Git Collaboration Workflow

Topics

1. git fetch, git merge, git pull
2. Simple Git Collaboration
3. git reset, git revert, git rebase

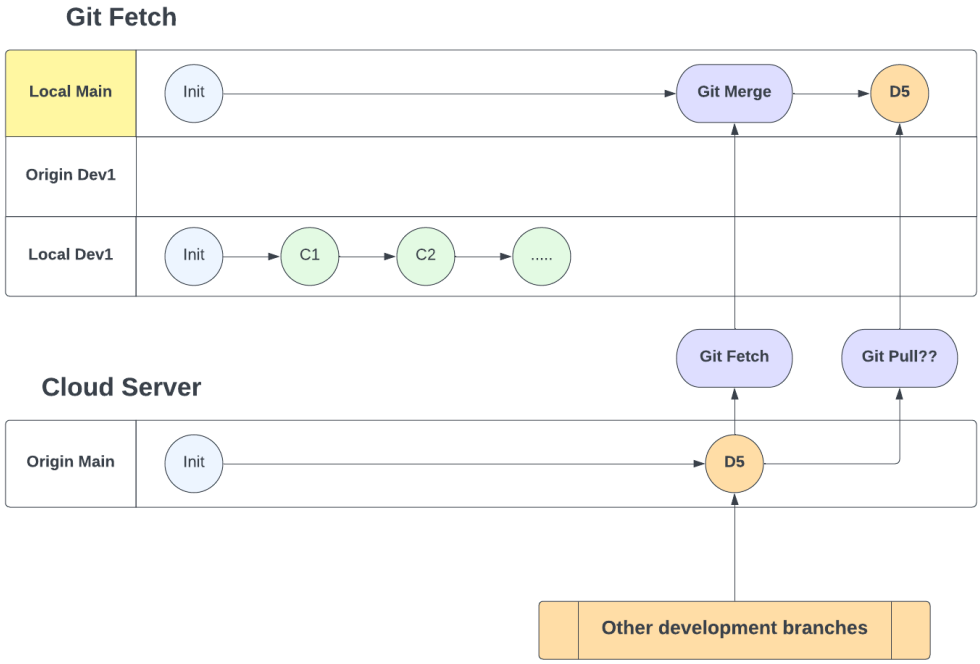
git fetch, git merge, git pull

git fetch:

The *git fetch* command is used to fetch the changes (metadata in *.git* directory) from the remote/origin repository and then stored the changes in the local repository without applying changes locally. Note that the local working directory stays unaltered or unaffected.

Benefit: since joining contents is a manual process, *git fetch* allows reviewing code before changing anything locally. The review process helps avoid merge conflicts.

NOTE: if you ever wonder what are the available branches currently exist on the remote server, *git fetch* offers a command to retrieve the information from all the remotes by using *git remote*.



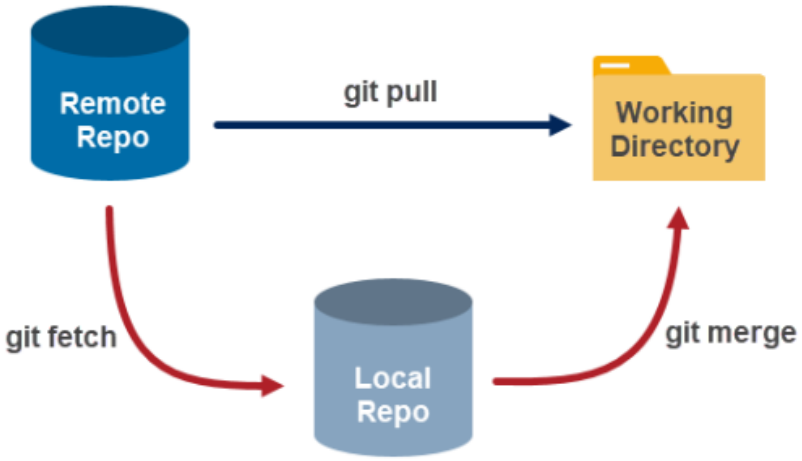
git merge:

The *git merge* command is primary used to combine two branches. It's also used to merge multiple commits into one history. Before merging, we should take some steps,

1. run *git status* to make sure pointing **HEAD** to the correct merge-receiving branch.

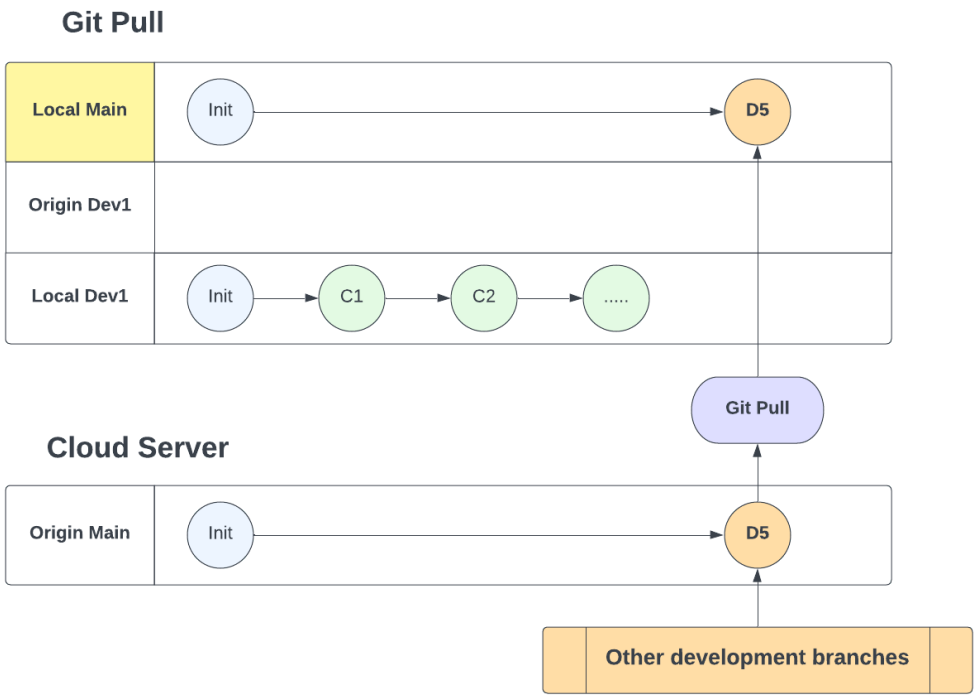
2. run *git fetch* to pull the latest remote commits.
3. run *git merge "branch name"* which the name of the branch to be merged into the receiving branch.

NOTE: *git merge* can be used to merge any fetched branch to any local branch. We can also use it to merge any local branches to bring commit history from one branch to another branch as an update function.



git pull:

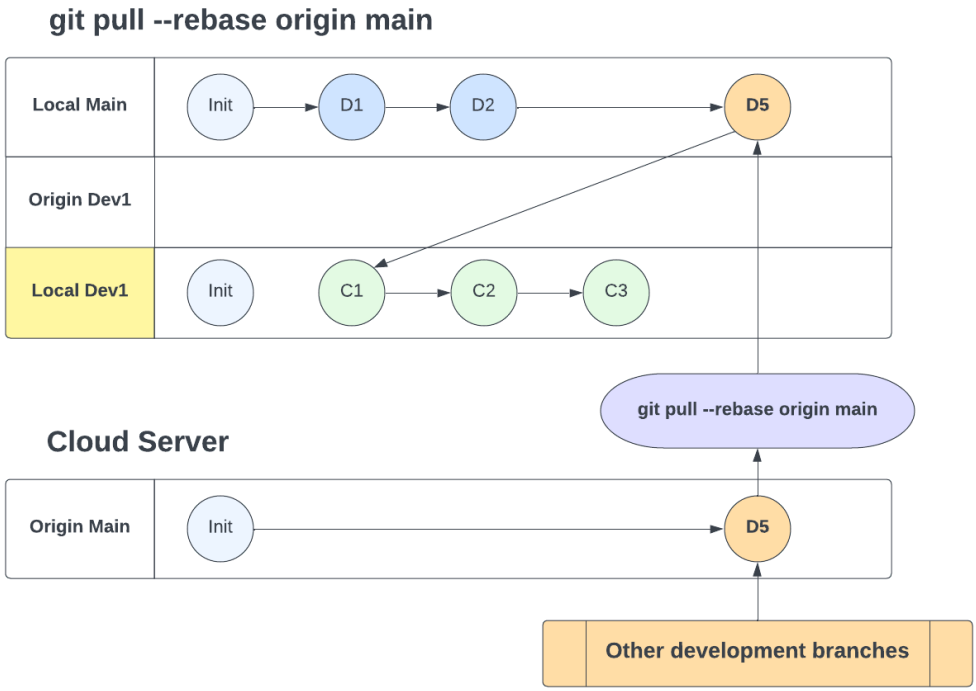
The *git pull* command is basically the combine of *git fetch* and *git merge* in one command, so that two branches are merged without reviewing. The downside is that *git conflict* can happen without reviewing the code prior merging. By default, *git pull* will update the current local branch from the origin main. Therefore, it is a good practice to combine the use of *git checkout "branch name"* and *git pull "branch name"* to explicitly define the local branch to get the update and the remote branch to pull the update.



The best practice of *git pull* when origin / remote main contains data or updates that do not exist in the local directory is to use *git pull --rebase origin main*. Note that this is only necessary when your task requires the updated part of the origin main branch, for example, a function that is fixed by a team member that is required for your analytic work. We can also achieve the same objective by using *git fetch origin + git rebase main* to pull updates from origin main to the local main, but the first option is a much cleaner solution, which most of the time is preferred.

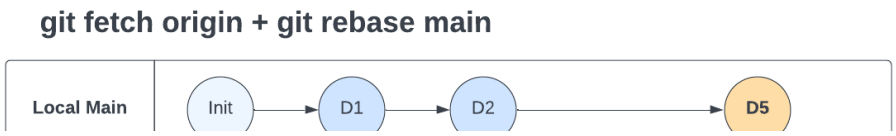
git pull --rebase origin main:

where *git pull --rebase origin main* pull changes from remote main branch at first, then rebase current branch on the top of it. The local branch is rebased on the origin main branch.



git fetch origin + git rebase main

where *git fetch origin* download remote files from the remote repository into your local branch and *git rebase main* takes the whole branch (the commits that are in your branch), and put on the top of your local repository. The local branch is rebased on local main branch.



Exercise:

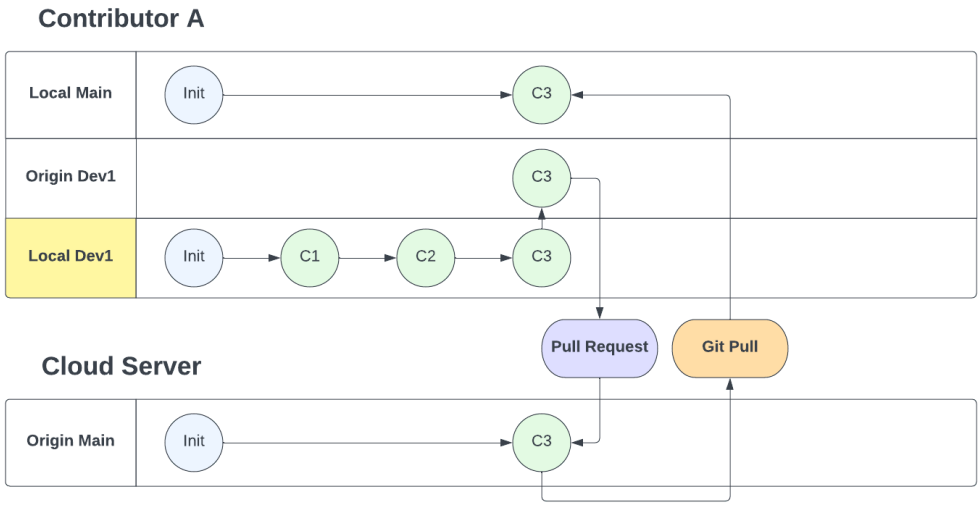
1. Run *git fetch --all* to download all branches from project repository.
2. Run *git branch -r* to check the remote branches that were fetched to the local repository folder.
3. Run *git diff main origin/"one of the remote branch name"* to see the different between the local main and the remote branch.
4. Run *git checkout -b "branch name" origin/"branch name"* to checkout the remote branch to the local.

Note: Using VIM to edit file in terminal,

1. Use *vim "file name"* to open a file with VIM.
2. Press **"i"** to enter insert mode to insert before the cursor.
3. After edited the file, use *Ctrl + c* to exit insert mode.
4. Use *:w* two write (save) the file, but don't exit. OR Use *:wqa* to write (save) and quite on all tabs.

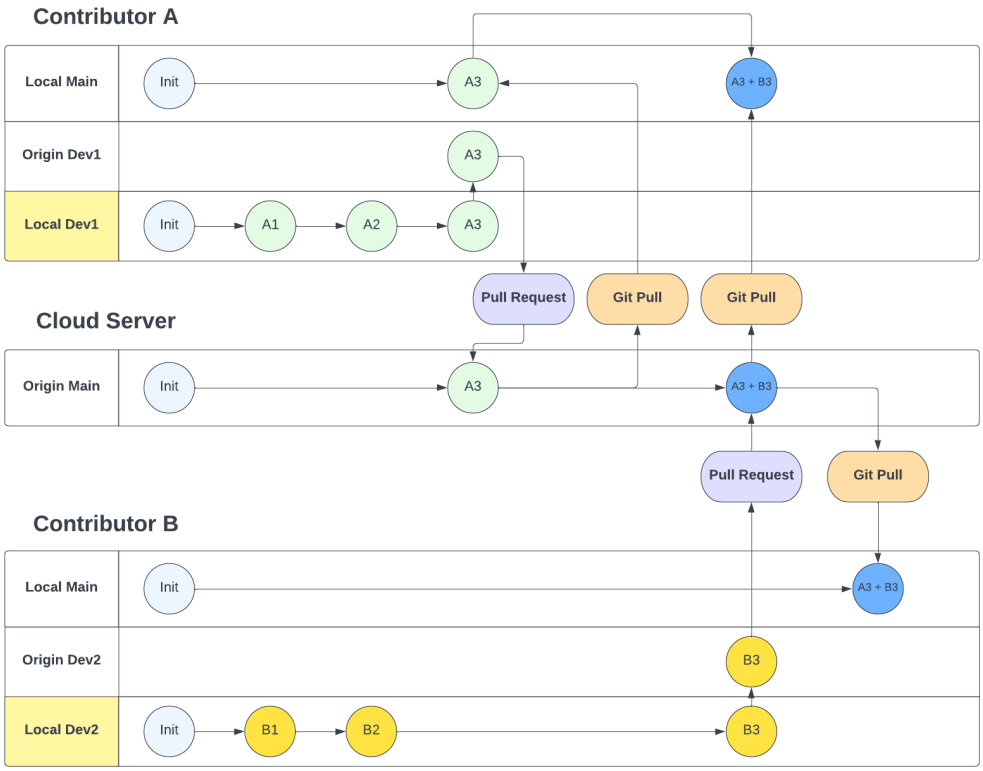
Simple Git Collaboration

When collaborating with a team on a project with Git, it's always a good practice to leave the local main branch untouched and only pull update directly from the remote/origin main branch. Here is an example of a simple git collaboration work flow.



Note that in any parallel development, the local main branch should serve as a base branch of any development branch. Unless a merge from other remote branches is necessary, the local main should be a standalone branch by itself during the development stage.

To demonstrate the multiple contributors scenario, assume there are two contributors are working on the same project simultaneously, a simple work flow should be follow to avoid any possible conflict in commit history. Here is an example.



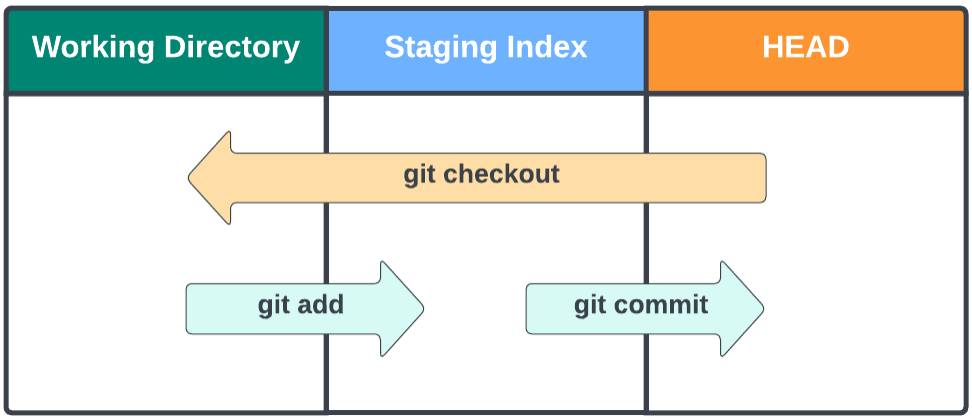
Note that when multiple contributors are working on the same project, git pull is necessary to

git reset, git revert, git rebase

As many of us have experienced with Git, there are often time we may want to roll back to one of the previous commits or history point to recover a lost file or go back to the previous stage of the development code that had over-written. Regardless of the reasons, we want to make sure to understand the basic Git rollback features before using them.

git reset:

git reset is a complex command that requires some Git internal management mechanisms, **HEAD**, **staging area (index)**, and the **working directory**.

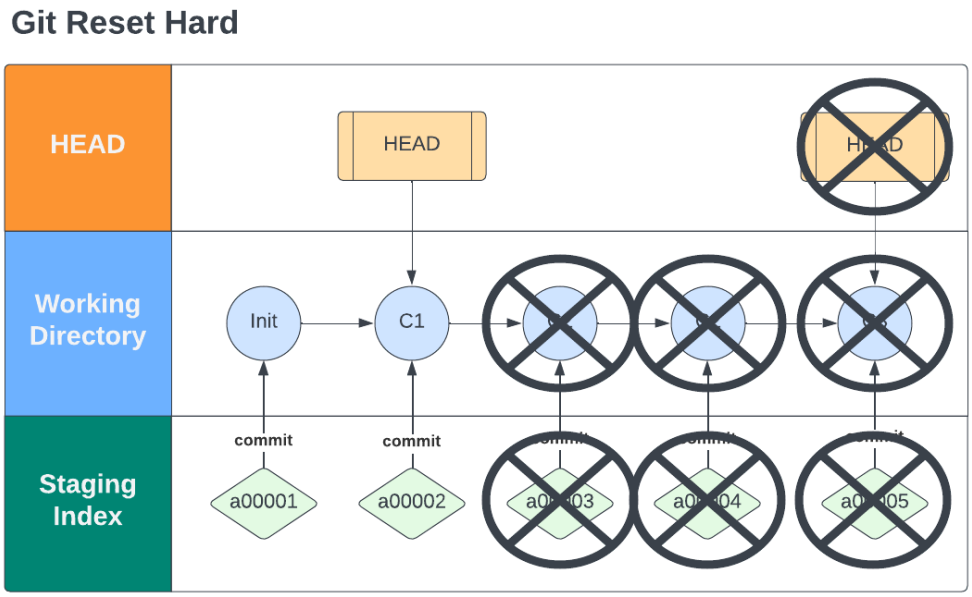


As demonstrated in the diagram, working directory is the place where you are currently working and the place where your files are present. Staging area (index) is where git tracks and saves all the changes in the files. The current branch in Git is referred to as HEAD. It points to the last commit which happened in the current checkout branch. It's treated as a pointer for any reference.

By using *git reset*, we can work in hard, soft, and mixed modes. Hard model is used to go to the pointed commit, the working directory gets populated with files of that commit, and the staging area gets reset. In soft reset, only the pointer is changed to the specified commit. The file of all the commits remain in the working directory and staging area before the reset. In mixed reset (default mode), the pointer and the staging area both get reset.

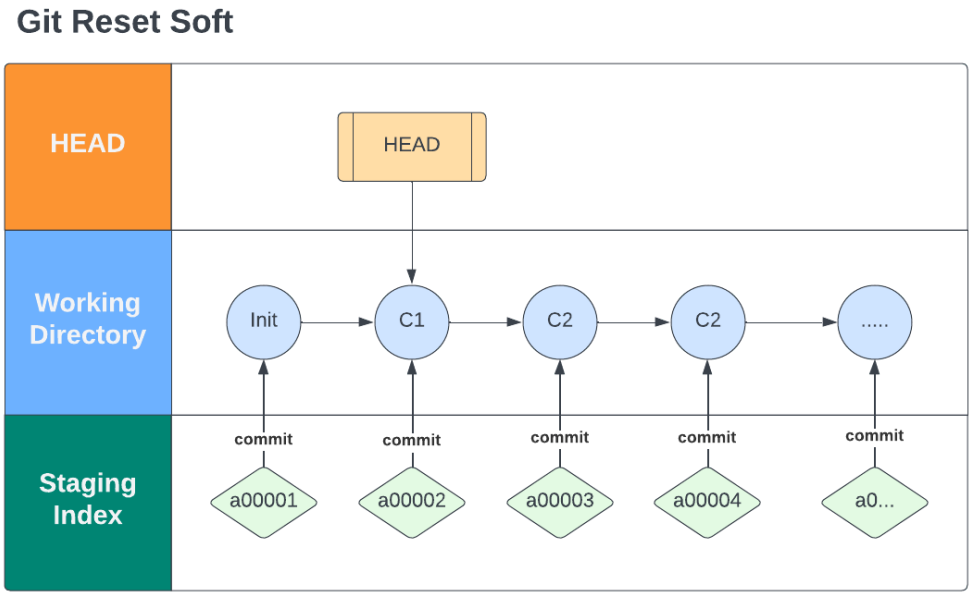
git reset --hard <SHA>

To reset the current HEAD to specific commit in the past, we only need to pass in the SHA hash value (minimum 6 characters of the hash value) to identify the commits in history. Note that hard reset does not preserve the history or data prior to the commit point for the reset. It's a risk to take for not having the history and data to work in a reset. Only use it if you are sure that all files and data are not relevant to the project.



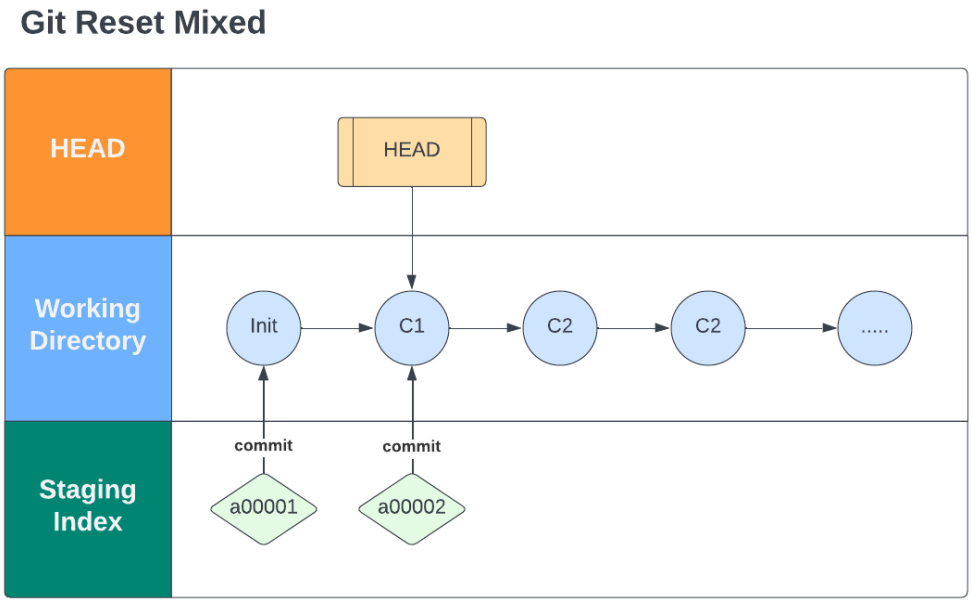
git reset --soft <SHA>

The soft reset allows us to move the HEAD back to the specific commit and preserve all history and data. Note that when doing a soft reset with Git, the previous commits are kept in the stage area.



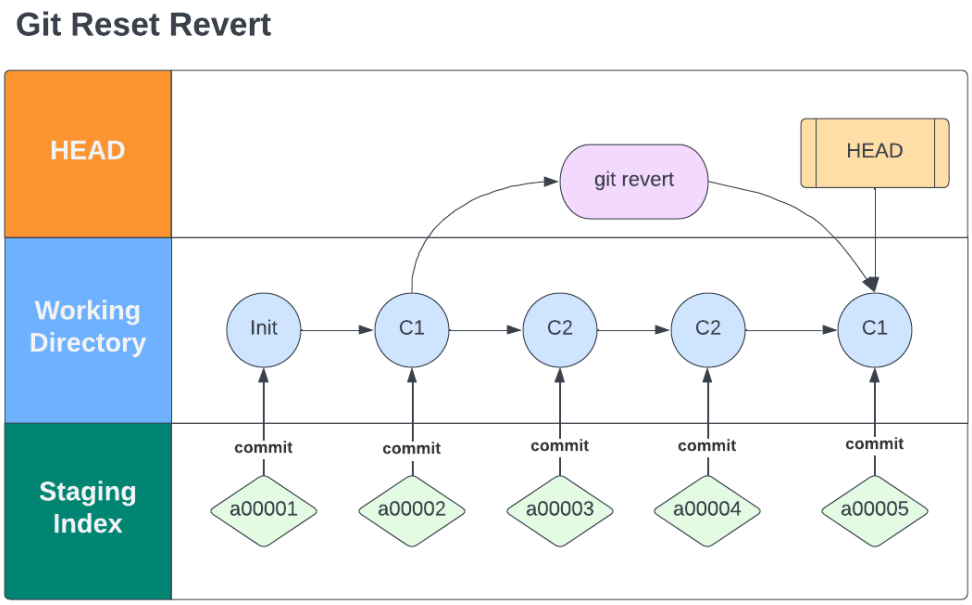
git reset <SHA>

The mixed reset is the default when using *git reset*. Similar to soft reset, the HEAD can be moved to the specific commit in history and preserve all data. The different between mixed and soft reset is that in mixed reset, the previous commits are not in the stage area, so it provides flexibility to modify the existing files before *git add* to add changes to the stage area.



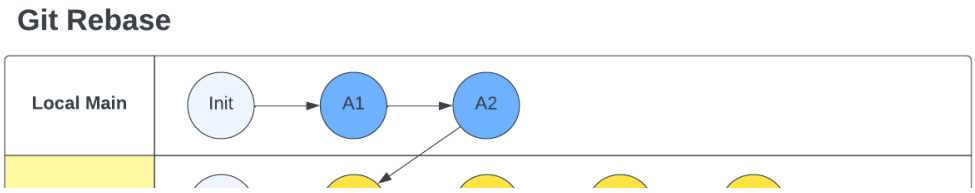
git revert <SHA>

The *git revert* command is basically a new commit to the current working directory. However, it does not "revert" back to the previous state of a project by removing all subsequent commits. Two important advantages of *git revert* over *git reset* is that it doesn't change the project history, which makes it a "safe" operation for commits that have already been published to the shared repository. Also, it is able to target an individual commit at an arbitrary point in the history, whereas *git reset* can only work backward from the current commit.



git rebase

Rebase can sometime be confusing to understand because it's very similar to a mrege. The goal of merging and rebasing is to take the commits from a development branch and put them on to a master branch or any other branch. The advantage of using *git rebase* instead of *git merge* is that *git merge* creates a more complex tree structure of the commit history, whereas *git rebase* allows us to move all the development branch commits on top of the main branch for a clean straight-line graph. It makes it easy to trace what commits went where.



Type *Markdown* and LaTeX: α^2