Session 2

git

GitHub

Agenda

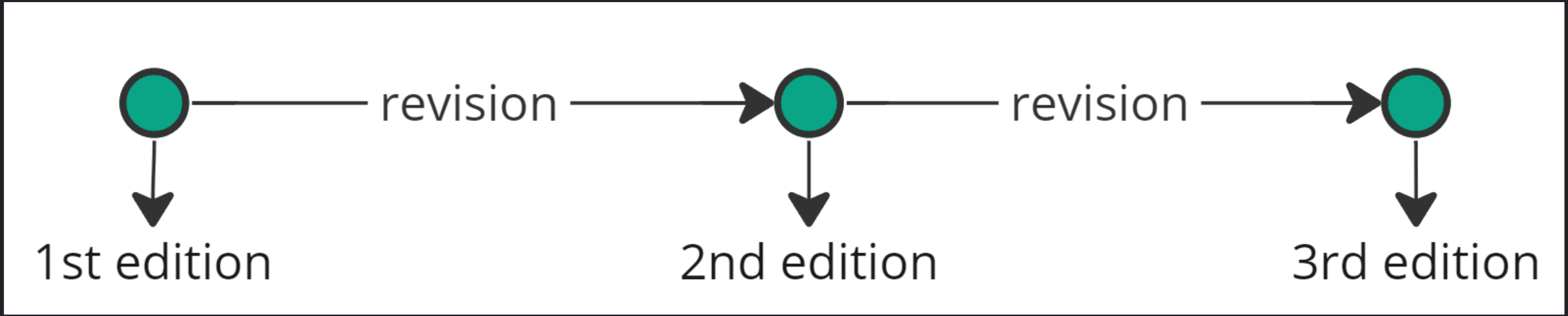
- What is Git?
- What are the Git components?
- How to Setup Git?
- How To Work UsingGit?
- How to work with Others Using Git & Github?

What is Git?

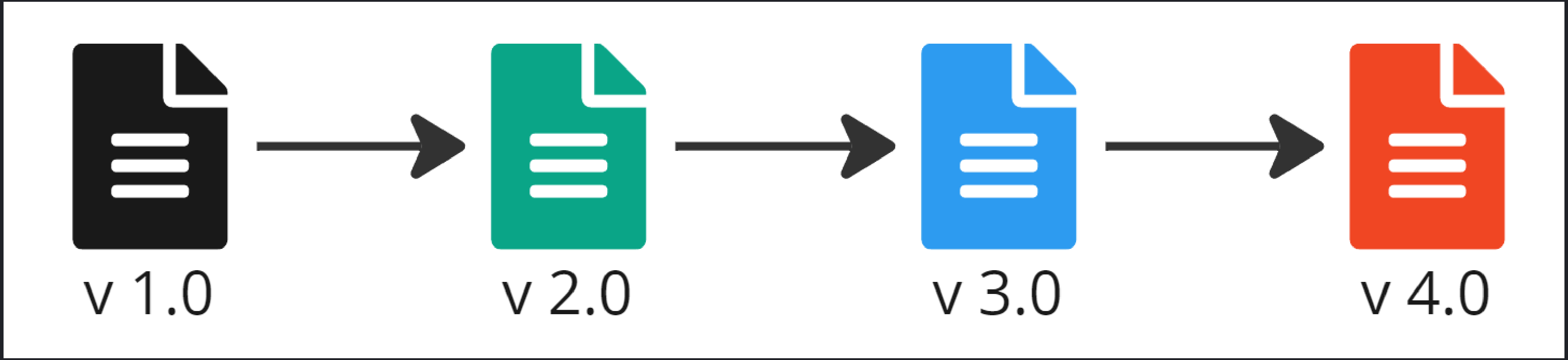
Git is a **D**istributed **V**ersion **C**ontrol **S**ystem **DVCS**.

What is a Version?

- A version is a stable release of your work.



- Renaming: Primitive way to manage versions.
  - We would store every version made with a different name or number to distinguish which one is the newest one.



- In simple cases Renaming will be sufficient, but it won't work with more than one file.
- That's where Version Control System Comes in Handy.

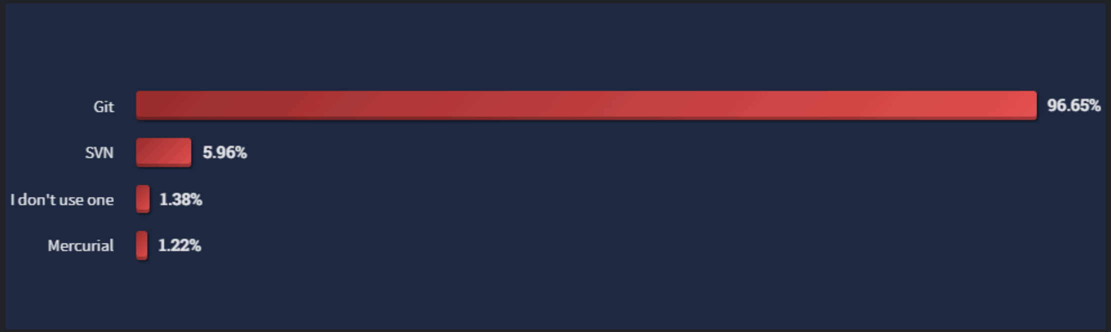
What is a Version Control System?

- Version Control System is a software that helps developers:

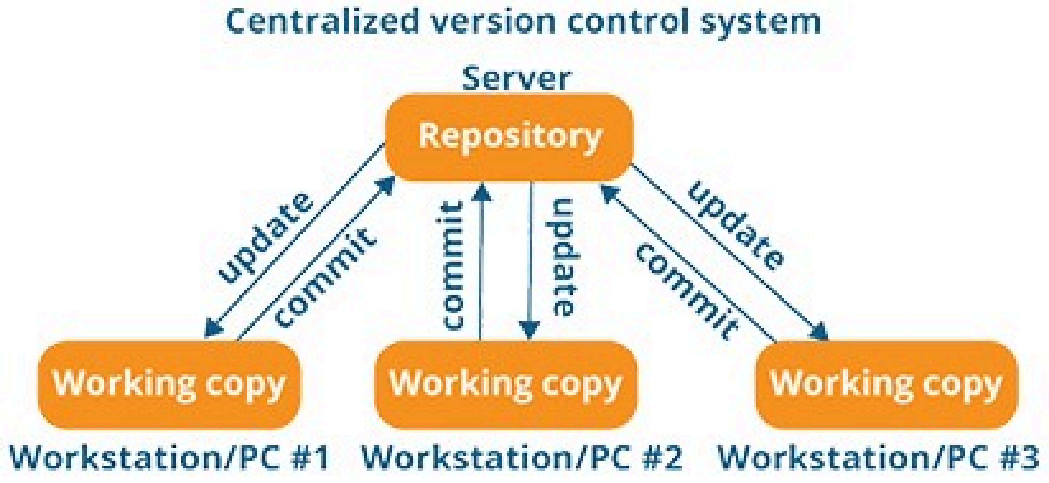
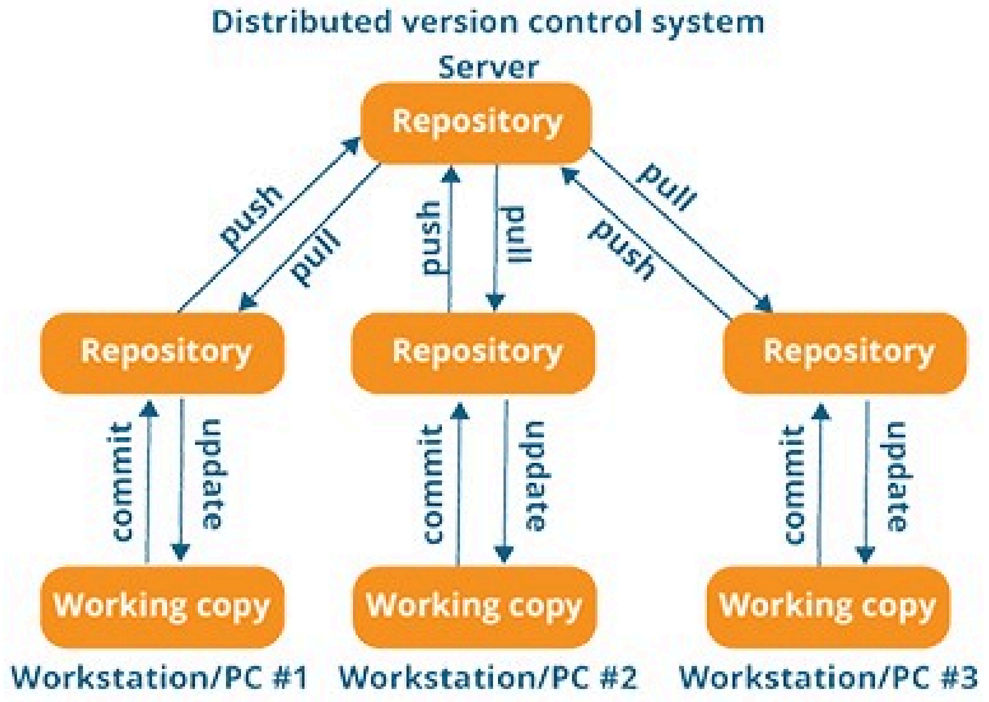
  → Keep track of all changes.
  → Access previous versions and restore them if necessary.
  → Allow people to work on the same code base simultaneously.

- Examples:
  - Git.
  - Subversion.
  - Mercurial.

- VCS survey:



Centralized Vs Distributed VCS


Centralized version control system
Server
Repository
update
commit
commit
update
commit
update
Working copy
Working copy
Working copy
Workstation/PC #1
Workstation/PC #2
Workstation/PC #3

Distributed version control system
Server
Repository
push
pull
push
pull
pull
push
Repository
Repository
Repository
commit
update
commit
update
commit
update
Working copy
Working copy
Working copy
Workstation/PC #1
Workstation/PC #2
Workstation/PC #3

- **CVCS:**
  - **Centralized History**
  - **Require Internet**
  - **Slow**
  - **More Conflicts**
- **DVCS:**
  - **Distributed History**
  - **Work Offline**
  - **Very Fast**
  - **Less Conflicts**

Git components

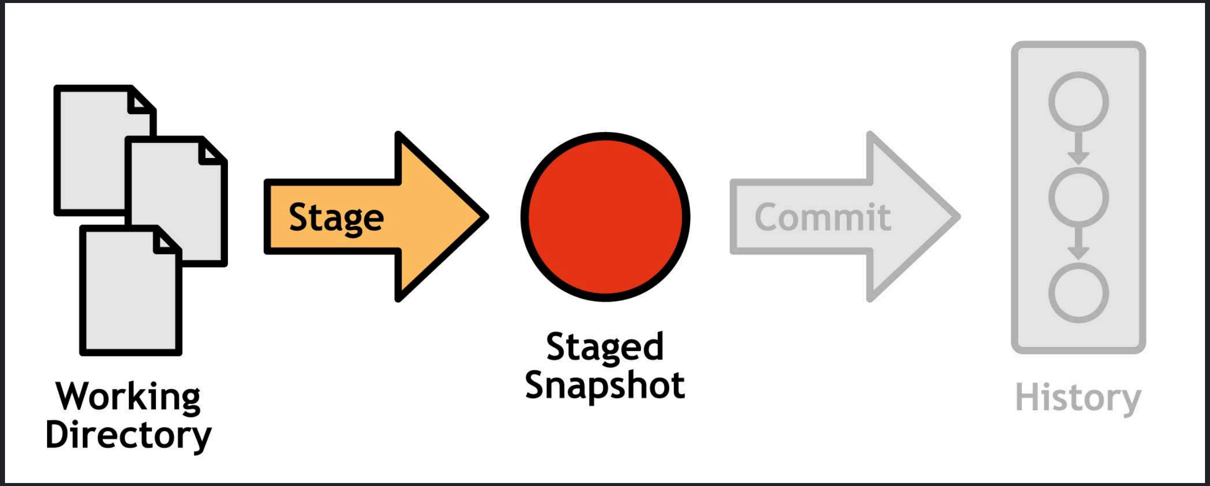
Git has 4 main components:

- Working Directory.
- Staging Area (Index).
- Commit History.
- Developments branches.

Working Directory

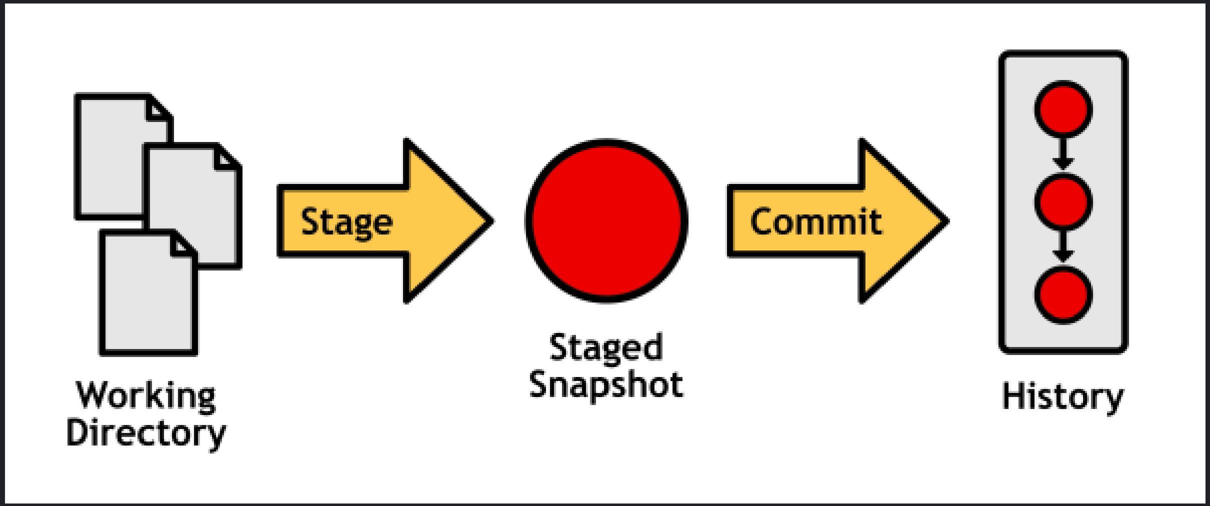
- It is the place where you actually edit files, compile code, and develop your project.
- You can treat the Working Directory as a normal folder. Except, you now have access to all sort of commands that can record, alter, and transfer the content of that folder.

Staging Area (Index)



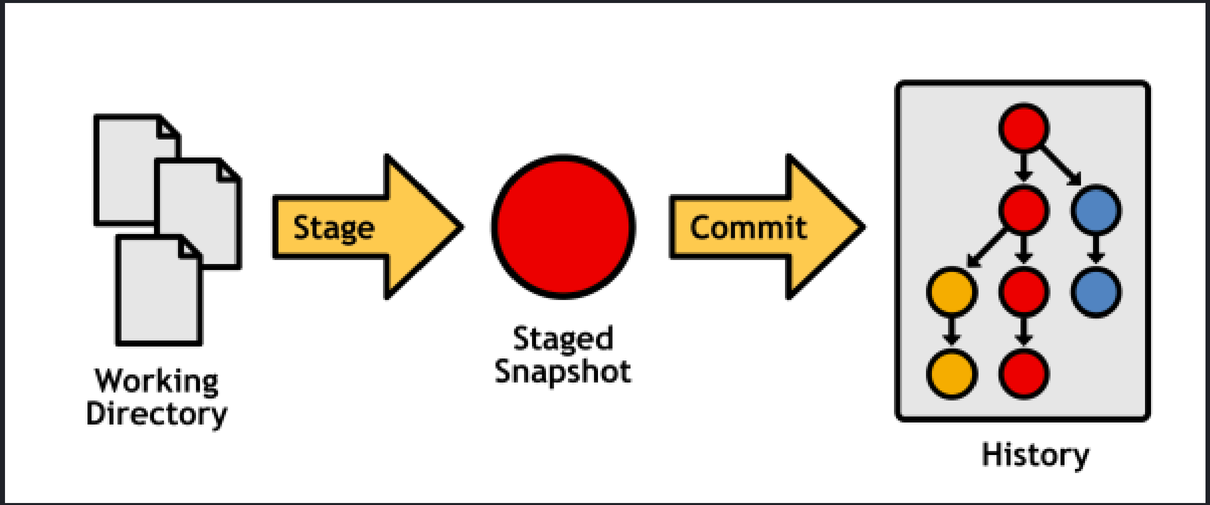
- It is an intermediary point between the Working Directory and the project History.
- Instead of forcing you to commit all changes at once, Git lets you group them into related change-sets.
- Staged changes are not yet part of your history.

Commit History



- Once you have configured your changes in the Staging Area, you can Commit it to the project History where it will remain as a safe revision.

Development Branches



- So far, we are still able to create linear project history, adding one commit on the top of another.
- Branches make it possible to develop multiple unrelated features in parallel.

How to Setup Git?

Git Installation

- git-scm

Git Configration

```
git config --global user.name KenzyAdel
git config --global user.email kenzyadel16@gmail.com
```

How To Work UsingGit?

Git Initialization

`git init`

- Ask Git to keep an eye on your project.
- git init command will create a new sub directory named .git that contains all of your necessary repository files.

Saving changes

1. **Manage Staging Area.**
2. **Viewing Status.**
3. **Ignoring Files.**
4. **Committing Staged Changes.**
5. **Viewing History.**
6. **Moving Inside History.**
7. **Tagging Commits.**

Add to/Remove from Staging Area

- `git add [what to add]`
  - File name: file1.txt
  - Pattern: "*.txt*", ".cpp"
  - Directory name: adds all the contents of the specified directory
  - -A, --all, and ".": adds all not tracked files

- `git rm --cached [what to remove]`
  - **File name: file1.txt**
  - **"*": removes all staged files.**
- **The option `--cached` removes changes only from the index (Staging Area).**

Viewing Status

- View the status of the working directory (**U**ntracked, **A**dded, **M**odified,**D**eleted)
- `git status`
    - `-s` : view status in Short format.

Ignoring Files

- In this case, we tell Git to stop tracking the history of the specified files.
- Create a file named ".gitignore" and specify the titles/pattern of the untracked files.
- ⚠️ Note:
  - You can't include any tracked/staged files in the .gitignore file.

Committing Staged Changes

- Record all staged changes to the repository and add them to the project history as checkpoints.
- `git commit`
  - `-m` : add a commit message (descriptive/clear).
  - Without the `-m` option, Git will open a text editor where you can write your commit message.

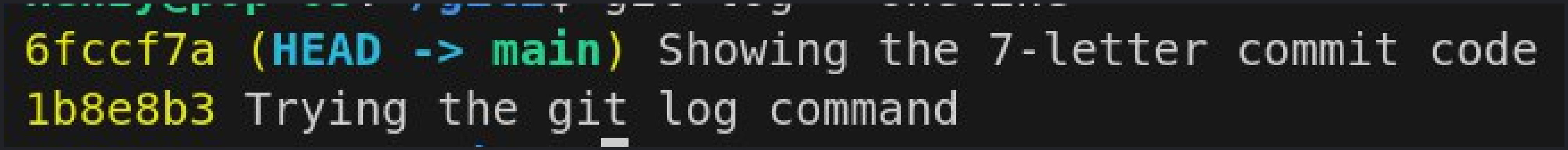
Viewing History

- Show commit logs/history including:
  - Commit code.
  - Author.
  - Date.
  - `HEAD` and `main` status.
- Commits will be represented in a reversed order (starting from the last commit).

- `git log`
  - `--oneline` : short representation (commit code and commit message).
  - `--graph` : graphical representation.

Moving Inside History

- `git checkout` : switch branches or restore working tree files
  - `7-letter commit code` : reference to the changes made in that commit.

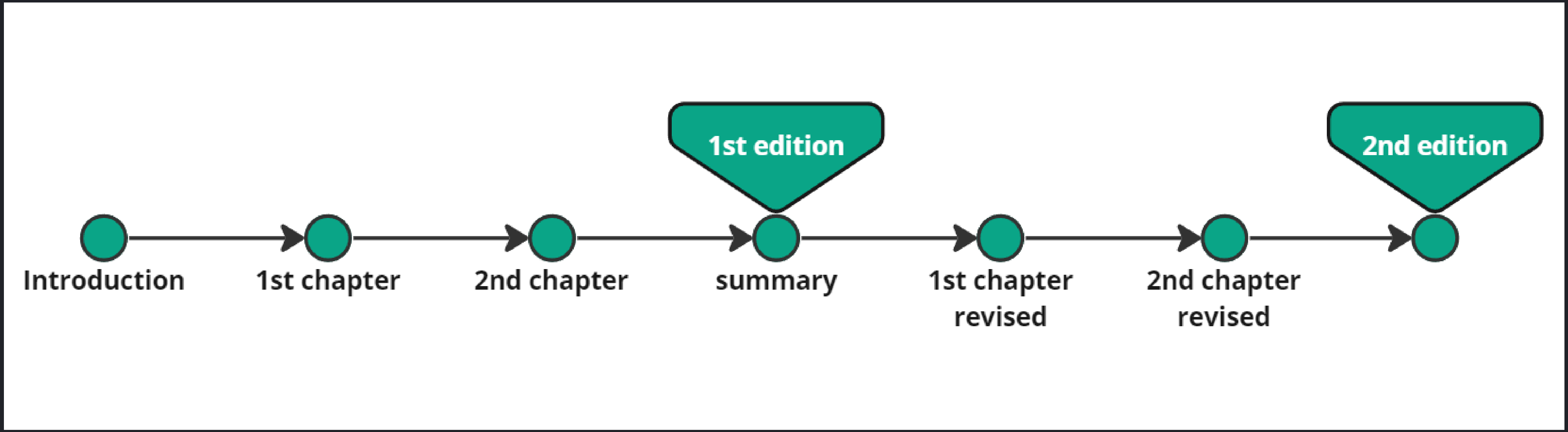


```
6fccf7a (HEAD -> main) Showing the 7-letter commit code
1b8e8b3 Trying the git log command
```

  - The `HEAD` Position will be changed to the specified commit code.

Tagging Commits

- Tag is a simple pointer to a commit, and they are useful for bookmarking important revisions like public releases.



- Instead of accessing commits using commit code you can create a tag to create a reference for your commit.
- Tagged commits created using the options [-a, -s, or -u] are called "**Annotated Tags**". They are meant for release.
  - Use the command `git tag -a [release-title]` to create annotated tags.

- **Lightweight Tag** is simply a name or a label for a commit.
- It will help you when you want to checkout this commit, as you will checkout the commit label instead of using the commit code.
  - To make a Lightweight tag use the command `git tag [tag-name] [commit code]` .

Useful Tag Commands

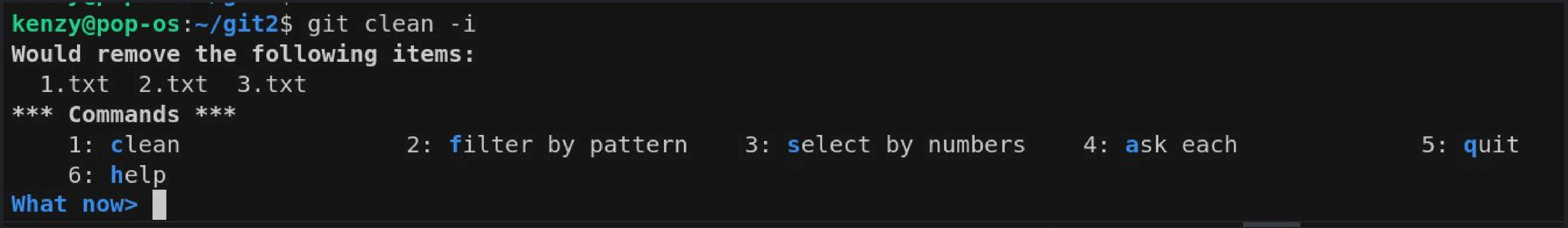
- `git tag` : list all tags you made.
- `git show [tag-name]` : list all information about the specified tag.
- `git tag [tag-name] -m [tag-message]` : leave a message with your annotated tag.

Undo Changes

- Working Directory
- Staging area
- Entire commit

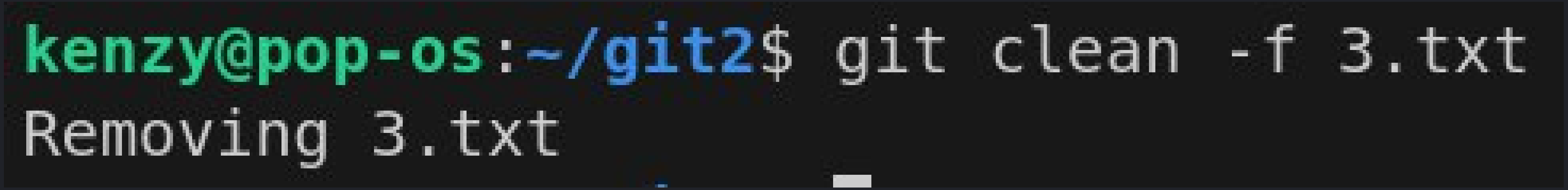
Undo Changes from Working Directory

- `git clean` : removes untracked files from the working directory. It has three main options:
  - `-i` : "Interactive".



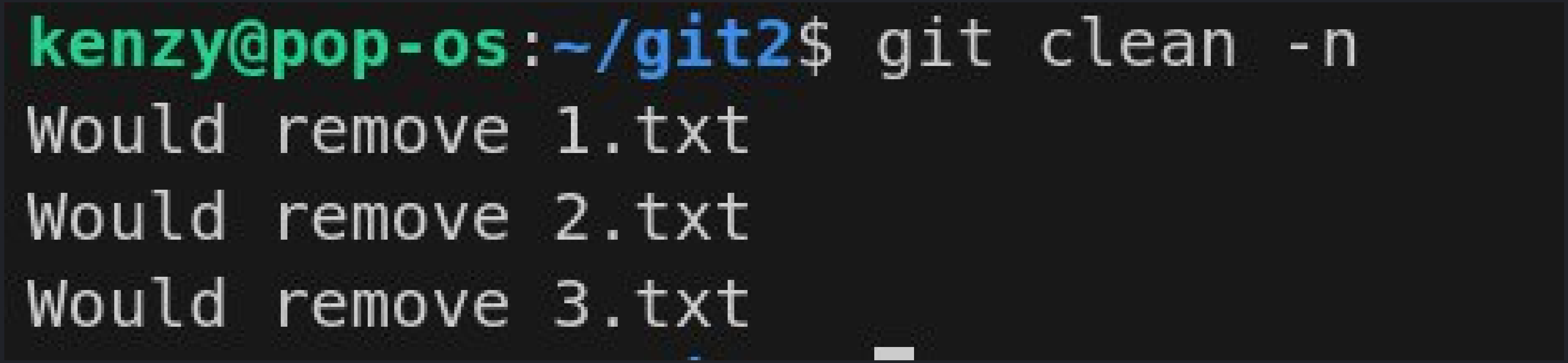
```
kenzy@pop-os:~/git2$ git clean -i
Would remove the following items:
  1.txt   2.txt   3.txt
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask each             5: quit
    6: help
What now> 
```

- `-f` : same as --force. You need to specify the file-name you want to be deleted as an argument or it will remove all untracked files.



```
kenzy@pop-os:~/git2$ git clean -f 3.txt
Removing 3.txt
```

- `-n` : same as --dry-run



```
kenzy@pop-os:~/git2$ git clean -n
Would remove 1.txt
Would remove 2.txt
Would remove 3.txt
```

- `git restore [file-name]` : removes all untracked changes from the working directory. It helps with the modified files.
    - Modified files in Git are those that have been changed since the last commit but the new changes haven't been staged or committed yet.

Undo Changes from Staging Area

- **You will remove changes from the staging area but you will not affect any thing in the working directory.**
- `git rm --cached [file-name]` **: useful when you add some changes in staging area but you haven't created any commits yet.**

- `git restore --staged [file-name]` : **useful when you have commits in your history but you have some changes in the staging area and you want to untrack them.**
    - **This command will use the last commit as a reference and will remove all uncommitted changes (Staging Area = History).**

- `git reset HEAD [file-name]` :
  - same as `git restore --staged [file-name]`
- ⚠️ Note:
  - You can remove changes from both the staging area and the working directory using the command `git restore --staged --worktree [file]`
    - (Working Directory = Staging Area = History)

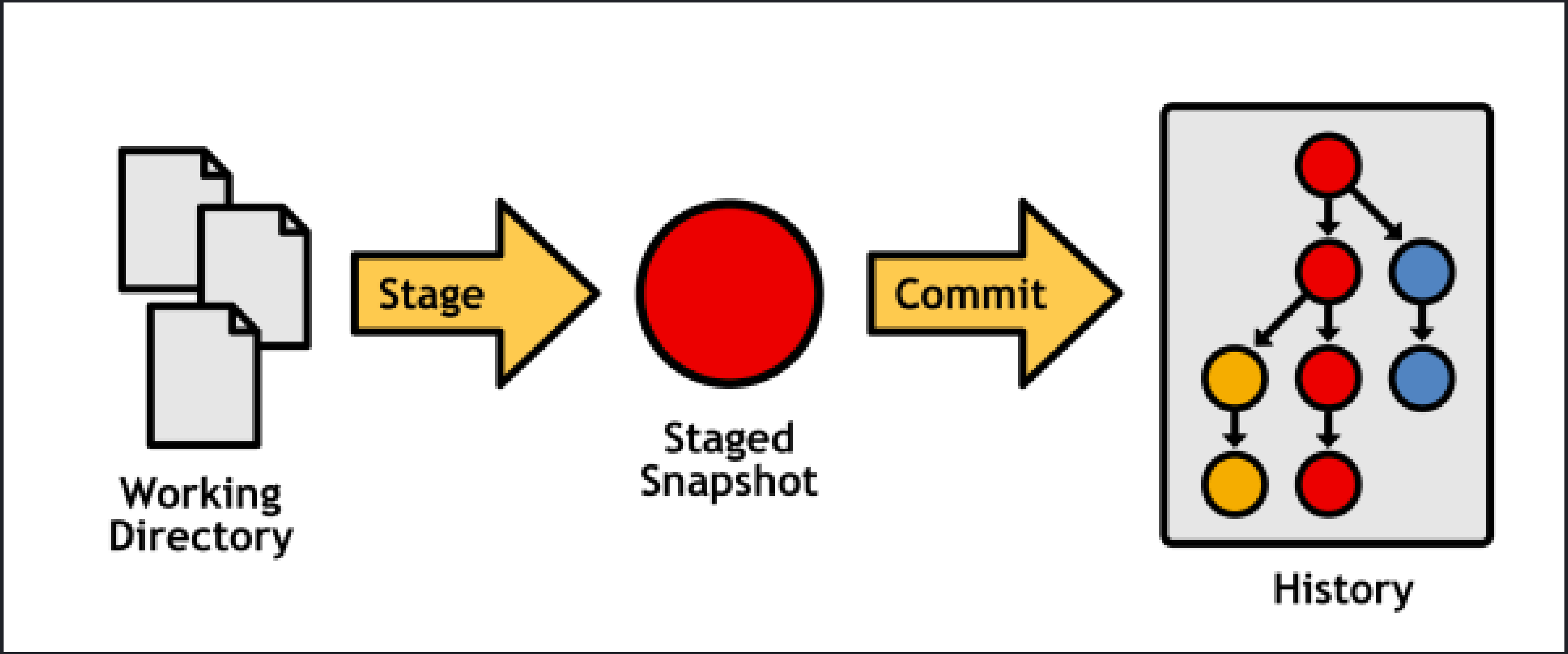
Undo an Entire Commit

Delete commit

- **We can't delete a commit exactly, but we can ignore it by resetting our timeline to a point where it wasn't there. Usually it will be the previous commit:**
  - `git reset --hard HEAD~1`
  - `git reset --hard [commit-code]`

- **When we use the `git reset --hard HEAD~1` HEAD and main go back to the previous commit.**
- **⚠️ Note:**
  - **If a commit is deleted then changes will be deleted from history, staging area, and working directory.**

Working with Branches



- **So far, we are still only able to create a linear project history, adding one commit on the top of another.**
- **Branches make it possible to develop multiple unrelated features in parallel by forking the project history.**

Branches

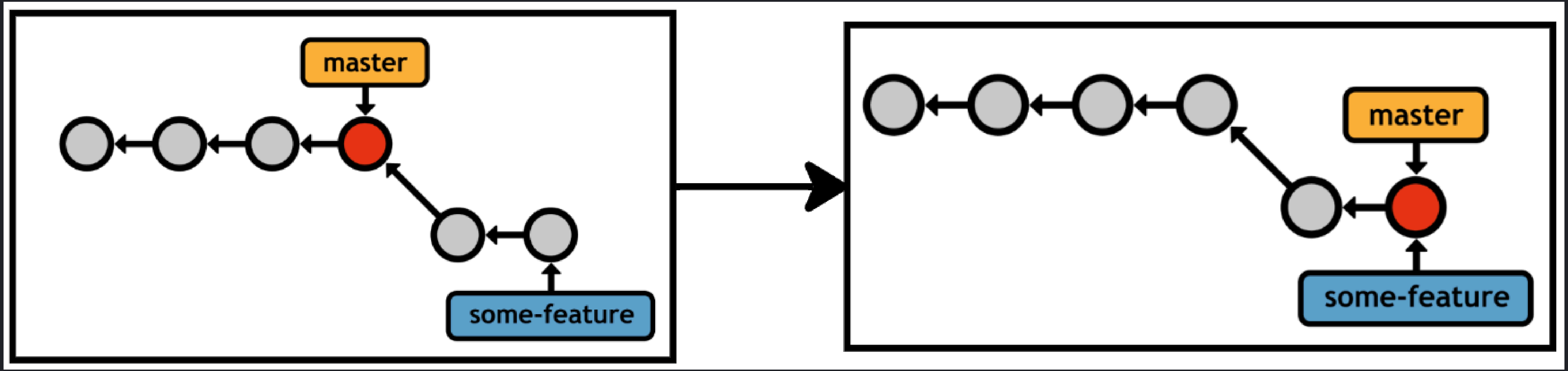
- **Create: you can use any of these commands to create a branch:**
  - `git branch [branch-name]` & `git checkout [branch-name]`
  - `git checkout -b [branch-name]` (create and move)
  - `git switch -c [branch-name]`

- **List:**
  - `git branch`
- **Rename:**
  - `git branch -m [new-name]`
- **Delete a branch that has no commits:**
  - `git branch -d [branch-name]`
- **Delete a branch that commits:**
  - `git branch -D [branch-name]`

Merging

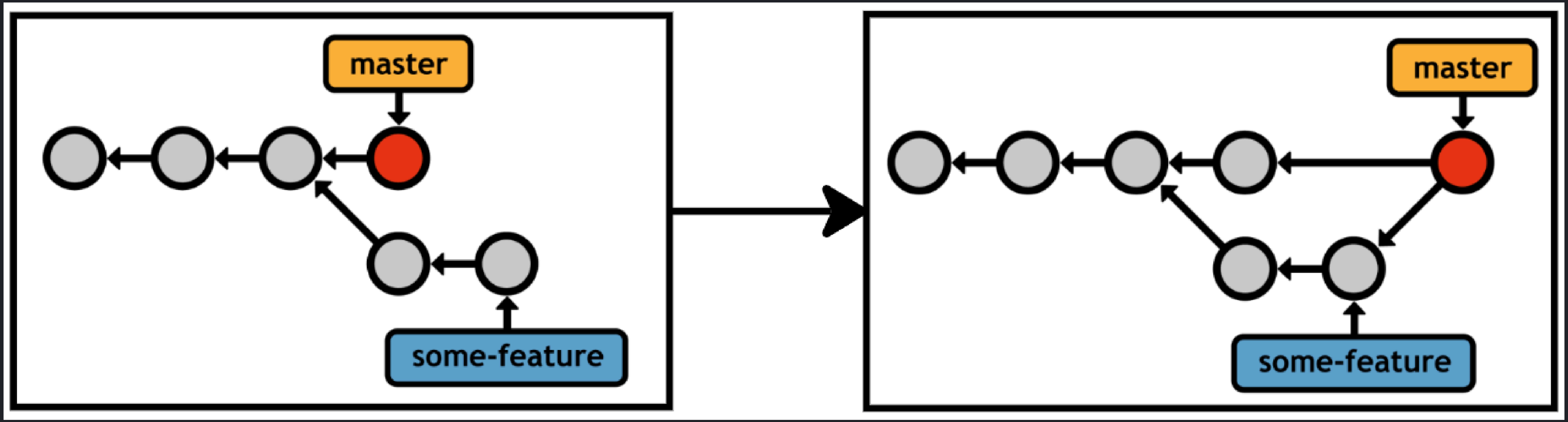
- Merging is the process of pulling commits from one branch into another.
- There are two main types of merges:
  - Fast Forward Merge.
  - Three-Way Merge.

Fast Forward Merge



- **We use the fast-forward merge process when there are no conflicting changes between the branches.**
- **You need to checkout main at first:**
  - `git checkout main`
- **Then you can merge the main branch to the feature:**
  - `git merge [feature-branch]`

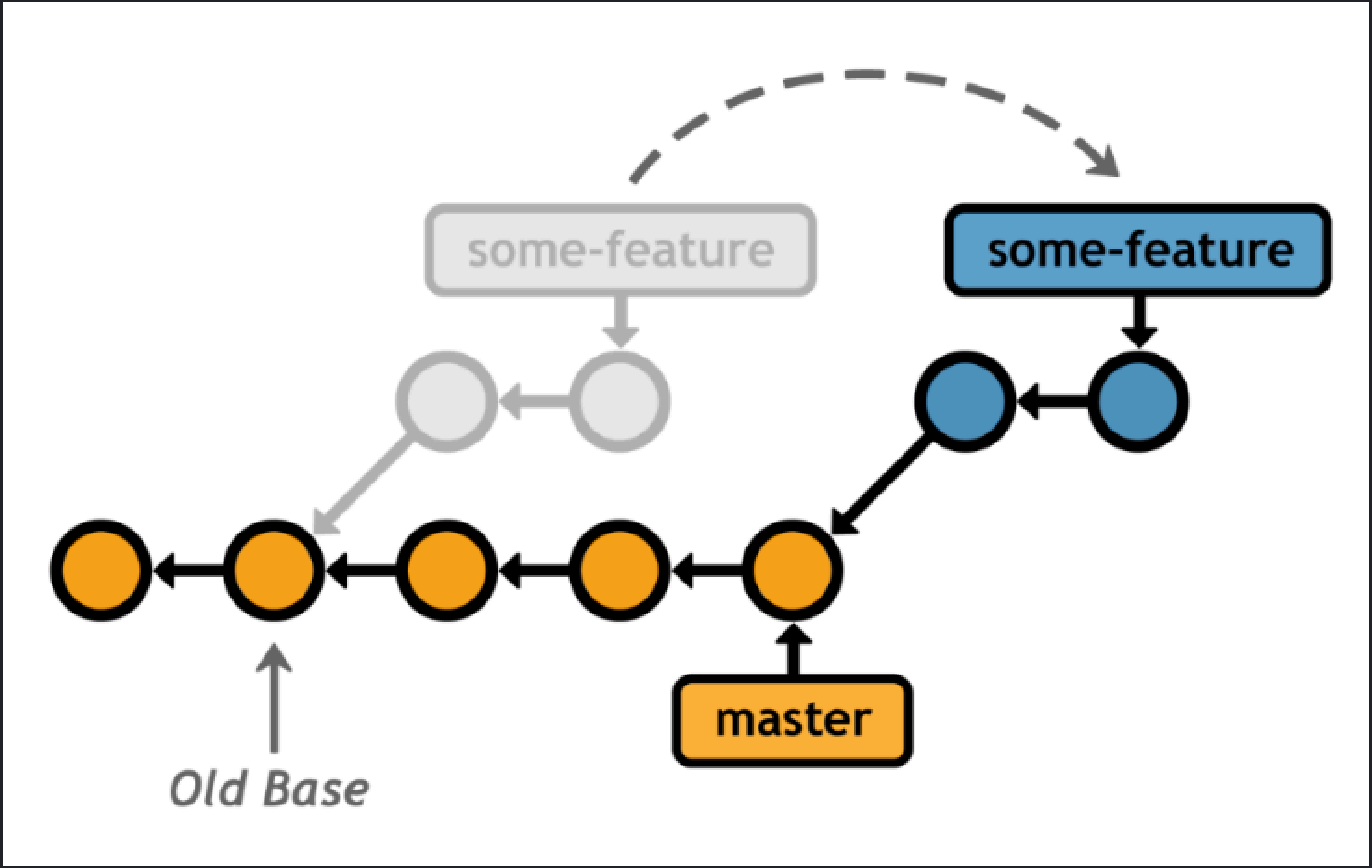
Three-Way Merge



- We use three-way merge to combine changes from two different branches when a fast-forward merge is not possible.
  - `git checkout main`
  - `git merge [feature-branch]`
- If there are conflicts, Git will stop the merge process and indicate the files with conflicts.

- **Identify conflict markers: Open the conflicted files and look for special markers like `<<<<<<<`, `=======`, and `>>>>>>>`.**
- **Resolve conflicts: Choose the desired changes from both branches or combine them as needed.**
- **Now stage and commit resolved files:**
  - `git add .`
  - `git commit -m "[Merging-message]"`

Rebasing



- **Rebasing: the process of moving a branch to a new base.**
- **You need to be on the branch that will be rebased:**
  - `git checkout [feature-branch]`
  - `git rebase main`**, then resolve any arising conflicts.**

- ⚠️ Note:
  - After rebasing the main will be left behind the rebased branch, so you need to do a fast-forward merge:
    - `git checkout main`
    - `git merge [feature-branch]` or `git rebase [feature-branch]`

`HEAD` Cases

- `HEAD` points to a branch ex: (`HEAD -> main`)
  - This is the normal scenario. In this case `HEAD` follows `main` in the project timeline.
  - When you want to checkout to a branch remember to specify the branch name and DO NOT checkout to the last commit in that branch.

- **HEAD points to a commit**
  - **This happens when you use the "git checkout [commit code]" command and it is called `detached HEAD` state.**
  - **It's called `detached HEAD` because the HEAD is now detached from any branch.**
  - **If you try to use the `git log` command, then any commit that is after the checked commit (`HEAD` position) will not be printed.**

- ⚠️ Note:
  - If you want to print all commits use the command `git log --all`.
  - Use the command "git branch [branch-name] [commit code]" to create a branch for the detached commits.

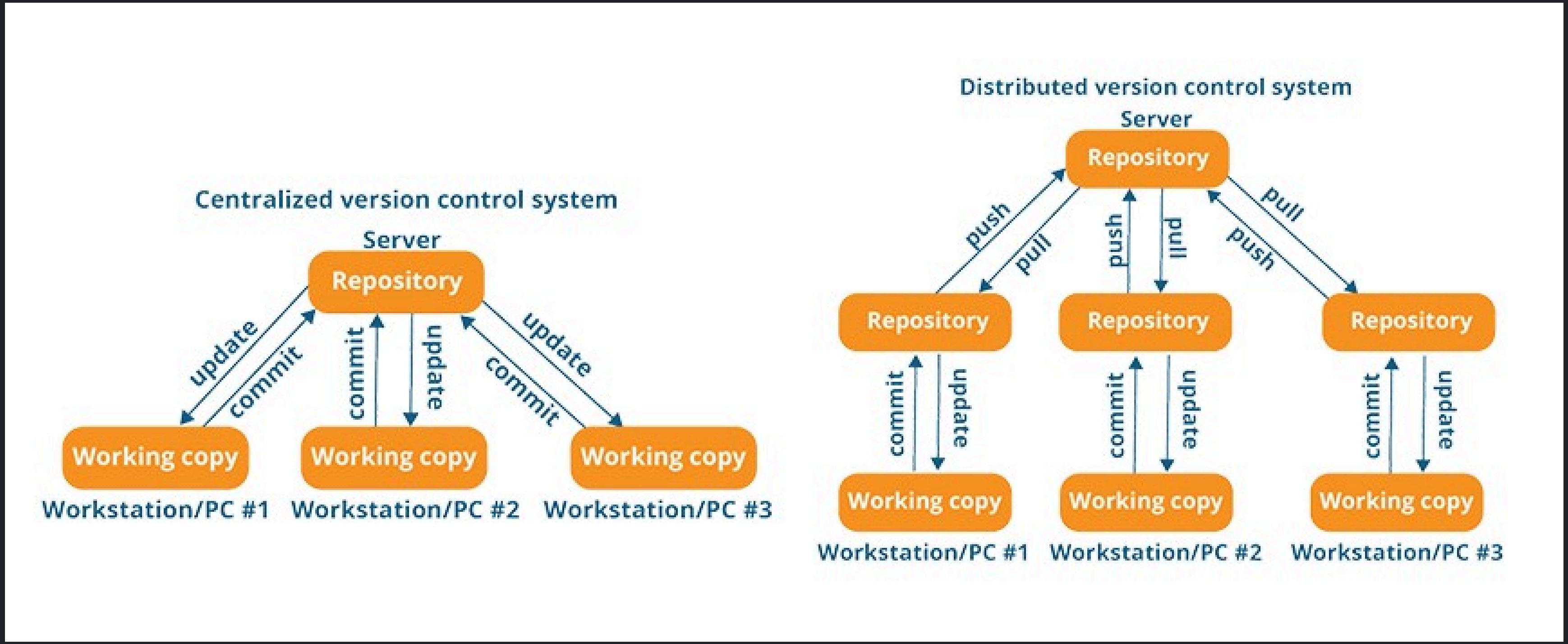
To sum up

- **Commit is a point in my timeline.**
- **HEAD is the present state (where I'm standing now in the project history).**
  - **It will point to a commit or a branch.**

How to work with Others Using Git & GitHub?

- GitHub is a platform that allows you to store and manage Git repositories online (remote repositories).
- While there are other platforms that can also host Git repositories, GitHub is the most widely used, and it's the one we will be using.

- **Remote repository: a version of your project hosted on a server or network.**



- In Centralized VCS there was only the remote repository where you commit your changes.
    - Any commit you make will require an internet connection, which may slow down the process.
    - If multiple people are committing their changes at the same time they will need to solve conflicts.

- In Distributed VCS every one in the team has his own copy of the project history.
- Commits are made locally (no need for internet access)till they are meaningful then you upload (push) them to the remote repository.
- You can download (pull) your team's changes whenever you want.
- Conflicts may happen when uploading or downloading changes but local commits will not lead to any conflicts.

New Remote Repository and Remote URL

1. **Create a repository on Github, give it a name, and a short description.**
2. **Specify whether it will be public or private.**
3. **Specify whether you want to create a README.md file or not.**

4. **Specify whether you want to create a .gitignore file or not.**
5. **Finally copy the URL and paste it in your terminal.**

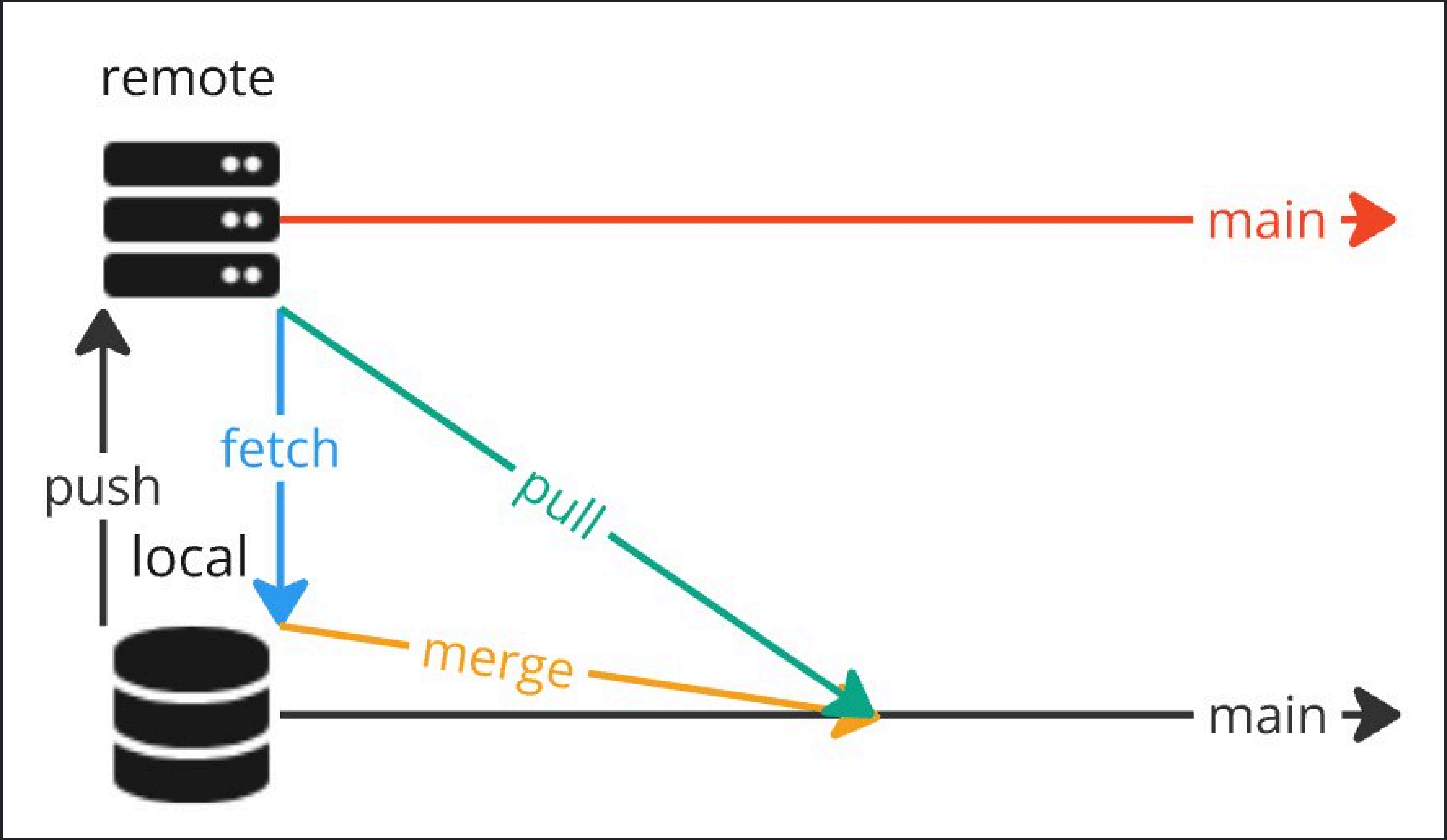
- ⚠️ **Note:**
  - **README.md: a markdown file that provides an introduction and explanation of the project.**

- **Add:**

    -
      ```
      git remote add [Remote-name] [Remote-URL]
      ```
- **Remove:**

    -
      ```
      git remote rm [Remote-name]
      ```
- **List remote repositories:**

    -
      ```
      git remote
      ```


remote
main
fetch
pull
push
local
merge
main

- **To link a remote repository with a local repository**
  - `git remote add origin [Remote-URl]`
  - `git branch -m main`
- **In the system's terminal use these commands:**
  - `cd [local-repo]`
  - `git push -u origin main`
- **This will ask you about your username on GitHub and your password.**

- ⚠️ **Note:**
  - **In this case it does not want the GitHub account password. It needs a Token.**
  - **To generate a token follow these steps:**
    - `Settings -> Developer settings -> Personal access tokens -> Tokens (classic)`.
  - **Copy the token and paste it the terminal.**
  - **To store your credentials use this command:**
    - `git config --global credentials.helper store`

- GitHub Issues: Your Project's To-Do List.
- You can give your issue a title, a detailed description, a label (bug, feature, enhancement, etc) and you can assign someone of your team to close it.
- Each issue has a number that you can use in your commit message to close it.
    - `closes #[issue-number]`

- You can link the issue to a Milestone.
- Milestone is a grouping of issues and pull requests that share a common goal or deadline.
- After finishing a milestone you can make a release using the annotated tags, then push your tag.
  - `git push origin [tag-name]`

- If you are not a contributor in a specific project, you can fork the project repository.
- Fork: creates a complete copy of the original project under your own account.
- You can make modifications in your fork and then create a pull request to contribute back to the original repository

- Pull Request: a proposal to merge changes from one branch into another.
- If you are assigned as a contributor in a project repository you can have a copy (a clone) of the remote repository on your disk:
    - `git clone [repository-URL]`

Thank you 😄