Resources:

<https://git-scm.com/book/en/v2>

<https://www.youtube.com/playlist?list=PLu0W_9lII9agwhy658ZPA0MTStKUJTWPi>

<https://www.youtube.com/playlist?list=PLu0W_9lII9agwhy658ZPA0MTStKUJTWPi>

<https://www.youtube.com/watch?v=rFRtsiQEJZw>

*Git* is a version control system which is used to maintain all the changes done in a file for future reference.

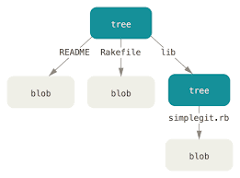
If one is working in a team, one must use Git to mention the previous changes done and their reasons which can be discussed with the team and mistakes could be recovered.

*GitHub* is a web-based cloud service to host your source code (Git repositories). It is a centralized system.

GitHub uses Git but Git does not use GitHub.

* Changes done in folder in a file which make a new version of the folder.
* Saving the whole folder as a zip file with even a small change made in a file will use up a lot of memory.
* Writing log of each file with date and time of changes made will take a lot of time finding a particular file of given time and date. With Git this can be done in a second’s time.
* Files can be easily recovered.
* Roll-back to changes will be done.
* Who introduced the changes and when will be maintained.
* Almost every operation by Git is local. Because we have the entire history of the project right there on your local disk.
* Git has integrity. It maintains SHA Checksum unique value for each file and if any changes will be made then this value also changes. It helps to maintain security of data and if any hacker tries to do changes then Checksum value will get changed and the receiver of the file will easily get to know this.
* Git generally only adds content. When you do actions in Git, nearly all of them only add data to the Git database.
* Git stores snapshot of the file and not only the differences, snapshot is the representation of the current state of the file. Each commit contains a pointer to its root tree, representing the state of the working directory at that time. The commit has a list of parent commits corresponding to the previous snapshots. A commit with no parents is a root commit and a commit with multiple parents is a merge commit. Commits also contain metadata describing the snapshot such as author and committer (including name, email address, and date) and a commit message. The commit message is an opportunity for the commit author to describe the purpose of that commit with respect to the parents.

The tree is the directory which contains list of changes done on each file in the directory. Each list is a child of the tree.



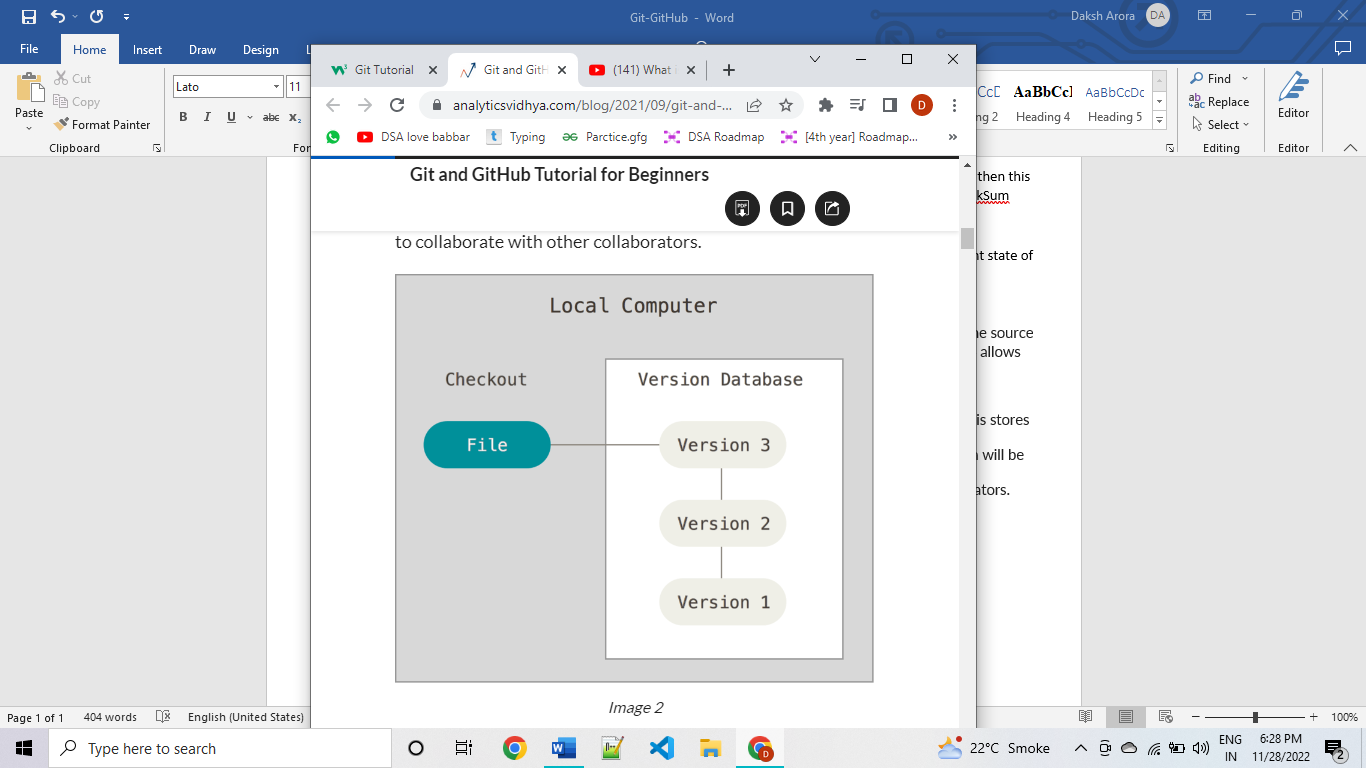
Version Control Systems (VSCs) are the software tools for tracking/managing all the changes made to the source code during the project development. It keeps a record of every single change made to the code. It also allows us to turn back to the previous version of the code if any mistake is made in the current version.

**Types of VSCs:**

1. **Local Version Control System:** Local Version Control System is in your local machine. It is stored in a database. If a user wants a particular version of folder, then one may query the database.

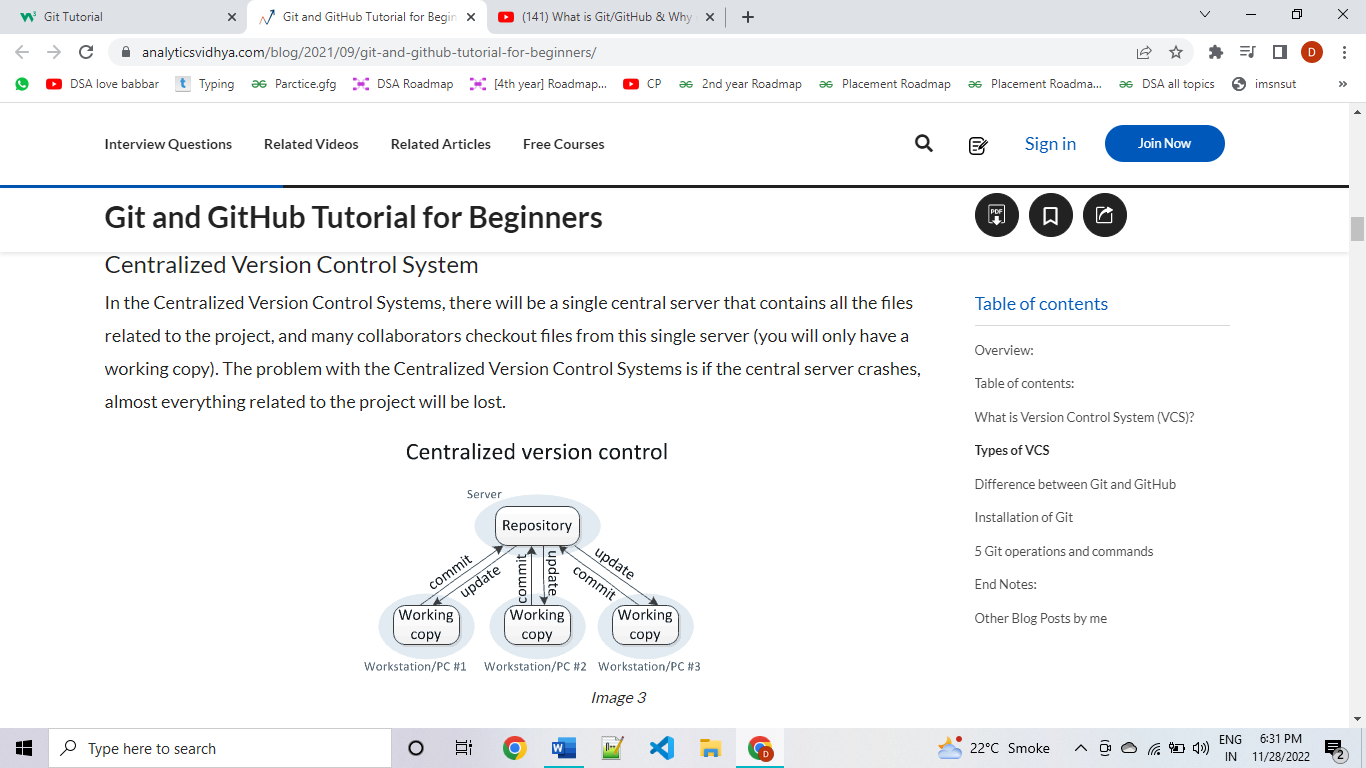
If the local machine crashes, it would not be possible to retrieve the files, and all the information will be lost. If anything happens to a single version, all the versions made after that will be lost.

Also, with the Local Version Control System, it is not possible to collaborate with other collaborators on other systems.



1. **Centralized Version Control System**: In the Centralized Version Control Systems, there will be a single central server that contains all the files related to the project, and many collaborators checkout files from this single server (you will only have a working copy). The problem with the Centralized Version Control Systems is if the central server crashes, almost everything related to the project will be lost.

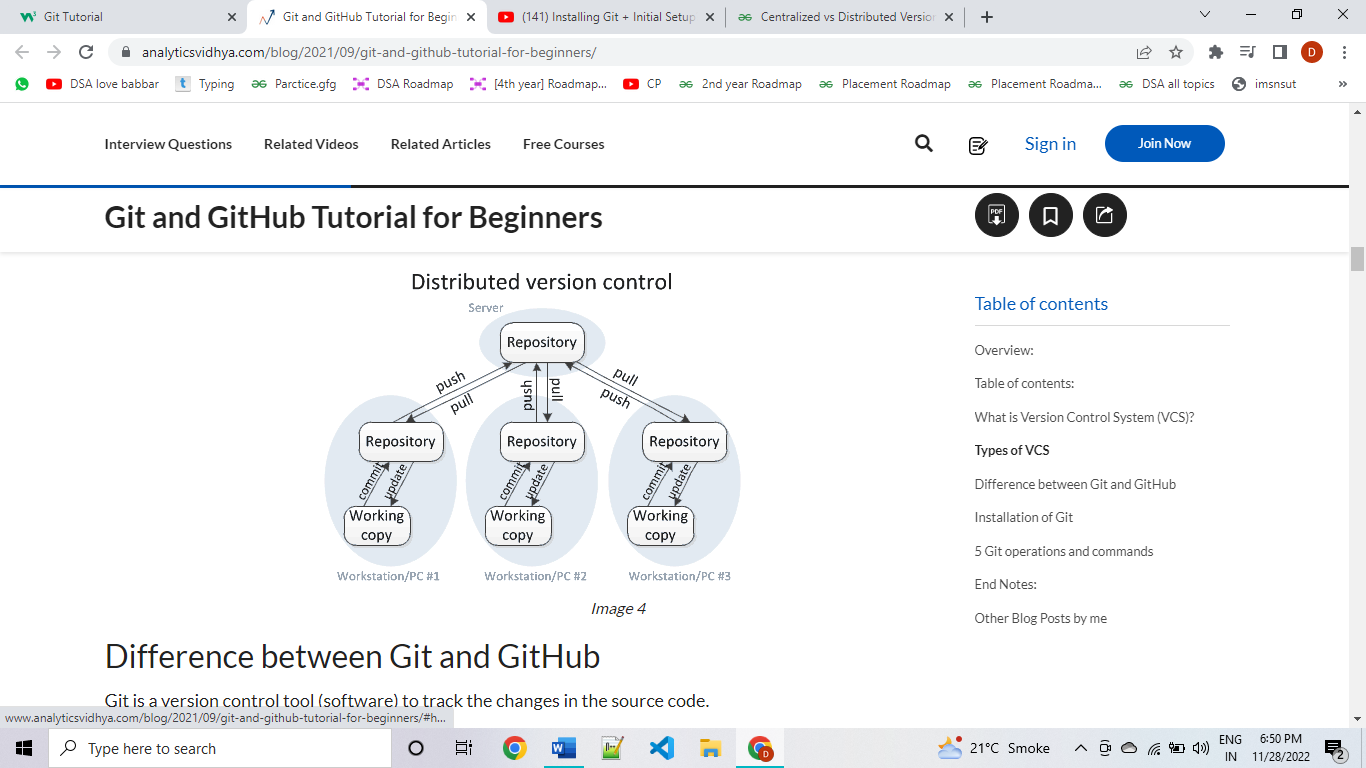
Here, the user only takes the latest folder from the repository and does changes and then commits the updated file.



1. **Distributed Version Control System:** The only major difference you will find here is, instead of one single repository which is the server, here every single developer or client has their own server and they will have a copy of the entire history or version of the code and all of its branches in their local server or machine.

So when you start working on a project, you clone the code from the master repository in your own hard drive, then you get the code from your own repository to make changes and after doing changes, you commit your changes to your local repository and at this point, your local repository will have ‘*change sets*’ but it is still disconnected with the master repository. Firstly, you commit all the changes in your own server or repository and then the ‘set of changes’ will merge to the master repository.

Getting the new change from the central repository is called “**pulling**” and merging your local repository’s ‘set of changes’ to the central repository is called “**pushing**”.



* DVCS is faster than CVCS because you do not need to communicate with the remote server for each command. You do everything locally which gives you the benefit to work faster than CVCS.
* Every client or user can work locally and disconnected which is more convenient than centralized source control and that is why it is called distributed.
* If the main server goes down or it crashes in DVCS, you can still get the backup or entire history of the code from your local repository or server where the full version of the code is already saved.
* Git is distributed version control system and GitHub is just a web-based hosting.

**Git Bash** is a Microsoft Windows application with a Git command-line tool/shell experience and utilities.

* Linux commands like ‘sudo’ or ‘apt-get’ work on Git Bash.
* For Windows, when installing Git through the installer, it is recommended you select the “Use Git from the Windows Command Prompt” option. This will allow you to use all git commands through your terminal (CMD, PowerShell, Anaconda) rather than having to use Git's personal terminal, Git Bash.
* Git also has GUIs like in VS Code editor, GitHub Desktop.
* ‘pwd’ command returns the present working directory.
* ‘ls’ command lists the name of all files in the pwd.
* To go to Desktop, make sure you are in its ancestor directory, for this use: ‘cd ~’ ‘~’ is the root directory, in case of Windows that is “This PC”.
* and then ‘cd Desktop’ makes Desktop directory as pwd.
* To go to a folder inside pwd use ‘cd folderName/’
* Not ‘cd c’ but ‘cd /c’ makes C directory as pwd.
* cd .. helps you move to previous directory.

To begin with, we will make an account on git bash by mentioning our user name and email because if somebody wants to know about the person who committed the changes, the person can contact that person using user name and email.

*To configure user name:*

*git config --global user.name "you"*

*To configure user email:*

*git config --global user.email "your@gmail.com"*

*To see what all configurations have been saved:*

*git config --list*

*To get user name:*

*git config --global user.name*

*To get user email:*

*git config --global user.email*

*To commit:*

*git commit -m “…”*

*To add* ***all*** *files to staging area*

*git add –a* or *git add .*

THREE STAGE ARCHITECTURE

Git involves three stages in which your file can reside in:

1. Working directory: This stores the file that we have pulled from the git directory(repository). The changes are made at this stage. It was changed since it was checked out but has not been staged, it is **modified**.

2. Staging Area: The files that you want to commit will come to staging area. The files that you do not want to include may be because of reasons such as – they may include errors. It has been modified and was added to the staging area, it is **staged**.

3. Committed: The files which are staged will be finally committed to the directory. If a particular version of a file is in git directory, it is called **committed**.

MAIN WORKING OF GIT COMMANDS in stages mentioned above

A directory of files is there with some location ‘C:\\a\\b’. Go to git bash and make location as pwd using “cd C:\\a\\b” command. Locations in Linux always use ‘\\’ in their path to go from folder to other. Now to make it a git repository to tell Git that it should track the files in this directory (now repository).

**Use** *git init* **to make it a git repository**

**Use** *git status* **to check the status of the git repository**

Now if you make any changes to any of the files, it will mention these untracked files in the status of the folder. Now to commit the changes in the git repository (basically you will acknowledge all the changes made and you will tell GIT that you want these changes to be made permanent).

So this includes **staging the files** which you to be committed **using** *git add --a* (all files).

To **stage one particular file use** *git add first.txt*

Now after the staging the **changes made must be permanently done or committed using** *git commit -m "Your message"*

Now to **check the account of all changes committed use** *git log*

To **untrack a folder use** *rm -rf .git*

To **clone a GitHub repository to local/remote git repository use (pulling)**

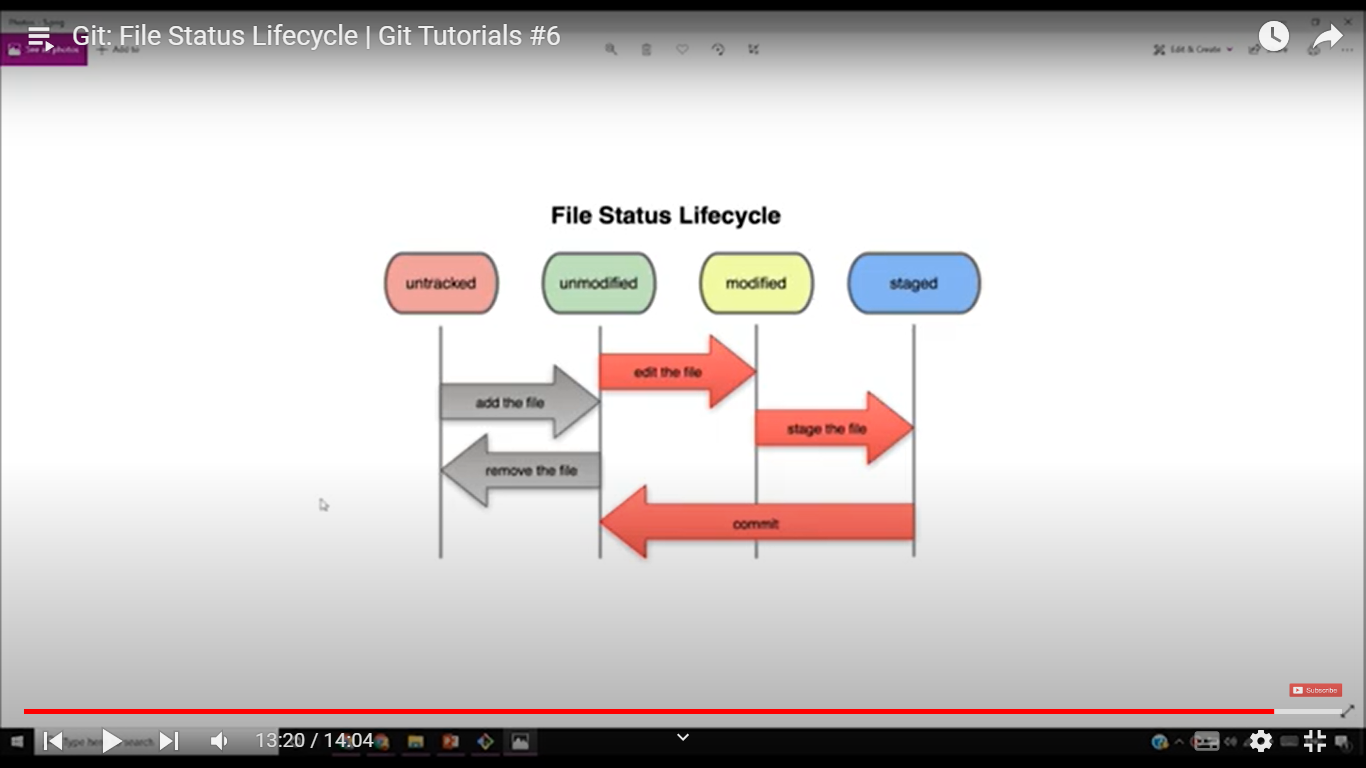
git clone <https://github.com/tensorflow/tensorflow> filename ->in which clone files are to be saved. This link from GitHub can be available by clicking on “Code” button from the repository on GitHub.

git pull (or git fetch + git merge ) is how you update that local copy with new commits from the remote repository.

git clone is used for just downloading exactly what is currently working on the remote server repository and saving it in your machine's folder where that project is placed.

A **repository** is a collection of files which are being tracked by Git.

**File Status Life Cycle**



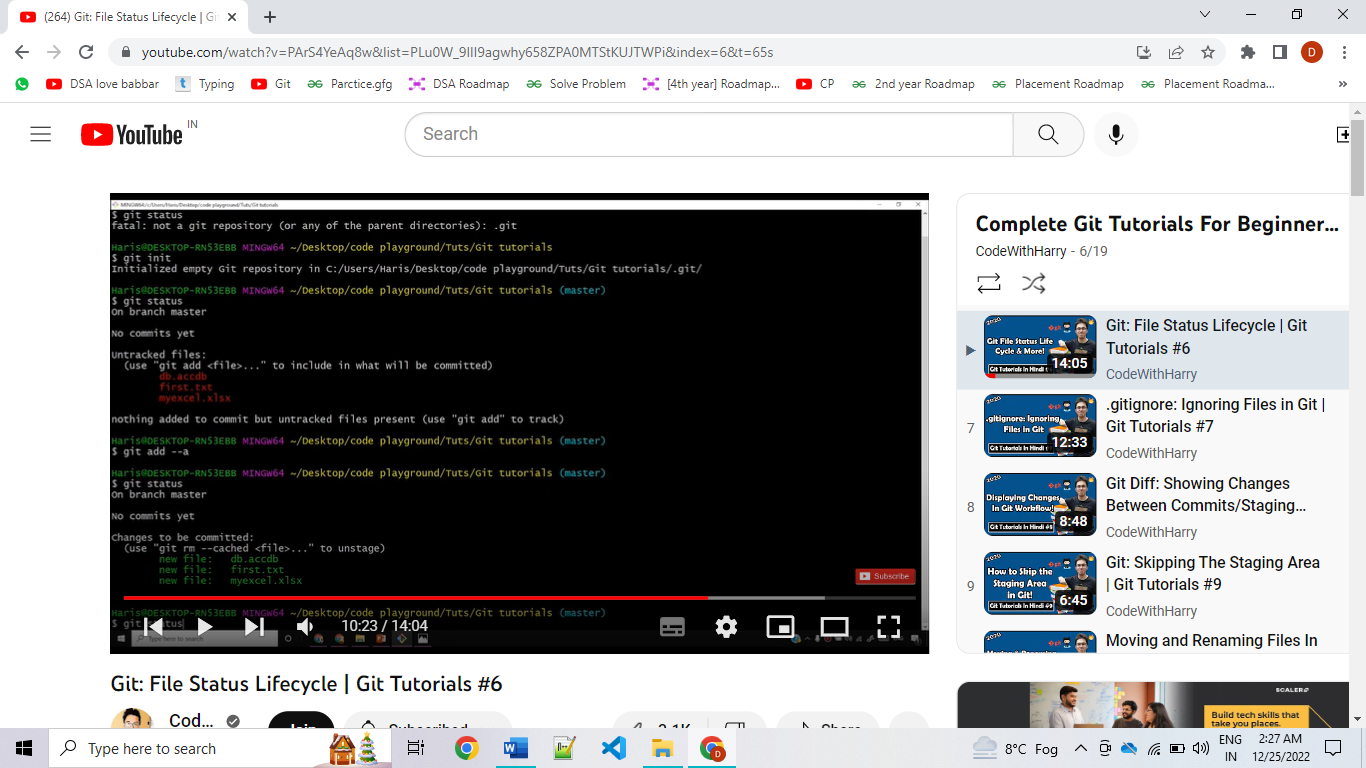
*Untracked:* Files come in this stage after we use command “*git init*”. Before this, files were not even a git file, so they did not exist in any stage of the cycle.

*Unmodified:* Files come in this stage when we use command “*git commit*”. This means the files are not modified by the user. In this stage the files are being tracked by Git. Note: Files after using “*git add*” command just after “*git init*” also come under this state.

*Modified:* Files come in this stage when the user has changed the file content and that is not tracked by Git. Changes done before committing when files are not in staged, such files also come in this stage. Now, if we commit then the changes staged before will only be tracked. But if the file is staged again using “git add” then the new changes will also be committed after using “*git commit*”.

*Staged:* Files come in this stage when we use command “*git add*” after modifying them (not after using “*git init”*, because then they come under unmodified category).

After staging the changes in files, the files are committed using “git commit” command and files come in “unmodified” stage because the changes done are now tracked by git.



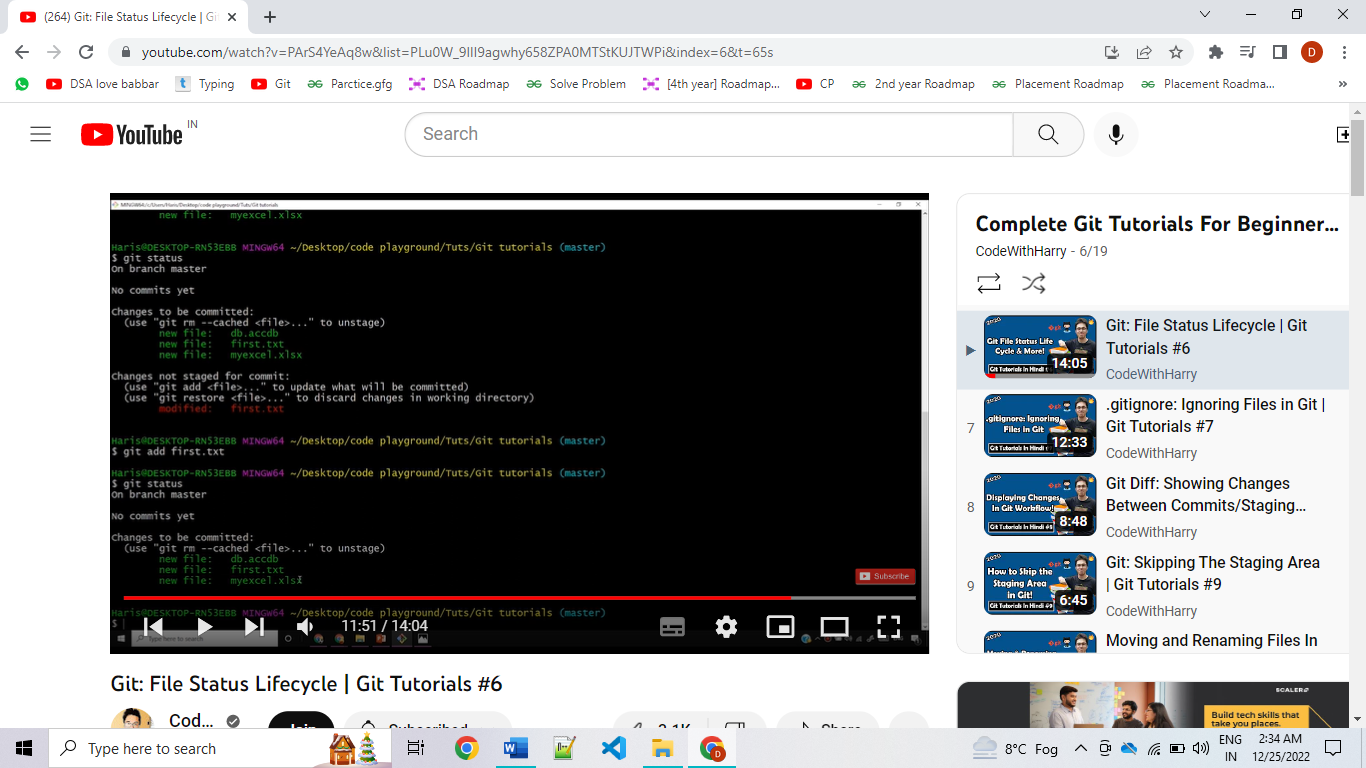
1. not a git repository: not there in any git stage.

2. Initialized as a git repository.

3. git status: tells that files are in “untracked” state.

4. git add –a: adds all the files to staging area and all files are in “unmodified” state. Thus, files before commit can also reside in “unmodified” stage (only after git init and git add).

5. git status: tells that all files are “staged” (but still not committed).



6. When some changes are done now, git status tells that a file is “modified” (goes from unmodified to modified), and the changes should be staged.

7. The changes in the modified file are staged using “git add”.

8. Now the status tells that all changes are staged (files have “staged” status).

9. At last all changes in the file are committed (file status changes from “staged” to “unmodified”).

*To create a new file*

touch file.ext

*To tell GIT to ignore some files and don’t tell if changes have been done*

Create a. gitignore text file. Write the name of the files in this. gitignore file with new lines.

*To ignore all files of particular extension*

Write \*.ext in .gitignore file

*To ignore a folder inside this folder anywhere with this name*

Write folderName/ in .gitignore file

*To ignore the outer folder only with the mentioned name*

Write /folderName/ in .gitignore file

*To ignore a folder at specific location (inside the root directory)*

Write InnerFolderName/folderName or Write /InnerFolderName/folderName in .gitignore file

Git by default ignores a empty folder, if we have a folder in which we have a file which we have ignored, then Git will treat this folder as empty and will simply ignore it.

*When you don’t want to ignore a file*

Write !README.md in .gitignore file

*To get differences between the changes committed/staged and the ones modified*

*git diff*

If no files are modified, then git diff gives nothing

*To get differences between the committed and staged files*

*git diff --staged*

*To skip the staging area and directly commit all files*

*git commit -a -m “…” or git commit -am “…”*

It only commits the tracked/modified files, but not the untracked ones.

*Show changes between two commits*

gi*t diff commit1\_id commit2\_id*

*To list all commit*

*git log*

*To delete a file (delete from the working directory and staging area both)*

*git rm fileName.ext*

Deletion changes will be staged automatically.

*To rename a file*

*git mv oldName.ext newname.ext*

Rename changes will be staged automatically.

If we modify a file and then add the file to .gitignore, the file will still be tracked and changes will be shown by git status. To untrack the file use: git rm --cached filename.ext (to remove a file from staging area only).

The file will be shown as deleted, now commit the deletion changes done.

This command deletes this file from the git repository, but not from the system directory. The system file will be shown as untracked and the git repository file as deleted.

*To get details of first n commits*

*git log -p -n*

n is the number of commits which should be shown

*To get number of line added/removed as diff with respective commits in respective files*

*git log –-stat*

*To get only the message associated with each commit with the respective commits*

*git log –-pretty=oneline*

*To get the author/creator of the file’s name and the message associated with each commit with the respective commits*

*git log –-pretty=short*

*To get the author/creator of the file’s name, name of committer and the message associated with each commit with the respective commits*

*git log –-pretty=full*

*To get all commits done in the last 2 days*

*git log –-since=2.days* (no blank spaces are entertained after ‘=’)

*To get all commits done in the last 2 weeks*

*git log –-since=2.weeks*

*To get all commits done in the last 2 months*

*git log –-since=2.months*

*To get all commits done in the last 2 years*

*git log –-since=2.years*

*To view log in a particular format*

*git log –-pretty=format:”%h -- %an”*

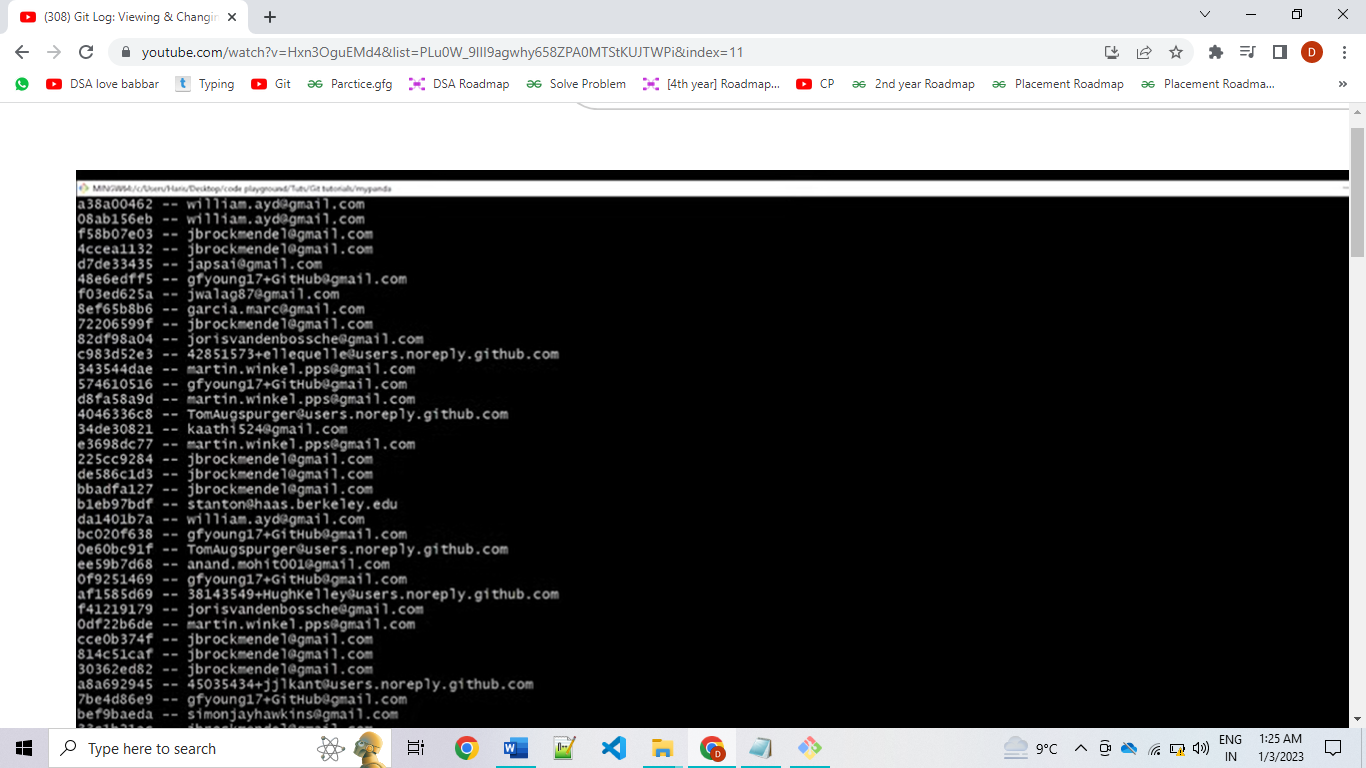
%h is for hash value of commit

%an is for Author Name

%ae is for Author Email

%cn is for Committer Name

%ce is Committer Email



*To do some changes in the last commit done by anybody, changes in message adding more modifications*

*git commit –-amend*

So if more changes are staged, at this commit those changes will be added to last commit and not be added as a new commit. To change message you will be shown a **Vim editor.** In Vim editor, to do changes press “i”. Then do modifications. To submit the changes press “esc” then write “:wq” at the box below.

*To unstage a file*

*git restore –-staged filename.ext*

*git restore –-staged .* (for all staged files)

*To unmodify a file, to change the current content of a file to the content that was commited*

*git checkout –- filename.ext* or *git restore filename.ext*

*git checkout -f* (for all files)

*git checkout -- .* (for all files)

It umodifies only the unstaged changes, not the staged ones.

*View a previous commit (READ only)*

*git checkout <commit id>*

*To go back to a previous commit and delete all commits ahead of it*

*git reset <commit id> {adding --hard is preferred}*

**Working on GitHub with Git**

If we “push” code from local system to GitHub, then the file will be saved to a GitHub repository. If we “pull” code from GitHub to local system, we can get a folder from GitHub.

GitHub just host the files that are tracked by Git in the form of a website so that it can be viewed and managed by multiple members of a group/organization. “remote” means web-host like: GitHub, it is remote connection to GitHub repository.

**To push a local directory to GitHub**

Step 1: Create a GitHub repository by clicking ‘+’ button on GitHub website.

Step 2: Add URL for “remote” in Git. This is done by using command: *git remote add origin* [git@github.com:you/repo\_name.git](mailto:git@github.com:you/repo_name.git) This link can be found by clicking on ‘Code’ button on GitHub repository. This command initializes this repository as ‘remote’ with the name of remote as ‘origin’. Now, any ‘push’ file performed will be done to this repository.

To verify remote use command: *git remote -v* or use *git remote*

To mention which branch would the local files be added to, use: git branch -M main

Step 3: To push the changes in your local repository to the remote repository you specified as the origin use command: *git push -u origin master*. (push to the master repository of origin)

Step 4: But, this says that you don’t have permission to access the repository. To get access to the GitHub repository to push/pull, one must generate SSH key and deploy that for the system that you want to grant access.

You can access and write data in repositories on GitHub.com using SSH (Secure Shell Protocol). When you connect via SSH, you authenticate using a private key file on your local machine. You can generate a new SSH key on your local machine. After you generate the key, you can add the key to your account on GitHub.com to enable authentication for Git operations over SSH for your system.

An SSH key is an access credential for the SSH (secure shell) network protocol. This authenticated and encrypted secure network protocol is used for remote communication between machines on an unsecured open network.

Step 5: To generate SSH key on local system use the below commands:

*ssh-keygen -t ed25519 -C <your email address that is registered on GitHub>* (creates a SSH key to access your whole account, not just one repository).

To register email address on GitHub, In the upper-right corner of any page, click your profile photo, then click Settings. In the "Access" section of the sidebar, click Emails. In "Add email address", type your email address and click Add. Verify your email address.

*eval "$(ssh-agent -s)"* (start the ssh-agent in the background)

*ssh-add ~/.ssh/id\_ed25519* (adds your SSH private key to the ssh-agent)

*tail ~/.ssh/id\_ed25519.pub* (returns the contents of id\_ed25519.pub which is the SSH key)

Step 6: Copy the SSH key. Go to settings in GitHub, then go to SSH and GPG keys. The click, add ‘New SSH key’. Add its title and paste the copied SSH key to the box below ‘Key’ heading. At last, click ‘Add SSH key’.

Step 7: Now use command git push -u origin master as explained in step 3.

*To remove a connection*

git remote remove <alias>

*To rename a connection*

git remote rename <old> <new>

Rather than writing a big command we can change the big command to a small substitute using aliasing in git

Syntax: *git config --global alias.substituteName ‘big command’*

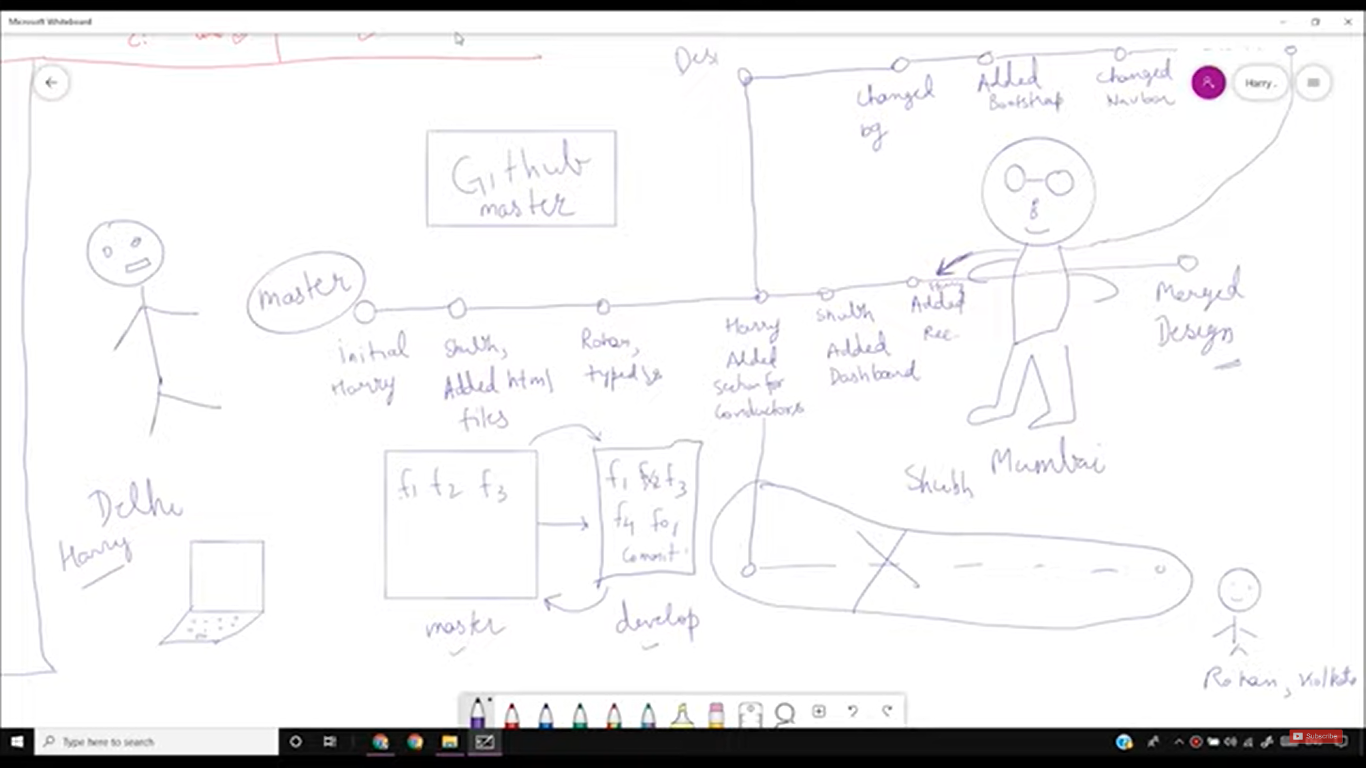
*git config --global alias.last 'log -p -1'*

*git config --global alias.st status*

**Branching in Git**

Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

Many companies try to develop their products by doing some changes to it. These changes could not be done in the main product files, because changing them may incur mistakes and can put the system down. We all have heard that server of big MNCs got down for one day and they must incur billion of losses. So, either change have to be done in a copy of the product. But that would waster a lot of memory space.



Branch is like a local space (or RAM) and main branch is like secondary memory (or HD). The branch can be merged with main branch, merging adds all files in branch to the main branch. “master branch” is the main branch by default.

git branch lists all the available branches.

git branch develop creates a new branch with name = “develop”.

git checkout master lets user switch to master branch.

git checkout -b develop to create a new branch and switch to it.

git merge branchName to merge a branch to the current branch.

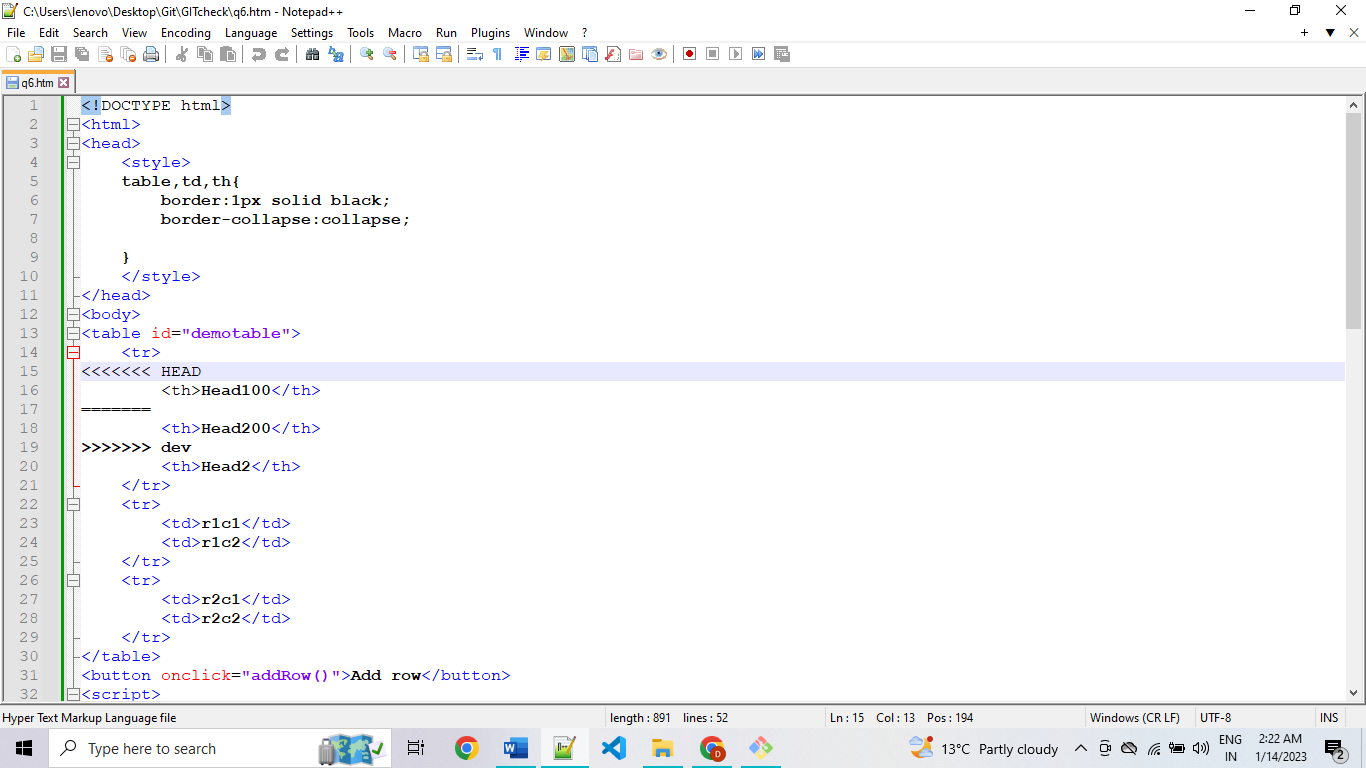
To merge branch A to branch B:

git checkout B

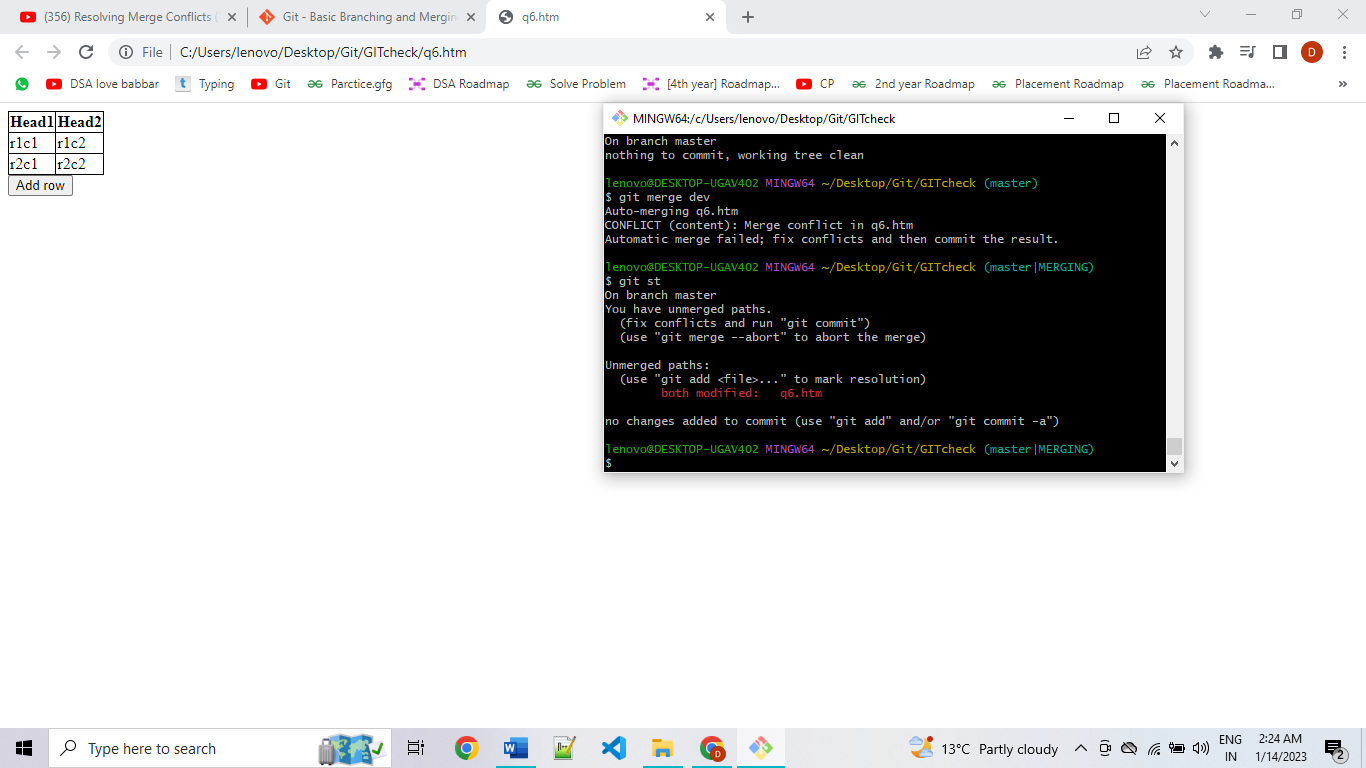
git merge A

**Merge Conflicts**

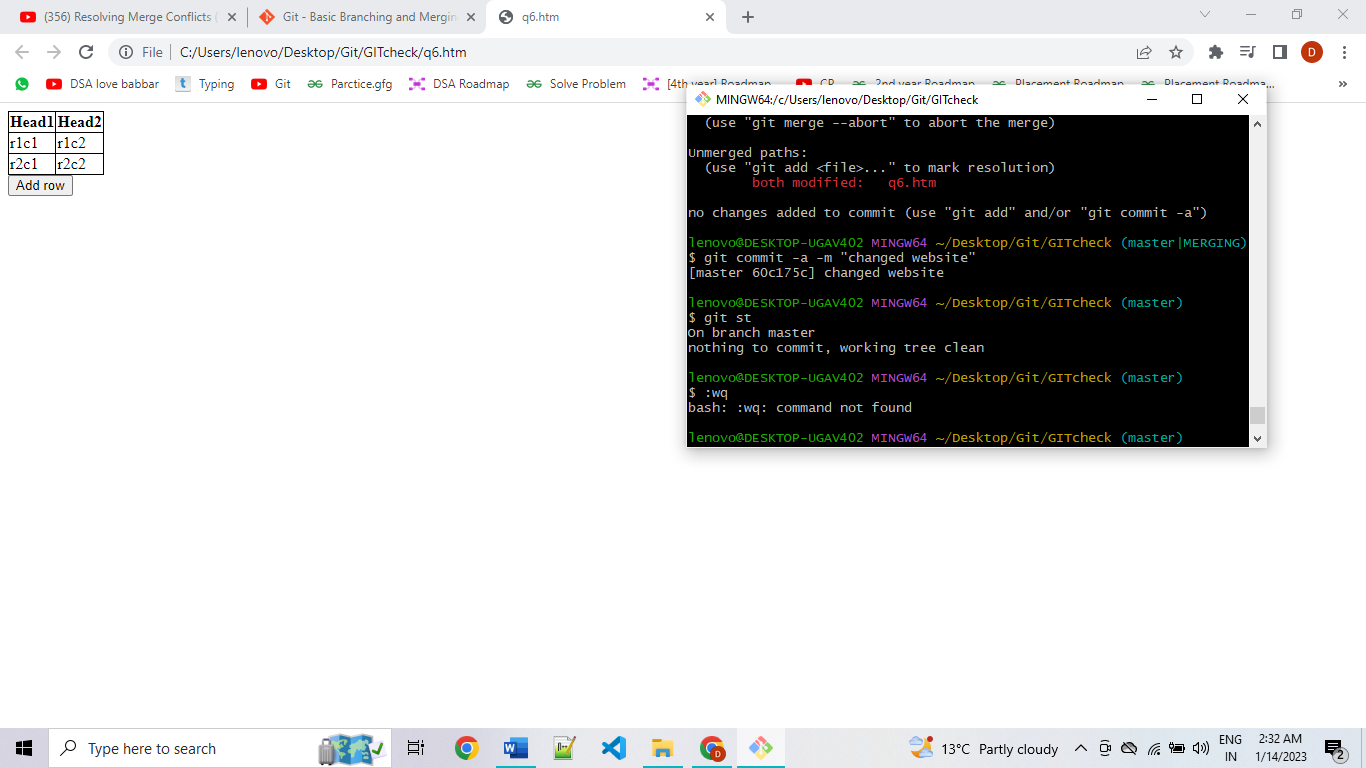
So, now merging branches may have conflicts. Conflicts include that GIT gets confused if it wants to keep the content of one branch or the main branch. The two contents will be shown as <<<< one content ====== second content >>>>.

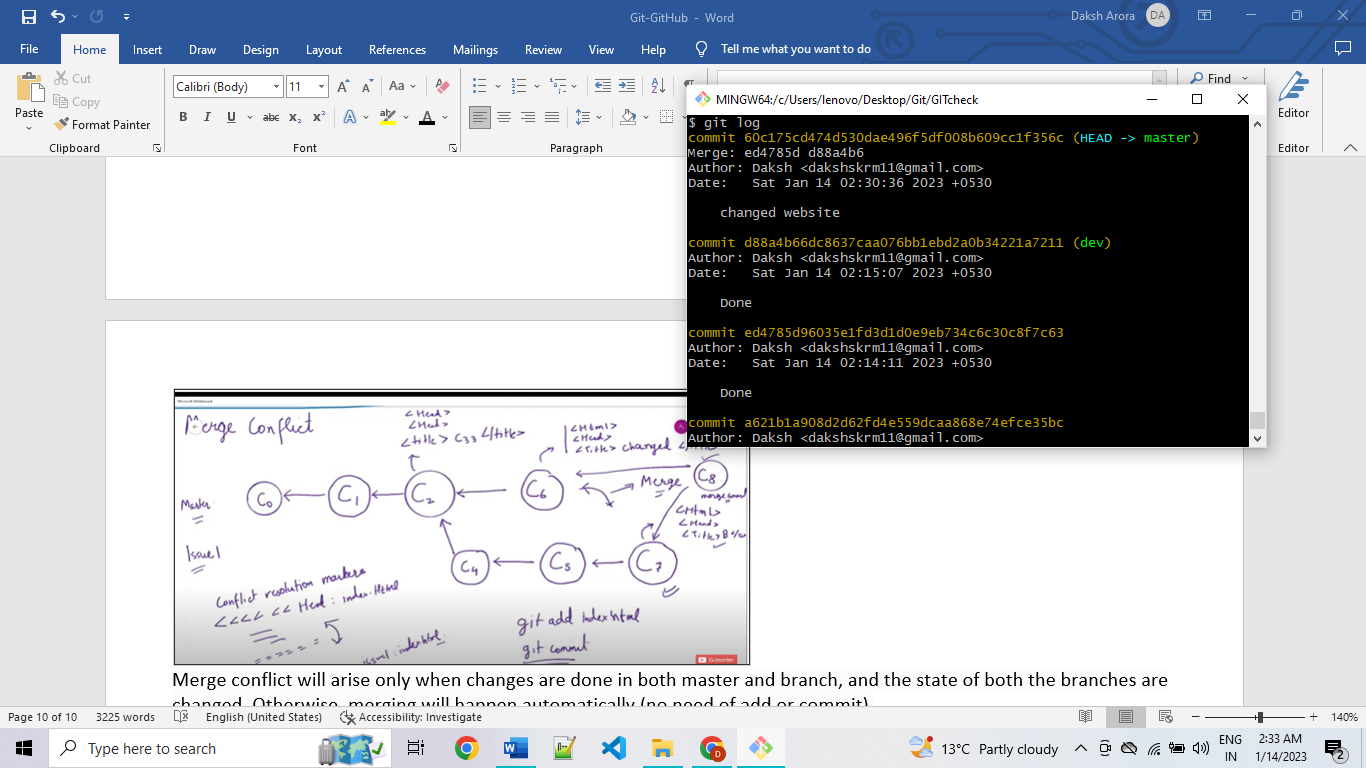


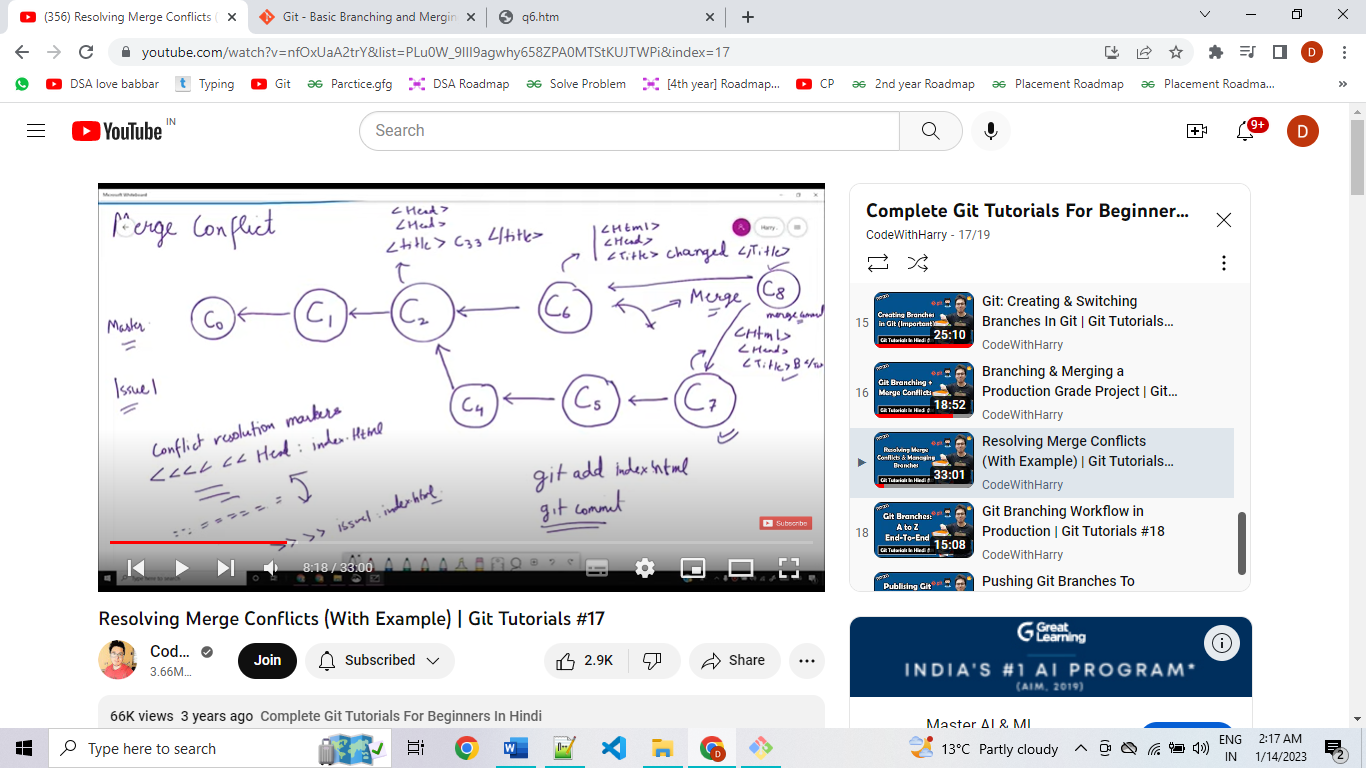
The user must remove this whole part and replace it with the content the user wants.

(git add and commit are required).

Merging is seen as an operation in main branch. After this replacing, stage the changes done and commit them (it will be seen as merge commit).







Merge conflict will arise only when changes are done in both master and branch, and the state of both the branches are changed. Otherwise, merging will happen automatically (no need of add or commit). \*\*If not changes have been done to main branch, then content of master will be changed to content of branch (if commit c6 was not done).

*git branch -v* returns list of last one commit (the commit hash and commit message) done by each branch.

*git branch –merged* returns the merged branches (here, master and dev).

*git branch –no-merged* returns the not merged branches.

Deleting branches

*git branch -d branchName* (deletes merged branches only, gives error if trying to delete not merged branches)

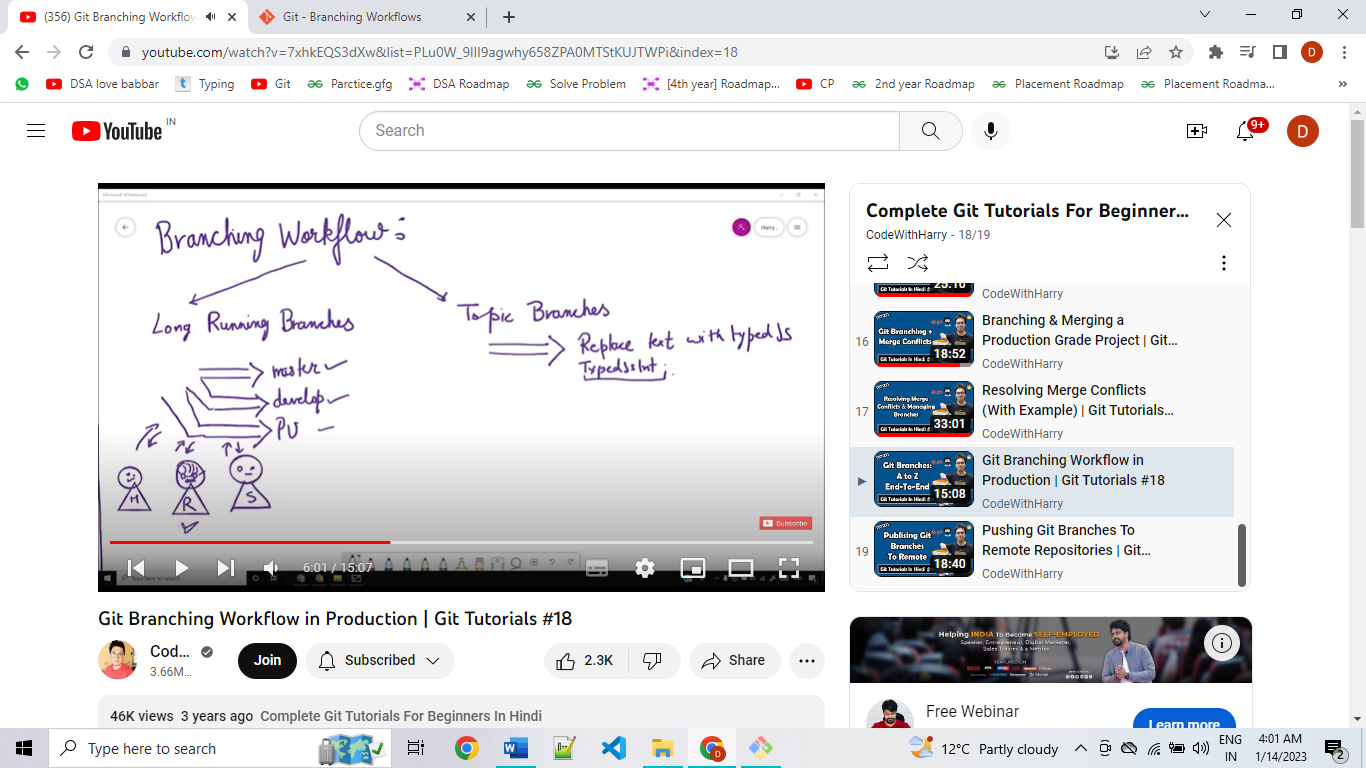
*git branch -D branchName* (deletes not merged branches also)

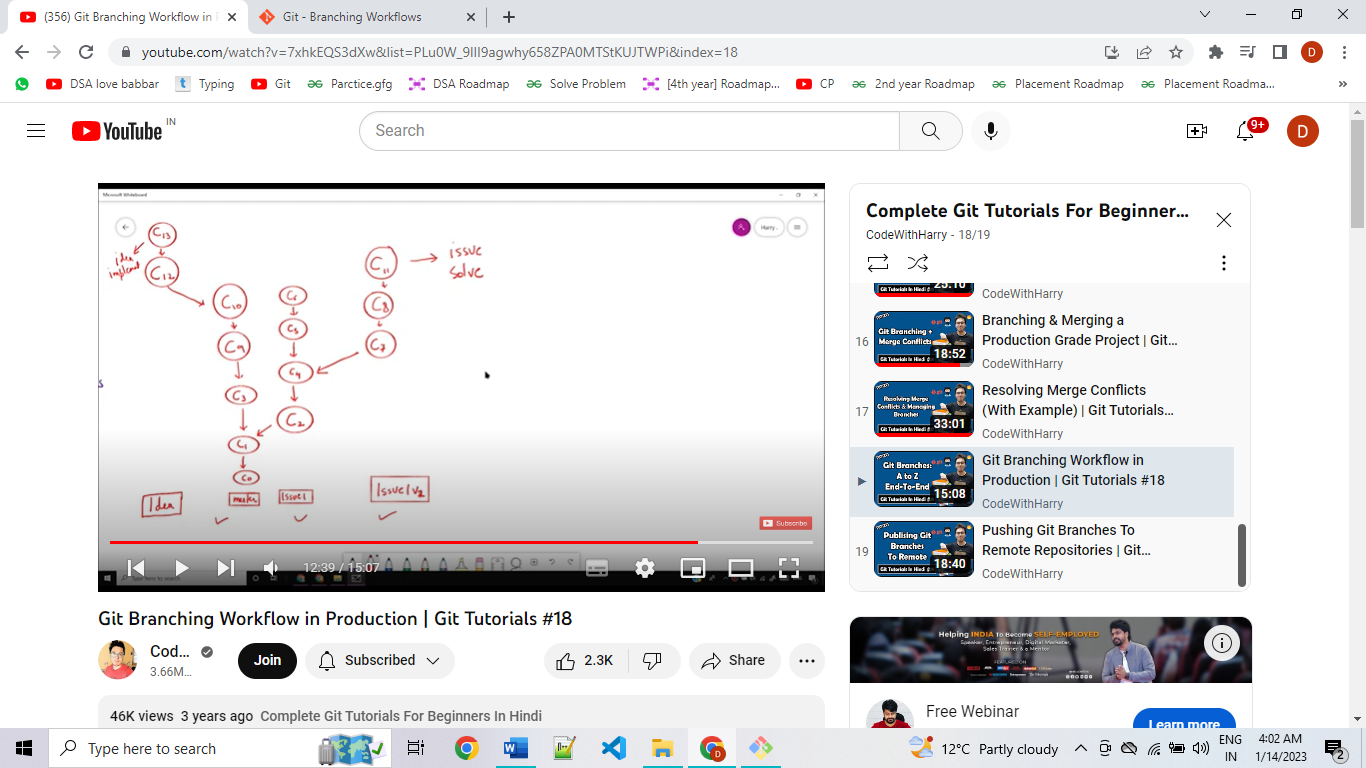
**Branching Workflow**

Types of branching workflows are:

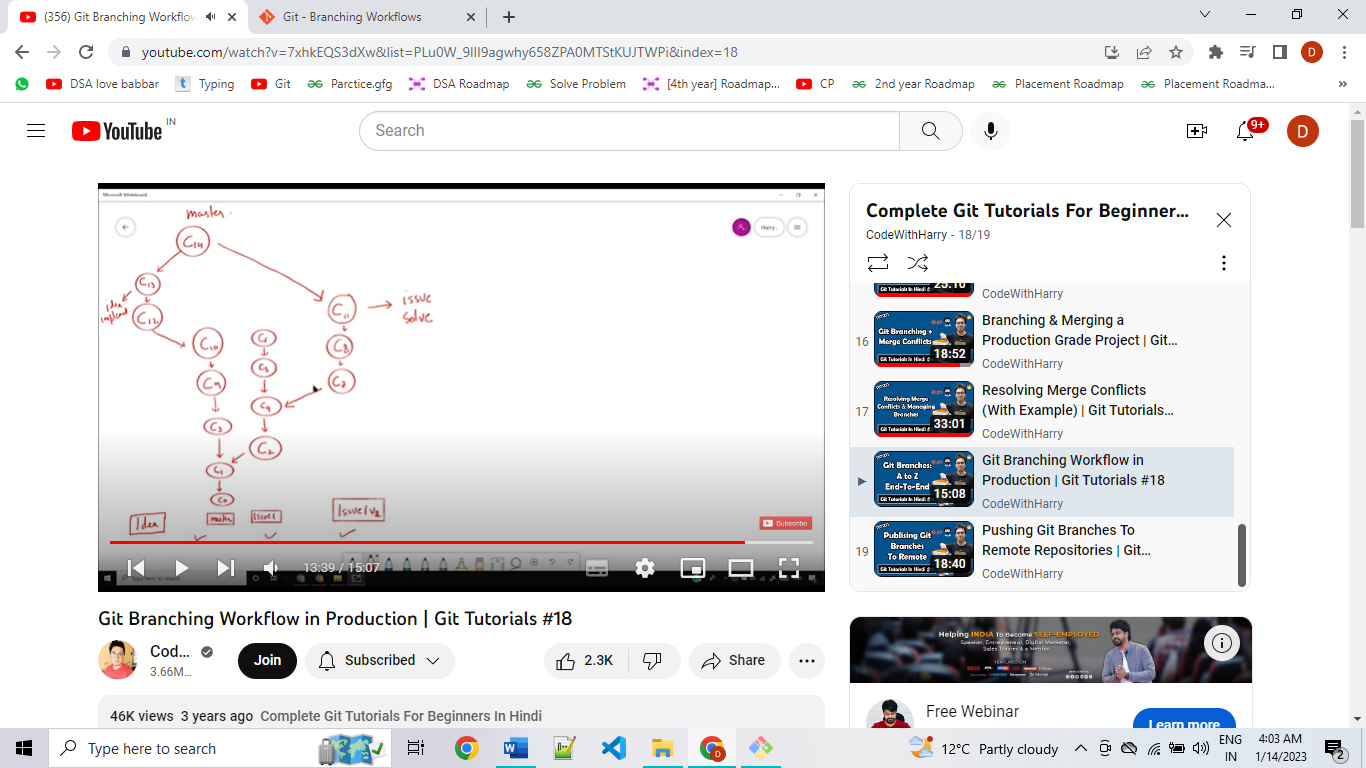
1. Long-Running Branches: This includes branches that are always open and that you use for different stages of your development cycle, you can merge regularly from some of them into others. They will be there for a long period of time until the product is running/useful.

2. Topic-Branch: A topic branch is a short-lived branch that you create and user for a single feature or related work. They will be there for a short period of time until the feature is not added or the issue is not resolved.





Now, if ‘idea’ and ‘issue v2’ branches are approved and must be merged to master, the git branch tree would look as below:



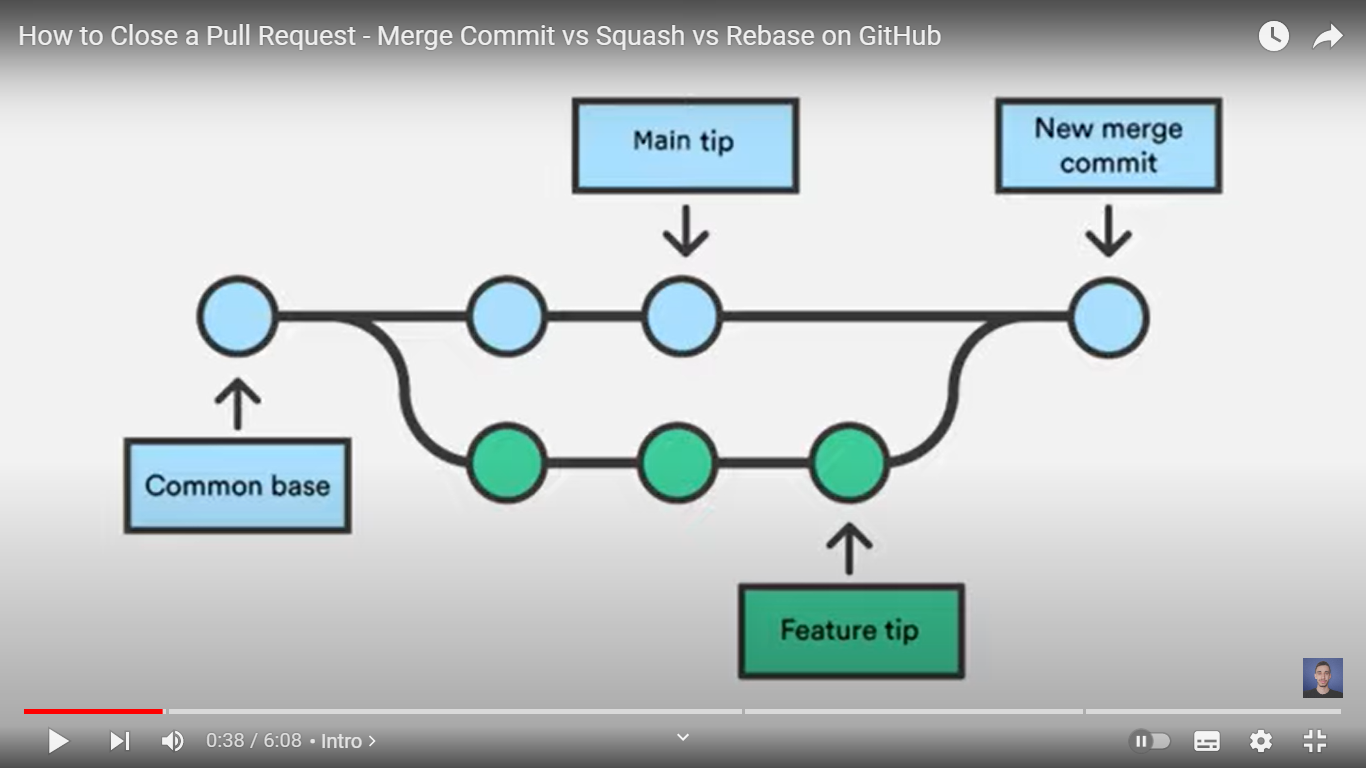
**How to push branch locally to GitHub**

* Use *git push origin branchName*
* To create a copy branch of the branch in GitHub use *git push origin branchName:copyBranchName*
* To delete a branch on GitHub (remote repository) use *git push -d origin branchName*
* *Git fetch* is the command that tells the local repository that there are changes available in the remote repository without bringing the changes into the local repository: *git fetch <remote> <branch>*
* Git Pull on the other hand brings the copy of the remote directory changes into the local repository: *git pull <remote> <branch>.*

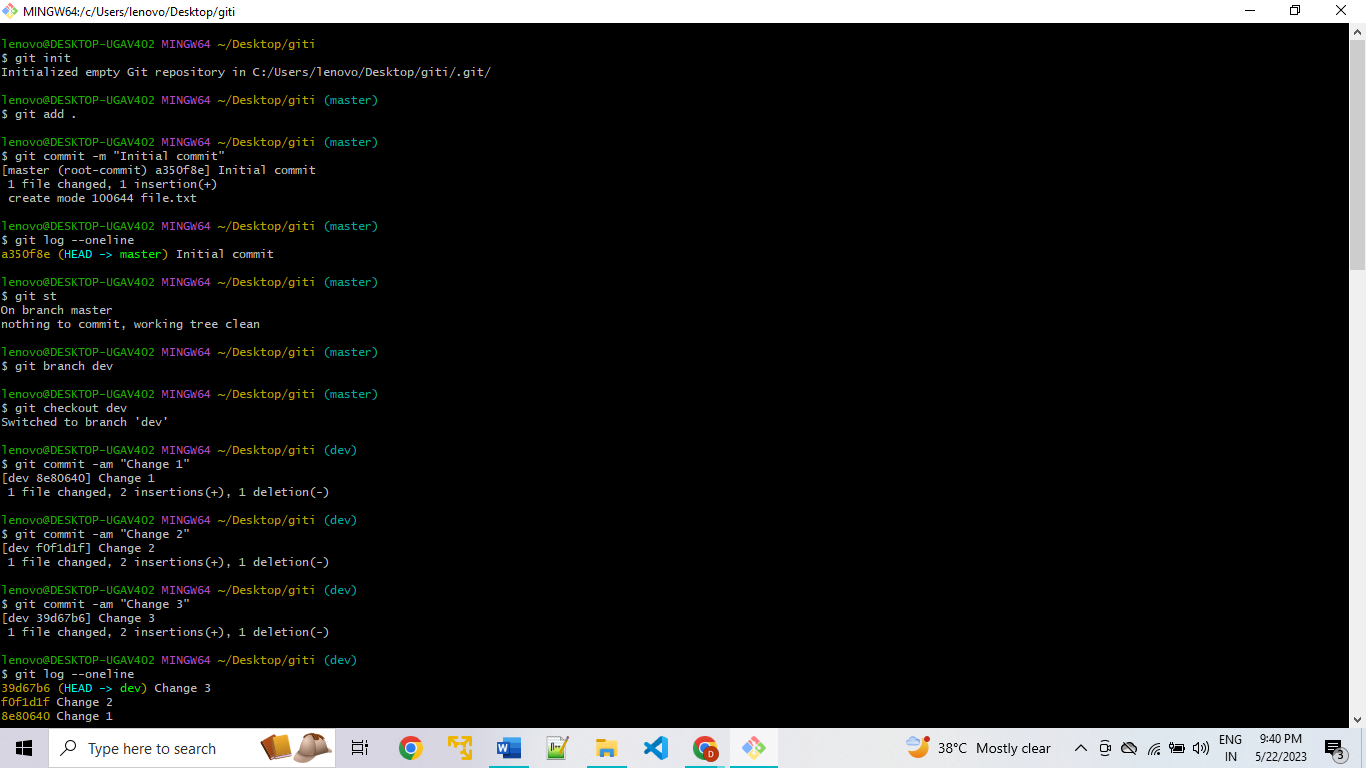
**Squash Merge**

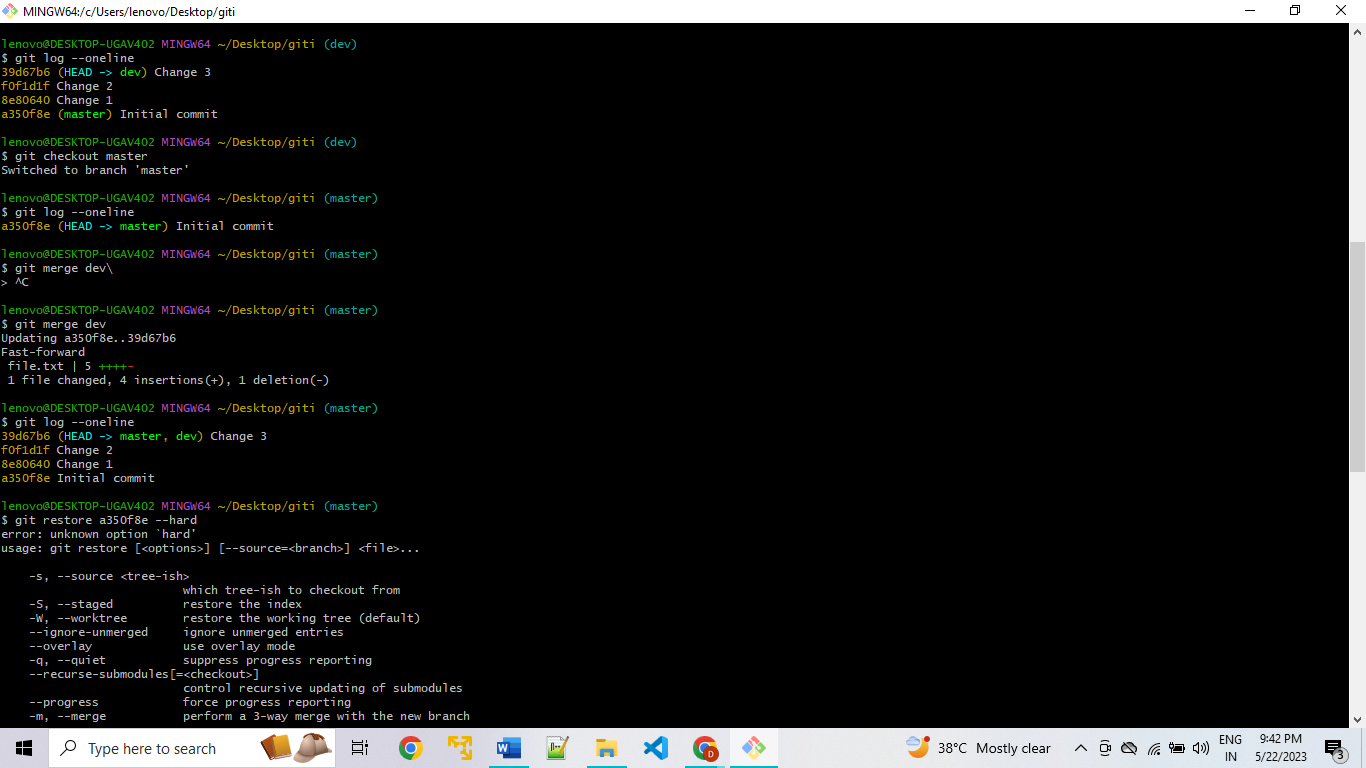
The question is, do you really need to keep track of every single commit? Including typos, missing files, formatting… if the answer is no, then you should consider Squash merge. When squashing a merge, the result is that you get rid of all commits on the branch and only add a single commit on main with all the content. As a result, the history is much cleaner and while working on a branch you can do all the commits you want.

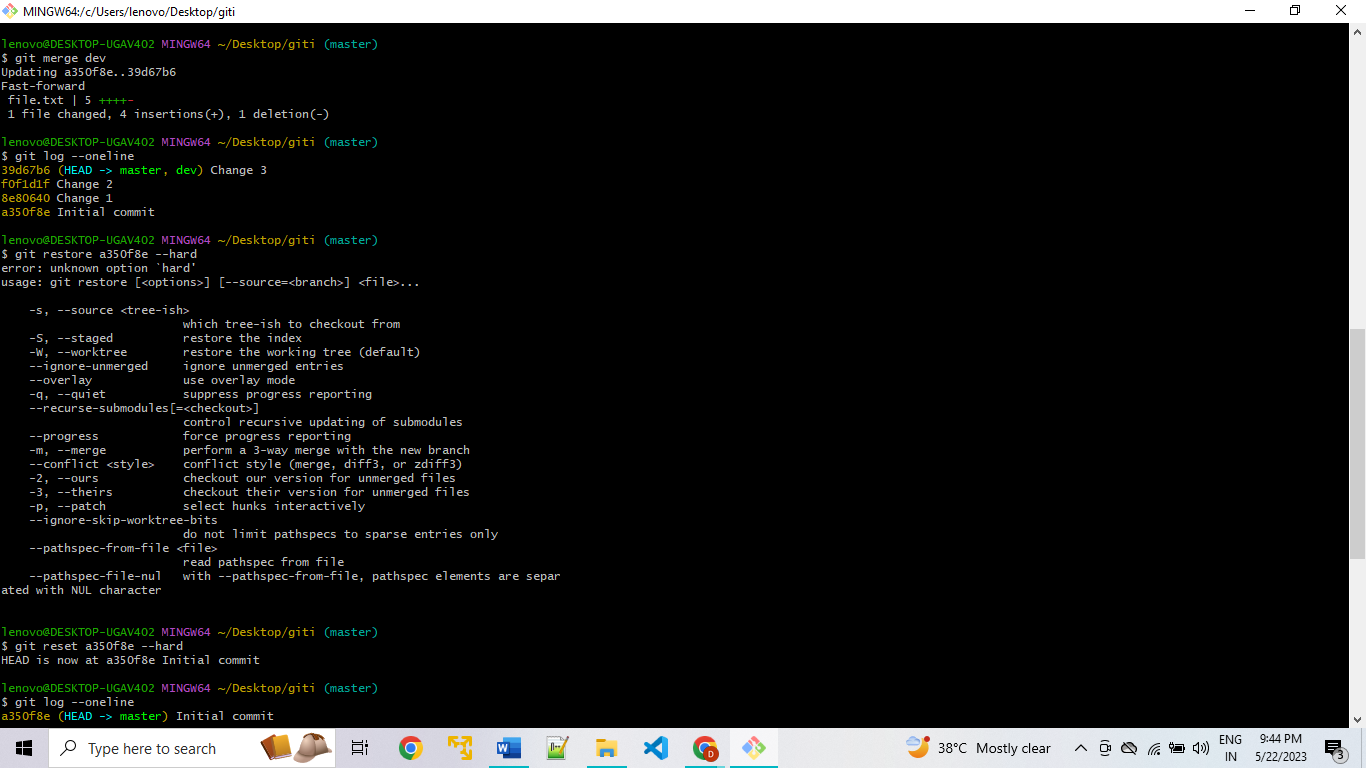
When we do normal merge operation on branches:



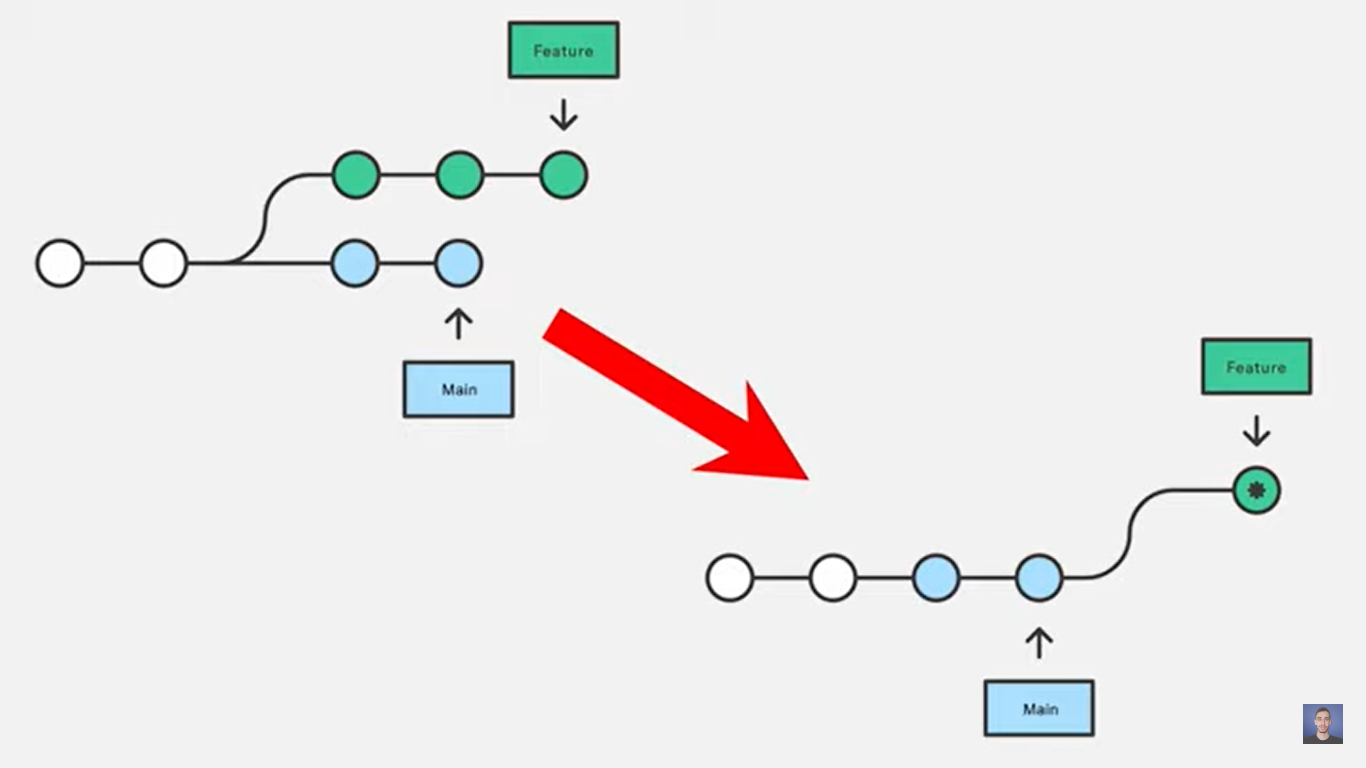
* The merge commit is done on the main branch.

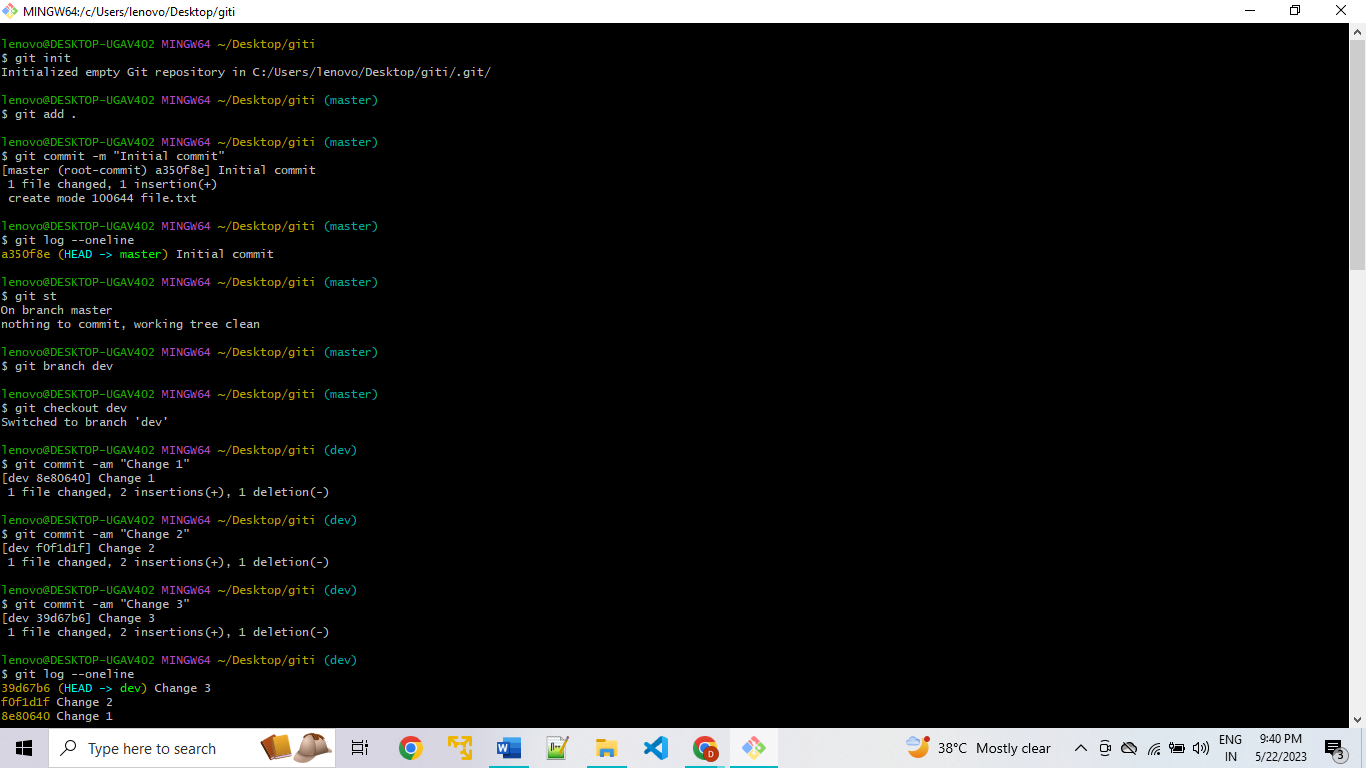


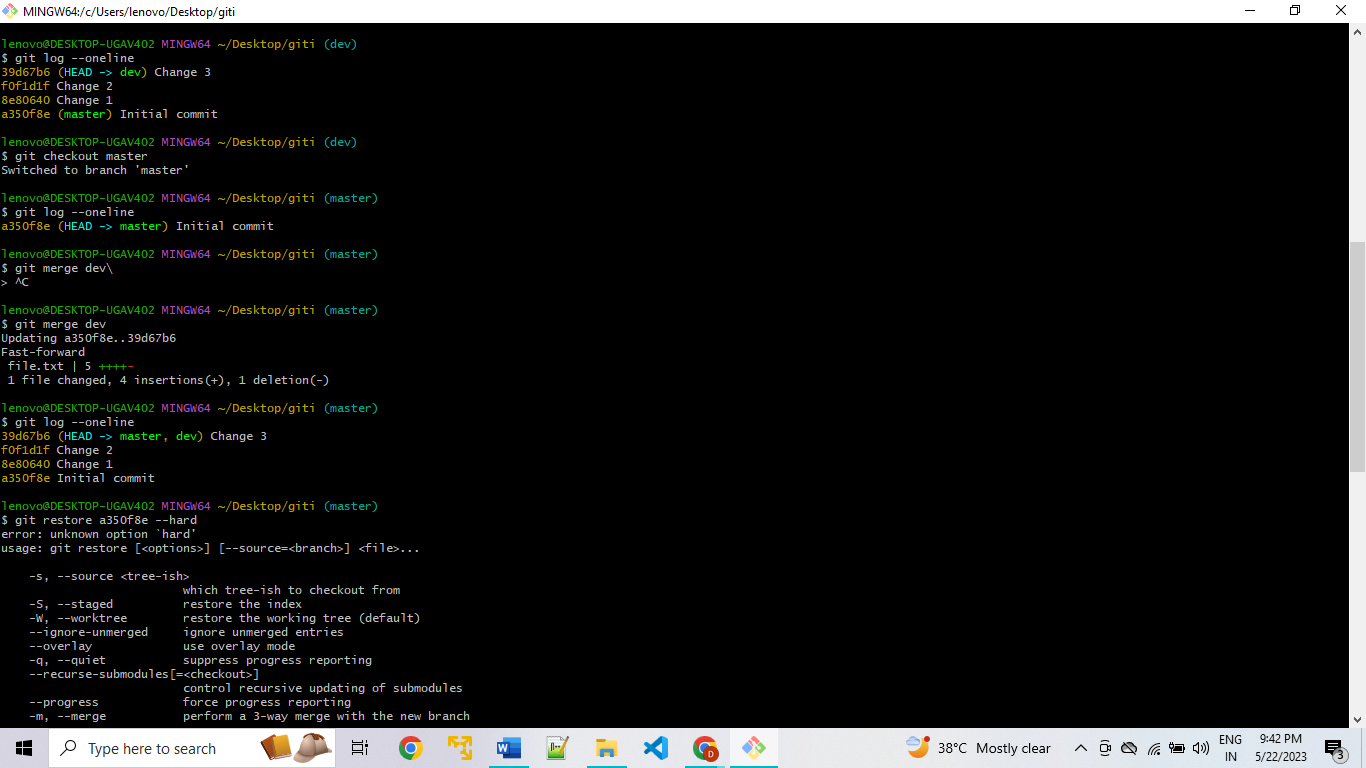


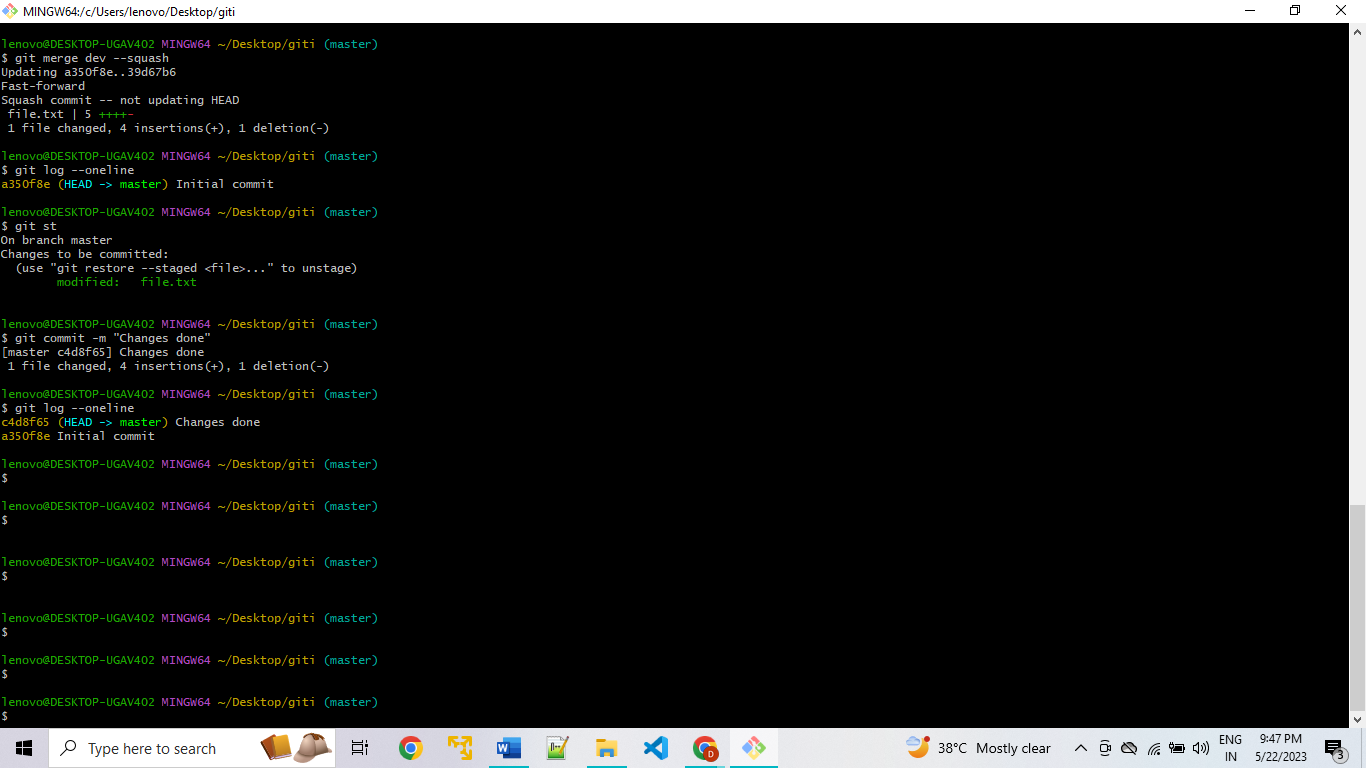


When we do merge squash operation on branches:



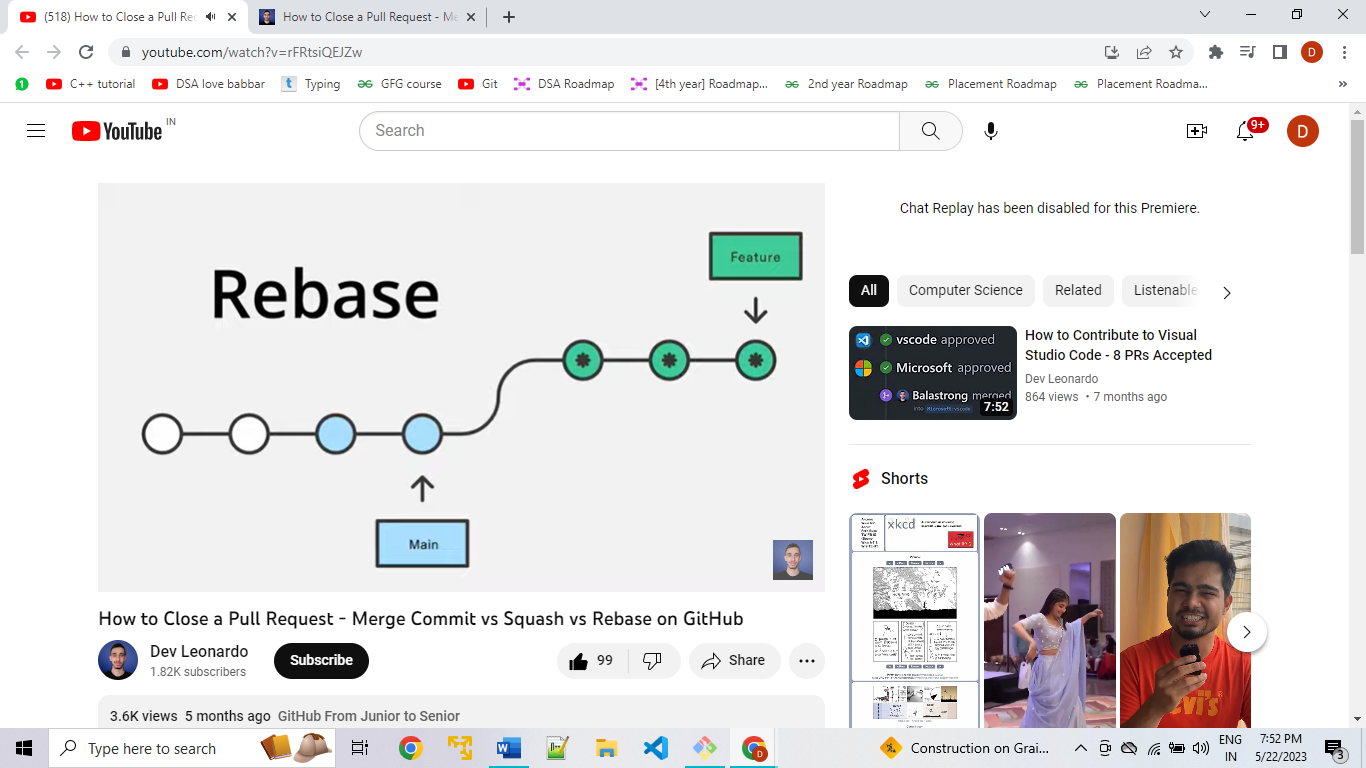




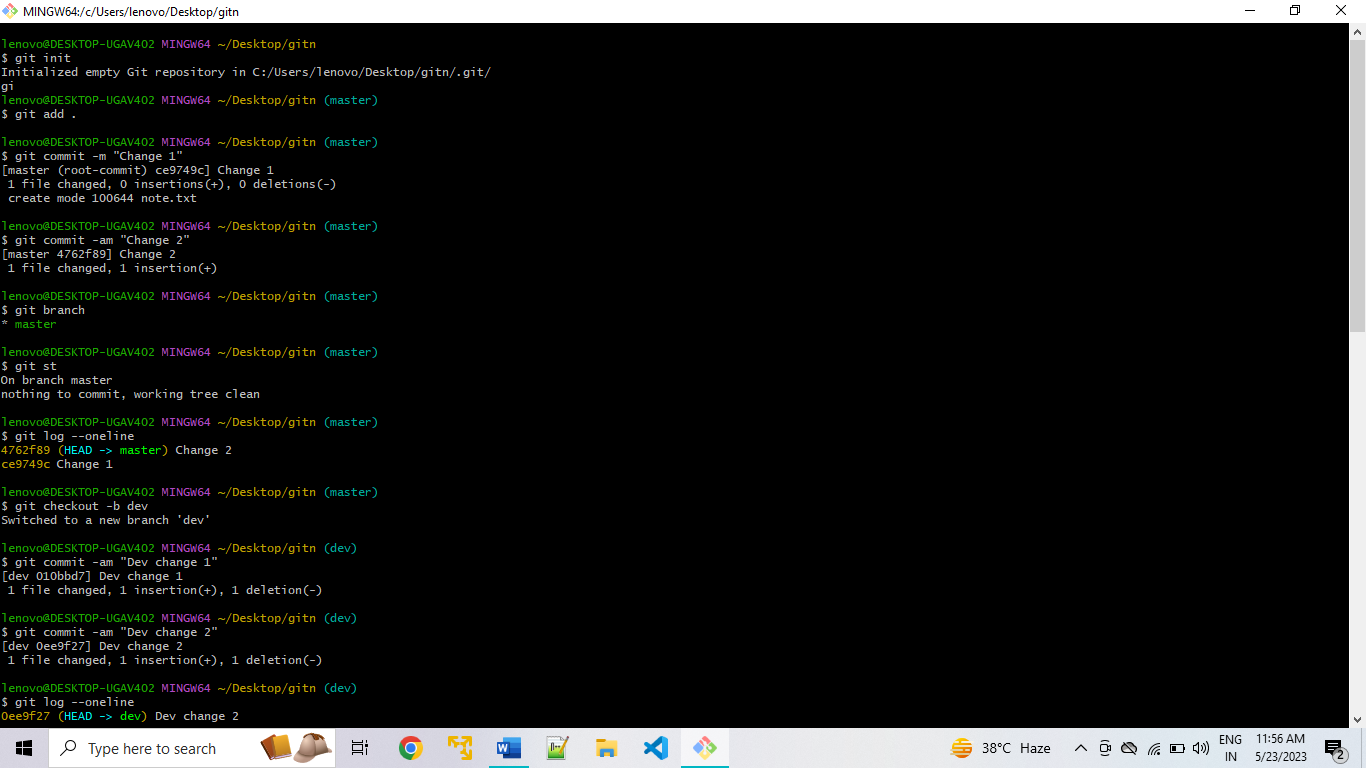


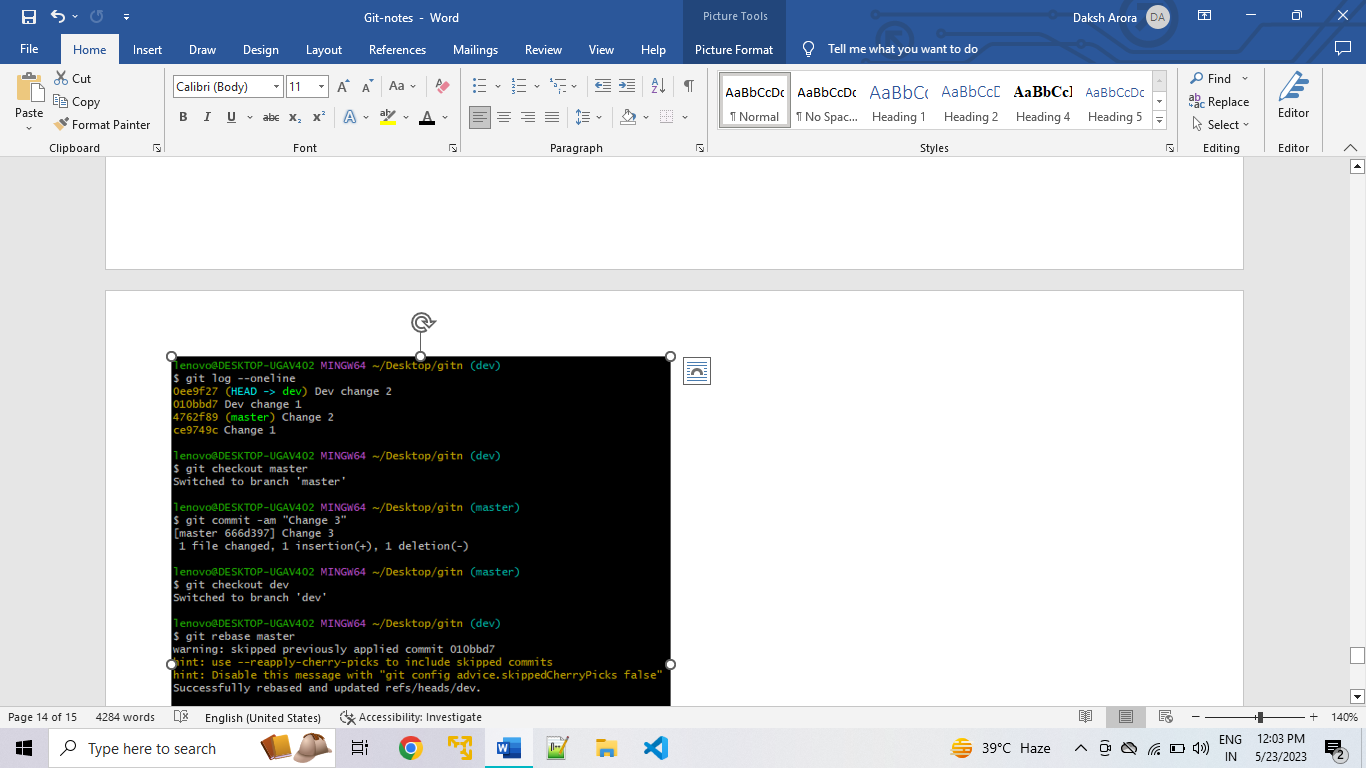
* After merge, no commits were added to “master” branch but a change was staged that would represent all commits in “dev” branch.

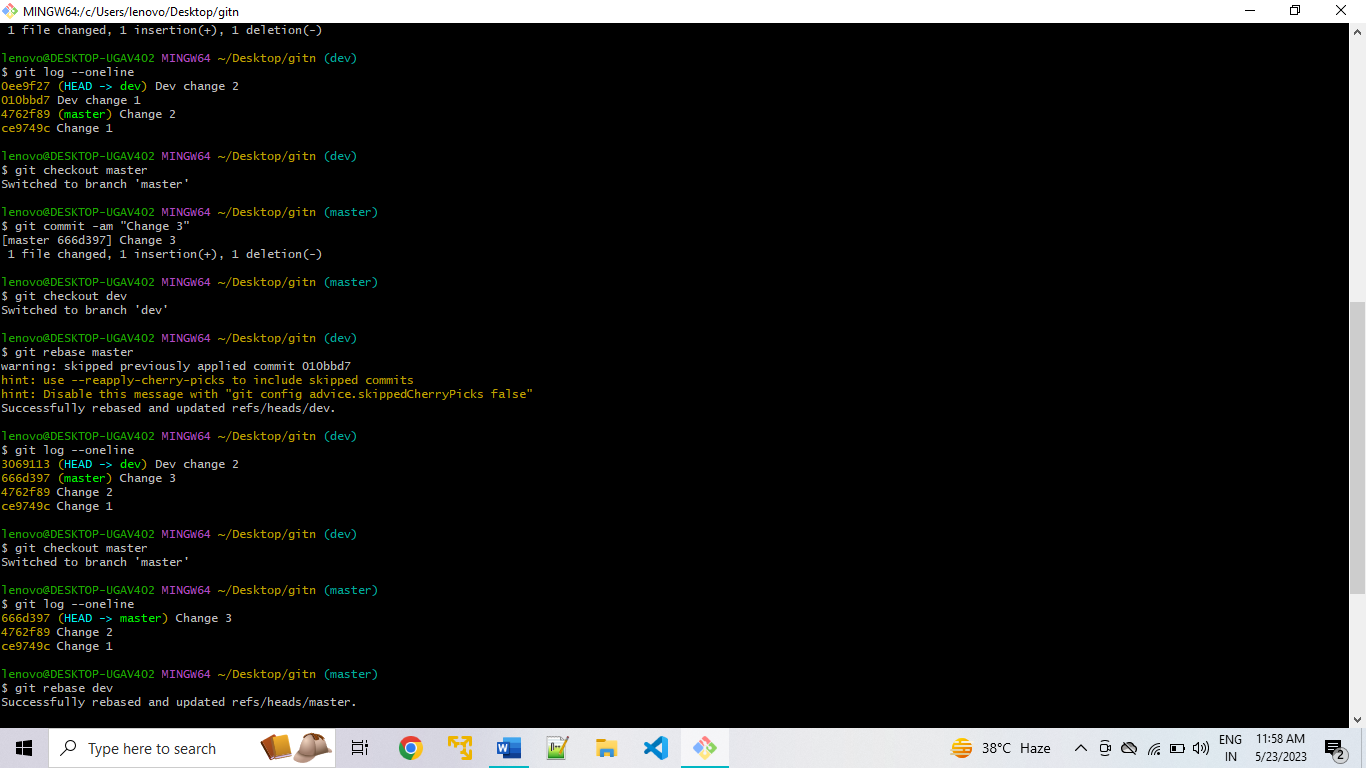
When we do rebase operation on branches:

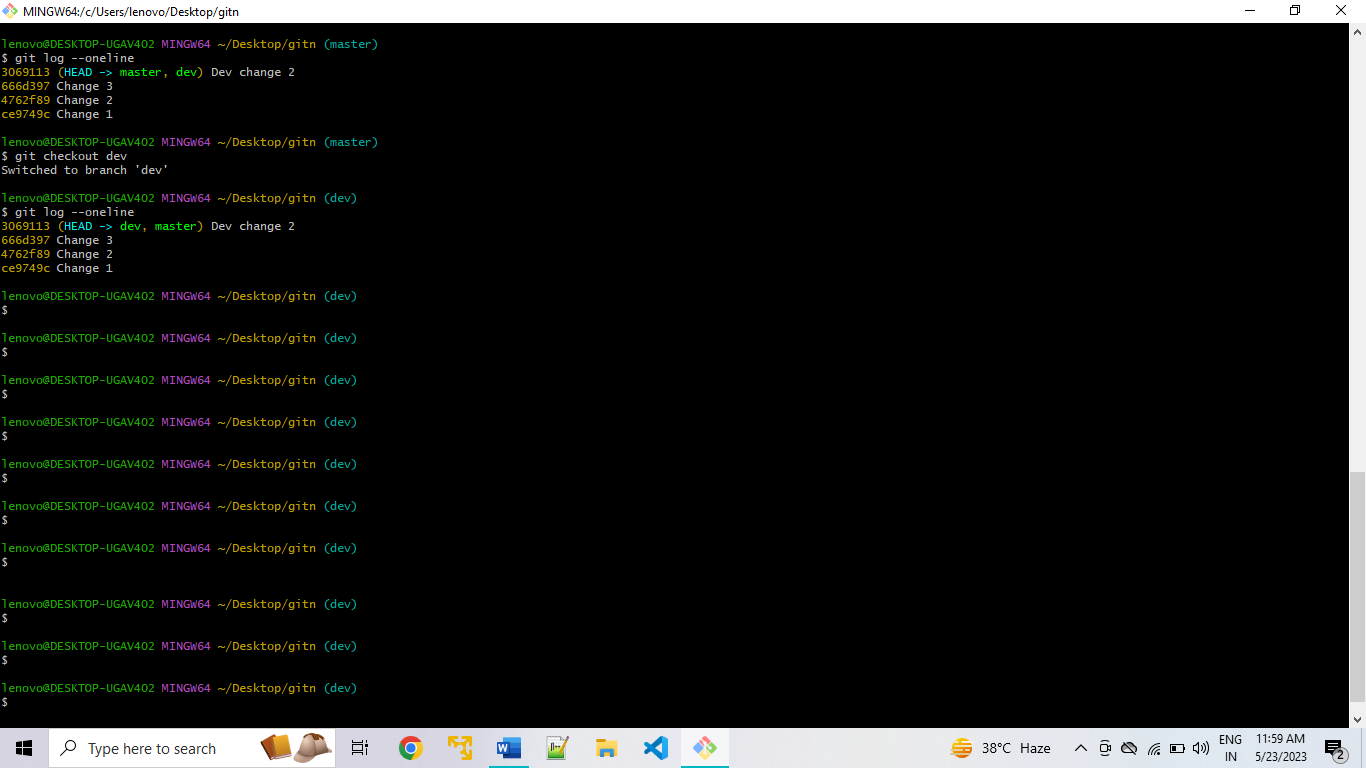


* This would mean solving merge conflict for each commit added on main branch.
* When performing an interactive rebase all commits except the first one (bottom in the list) can be dropped.









* The end objective for merge and rebase commands is same, but their usage varies.

|  |  |
| --- | --- |
| Merge | Rebase |
| Git merge is a command that allows you to merge branches from Git. | Git rebase is a command that allows developers to integrate changes from one branch to another. |
| In Git Merge logs will be showing the complete history of the merging of commits. | Logs are linear in Git rebase as the commits are rebased |
| All the commits on the feature branch will be combined as a single commit in the master branch. | All the commits will be rebased and the same number of commits will be added to the master branch. |
| Git Merge is used when the target branch is shared branch | Git Rebase should be used when the target branch is private branch |