Project Management Using Git

Prepared by -

- Kartik Joshi (12111006)
- Utpal Tiwari (12111017)
- Aryan Gupta (12111030)
- Siddh Manek (12111114)
- Vrishav Garg (12111051)

Introduction to Git and GitHub

Git

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It outclasses SCM tools like Subversion, CVS Perforce, and ClearCase with features like cheap local branching, convenient staging area, and multiple workflows.

GitHub

GitHub is a Platform and Cloud-Based service for Software Development and version control using git, allowing developers to store and manage their code. It provides the distributed version control of git, access control, bug tracking, software features requests, task management, continuous integration (CI), and wikis for every project. Headquartered in California, it has been a subsidiary of Microsoft since 2018.

Why use Git ?

1. **Version Control**: Git allows you to track changes to your codebase over time. This means you can easily revert to previous versions if something goes wrong, compare changes between versions, and understand the evolution of your code.

2. **Collaboration**: It allows for seamless collaboration by managing and merging changes made by different team members, avoiding conflicts and ensuring that everyone is working on the latest version.

3. **Branching and Merging**: This allows developers to work on different features or fixes without affecting the main codebase. Later, these branches can be merged back together, keeping the changes organized and controlled.

4. **Backup and Recovery**: With Git, your code is stored in a distributed manner across multiple locations. This helps safeguard your code against data loss, as each developer has a full copy of the repository and its history.

5. **Code Review**: Git integrates well with code review processes. Developers can create pull requests (or merge requests) to propose changes and have them reviewed by colleagues before they are merged into the main codebase.

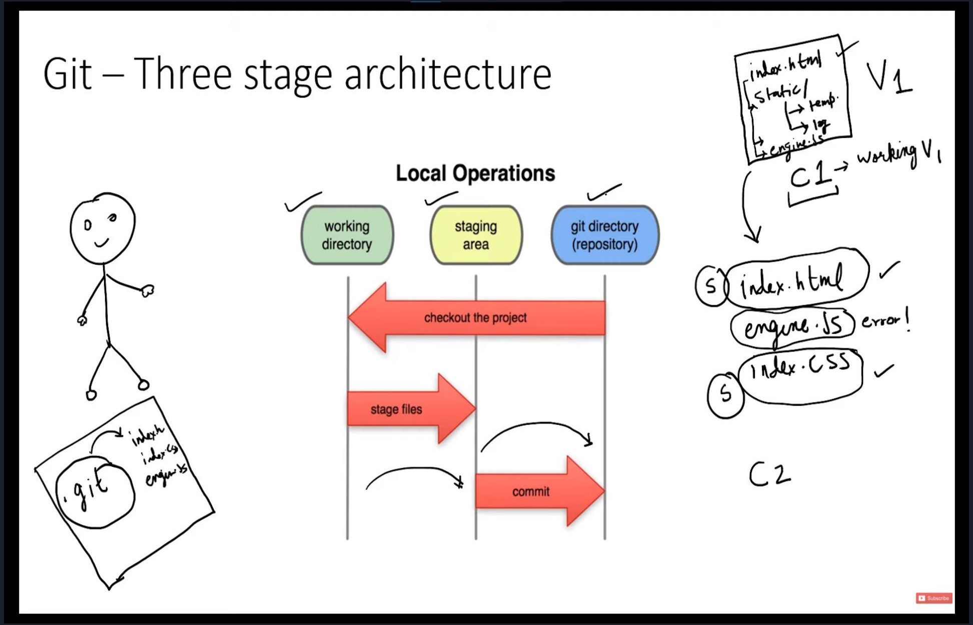
6. **Open Source Community**: Many open-source projects use Git, making it easier to contribute to and collaborate on projects with a global community.

7. **History and Accountability**: Git maintains a detailed history of every change made to the codebase, including who made the changes and when. This can be valuable for tracking down the source of bugs, analyzing past decisions, and providing accountability.

8. **Ecosystem and Tools**: Git has a wide range of tools, integrations, and services that make it easier to work with, such as GitHub, GitLab, and Bitbucket. These platforms provide additional features like issue tracking, continuous integration, and more.

9. **Learning and Industry Standard**: Git is one of the most popular version control systems and has become an industry standard in software development.

Git Three Stage Architecture



Working Directory:
The working directory is where you make modifications to your project files. It's simply the folder on your local machine where you edit, add, or delete files. When you create or modify a file, it exists in the working directory.

Staging Area (Index):

The staging area, also known as the index, acts as a buffer between the working directory and the repository. It is a conceptual space that holds the changes you want to include in the next commit. When you modify files in the working directory, you need to explicitly add them to the staging area before they become part of the next commit. This allows you to control which changes will be included in the commit.

Repository (Commit History):

The repository contains the complete history of your project, including all the commits you've made. Each commit represents a snapshot of the project's state at a specific point in time. It contains the changes from the staging area along with a commit message describing the changes made. Commits are permanent and immutable, ensuring that the project's history is well-documented and can be revisited at any time.

A typical workflow in Git follows these three stages:

Modify Files

You start by making changes to the files in the working directory. This could involve creating new files, modifying existing ones, or deleting files.

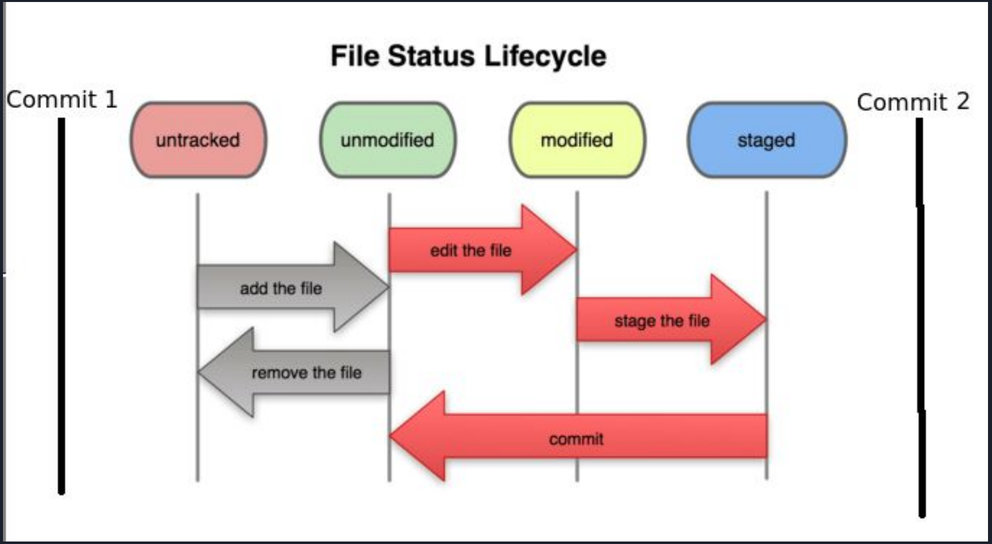
Stage Changes

Once you're satisfied with the changes you've made and want to save them as a new version, you add those changes to the staging area using the **git ‘add’ command.** This allows you to selectively choose which changes to include in the next commit.

Commit Changes

After staging the changes, you create a commit with a message describing the changes you made. The **git commit command** is used for this purpose. The changes in the staging area become part of the commit, and the commit is added to the repository, becoming a permanent part of the project's history.

File Status Life-Cycle



Untracked:

A file is in the "Untracked" state when it is newly created in the working directory or if Git has not been instructed to track changes to that file. These files are not yet part of Git's version control system, and Git is unaware of any changes made to them.

Unmodified:

All the changes done after it has been in the tracked are checked.

If no changes done then file remains in unmodified stage.

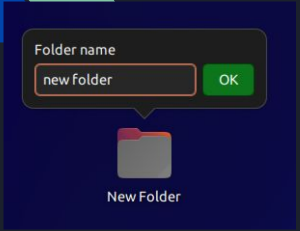
Modified:

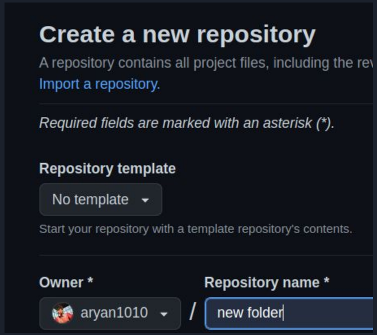
After you have created or modified a tracked file in the working directory, Git recognizes it as "Modified." This means that the file's content has changed since the last commit.

Staged (Changes to be committed):

To include the changes made to a modified file in the next commit, you need to add them to the staging area. The staging area is a space where you assemble changes that you want to be included in the next commit.

Creating and Cloning the Repository




Create a new repository

A repository contains all project files, including the re

Import a repository.

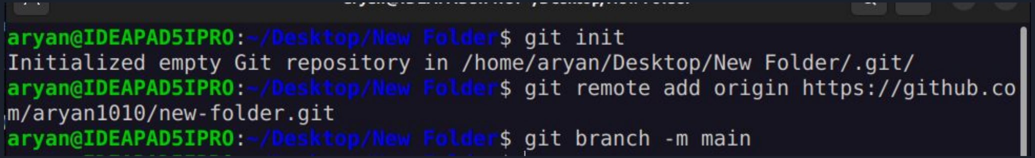
Required fields are marked with an asterisk (*).

Repository template

No template

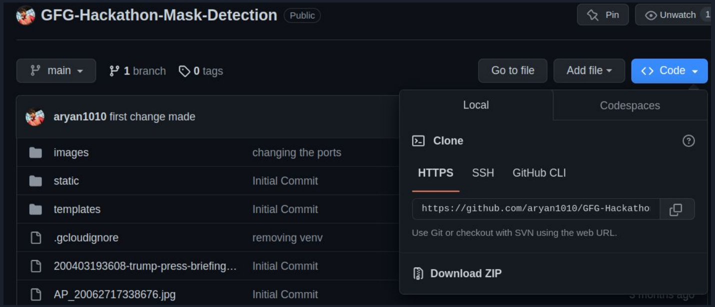
Start your repository with a template repository's contents.

Owner * aryan1010 / Repository name * new folder


```
aryan@IDEAPAD5IPRO:~/Desktop/New Folder$ git init
Initialized empty Git repository in /home/aryan/Desktop/New Folder/.git/
aryan@IDEAPAD5IPRO:~/Desktop/New Folder$ git remote add origin https://github.co
m/aryan1010/new-folder.git
aryan@IDEAPAD5IPRO:~/Desktop/New Folder$ git branch -m main
```

Create new folder repository on github and on PC

Git init-Initialize git in that folder
Git remote -Connecting the git folder to github repo
Git Branch - Creating branch main to work on


GFG-Hackathon-Mask-Detection Public

main · 1 branch · 0 tags

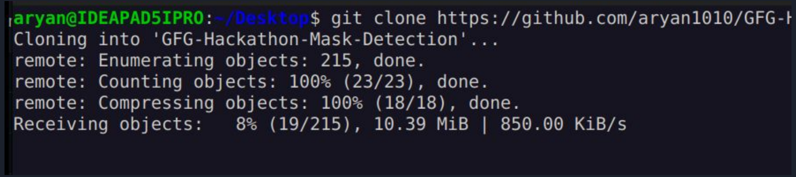
aryan1010 first change made

images	changing the ports
static	Initial Commit
templates	Initial Commit
.gcloudignore	removing venv
200403193608-trump-press-briefing...	Initial Commit
AP_20062717338676.jpg	Initial Commit

Clone — HTTPS

https://github.com/aryan1010/GFG-Hackatho

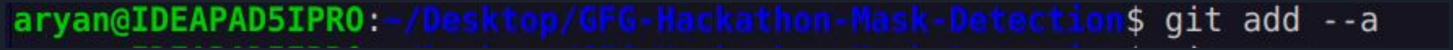
Use Git or checkout with SVN using the web URL.


```
aryan@IDEAPAD5IPRO:~/Desktop$ git clone https://github.com/aryan1010/GFG-H
Cloning into 'GFG-Hackathon-Mask-Detection'...
remote: Enumerating objects: 215, done.
remote: Counting objects: 100% (23/23), done.
remote: Compressing objects: 100% (18/18), done.
Receiving objects:   8% (19/215), 10.39 MiB | 850.00 KiB/s
```

Git clone-Download the repository

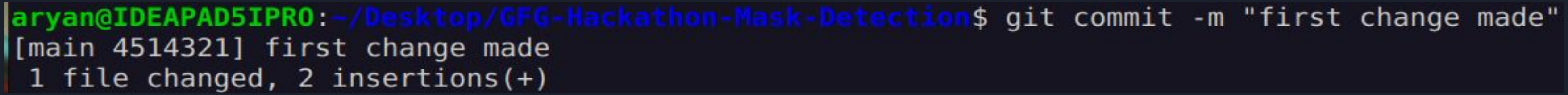
Find the repo and copy the link

Working with Git Commands



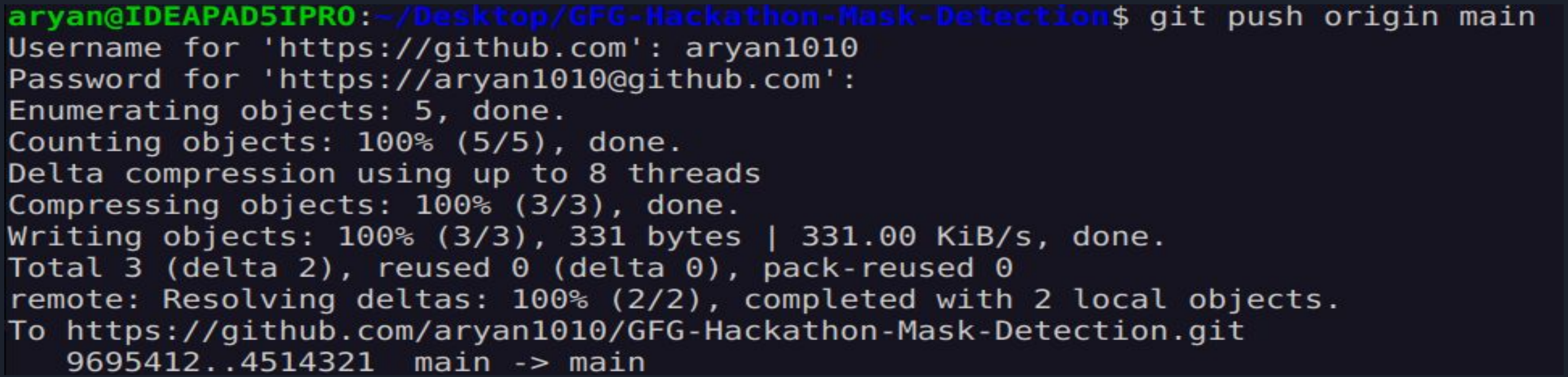
```
aryan@IDEAPAD5IPRO:~/Desktop/GFG-Hackathon-Mask-Detection$ git add --a
```

Git add - Used staging the changes



```
aryan@IDEAPAD5IPRO:~/Desktop/GFG-Hackathon-Mask-Detection$ git commit -m "first change made"
[main 4514321] first change made
 1 file changed, 2 insertions(+)
```

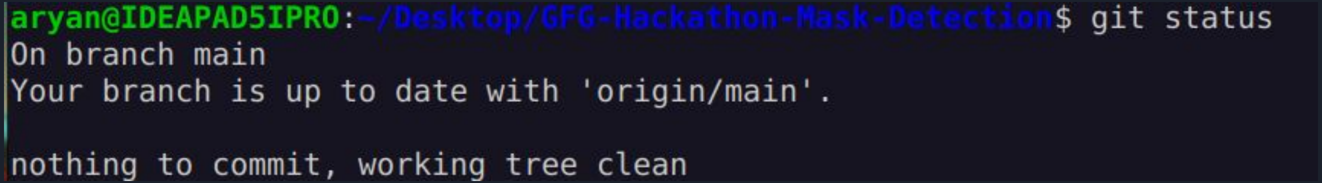
Git commit - Used to commit the changes



```
aryan@IDEAPAD5IPRO:~/Desktop/GFG-Hackathon-Mask-Detection$ git push origin main
Username for 'https://github.com': aryan1010
Password for 'https://aryan1010@github.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 331 bytes | 331.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/aryan1010/GFG-Hackathon-Mask-Detection.git
   9695412..4514321  main -> main
```

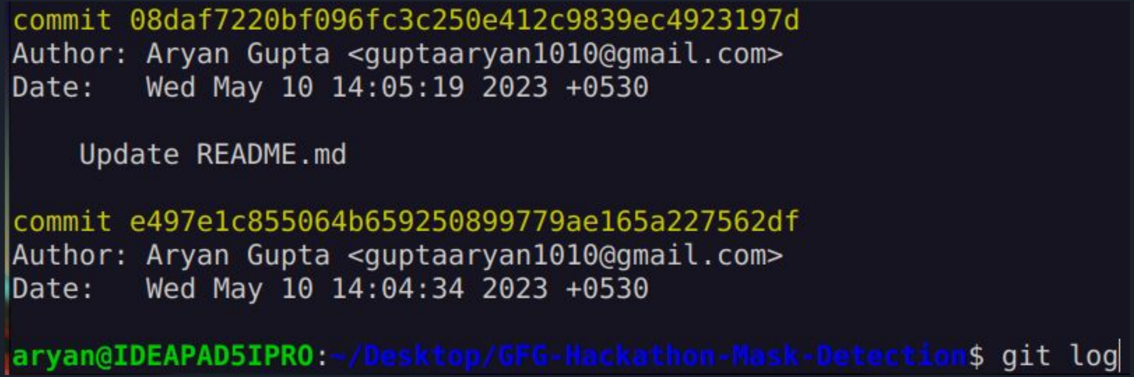
Git push- Used to push the changes in repository

Working with Git Commands



```
aryan@IDEAPAD5IPRO:~/Desktop/GFG-Hackathon-Mask-Detection$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

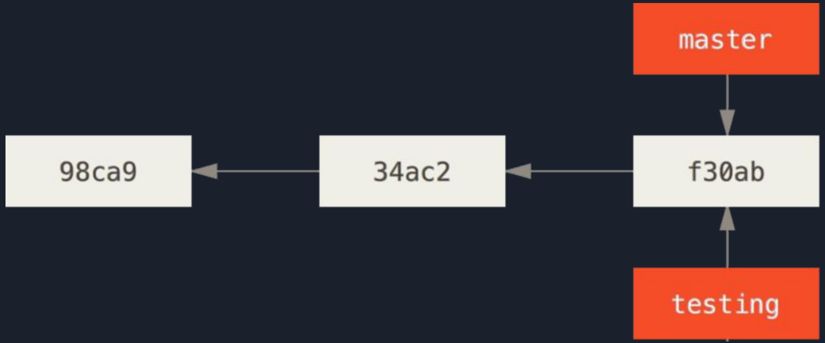
Git status - displays the state of the working directory and the staging area



```
commit 08daf7220bf096fc3c250e412c9839ec4923197d
Author: Aryan Gupta <guptaaryan1010@gmail.com>
Date:   Wed May 10 14:05:19 2023 +0530

    Update README.md

commit e497e1c855064b659250899779ae165a227562df
Author: Aryan Gupta <guptaaryan1010@gmail.com>
Date:   Wed May 10 14:04:34 2023 +0530

aryan@IDEAPAD5IPRO:~/Desktop/GFG-Hackathon-Mask-Detection$ git log
```

Git log - Used to show the history of commits

Git Branching



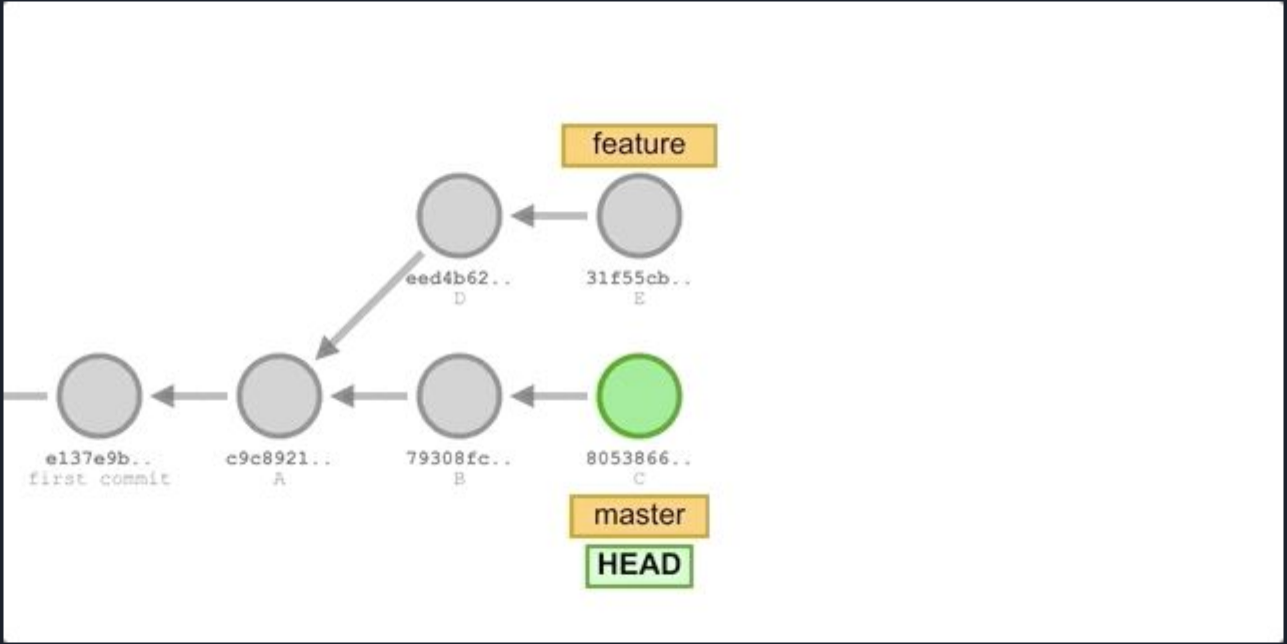
Branching means you diverge from the main line of development and continue to do work without messing with that main line.

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.

When we create a new branch, a new pointer for you to move around. Let's say you want to create a new branch called *testing*. This creates a new pointer to the same commit (i.e. commit with id **f30ab**) you're currently on.

Git knows on which branch we are on by maintaining a pointer named **HEAD**. Whenever we switch from one branch to another, the HEAD pointer moves and points to the new branch.

Pointers Related to Branches in Git

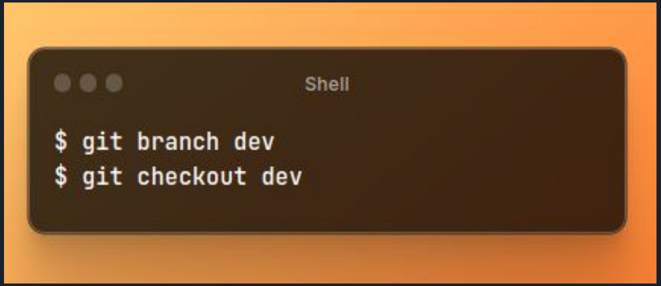


Source: Medium

Creating and switching Branches in Git

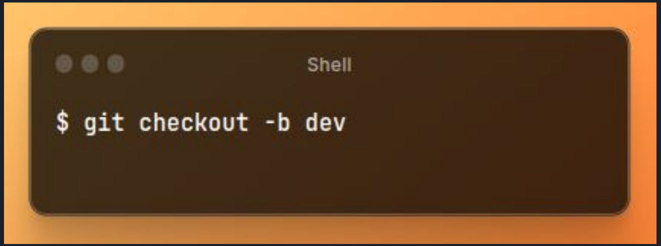
Git has two ways to perform the creating and switching branches:

1. **Using git branch and git checkout -**
   The git branch command creates a new branch but doesn't switch to the new branch. The git checkout command switches to the new branch.



```
$ git branch dev
$ git checkout dev
```

2. **Using git checkout -**
   The git checkout command with -b option creates a new branch along with switching to that.



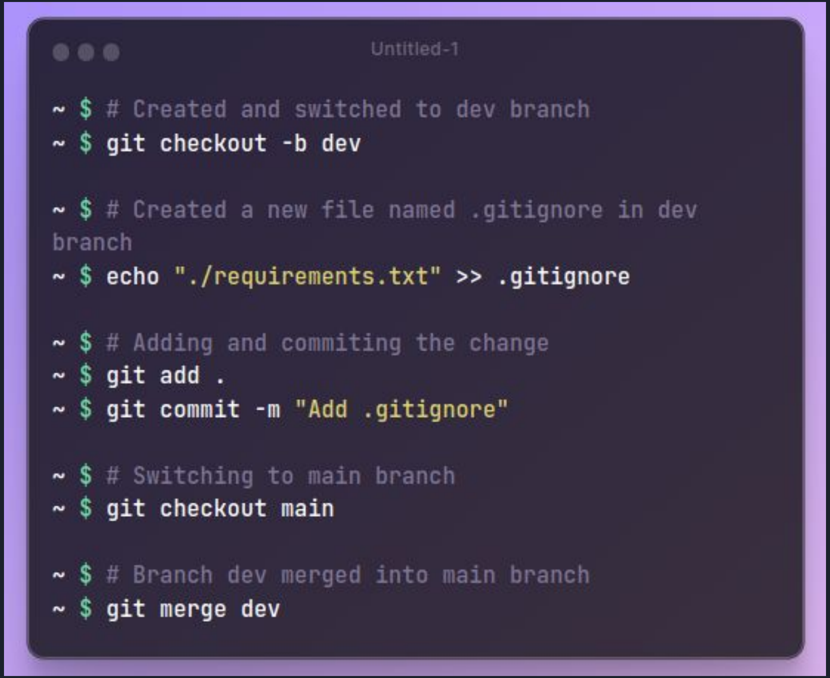
```
$ git checkout -b dev
```

Merging Branches in Git

The **git merge** command lets you take the independent lines of development created by **git branch** and integrate them into a single **branch**.

There are four major types of merging:

1. Standard Merging
2. Fast-forward Merging
3. Squash and Merge
4. Rebase and merge

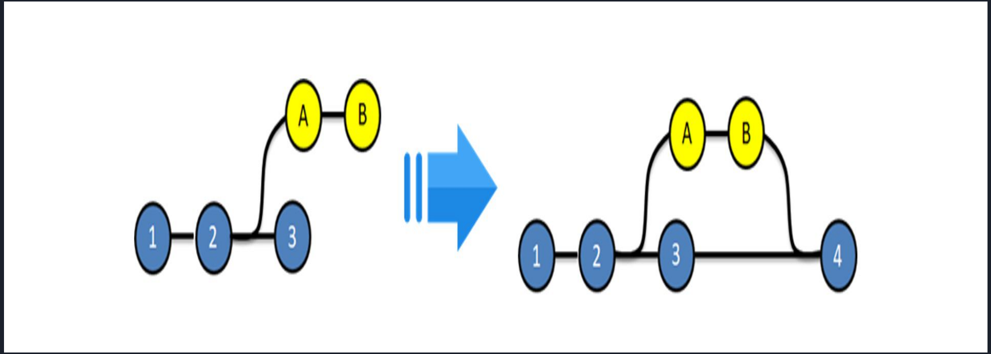

Untitled-1

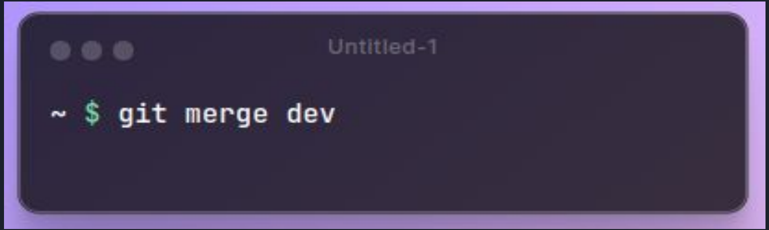
```
~ $ # Created and switched to dev branch
~ $ git checkout -b dev

~ $ # Created a new file named .gitignore in dev
branch
~ $ echo "./requirements.txt" >> .gitignore

~ $ # Adding and commiting the change
~ $ git add .
~ $ git commit -m "Add .gitignore"

~ $ # Switching to main branch
~ $ git checkout main

~ $ # Branch dev merged into main branch
~ $ git merge dev
```

Standard Merging

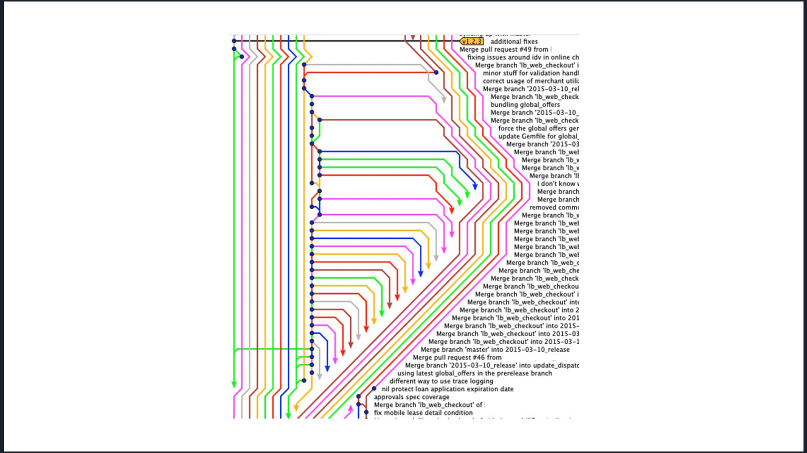
Git merge is one of the merging techniques in git, in which the logs of commits on branches are intact.

The logs are detailed but very clumsy and not so developer-friendly.





```
~ $ git merge dev
```

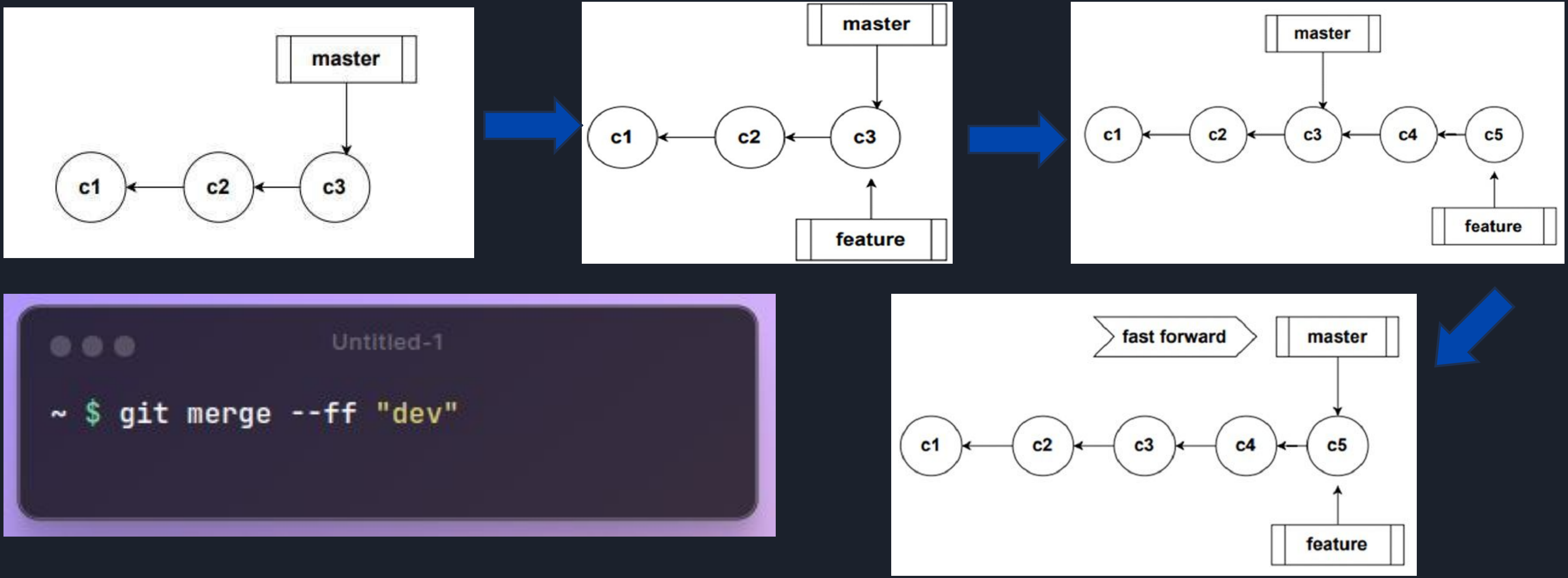


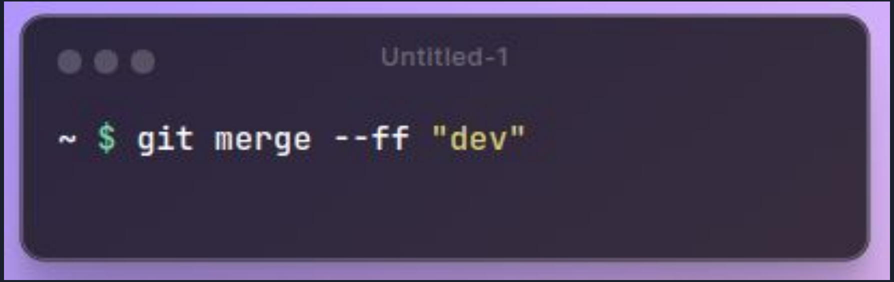
```
v1.2.3  additional fixes
Merge pull request #49 from
fixing issues around idv in online ch
Merge branch 'lb_web_checkout' i
minor stuff for validation handl
correct usage of merchant utili
Merge branch '2015-03-10_rel
Merge branch 'lb_web_check
bundling global_offers
Merge branch '2015-03-10_
Merge branch 'lb_web_check
force the global offers ger
update Gemfile for global_
Merge branch '2015-03
Merge branch 'lb_wel
Merge branch 'lb_v
Merge branch 'lb_v
Merge branch 'lt
I don't know v
Merge branch
Merge branch
removed comm
Merge branch 'lb_v
Merge branch 'lb_wel
Merge branch 'lb_wel
Merge branch 'lb_wel
Merge branch 'lb_wel
Merge branch 'lb_wel
Merge branch 'lb_web_c
Merge branch 'lb_web_che
Merge branch 'lb_web_check
Merge branch 'lb_web_checkou
Merge branch 'lb_web_checkout' i
Merge branch 'lb_web_checkout' into
Merge branch 'lb_web_checkout' into 2
Merge branch 'lb_web_checkout' into 201
Merge branch 'lb_web_checkout' into 2015-
Merge branch 'lb_web_checkout' into 2015-03
Merge branch 'lb_web_checkout' into 2015-03-1
Merge branch 'master' into 2015-03-10_release
Merge pull request #46 from
Merge branch '2015-03-10_release' into update_dispatc
using latest global_offers in the prerelease branch
different way to use trace logging
nil protect loan application expiration date
approvals spec coverage
Merge branch 'lb_web_checkout' of
fix mobile lease detail condition
```

Fast Forwarding

Fast forward merge can be performed when there is a direct linear path from the source branch to the target branch. In fast-forward merge, git simply moves the source branch pointer to the target branch pointer without creating an extra merge commit.

if two branches have not diverged and there is a direct linear path from the target branch to the source branch, Git runs a fast forward merge.

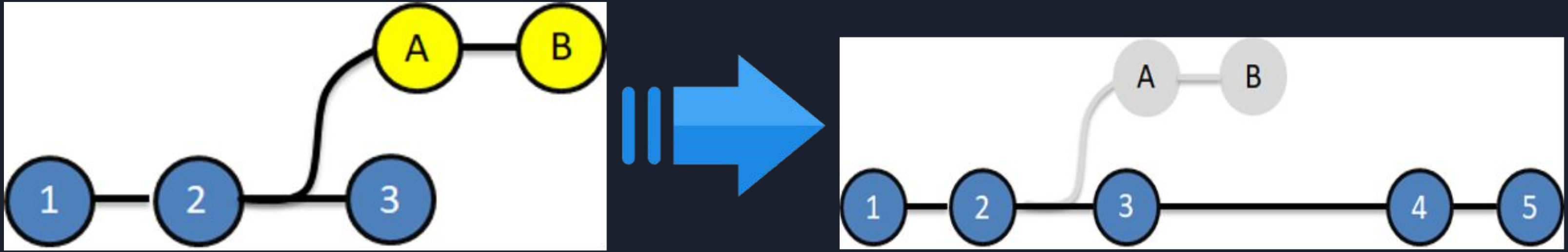


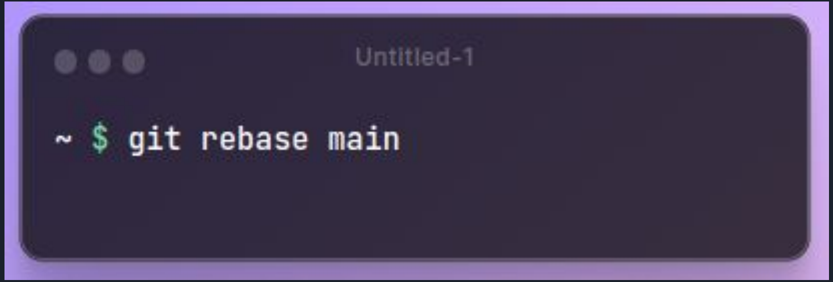


```
~ $ git merge --ff "dev"
```

Rebasing

Git Rebase is similar to git merge, but the logs are modified after merge in this technique. Git rebase was introduced to overcome the limitation of merging, i.e., to make logs of repository history look linear.

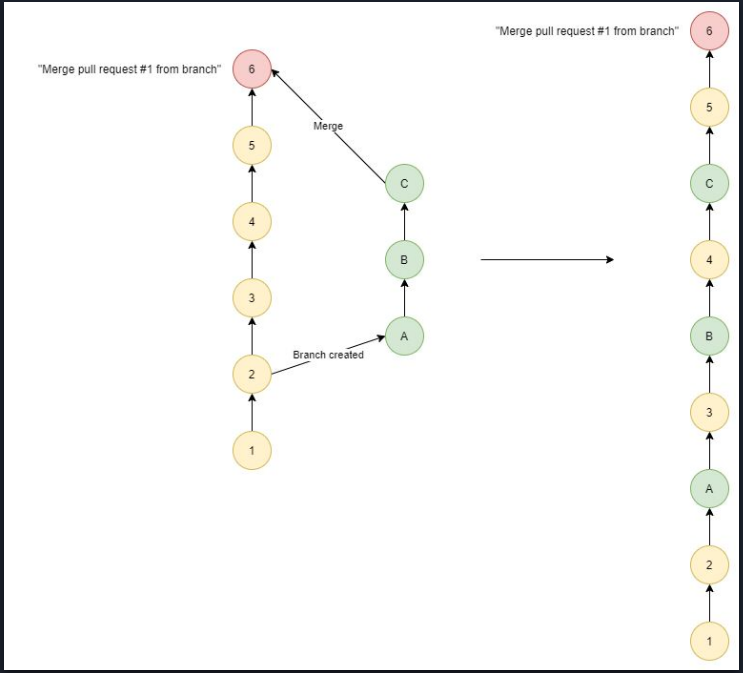


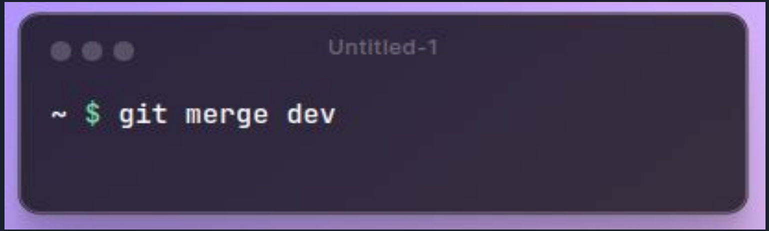


```
~ $ git rebase main
```

Standard Merging

A standard merge will take each commit in the branch being merged and add them to the history of the base branch based on the timestamp of when they were created.

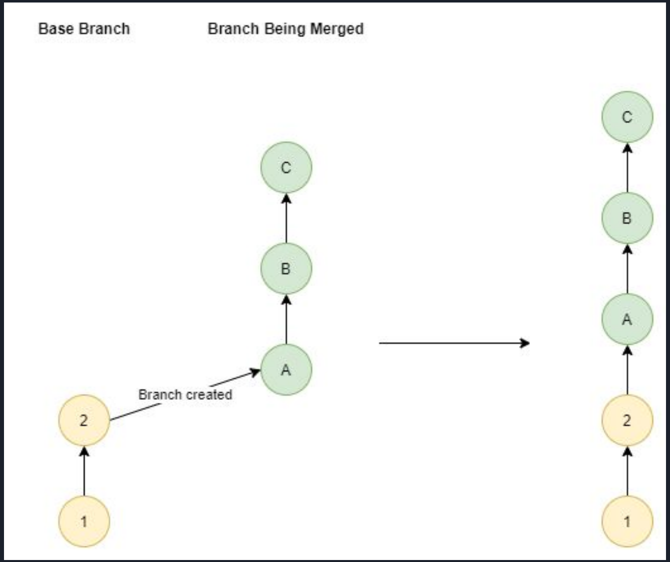

"Merge pull request #1 from branch"
6
Merge
5
C
4
B
3
A
Branch created
2
1
"Merge pull request #1 from branch"
6
5
C
4
B
3
A
2
1

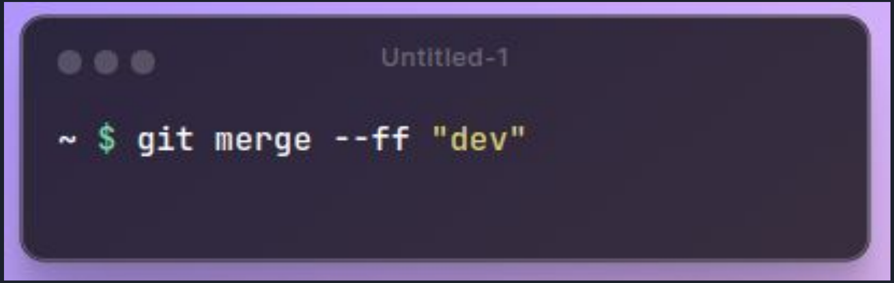


```
~ $ git merge dev
```

Fast Forward Merging

This is as if you made the commits directly on the base branch. The idea is because no changes were made to the base branch there's no need to capture a branch had occurred.A

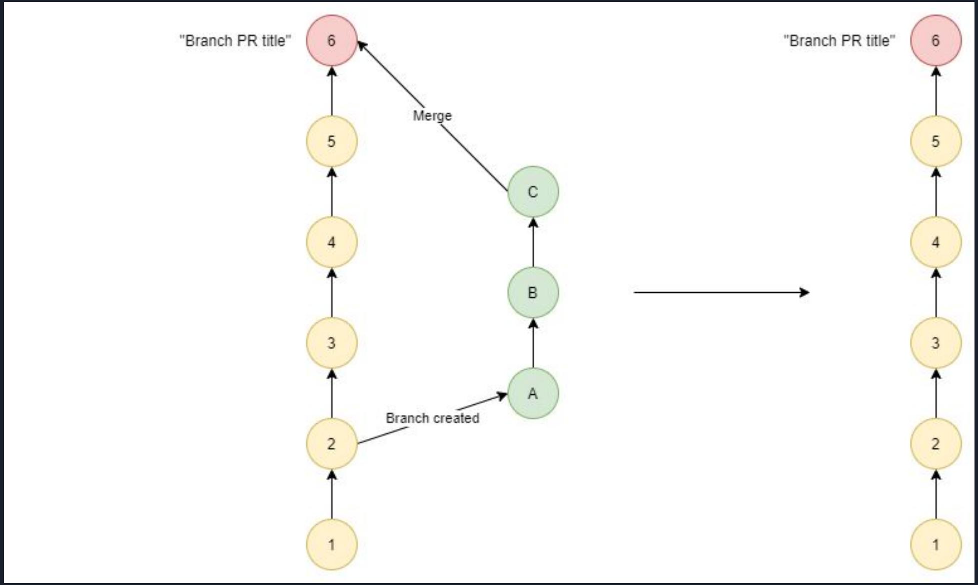




```
~ $ git merge --ff "dev"
```

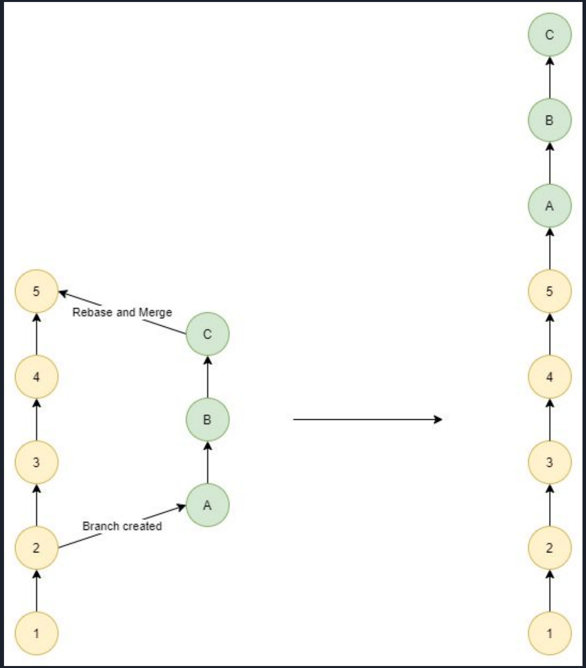
Squash and Merge

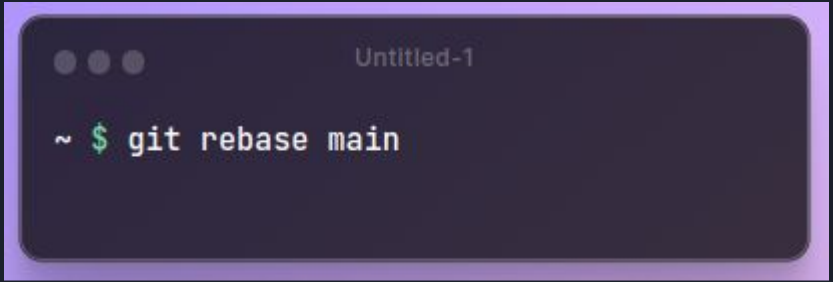
Squash takes all the commits in the branch (A,B,C) and melds them into 1 commit. That commit is then added to the history, but none of the commits that made up the branch are preserved



Rebasing

A rebase and merge will take where the branch was created and move that point to the last commit into the base branch, then reapply the commits on top of those changes.





```
~ $ git rebase main
```

Merge Conflicts in Git

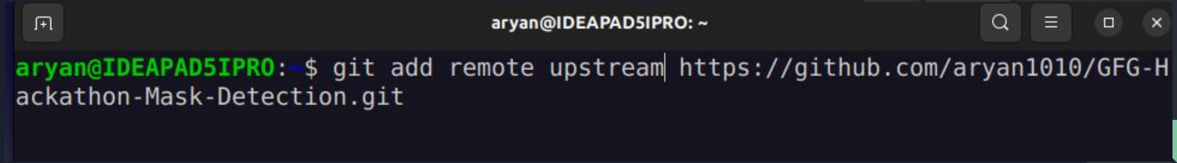
Conflicts generally arise when two people have changed the same lines in a file, or if one developer deleted a file while another developer was modifying it. In these cases, Git cannot automatically determine what is correct. Conflicts only affect the developer conducting the merge, the rest of the team is unaware of the conflict. Git will mark the file as being conflicted and halt the merging process. It is then the developers' responsibility to resolve the conflict.

Conflicts can be resolved using the code editors by the developer who is performing the merge by consulting with the collaborating team that which part should be kept and which one should be removed.

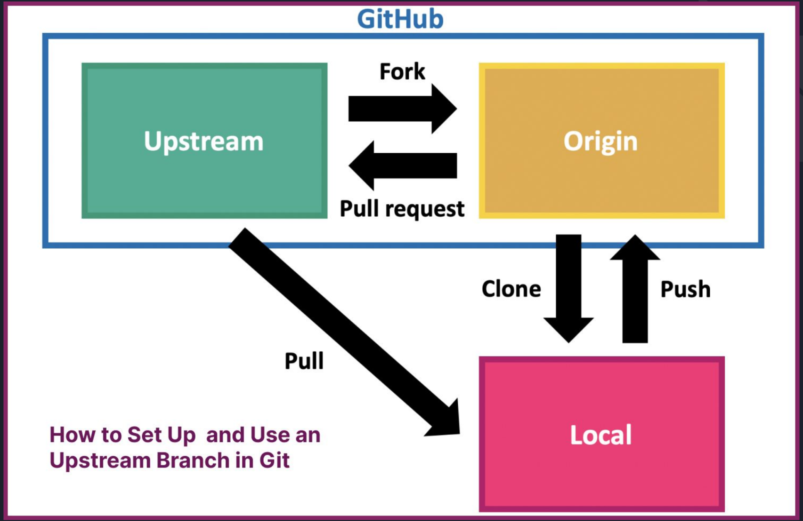
Use meld for merging conflicts

Working with Fork(Git Pull and Fetch)

Create a new fork & clone the repository as stated previously



```
aryan@IDEAPAD5IPRO:~$ git add remote upstream https://github.com/aryan1010/GFG-H
ackathon-Mask-Detection.git
```



What’s Next?