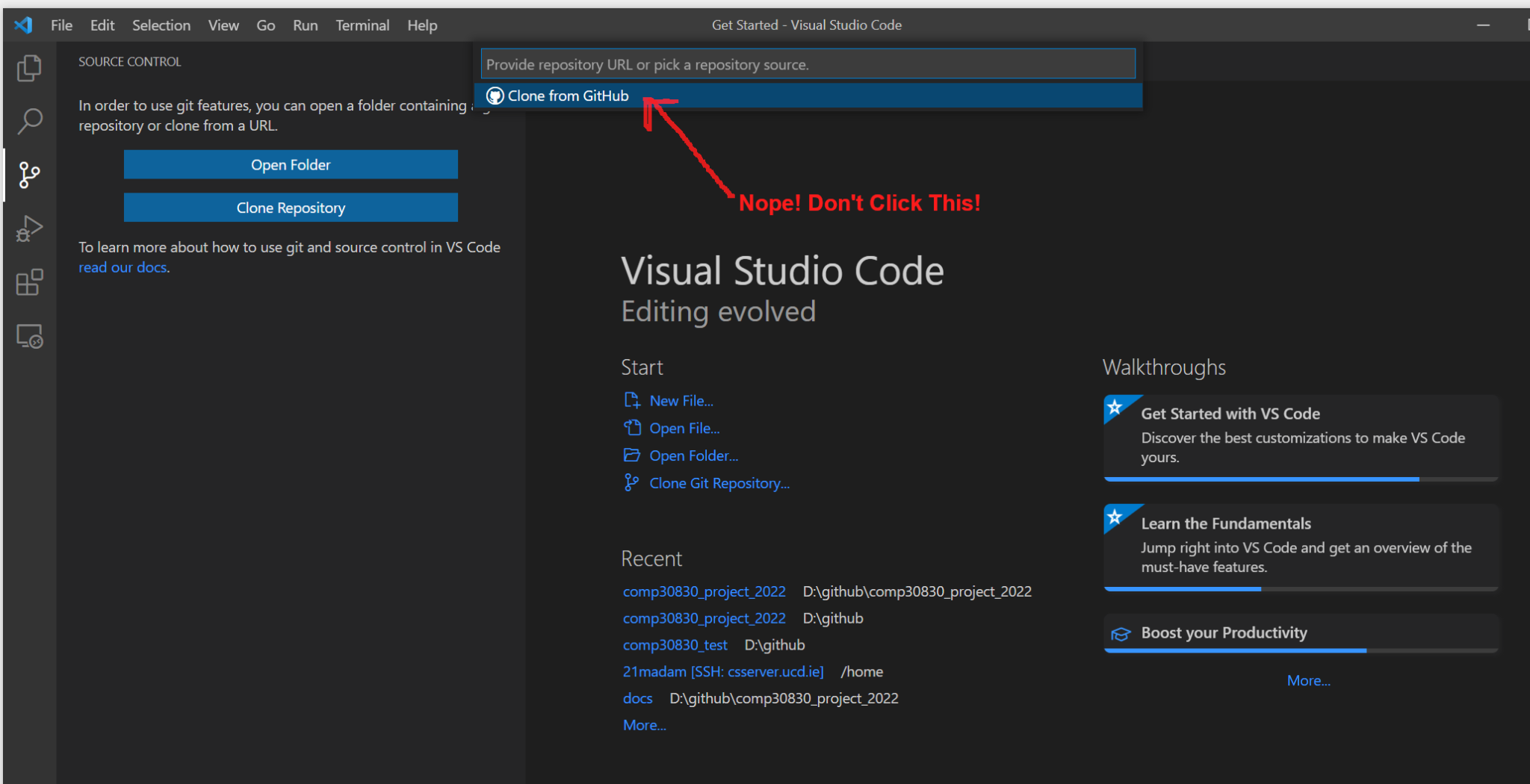


Fig. 3: Booby Trap - Don’t Click the Link

Instead just paste the URL into the text box and a new option will appear to clone from that URL. That’s the one you want.

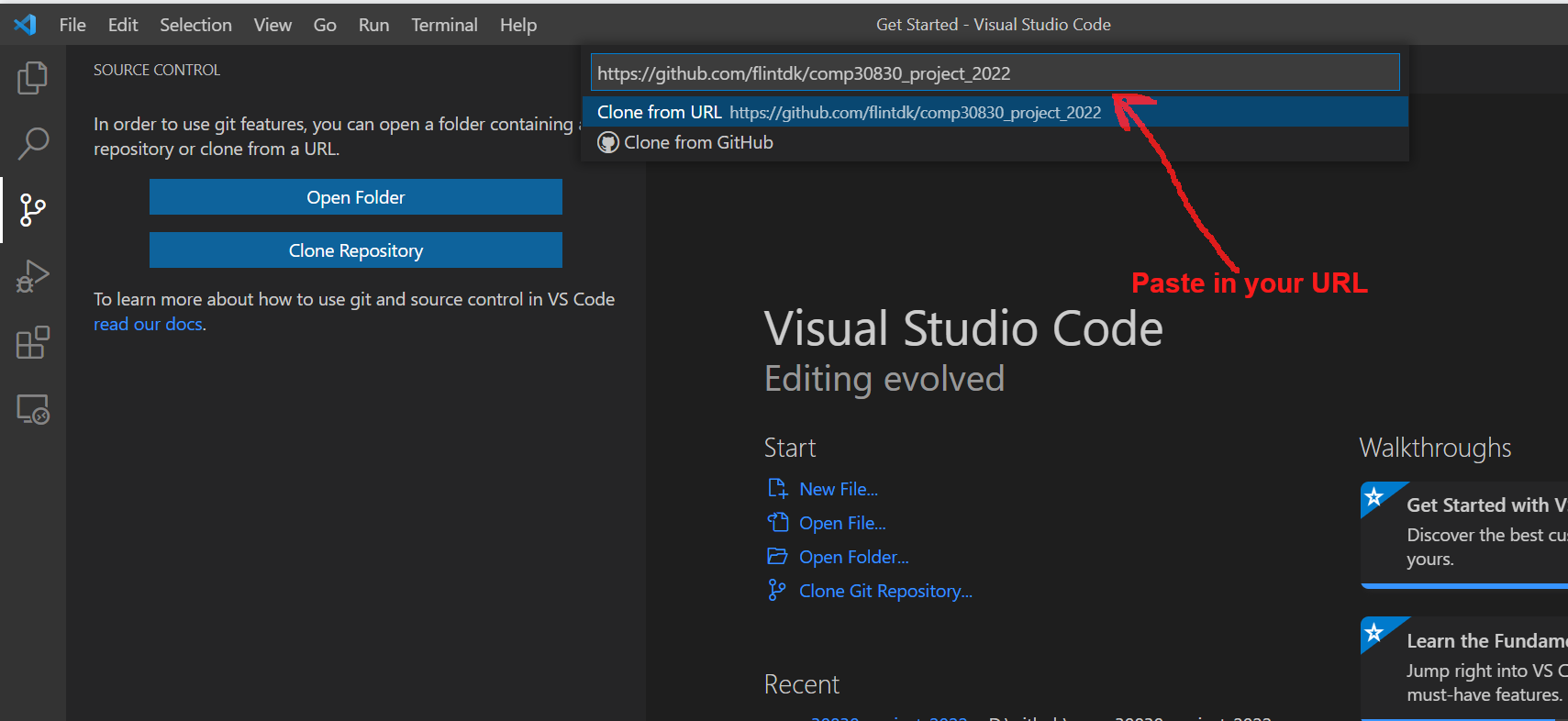


Fig. 4: Clone from URL

When you hit enter you get a standard folder chooser dialog box. You just select the folder on your PC where you want the repository created (I use D:\github on my PC). A new folder for the repository will be created in the folder you choose and all the files will be placed in there.

You’re done! You have your fresh clean copy of the remote repository.

## Making a branch

We want to make branches, check stuff out, mess around, but keep it simple, right? Well - Git is not simple. It has an extremely flexible architecture that lets you branch locally in a repository (and not track remotely), to branch remotely and not track locally, etc.. We don’t want the fancy stuff. What we want is:

-> If we create a branch locally to create one remotely and for it to track.

Luckily - this is “sort of” the default behaviour \*\*\*as long as you’re careful when you create your branches\*\*\*. There are lots of ways to create branches. What follows works. If you deviate from the below/take a shortcut/whatever then the best of luck with it! But you’re on your own….

Let’s say we want to make a feature branch (dev and main already exist). In this example I’ll create a branch called “feature/myFeature”. This branch will be from the dev branch and the plan will be to merge it back there later. First: make sure you’re on the dev branch. Select the Source Control view from the controls on the left and double check (see fig. 5). If the branch doesn’t say ‘dev’, then click it and select ‘dev’ to switch to the ‘dev’ branch.

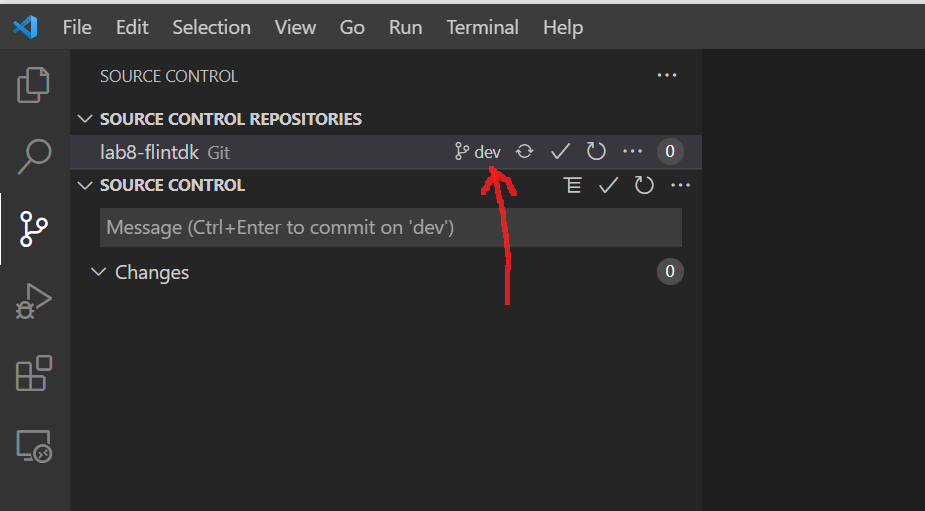


Fig. 5: Make sure this says ‘dev’

Next, on Source Control Repositories take the option to “create a new branch from…”. In general you’d think you could just take the option to “create branch” and it would create a branch from the current branch. But I had some odd behaviour when testing this so play it safe - take the option “create branch from…”

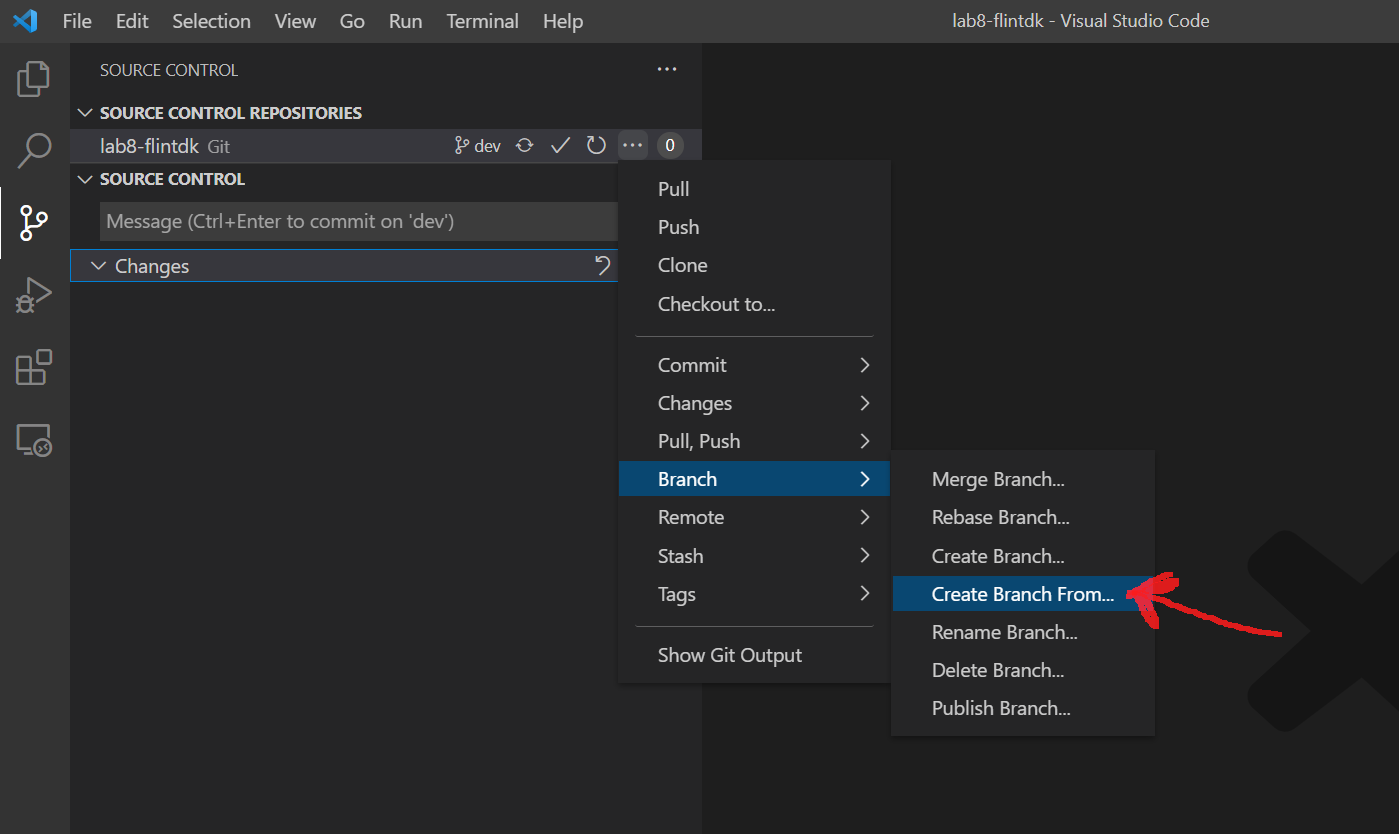


Fig. 6: Create Branch From

You’ll be asked first to enter a name for your new branch. Go ahead and enter one (I entered feature/myFeature)

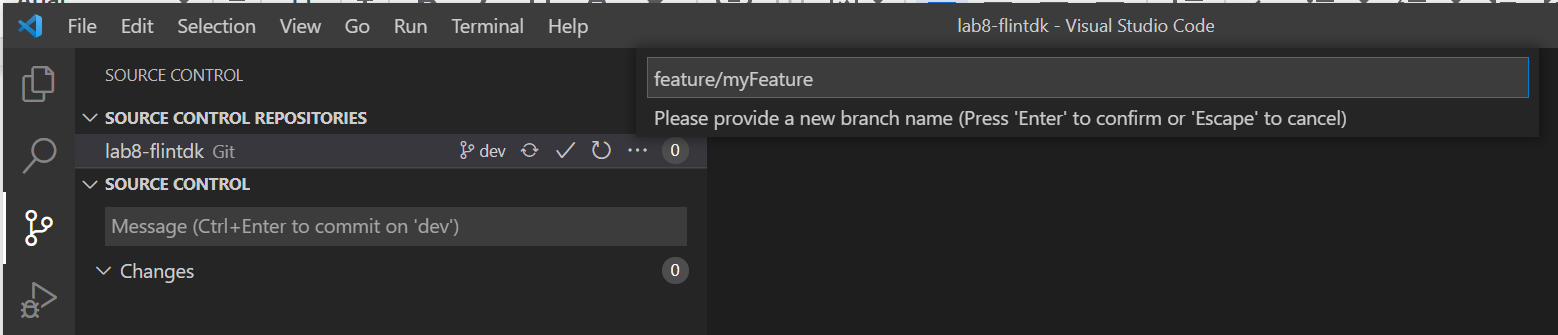


Fig. 7: Naming your new branch

Once you’ve entered the name for your new branch you will get a second dialog box asking you to choose where you want to make your new branch from \*\*\*this is the important bit\*\*\*. You need to select \*\*\*the remote ‘dev’ branch\*\*\*.

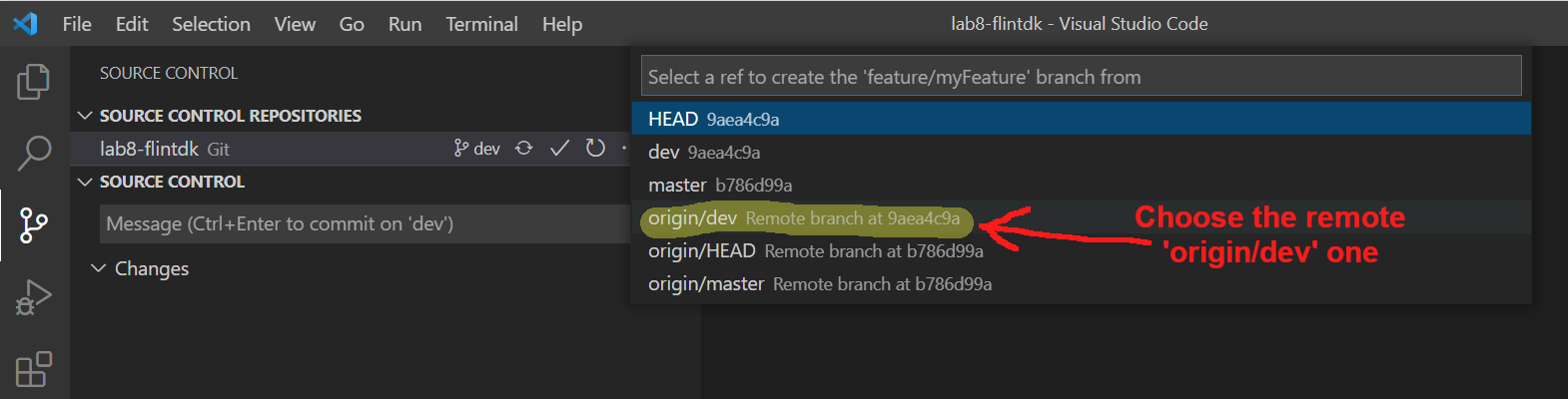


Fig. 8: Branch from the remote ‘dev’ branch

Once you’ve done all that just click the massive publish branch button and you’re done!

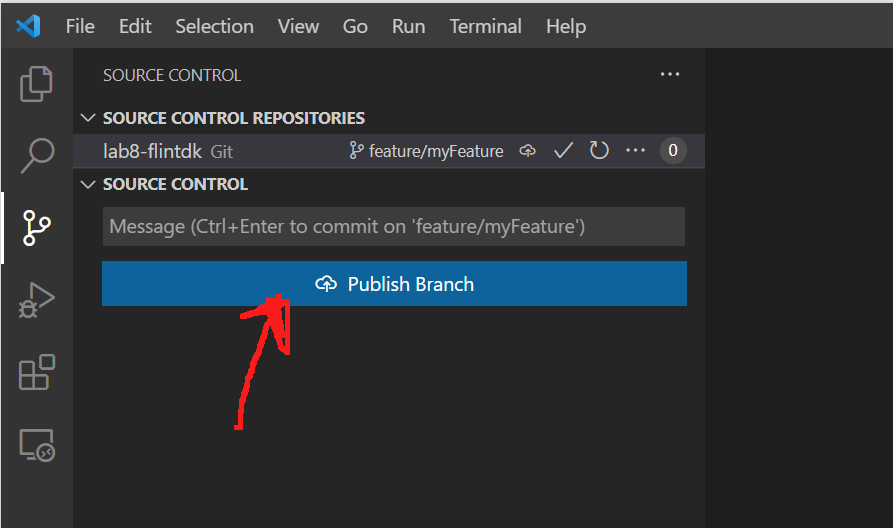


Fig. 9: Publish Branch

You should be able to see your branch on github, everyone else should be able to see your branch on github and most importantly when you commit to the branch locally VS Code will know that you want the changes to track remotely too.

At this point I went and did some test commits just so I had some activity on my new branch. I created some new files. But I also updated some files to make sure there would be a clash when I merged stuff back to dev later.

## Merging a branch.

This is the part about source control that everyone hates. But it doesn’t have to be insurmountable. Good branching and commit cadence will help reduce the size of conflicts.

Branching in Git works by merging from a source branch into an active branch. For example, if you want to merge changes:

-> from a feature branch

-> into the master branch

… you need to checkout the master branch to make it active and ***then*** select the feature branch as the source. You are essentially “pulling” changes from a branch into the active branch.

I wanted to merge myFeature into the ‘dev’ branch so I switched to the ‘dev’ branch (just go to Source Control and click the branch you want on the Source Control Repositories list to do this).

To begin merging a branch you must first select the branch you want to merge:

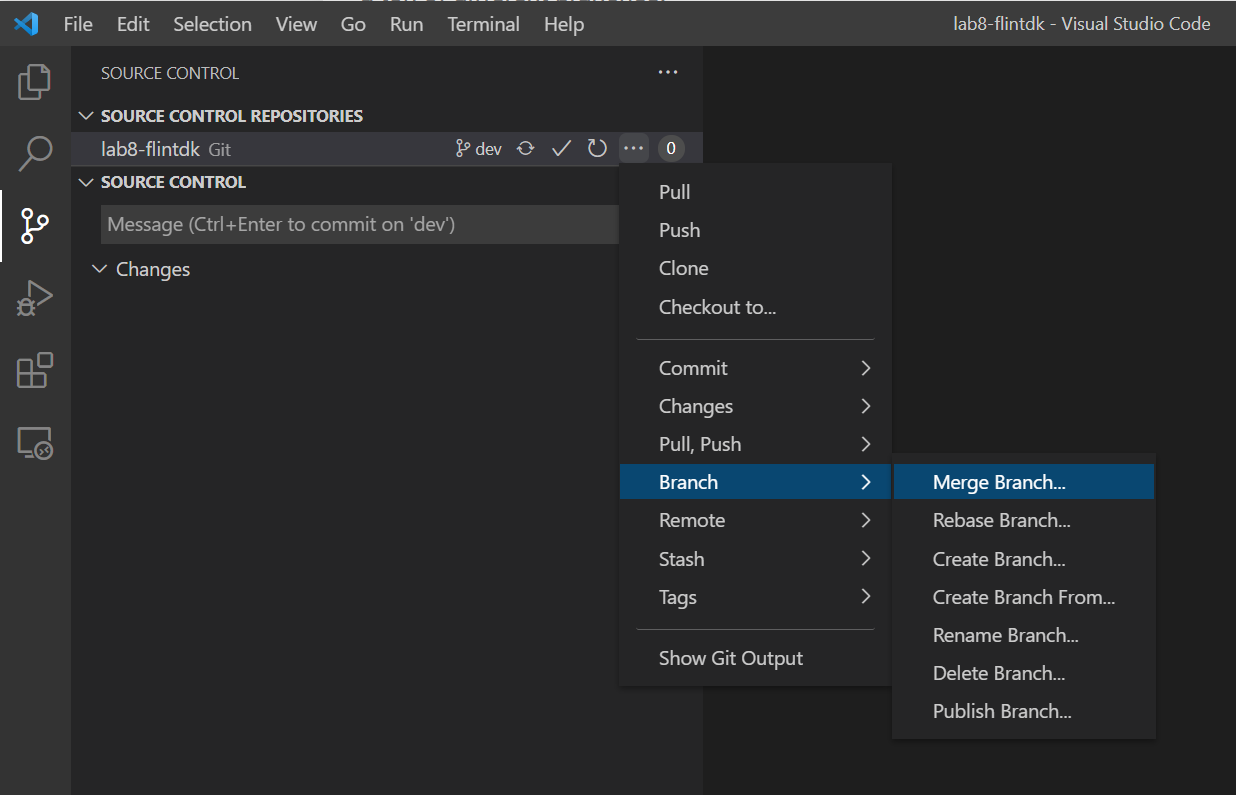


Fig. 10: Select the Branch you want to Merge

I found it safest to alway explicitly tell VS code “use the remote branch”. So you can see (below) that I selected the remote copy of my branch (‘origin/myFeature’). Other approaches probably work - I just decided to do everything the same way as it appeared to be working.

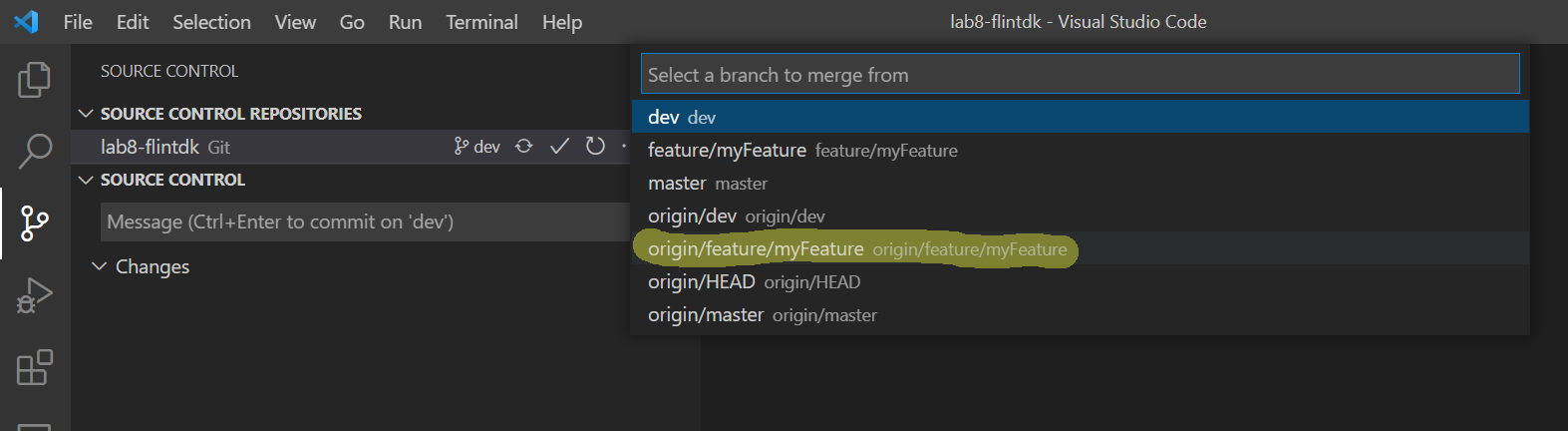


Fig. 11: Select the remote (‘origin/<name>’) Branch to Merge

Now for the fun. You’ll get a nice little notice on screen that tells you that you are merging remote tracking branch “feature/myFeature” into dev. This is great - we’re coming back to this section in just a minute.

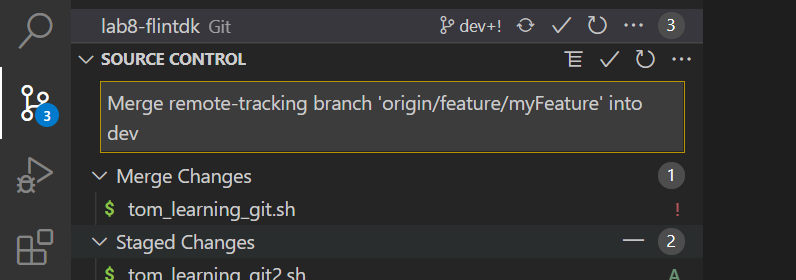


Fig. 12: Merging Information Message

However - you’ll also get a message - where applicable - telling you there are merge conflicts and prompting you to open Git Log to resolve them:

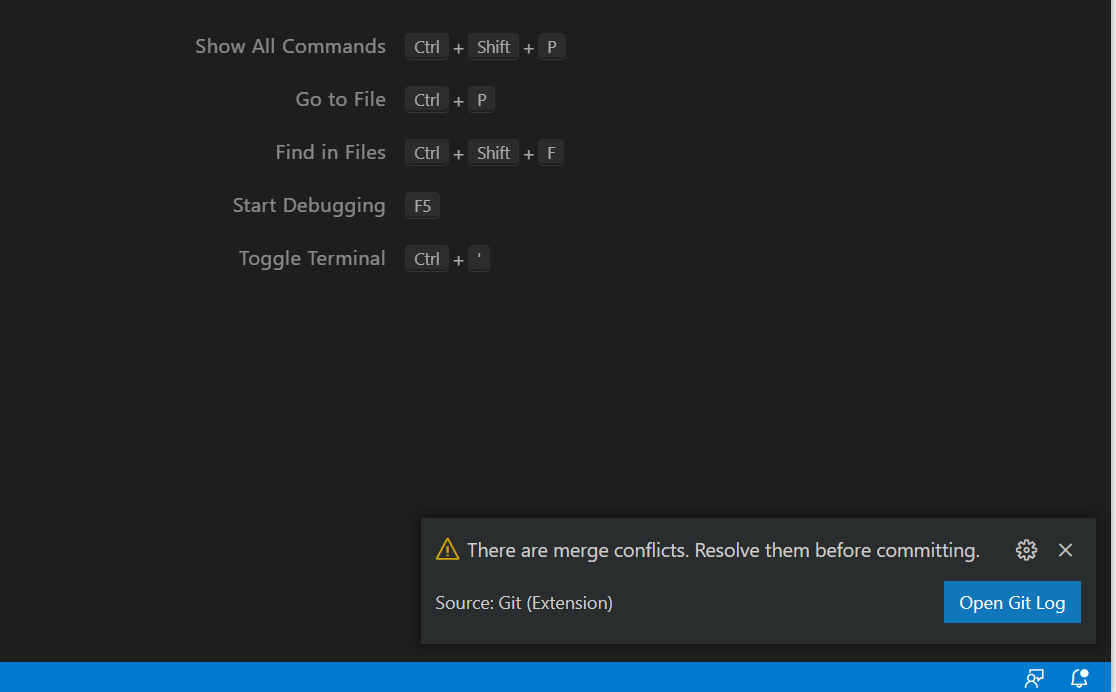


Fig. 13: Merge conflicts. Open Git Log

Opening Git Log… is pretty much useless. It literally opens up a Git Log text file. You put your nerd glasses on and go digging. Lovely. Just forget this.

OK - back to the Merging info. Message we saw above. This is where the magic happens. You’ll see a list of changes to the ‘dev’ branch that have resulted from attempting to merge the ‘feature/myFeature’ branch. Changes that don’t cause conflict just appear as ‘staged changes’. But your conflicts are listed as ‘Merge Changes’ and have a nice little red exclamation mark next to them, telling you you need to do something!

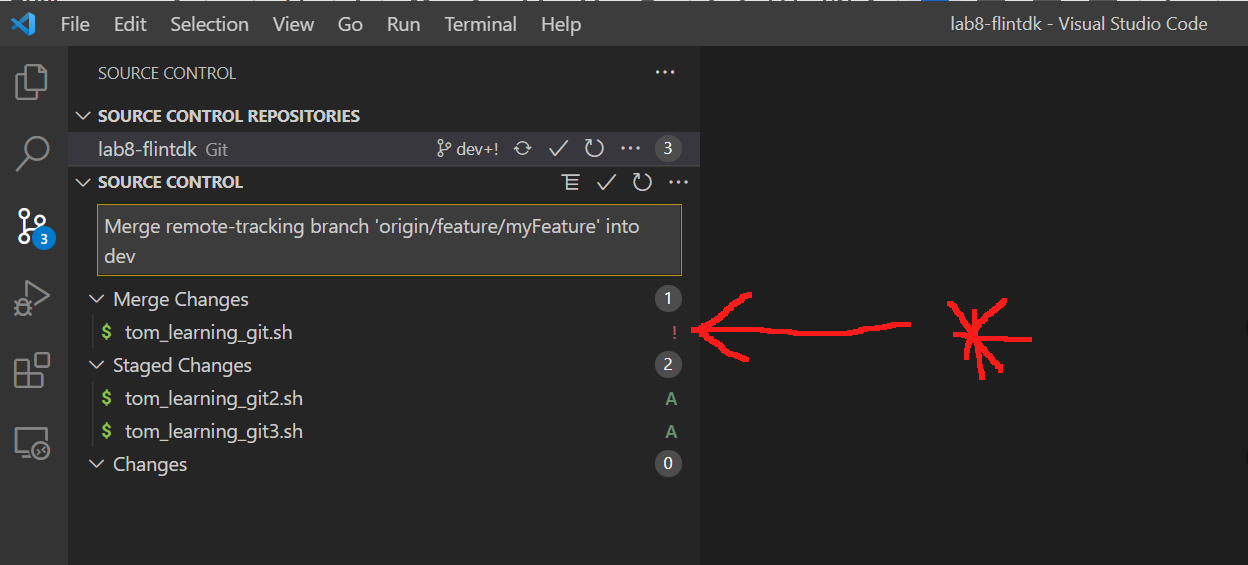


Fig. 14: VS Code Shows You where the merge conflicts are!

To get going, just click the file (or files) that have merge conflicts. An merge resolution editor will open. Sadly, VS Code does not use the nice “side by side” comparison editors that I’m used to using. It uses the github native view of the world. Tough luck buttercup - you just have to get used to this.

* <<<<<<< HEAD indicates the starting point of your changes.
* >>>>>>> [source\_branch\_name] indicates the end point of your coworker’s changes.
* ======= is a separator for the conflicting changes.

The screen will show you the original version of the code highlighted in green - and the incoming changes highlighted in blue.

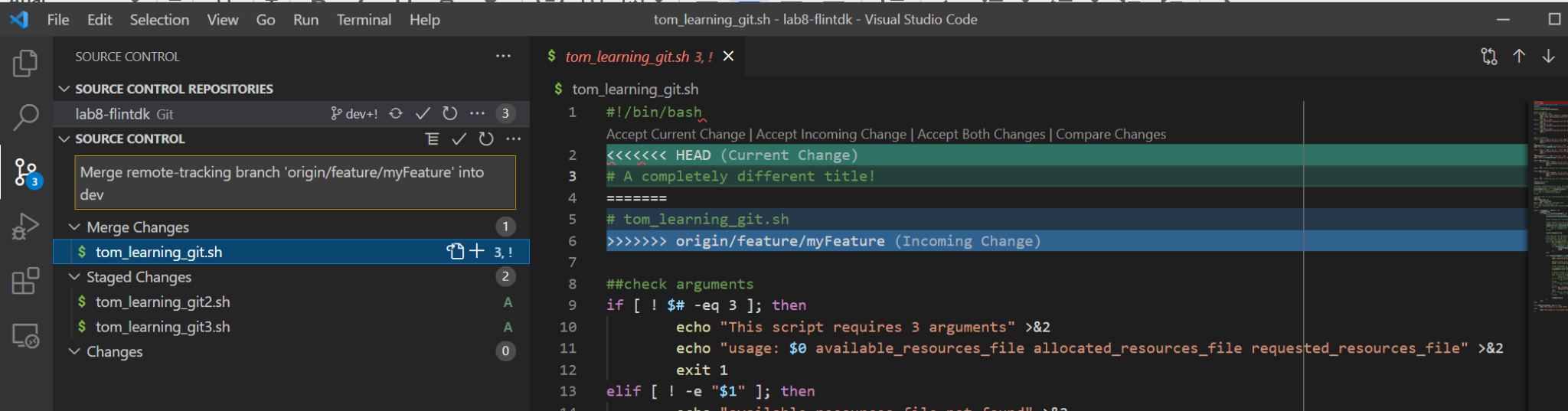


Fig. 15: Git Showing you a conflict on merge.

If the conflict is simple to resolve, you can manually edit the file to remove the markers and only retain the line that you want. Otherwise you have a few options per conflict.

The ‘nuclear’/quick options are:

* Accept Current Change to remove all markers and the incoming changes from the branch we’re trying to merge in (a.k.a. the ‘source branch’, or ‘feature/myFeature)
* Accept Incoming Change to remove all markers and the currently checked out branch’s changes in favour of the source branch
* Accept Both Changes to remove all markers and retain all changes

If you want to “Accept All Current” or “Accept All Incoming” you can use a right-click…

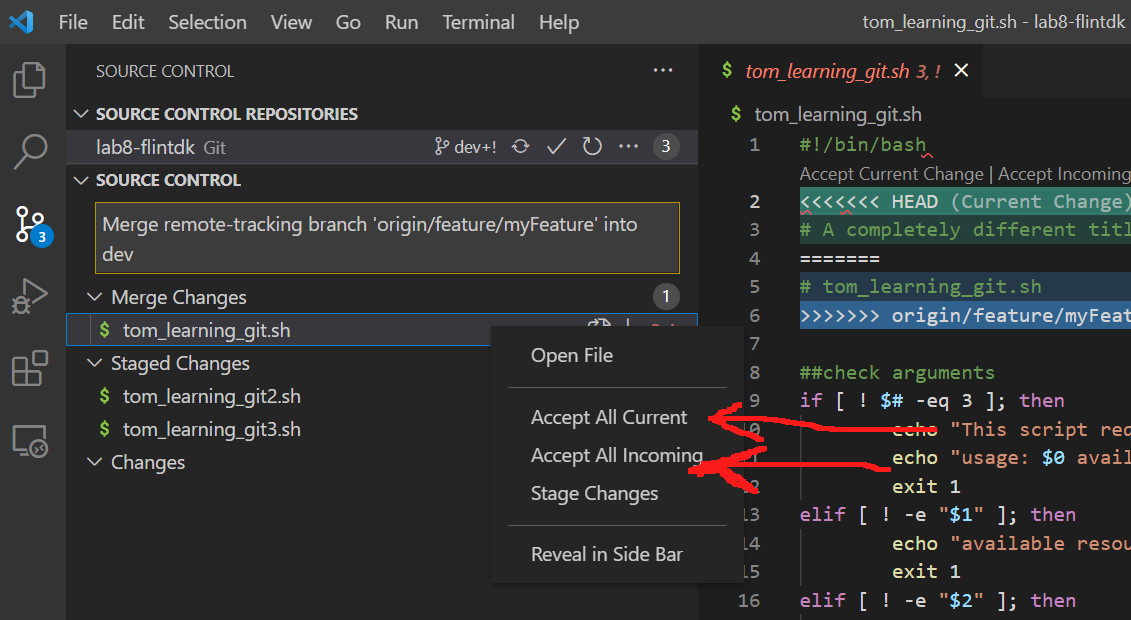


Fig. 16: GUI options for ‘Accept All Current’, and ‘Accept All Incoming’

But if you want ‘Accept Both Changes’ or ‘Compare Changes’ you need to look closer…

You can either use the command palette, for e.g.:

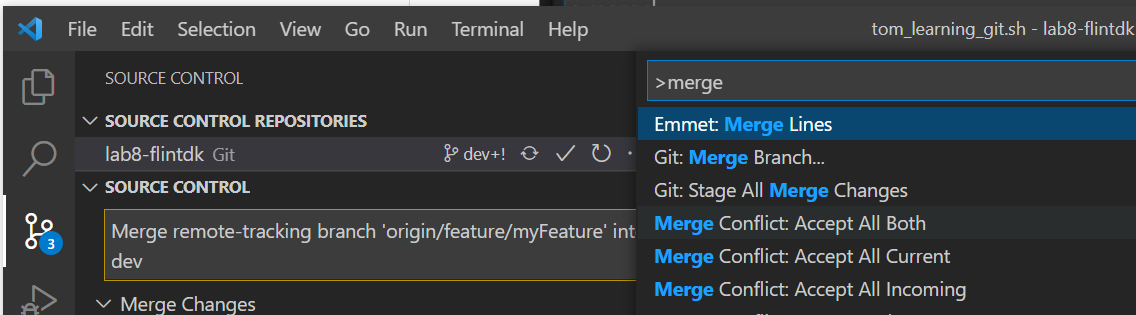


Fig. 17: Using the Command Palette for ‘Accept All Both’

… or if you look closely at the screen there are actually on-screen prompts above each conflict in a file offering you all the common options - like the one you’ve most likely been after from the very beginning ‘‘Compare Changes”. Compare Changes will open a diff viewer so that you can see changes side-by-side

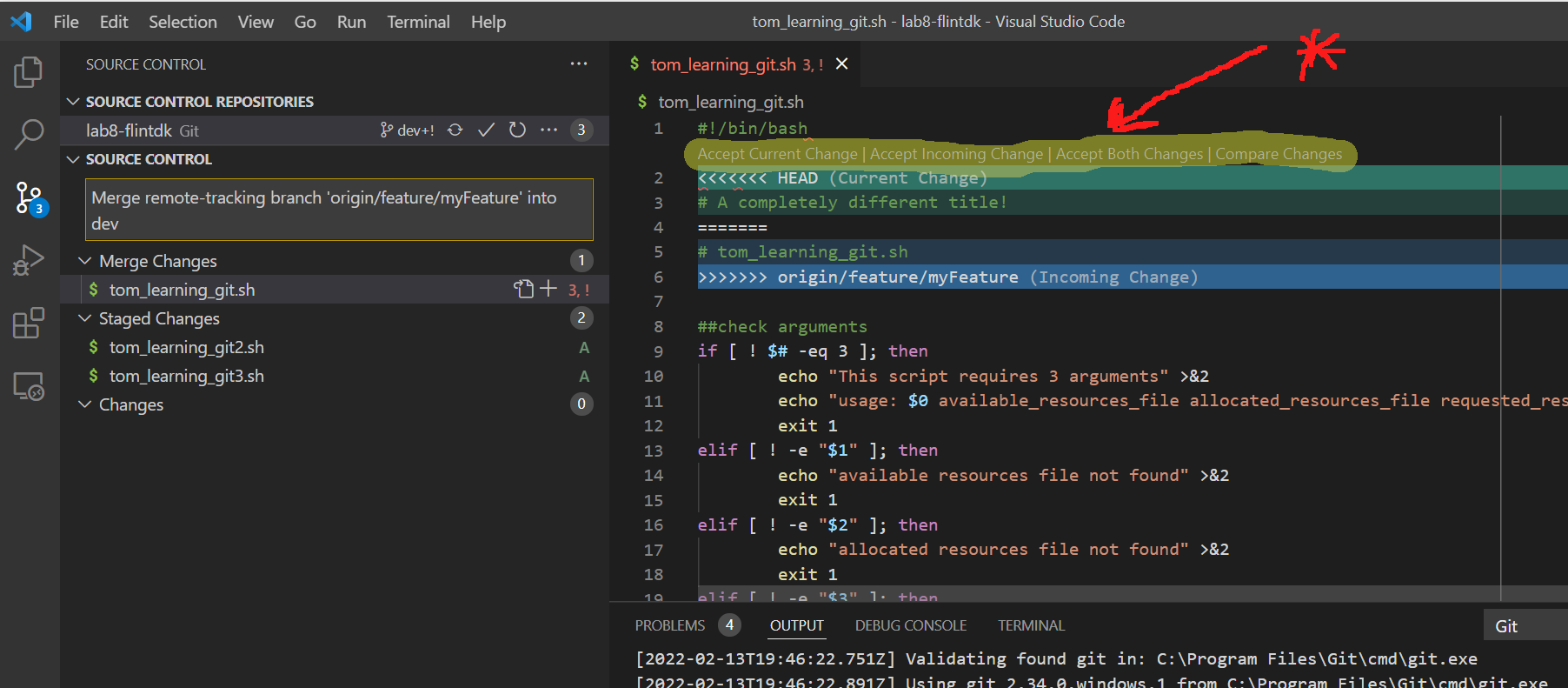


Fig.18: Where to Find the Common Merge Tool Links

The Compare editor is pretty basic - it just does a side-by-side comparison. However you get used to it and if you use both the Compare editor and the Git Merge view side by side it’s pretty easy to move through the conflicts, looking at them, taking action and moving on.

I think it’s a good idea to compare changes before making a choice so that you get in the habit of double checking.

After making your decision on all conflicts in all files, you can finish up the merge by saving the files, staging, and committing as normal!

## Troubleshooting

### Resolving merge conflicts using “Pop” and “Stash”

If you’re facing a merge conflict when Github won’t allow you to synchronise your feature branch with the dev branch, then the following might help:

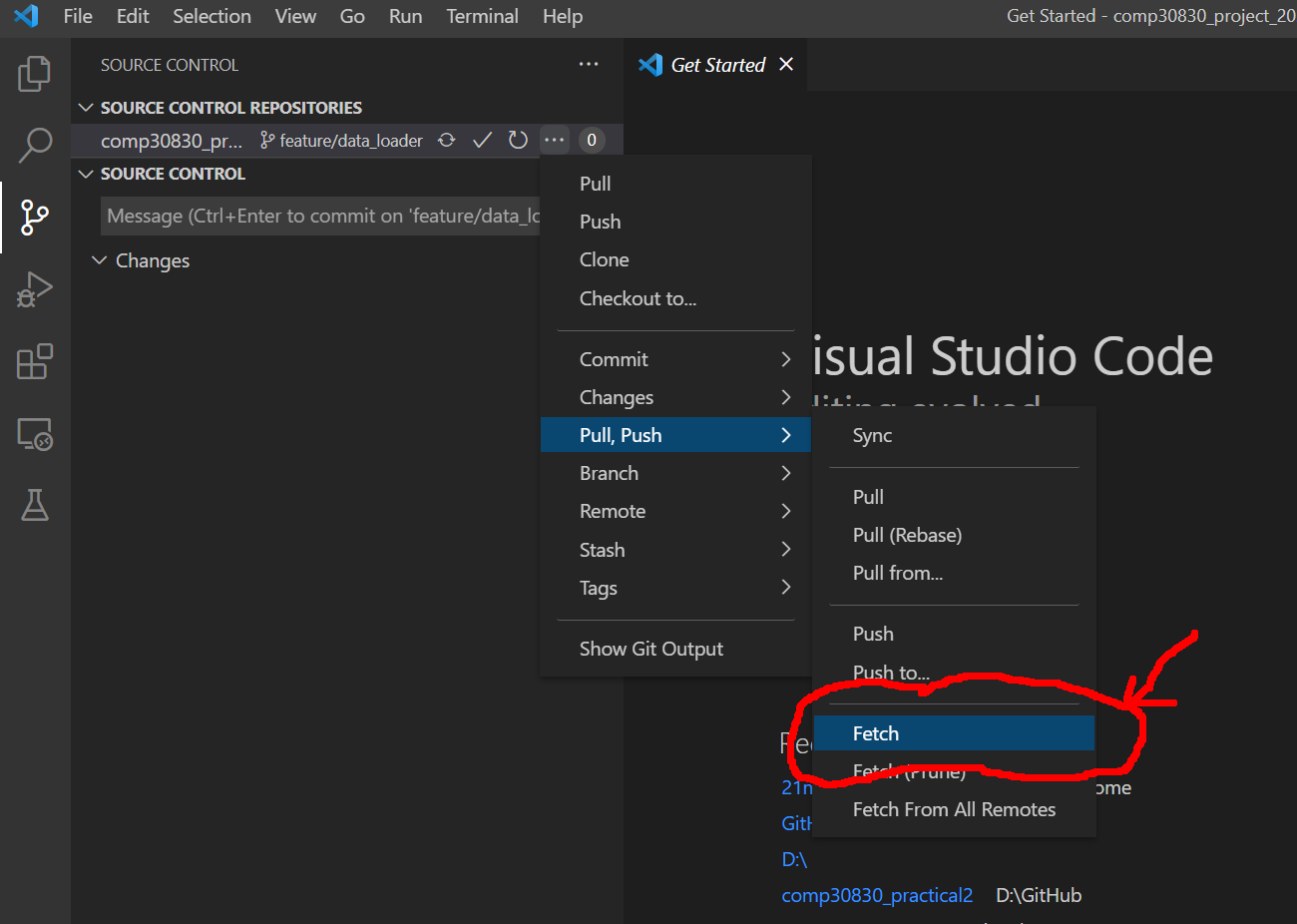
1. Stash the changes you have made to your branch
2. Synchronise your branch with the dev branch
3. Pop the changes you had made to your branch
4. Commit changes

Note: If anyone faces this issue, would be nice if you could throw in some screenshots

## 

## Tips/Tricks:

Wondering why a new github branch is not appearing in VS Code? VS Code doesn’t reload all the remote branches by default. Click the three dots beside the repository in question, select the Push, Pull menu and then select Fetch. That should refresh everything.



Fi. 19: Using ‘Fetch’ to Refresh Repository Branches in VS Code