Git Branch Workflows for Effective Collaboration

Taras Zakharko for the “Data Science for Linguists 2025 class”
University of Zurich

Introduction to branches

Suppose that you and your teammate are working together on a report. You are responsible for carrying out a new statistical analysis while your teammate is rewriting the introduction. To make things simpler, you each work on your own copy of the report. Once you are both done, you meet to discuss the edits and agree what to keep and what to change in the new common version of the report.

Branches in Git serve the same purpose. They are labels assigned to particular revisions (commits) that make it easy to track and discuss a specific change. For example, in the above scenario your branch could be **`new_analysis`** and your teammate's branch could be **`introduction_rewrite`**.

Branches are a powerful tool for organizing collaborative workflows. A common way to utilize branches is to associate them with specific work packages and/or team members, keeping edits separate until they are ready. Branches are also important as points of control — they allow you to separate work-in-process from the final contribution and give your teammates an opportunity to provide feedback or contribute their own changes before the entire set of changes is made part of the official shared project. Platforms like GitHub come with a rich set of utilities for managing branches, such as change tracking, discussions, reviews, and contributor roles.

The main branch

You have already worked with Git branches — every Git project has a default branch called **`main`** (sometimes **`master`**). This is a convention used to represent the clean, agreed-upon version. For example, in your project the `main` branch could track the latest working/coherent version, while partially completed work packages could be tracked in their respective transient branches.

Branches in a distributed environment

Remember that Git is a distributed system. Every team member works with their own copy of the repository and synchronizes it on demand, either sharing their new changes or fetching changes made by others. This distributed nature of Git can make it difficult to keep track of what's happening in the project, and branches are no exception. It is important to keep in mind that a branch is a version within a specific repository, so in a distributed system each copy of the repository has its own branches which are not always synchronized.

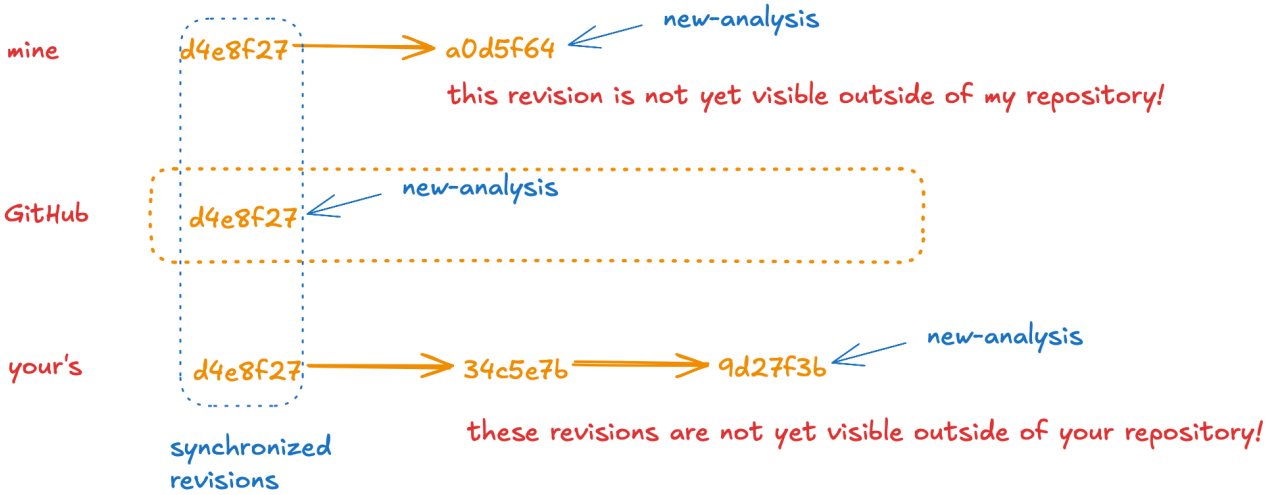

Fig 1. Branches in a distributed environment

For example, suppose that you and your teammate are working on a shared GitHub repository. Recall that each Git project has a default branch called `main`. In this particular case, there are actually three distinct **`main`** branches — the one on your computer, the one on your teammate's computer, and the one on GitHub. All these **`mains`** can be different, for example you could have new changes on your computer's **`main`** that you have not shared with GitHub yet. The same goes for any other branch you or your teammates create (e.g. **`new_analysis`**).

The simplest way to deal with all this confusion is to always remember that everyone's project repository is a copy, and explicit synchronization is needed to share data. Rather than talking about a branch called **`new_analysis`**, think of separate branches with the same label such as my **`new_analysis`**, your **`new_analysis`**, and GitHub's **`new_analysis`**. For example:

- I committed a change to my **`new_analysis`** branch
- I am pushing my **`new_analysis`** branch to GitHub, so that GitHub's **`new_analysis`** is now identical to mine
- I remind my teammates to pull GitHub's **`new_analysis`** into their own **`new_analysis`**, so that they can get up-to-date shared changes
- I check the difference between my **`new_analysis`** and GitHub's **`new_analysis`** to check if someone has shared new updates of the new analysis

This makes it easy to keep in mind the distributed nature of Git and the need for synchronization.

Example branch workflow

Let's walk through how branches work in a real team scenario.

Maria's team is working on a project analyzing word frequencies in political speeches. The team decides they need to add sentiment analysis to their R code. Maria volunteers to take on this task.

- Maria creates a new issue on GitHub, where she mentions what needs to be done and proposes her plan of action. The team can comment on this issue and suggest their own ideas.
- Maria creates a new branch **`sentiment_analysis`** on her copy of the project. This branch currently only exists on her computer and is not yet shared with GitHub or her team.
- Over the next few days, Maria works on implementing the sentiment analysis, writing some code using the **`sentimentr`** package. She commits frequently. All the work done on the **`sentiment_analysis`** branch is not yet visible to others.
- Once Maria is satisfied with what she did, she pushes her **`sentiment_analysis`** to GitHub. This creates a copy of her branch for everyone to see, visible under **`sentiment_analysis`** on GitHub.
- At the same time, Maria creates a new pull request on GitHub. A pull request is a mechanism to signal "my changes are ready, please integrate them into the project". She requests that changes on **`sentiment_analysis`** are integrated into **`main`** and asks her teammates for a review. A pull request opens a new discussion, allowing others to see the changes and participate in a conversation.
- One of the team members has an idea how to improve some of the code Maria wrote. After a brief conversation, she agrees that this is a better approach and makes appropriate edits to her **`sentiment_analysis`**. She then pushes the branch to GitHub, making these edits visible to all. GitHub automatically presents the latest changes in the pull request.
- Everyone is satisfied, and the team leader gives Maria a green light to go ahead and merge (integrate) her changes into the **`main`**. But there is a problem! Another pull request was recently merged in, editing the same lines of code that Maria has changed. This creates a merge conflict, which Git cannot resolve automatically. Maria is asked to deal with it.
- Maria updates her **`sentiment_analysis`** using the latest **`main`** from GitHub and fixes the merge conflict manually. Luckily, the fix is trivial; she can just put her code on a new line and

keep all other alterations. She instructs Git that the conflict has been resolved and pushes the fixed commit to GitHub.

- Now **`sentiment_analysis`** on GitHub has the latest changes and no conflicts with the main branch. The team manager presses the "merge" button in the GitHub interface, which integrates Maria's work into the main branch.
- Maria and everyone else can now delete the **`sentiment_analysis`** branch as it is not needed anymore. She can also close the issue (in fact, GitHub already did it as it was linked with her working branch!)
- Everyone is impressed with Maria's work, and the team goes to a bar to celebrate.

Example branch workflow

Creating and managing branches is really easy with GitHub desktop! Just follow the instructions below.

To create a branch

1. Click on the "Current Branch" dropdown at the top of the window
2. Click on the "New Branch" button
3. Enter a descriptive name for your branch (e.g., “sentiment_analysis")
4. Choose which branch you want to base your new branch on (usually “main")
5. Click "Create Branch”

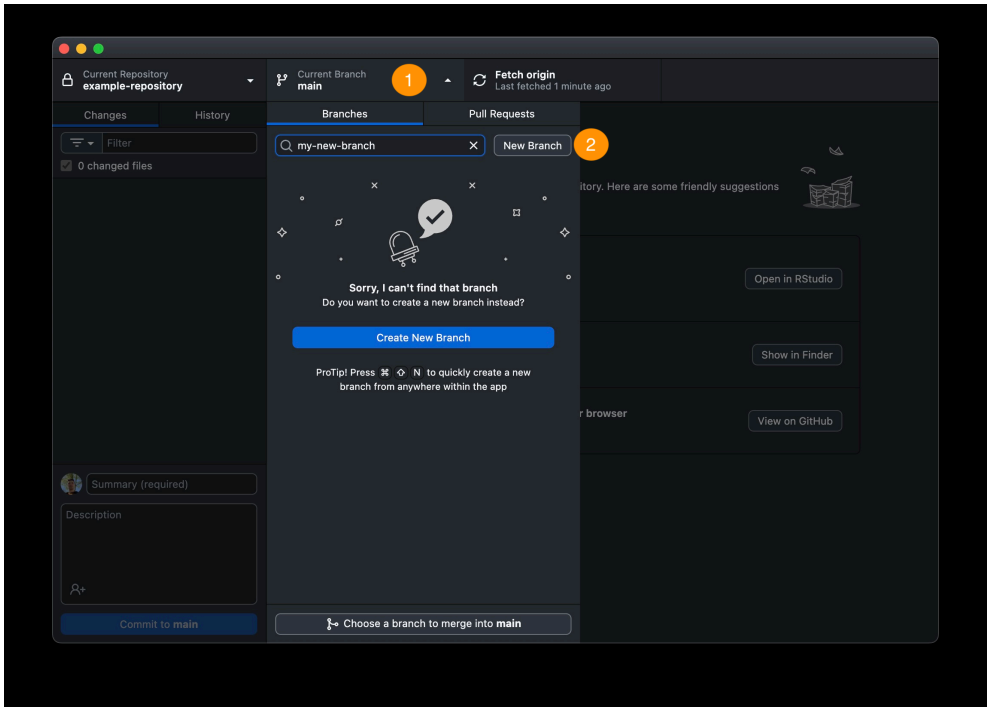

Fig 2. Creating Branches in GitHub Desktop

To switch between branches

1. Click on the "Current Branch" dropdown
2. Select the branch you want to switch to from the list
3. GitHub Desktop will update your working files to match the selected branch

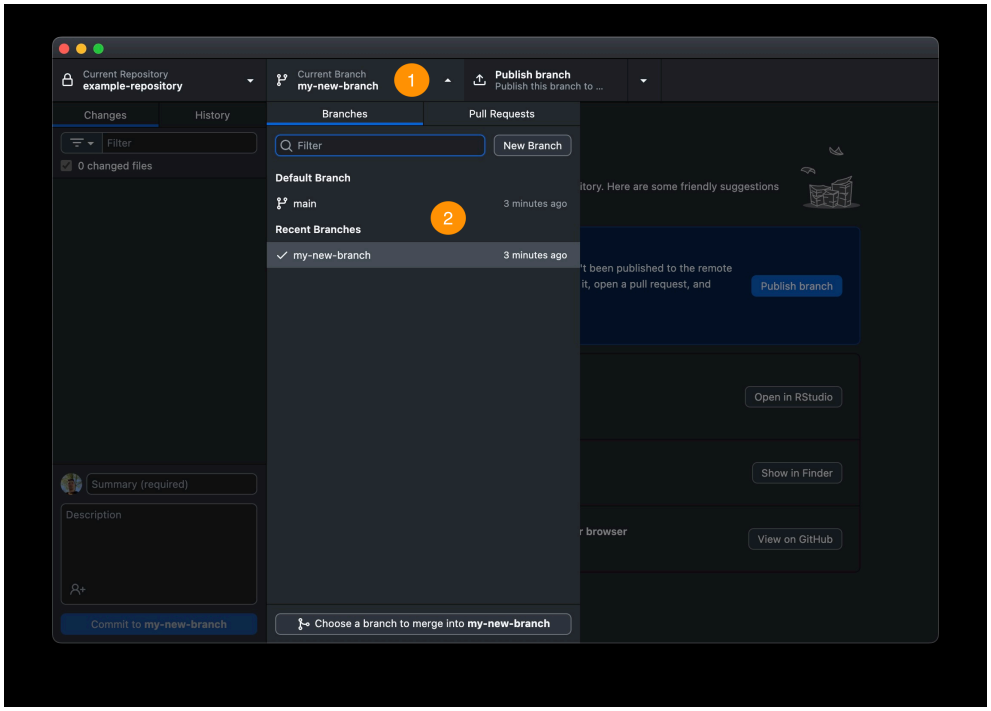

Fig 3. Switching Branches

To publish a branch to GitHub

1. Make your changes and commit them to your local branch as usual
2. Click the "Publish branch" button to share your branch with GitHub
3. Your branch is now available for others to see, pull, or comment on

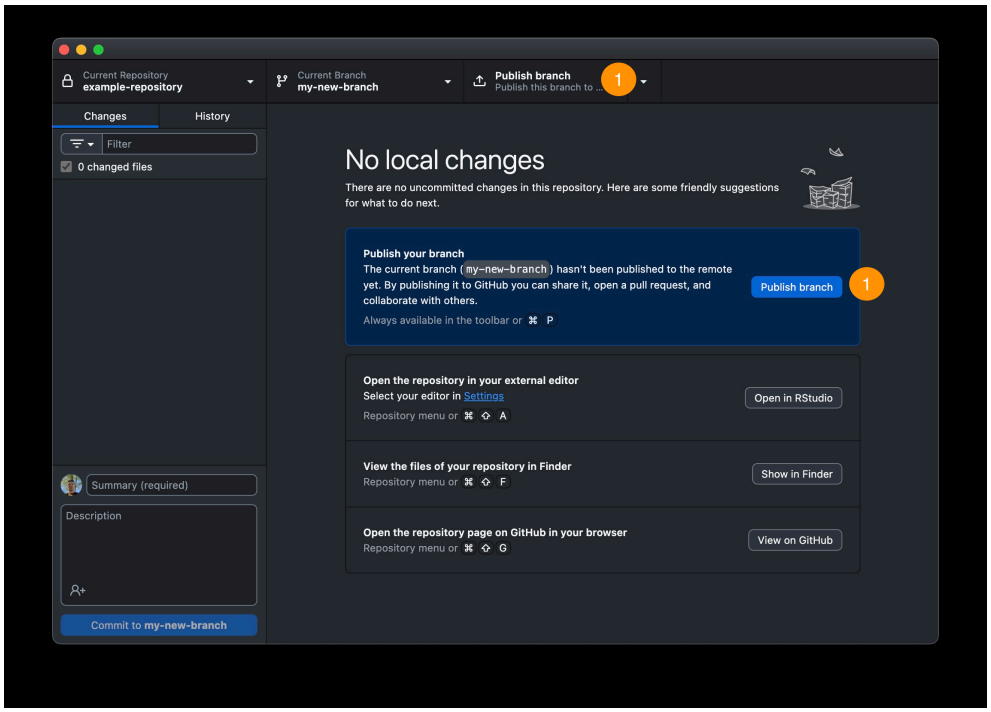

Fig 4. Publishing a Branch

To open a pull request on GitHub

1. After publishing your branch, click the "Preview Pull Request" button (Fig 5.)
2. This will open a preview interface with a “Create Pull Request” button
3. This will take you to GitHub in your web browser
4. Fill in the title and description explaining your changes (Fig 6.)
5. Request reviewers if needed
6. Click "Create pull request" to submit it for review

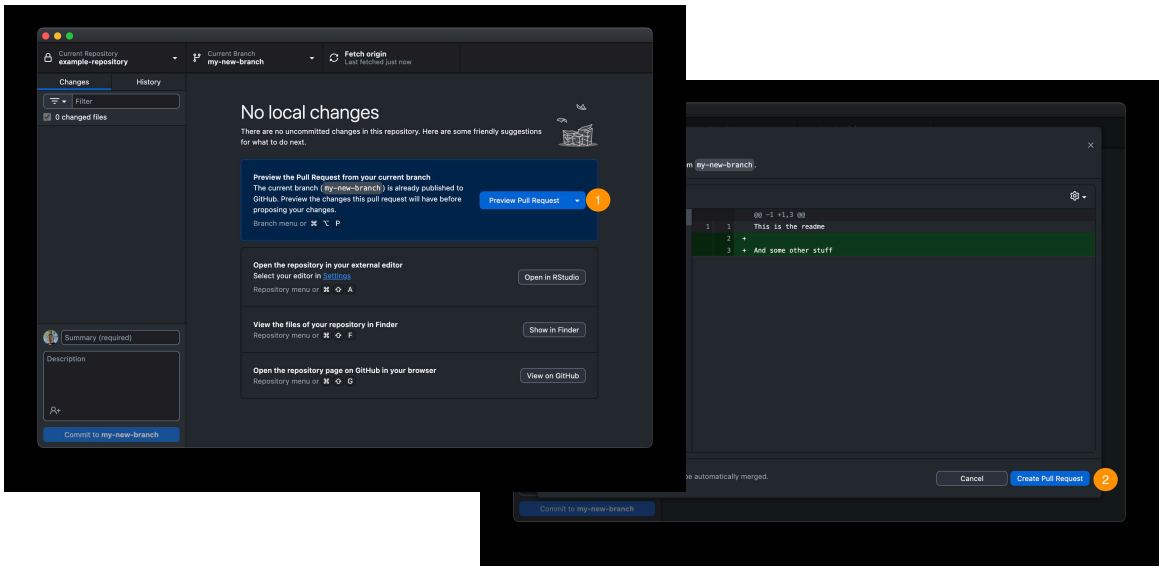

Fig 5. Initiating a Pull Request from GitHub Desktop

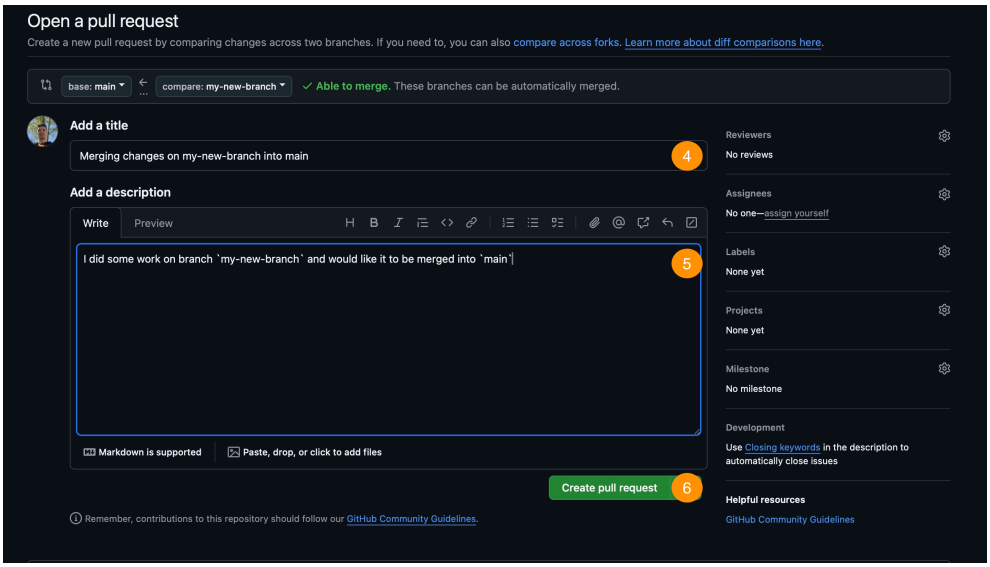

Fig 6. Opening a Pull Request on GitHub

To update your branch with changes from main

1. Switch to your branch in GitHub Desktop
2. Click on "Branch" in the top menu
3. Select "Update from main" (or another branch you want to update from)
4. If there are conflicts, GitHub Desktop will notify you

Alternatively, you can also use the "Choose a branch to merge into ..." from the branch selection panel.

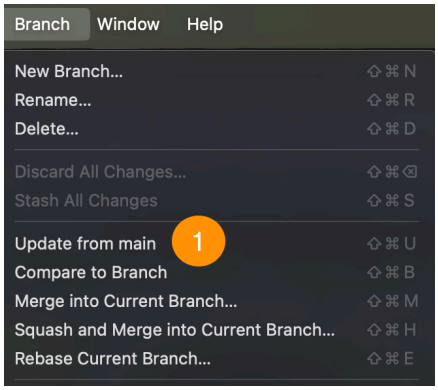

Fig 7. Updating from main

To resolve a merge conflict

1. When GitHub Desktop notifies you of conflicts, click "Resolve conflicts"
2. This will open a text editor where you can manually fix the conflicting code
3. Save the file once conflicts are resolved
4. Return to GitHub Desktop and continue with the merge

To delete a branch

1. After your changes have been merged, switch to another branch in GitHub Desktop
2. Click on the "Branch" menu at the top
3. Select "Delete..." and choose the branch you want to remove
4. Confirm the deletion

Navigating GitHub pull requests

GitHub pull requests are a powerful tool for summarizing, reviewing, and managing changes. A pull request (also called merge request) is a request to merge changes on a branch (usually a temporary work in progress branch) into another branch (usually `main`). Opening a pull request is easy — you can do it straight from GitHub Desktop once you publish your branch.

The pull request interface in GitHub provides several useful elements:

1. **Conversation tab**: A central place for team discussions about the changes, where comments can be added and responded to
2. **Commits tab**: Shows all commits included in the pull request
3. **Files changed tab**: Displays a side-by-side comparison of the code before and after your changes
4. **Review features**: Team members can approve changes, request modifications, or simply comment on specific lines of code

5. **Status checks**: Automated tests and checks can run on your code to ensure it meets quality standards

This interface creates a collaborative environment where team members can review all changes in one place, discuss implementation details, suggest improvements, and ultimately approve the changes before they're merged into the main codebase.

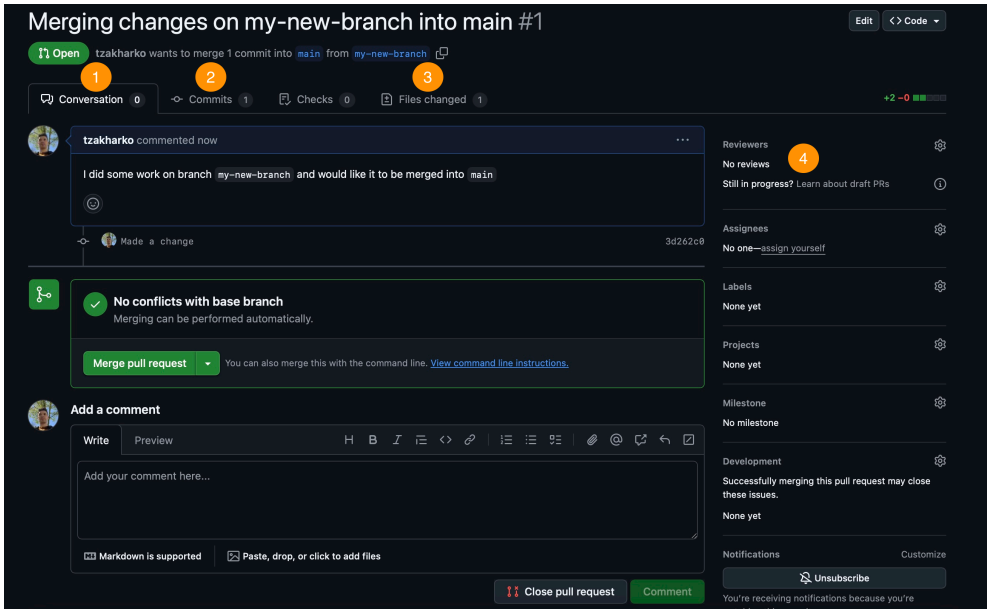

Fig 8. GitHub Pull Request Interface

Managing your workflows with roles and protected branches

You can set up your GitHub project to make it impossible to commit new changes to the **`main`** branch without going through a pull request first. This is a powerful capability since it forces review on any change that makes it into the final project. The best practice is to protect the **`main`** branch (this will disable direct push functionality to GitHub's **`main`** branch) and designate a member of a team as a merge manager. Only the merge manager is actually allowed to merge branches into main, all other contributors do not have this ability. The merge manager is responsible for reviewing every pull request and making sure that things are working as intended and done up to the standard. They can then choose to approve the pull request and merge the changes into **`main`**, request additional changes from the contributor, or reject the work outright.

This structured approach ensures quality control and maintains the integrity of your codebase throughout the development process. It also promotes accountability and creates clear roles within the team, which is especially valuable for larger projects with multiple contributors.

Note that when working with such a protected workflow, you can still make commits to your local **`main`** branch (recall that in the distributed git model your repository is a copy and not directly connected to the shared GitHub!). However, this will make your **`main`** incompatible with GitHub's **`main`** since you are not allowed to do direct synchronization. One way to repair this is to explicitly reset your commit history using GitHub Desktop — in the changes tab navigate to the last correct commit and select "Reset to commit" from the context (right-click) menu. You can then create a new branch from your local changes and submit that as a pull request following the proper workflow.