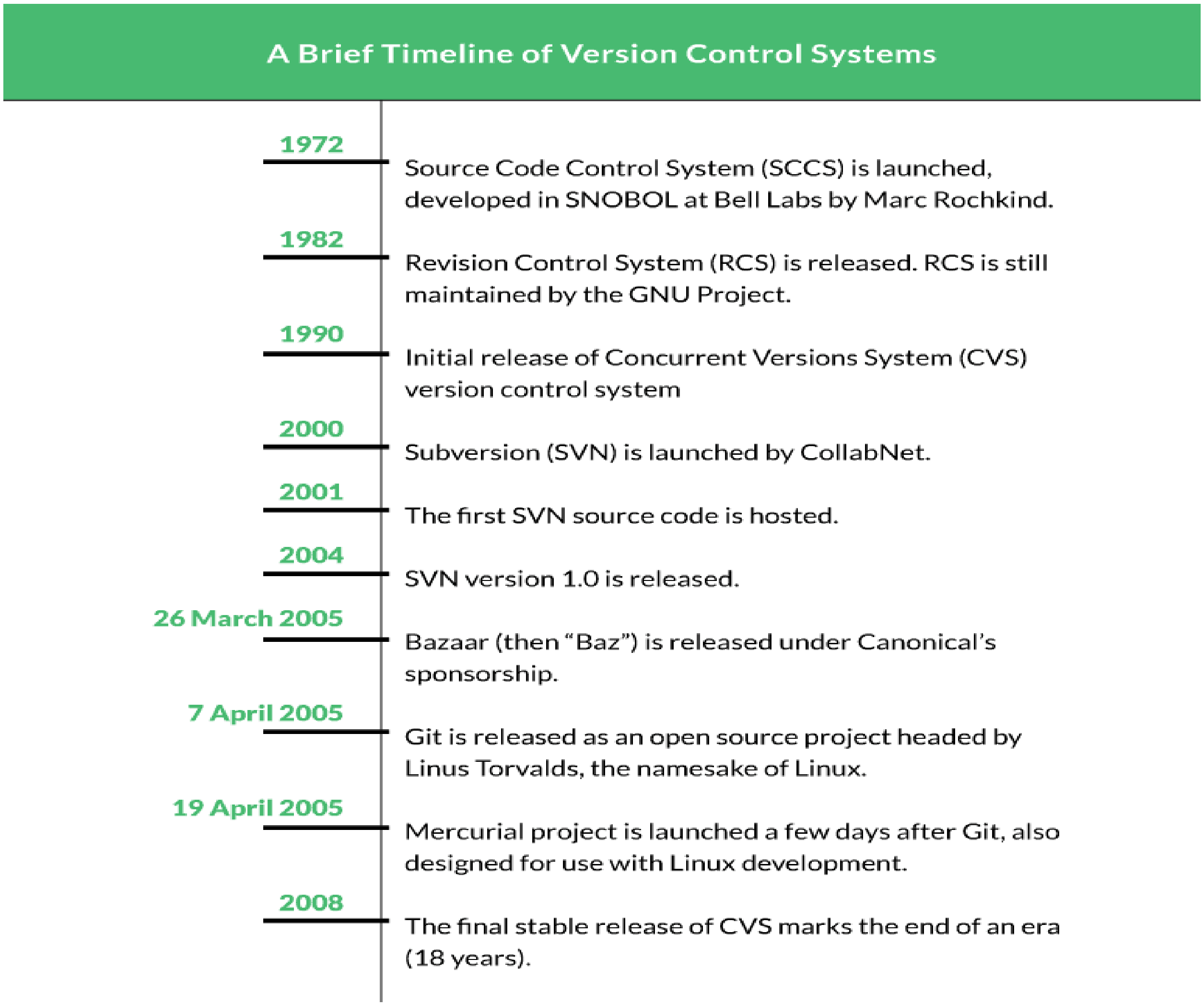
Version Controlling

This is the process of maintaining multiple versions of the code All the team members upload their code (check in) into the remote version controlling system. The VCS accepts the code uploads from multiple team members and integrates it so that when the other team members download the code they will be able to see the entire work down by the team.



A Brief Timeline of Version Control Systems

1972	Source Code Control System (SCCS) is launched, developed in SNOBOL at Bell Labs by Marc Rochkind.
1982	Revision Control System (RCS) is released. RCS is still maintained by the GNU Project.
1990	Initial release of Concurrent Versions System (CVS) version control system
2000	Subversion (SVN) is launched by CollabNet.
2001	The first SVN source code is hosted.
2004	SVN version 1.0 is released.
26 March 2005	Bazaar (then "Baz") is released under Canonical's sponsorship.
7 April 2005	Git is released as an open source project headed by Linus Torvalds, the namesake of Linux.
19 April 2005	Mercurial project is launched a few days after Git, also designed for use with Linux development.
2008	The final stable release of CVS marks the end of an era (18 years).

VCS's also preserve older and later versions of the code so that at any time, we can switch between whichever version we want.

VCS's also keep a track of who is making what kind of changes.

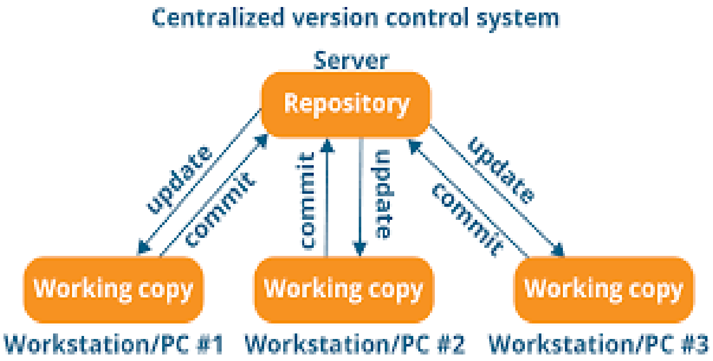
VCS's are categorized into 2 types: -

1 Centralized version controlling

2 Distributed version controlling

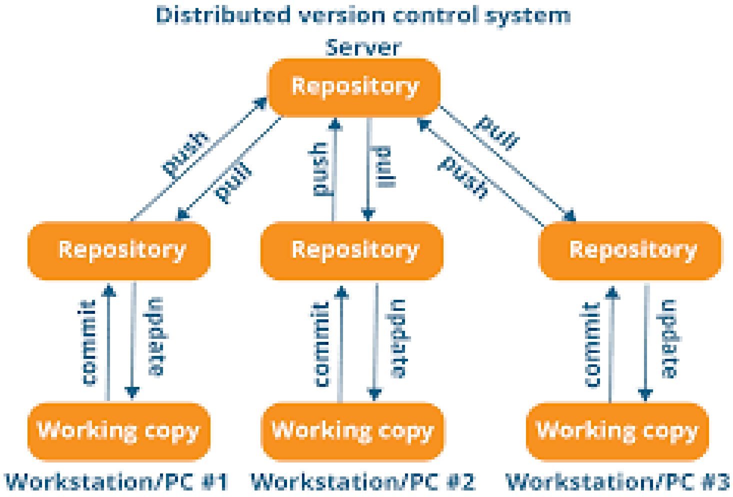
Centralized Version controlling:

Here we have a remote server (code repository) into which all the team members check in the code and all the features of version controlling are implemented in this remote server.



Distributed version controlling:

Here we have a local repository installed on every team member's machines where version controlling happens at the level of individual team members form where it is uploaded into a remote server where version controlling happens for the entire team.



Revision Control System (RCS):

A revision control system (RCS) is an application capable of storing, logging, identifying, merging or identifying information related to the revision of software, application documentation, papers or forms. Most revision control systems store this information with the help of a differential utility for documents.

A revision control system is an essential tool for an organization with multi-developer tasks or projects, as it is capable of identifying issues and bugs and of retrieving an earlier working version of an application or document whenever required.

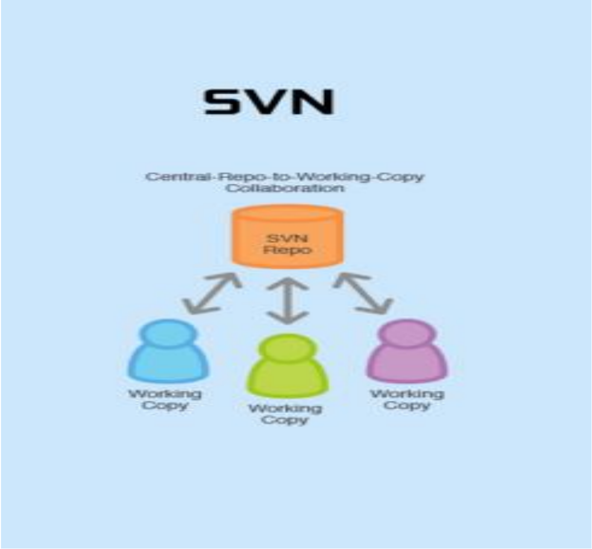
A revision control system is also known as a version control system.

Concurrent Versions System(CVS):

CVS stands for Concurrent Versions System. It is a free version control system. It saves the changes made to the files. Therefore, the developer can compare the versions. It allows multiple developers to work on the same project simultaneously. Also, this improves collaboration among the team members.

SubVersion(SVN):

SVN, known as Subversion or Apache Subversion, represent the most popular centralized version control system. A centralized system stores all files and historical data on a central server. When developers commit their changes, the changes directly are committed to the central server repository. SVN has a global revision number, it is a source code's snapshot.



Git Stages:

On the local machine git uses three sections

1 Working directory

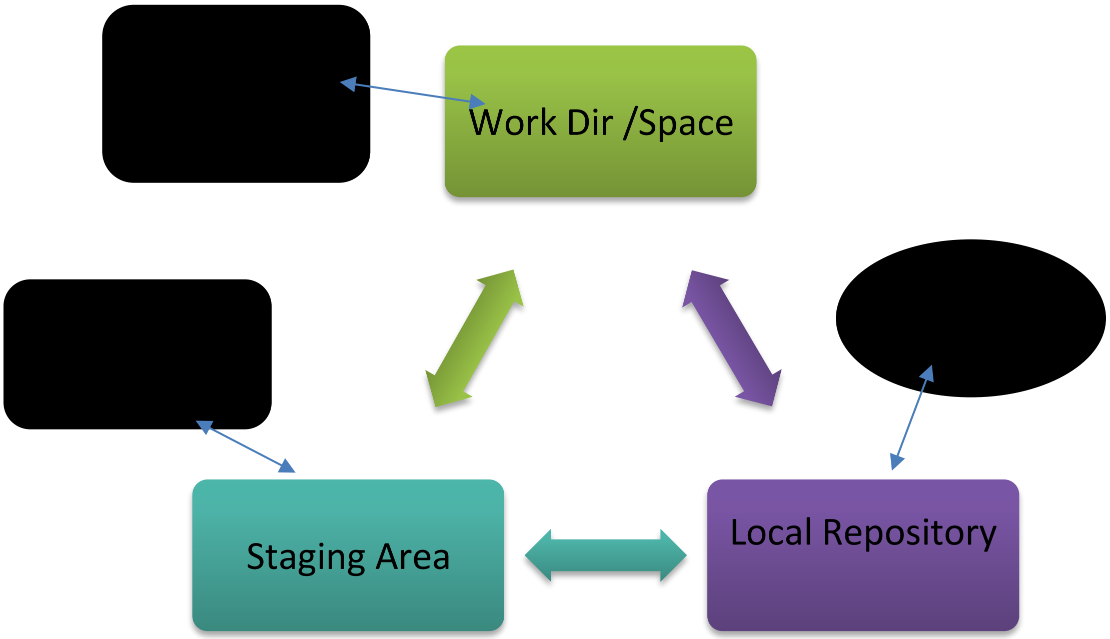
2 Staging Area

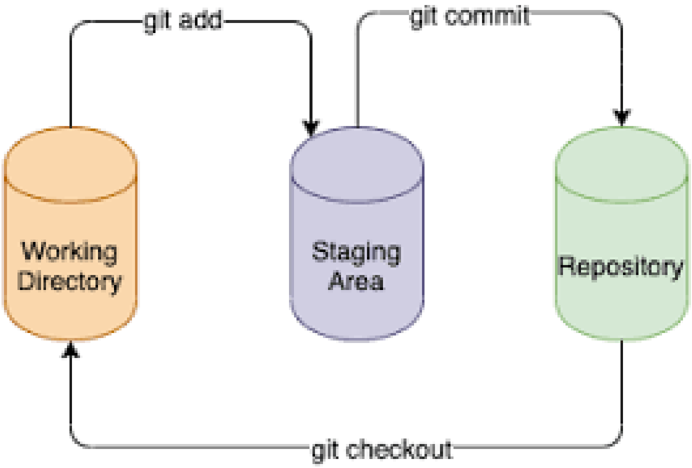
3 Local repository

Working directory is the location where all the code is created Initially all the files present here are called as untracked files.

Staging area is the location where file indexing happens and it is the buffer area of git and the files are called as indexed files.

Local repository is where version controlling happens and the files are called as committed files.





Setting up git on Windows:

1 Download git from

https://git-scm.com/downloads

2 Install it, as like normal software we installed.

3 Open gitbash and execute the git commands.

Setting up git in ubuntu linux servers:

1 First we Update the apt repository in Linux server.

```
sudo apt-get update
```

2 After apt repository update, we Install git on Linux sever.

```
sudo apt-get install -y git
```

Configuring User:

Configuring user and email globally for all users on a system

```
git config --global user.name " ur name"
git config --global user.email "ur mail-id"
```

How to use Git?

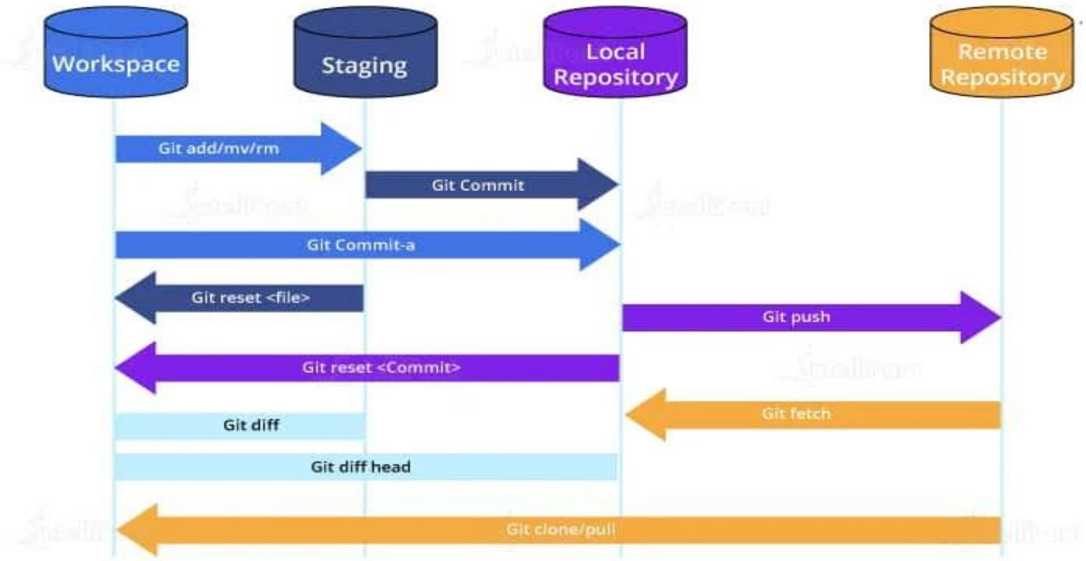
Much of Git's popularity is attributed to its user-friendly nature. You can check out these simple concepts to perform certain basic operations and get acquainted with Git basics.

Commit: This is an object that takes the current state of the repository.

Pull: This operation copies the changes made to a project to the local repository from the remote one.

Push: This operation copies the changes made to a project to the remote repository from the local one.

Now, let's take a look at some of the basic and commonly used commands in Git.



Git add:

This command allows you to add files in the Git staging area. The file must be added to the index of Git before being available to commit to any particular repository. You can use this command to add directories, files, etc. It's very important to know what you are staging and committing.

git add -A: stages all files, including new, modified, and deleted files, including files in the current directory and in higher directories that still belong to the same git repository

git add . : adds the entire directory recursively, including files whose names begin with a dot

git add -u: stages new and modified files only, NOT deleted files

	New files	Modified files	Deleted files	Files with names beginning with a dot	Current directory	Higher directories
git add -A	Yes	Yes	Yes	Yes	Yes	Yes
git add .	Yes	Yes	Yes	Yes	Yes	No
git add -u	No	Yes	Yes	Yes	Yes	Yes

Git Commit:

The "commit" command is used to save your changes to the local repository.

The commit is not automatically transferred to the remote server. Using the "git commit" command only saves a new commit object in the local Git repository. Exchanging commits has to be performed manually and explicitly (with the "git fetch", "git pull", and "git push" commands).

Important Options:

-m <message>

Sets the commit's message. Make sure to provide a concise description that helps your teammates (and yourself) understand what happened.

-a

Includes all currently changed files in this commit. Keep in mind, however, that untracked (new) files are not included.

--amend

Rewrites the very last commit with any currently staged changes and/or a new commit message. Git will rewrite the last commit and effectively replace it with the amended one. Note that such a rewriting of commits should only be performed on commits that have *not* been pushed to a remote repository, yet.

**For practice here some content is there for execute commands **

1) To initialize the working dir as a git repo

```
git init
```

This will create a hidden folder called. git where it will store all configurations for git to work.

2) To send a file into from working dir to staging area

```
git add <filename>
```

To send multiple files

```
git add file1 file2 file3
```

To send all files

```
git add .
```

“. “represents current working dir

3) To bring files from staging area to working dir

```
git rm --cached <filename>
```

(or)

```
git reset <filename>
```

4) To send files from staging area to local repository

```
git commit -m "Some msg"
```

5) To check the status of working dir and staging area files

```
git status
```

6) To see the commits done on the local repository

```
git log
```

To see this output in simple one line format

```
git log --oneline
```

Ignoring Content:

```
.gitignore
```

This is a special configuration file that is used to store private files info. Any file whose name is stored in .gitignore will not be accessed by git.

1) Create few file

```
touch file1 file2 file3 file4
```

2) Check the git status

```
git status
```

It will show the above 4 files as untracked

3) Create .gitignore and store the above 4 filenames in it

```
cat > .gitignore
file1
file2
file3
file4
```

4) Check the status of git

```
git status
```

It will no longer show file 1-4

GitHub:

GitHub is an online software development platform used for storing, tracking, and collaborating on software projects. It enables developers to upload their own code files and to collaborate with fellow developers on open-source projects. GitHub also serves as a social networking site in which developers can openly network, collaborate, and pitch their work.

Working on the GitHub:

This is the remote repository into which the code is uploaded and this process is called as checkin.

1) Signup for a github account.

2) Sign in into that account

3) Click on + on top right corner

4) Click on New repository

5) Enter some repository name

6) Select Public or Private

7) Click on Create repository

8) Go to Push an existing repository from command line and copy paste the commands & Enter username and token of github.

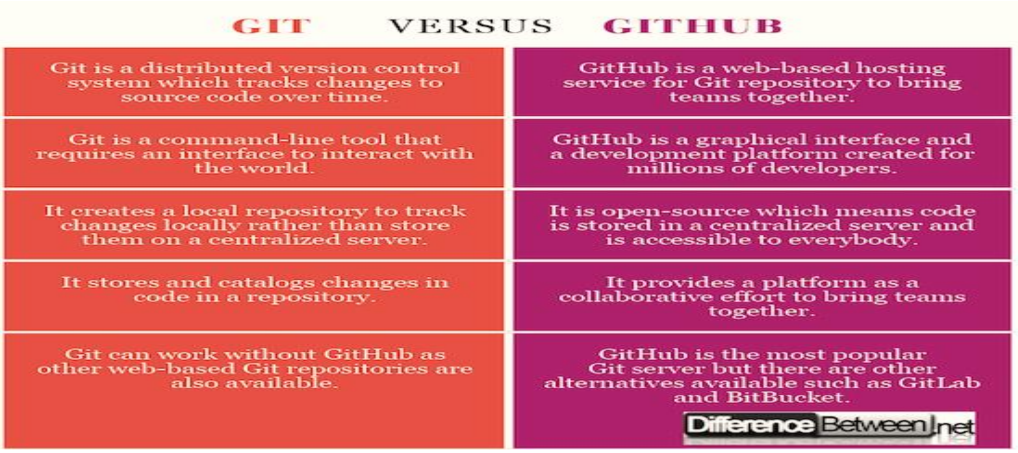
Downloading the code from the remote github. This can be done in three ways. They are

git clone

git fetch

git pull

Difference B/W Git & GitHub:



GIT VERSUS GITHUB

Git is a distributed version control system which tracks changes to source code over time.	GitHub is a web-based hosting service for Git repository to bring teams together.
Git is a command-line tool that requires an interface to interact with the world.	GitHub is a graphical interface and a development platform created for millions of developers.
It creates a local repository to track changes locally rather than store them on a centralized server.	It is open-source which means code is stored in a centralized server and is accessible to everybody.
It stores and catalogs changes in code in a repository.	It provides a platform as a collaborative effort to bring teams together.
Git can work without GitHub as other web-based Git repositories are also available.	GitHub is the most popular Git server but there are other alternatives available such as GitLab and BitBucket.

Branching in Git

This is a feature of git using which we can create separate branches for different functionalities and later merge them with the main branch also known as the master branch. This will help in creating the code in an uncluttered way. It allows separate work to Developers.

1) To see the list of local branches

```
git branch
```

2) To see the list all branches local and remote

```
git branch -a
```

3) To create a branch

```
git branch <branch_name>
```

4) To move into a branch

```
git checkout <branch_name>
```

5) To create a branch and also move into it

```
git checkout -b <branch_name>
```

6) To merge a branch

```
git merge <branch_name>
```

7) To delete a branch that is merged

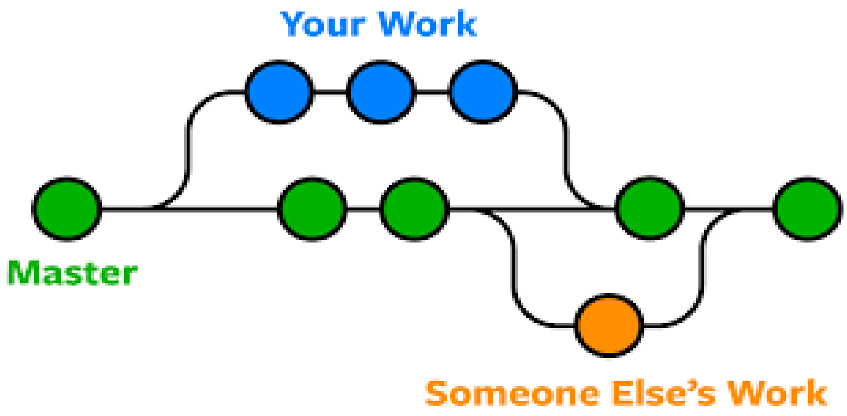
```
git branch -d <branch_name>
```

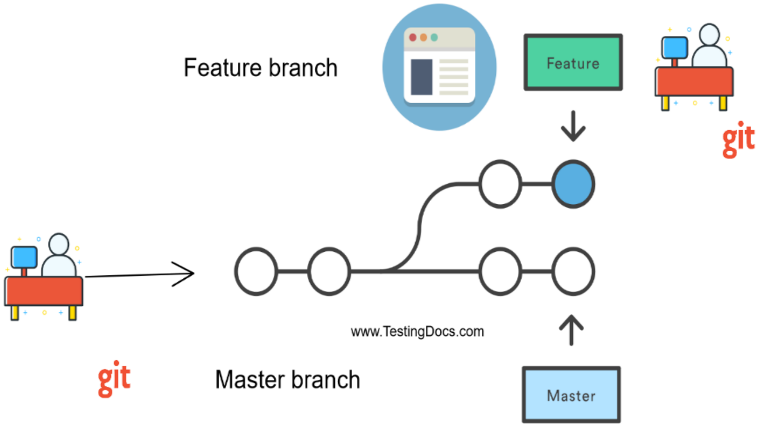
This is also called as soft delete

8) To delete a branch that is not merged

```
git branch -D <branch_name>
```

This is also known as hard delete





Git branching

Git Repositories:

You can own repositories individually, or you can share ownership of repositories with other people in an organization.

You can restrict who has access to a repository by choosing the repository's visibility. For user-owned repositories, you can give other people collaborator access so that they can collaborate on your project. If a repository is owned by an organization, you can give organization members access permissions to collaborate on your repository. For more information, see "Permission levels for a personal account repository" and "Repository roles for an organization."

With GitHub Free for personal accounts and organizations, you can work with unlimited collaborators on unlimited public repositories with a full feature set, or unlimited private repositories with a limited feature set. To get advanced tooling for private repositories, you can upgrade to GitHub Pro, GitHub Team, or GitHub Enterprise Cloud.

You can use repositories to manage your work and collaborate with others.

- You can use issues to collect user feedback, report software bugs, and organize tasks you'd like to accomplish.
- You can use GitHub Discussions to ask and answer questions, share information, make announcements, and conduct or participate in conversations about a project.
- You can use pull requests to propose changes to a repository.
- You can use project boards to organize and prioritize your issues and pull requests.

About repository visibility

You can restrict who has access to a repository by choosing a repository's visibility: public or private.

When you create a repository, you can choose to make the repository public or private. Repositories in organizations that use GitHub Enterprise Cloud and are owned by an enterprise account can also be created with internal visibility.

- Public repositories are accessible to everyone on the internet.
- Private repositories are only accessible to you, people you explicitly share access with, and, for organization repositories, certain organization members.

Organization owners always have access to every repository created in an organization. For more information, see "Repository roles for an organization." People with admin permissions for a repository can change an existing repository's visibility. For more information, see "Setting repository visibility."

Setting repository visibility:

1)Making a repository private

- GitHub will detach public forks of the public repository and put them into a new network. Public forks are not made private.
- If you're using GitHub Free for personal accounts or organizations, some features won't be available in the repository after you change the visibility to private. Any published GitHub Pages site will be automatically unpublished. If you added a custom domain to the GitHub Pages site, you should remove or update your DNS records before making the repository private, to avoid the risk of a domain takeover. For more information, see "GitHub's products and "Managing a custom domain for your GitHub Pages site."
- GitHub will no longer include the repository in the GitHub Archive Program. For more information, see "About archiving content and data on GitHub."
- GitHub Advanced Security features, such as code scanning, will stop working. For more information, see "About GitHub Advanced Security."

2)Making a repository public

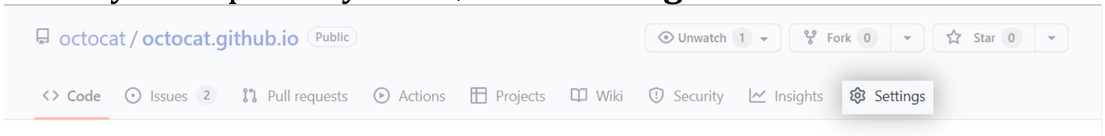
- GitHub will detach private forks and turn them into a standalone private repository. For more information, see "What happens to forks when a repository is deleted or changes visibility?"

- If you're converting your private repository to a public repository as part of a move toward creating an open source project, see the Open Source Guides for helpful tips and guidelines. You can also take a free course on managing an open source project with GitHub Skills. Once your repository is public, you can also view your repository's community profile to see whether your project meets best practices for supporting contributors. For more information, see "Viewing your community profile."
- The repository will automatically gain access to GitHub Advanced Security features.

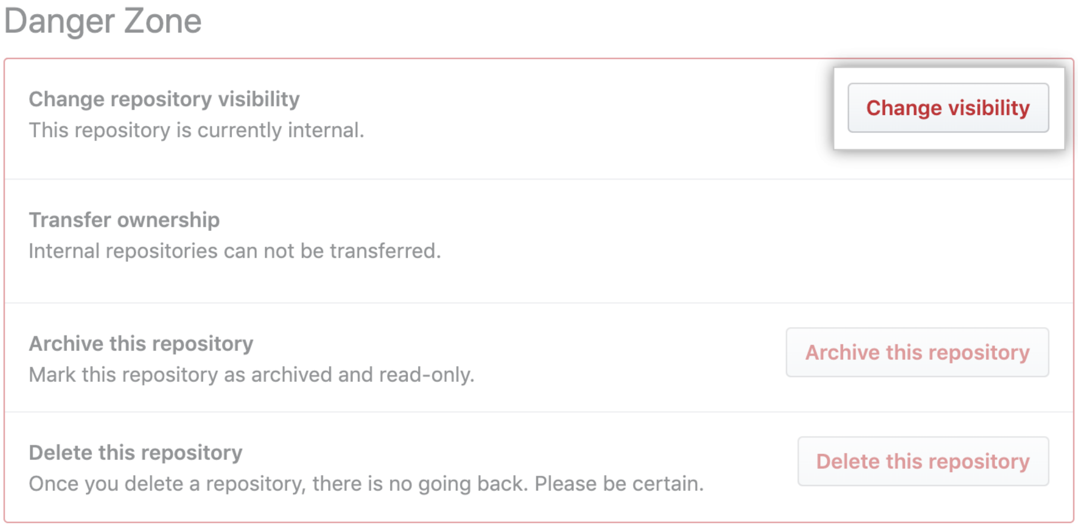
For information about improving repository security, see "Securing your repository."

Changing a repository's visibility:

1. On GitHub.com, navigate to the main page of the repository.
2. Under your repository name, click **Settings**.



3. Under "Danger Zone", to the right of to "Change repository visibility", click **Change visibility**.


Danger Zone

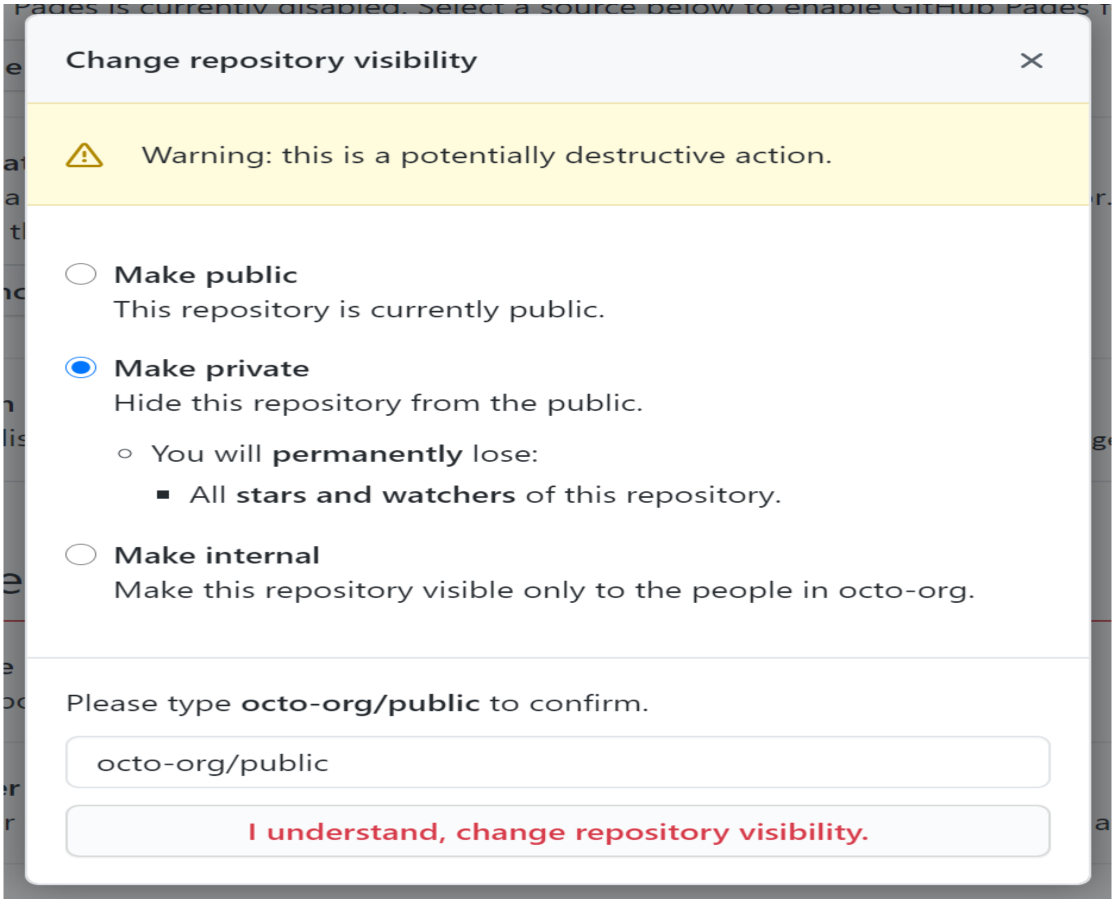
Change repository visibility
This repository is currently internal.

Transfer ownership
Internal repositories can not be transferred.

Archive this repository
Mark this repository as archived and read-only.

Delete this repository
Once you delete a repository, there is no going back. Please be certain.

4. Select a visibility.



Pages is currently disabled. Select a source below to enable GitHub Pages f

Change repository visibility ×

⚠ Warning: this is a potentially destructive action.

○ **Make public**
This repository is currently public.

● **Make private**
Hide this repository from the public.

- You will **permanently** lose:
  - All **stars and watchers** of this repository.

○ **Make internal**
Make this repository visible only to the people in octo-org.

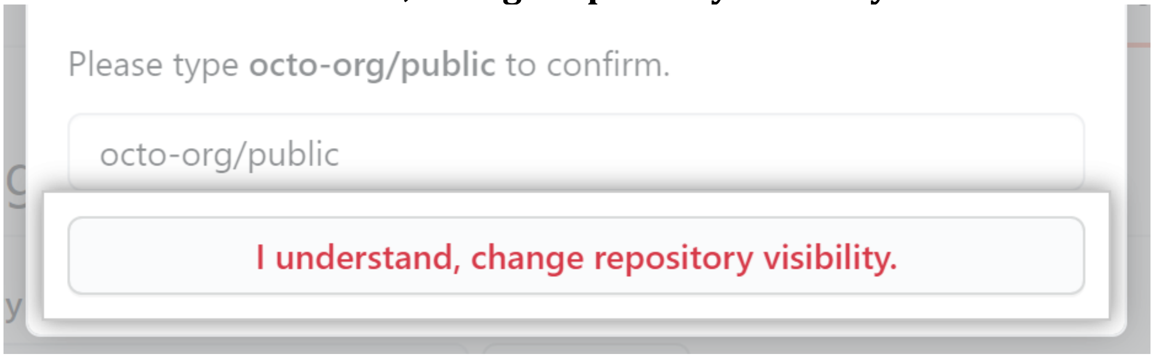
Please type **octo-org/public** to confirm.

octo-org/public

I understand, change repository visibility.

5. To verify that you're changing the correct repository's visibility, type the name of the repository you want to change the visibility of.

6. Click **I understand, change repository visibility**.



Please type **octo-org/public** to confirm.

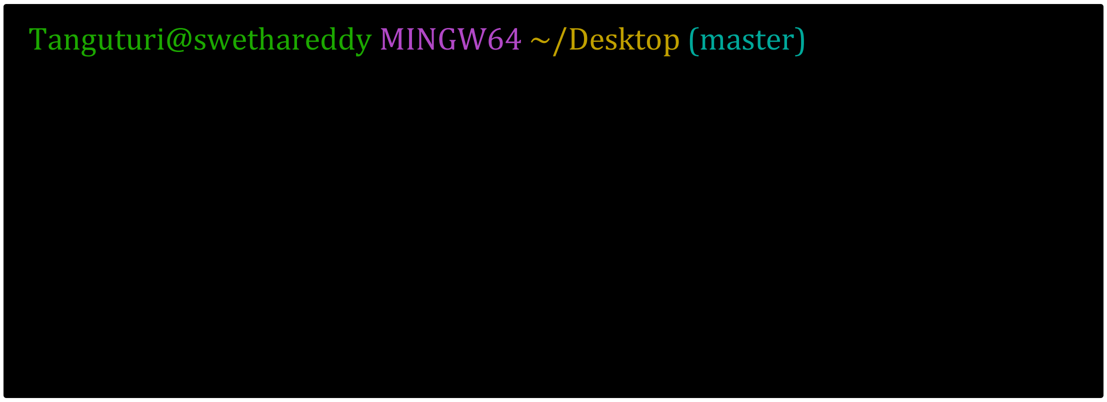
octo-org/public

I understand, change repository visibility.

Git Clone:

This will download all the code from the remote repository into the local repository and it is generally used only once when all the team members want a copy of the same code.

Syntax: git clone <remote_git_repo_url>



```
Tanguturi@swethareddy MINGW64 ~/Desktop (master)
```

Git Fetch:

This will download only the modified files but it will place them on a separate branch called as "remote branch", we can go into this remote branch check if the modifications are acceptable and then merge it with the main branch.

Steps for execute:

1) Open the github

2) Go to the repository that we uploaded

3) Select a file and edit it--->Click on commit changes

4) Open git bash

5) git fetch

6) To see the name of remote branch

   git branch –a

7) To switch into this branch

```
git checkout <branch_name_from_step6>
```

8) View the modified file

```
cat filename
```

9) If these modifications are ok then merge with main branch

```
git checkout main
git merge <branch_name_from_step6>
```

Git Push:

Git push uploads all local branch commits to the corresponding remote branch. git push updates the remote branch with local commits. It is one of the four commands in Git that prompts interaction with the remote repository. You can also think of git push as update or publish.

Common usages and options for git push:

- git push -f: Force a push that would otherwise be blocked, usually because it will delete or overwrite existing commits *(Use with caution!).*
- git push -u origin [branch]: Useful when pushing a new branch, this creates an upstream tracking branch with a lasting relationship to your local branch.
- git push --all: Push all branches.
- git push --tags: Publish tags that aren't yet in the remote repository.

Git Pull:

This will download only the modified files and merge them with our local branches.

1 Open the github

2 Go to the repository that we uploaded

3 Select a file and edit it--->Click on commit changes

4 Open git bash

5 `git pull`

We can see the modified files on the main branch

Git Merge:

Merging always happens bases on the time stamps of the commits

1) Create few commits on master

```
touch f1
git add .
git commit -m "a"
touch f2
git add .
git commit -m "b"
```

2) Check the git commit history

```
git log --oneline
```

3) Create a test branch and create few commits on it

```
git checkout -b test <test is a Branch name>
touch f3
git add .
git commit -m "c"
touch f4
git add .
git commit -m "d"
```

4) Check the commit history

```
git log --oneline
```

5) Go back to master and create few more commits

```
git checkout master <Master is main name >
touch f5
git add .
git commit -m "e"
touch f6
git add .
git commit -m "f"
```

6) Check the commit history

```
git log --oneline
```

7) Merge test with master

```
git merge test <test is a Branch name>
```

8)Check the commit history

```
git log --oneline
```

Git Fork:

A fork is a rough copy of a repository. Forking a repository allows you to freely test and debug with changes without affecting the original project. One of the excessive use of forking is to propose changes for bug fixing. To resolve an issue for a bug that you found, you can:

- Fork the repository.
- Make the fix.
- Forward a pull request to the project owner.

Forking is not a Git function; it is a feature of Git service like GitHub.

When to Use Git Fork:

Generally, forking a repository allows us to experiment on the project without affecting the original project. Following are the reasons for forking the repository:

- Propose changes to someone else's project.
- Use an existing project as a starting point.

Git Stash:

Git stash is a built-in command with the distributed Version control tool in Git that locally stores all the most recent changes in a workspace and resets the state of the workspace to the prior commit state.

A user can retrieve all files put into the stash with the git stash pop and git stash apply commands. Git stash acts as a mechanism to locally version files without those versions being seen by other developers who share the same git repository.

The most common git stash options include:

git stash push: Creates a new stash and rolls back the state of all modified files;

git stash pop: Takes the files in a stash, places them back into the development workspace and deletes the stash from history;

git stash apply: Takes the files in a stash and places them back into the development workspace, but does not delete the stash from history.

git stash list: Displays the stash history in chronological order.

git stash clear: Removes all entries in the git stash history.

Git rebase:

This is called as fast forward merge where the commits coming from a branch are projected as the top most commits on master branch.

1) Implement step1-6 below

1 Create few commits on master

```
touch f1
git add .
git commit -m "a"
touch f2
git add .
git commit -m "b"
```

2 Check the git commit history

```
git log –oneline
```

3 Create a test branch and create few commits on it

```
git checkout -b test <test is a Branch name>

touch f3

git add .

git commit -m "c"

touch f4

git add .

git commit -m "d"
```

4 Check the commit history

```
git log --oneline
```

5 Go back to master and create few more commits

```
git checkout master

touch f5

git add .
git commit -m "e"

touch f6

git add .

git commit -m "f"
```

6 Check the commit history

```
git log –oneline
```

2) To rebase test with master

```
git checkout test <test is a Branch name>

git rebase master

git checkout master

git merge test <test is a Branch name>
```

3 Check the commit history

```
git log –oneline
```

Git Cherry-pick:

git cherry-pick is a powerful command that enables arbitrary Git commits to be picked by reference and appended to the current working HEAD. Cherry picking is the act of picking a commit from a branch and applying it to another. git cherry-pick can be useful for undoing changes. This is used to selectively pick up certain commits and add them to the master branch.

1 On master create few commits

a--->b

2 Create a test branch and create few commits

```
git checkout -b test
```

a--->b--->c--->d--->e--->f--->g

3 To bring only c and e commits to master

```
git checkout master

git cherry-pick c_commitid e_commitid
```

Git reset:

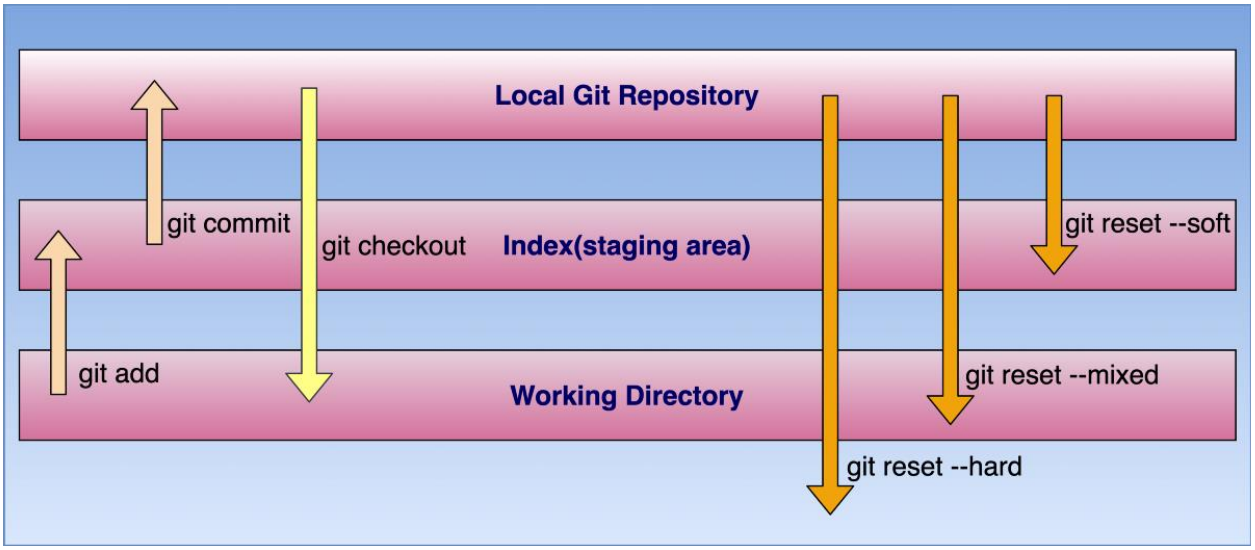
This is a command of git using which we can toggle between multiple versions of git and access whichever version we want.

Reset can be done in 3 ways

1 Hard reset- it will take 3steps back from commit operation.

2 soft reset- it will take 1step back from commit operation.

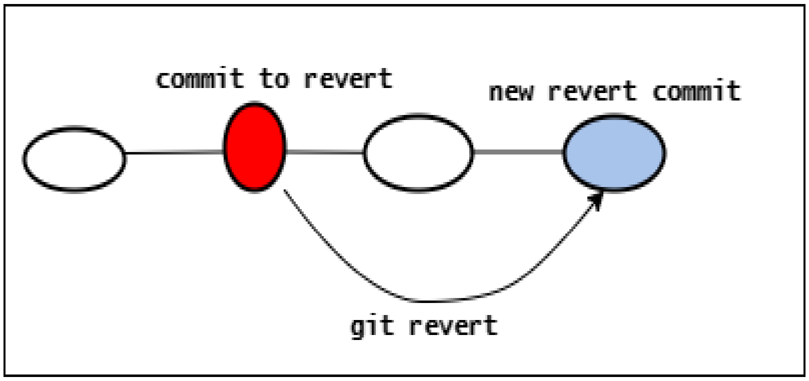
3 Mixed reset- it will take 2step back from commit operation.



Git Revert:

git revert to record some new commits to reverse the effect of some earlier commits (possibly faulty ones). It is an "undo" command, but technically it is much more than that. Git revert does not delete any commit in this project history. Instead, it inverts the changes implemented in a commit and appends new commits with the opposite effect. This process helps Git remove the unwanted commit from the codebase and retain the history of every commit and the reverted one. This makes it a handy command, especially when collaborating on a project.

Git revert undoes changes in a project commit history without tampering with it. When reverting, this operation takes the specific commit, inverts the changes from that commit, and implements a new reverse commit—only removing the changes tied to the reverted commit.



Git Merge Conflict:

When working with version control systems such as Git, most merge conflicts resolve automatically. However, there are situations where git merge is unable to resolve an issue.

Some examples of merge conflicts include:

- Changing the same lines of code in a file.
- Removal of files while changes happen in another place.

Since the problem happens locally and the rest of the project members are unaware of the issue, resolving the conflict is of high priority and requires an immediate fix.

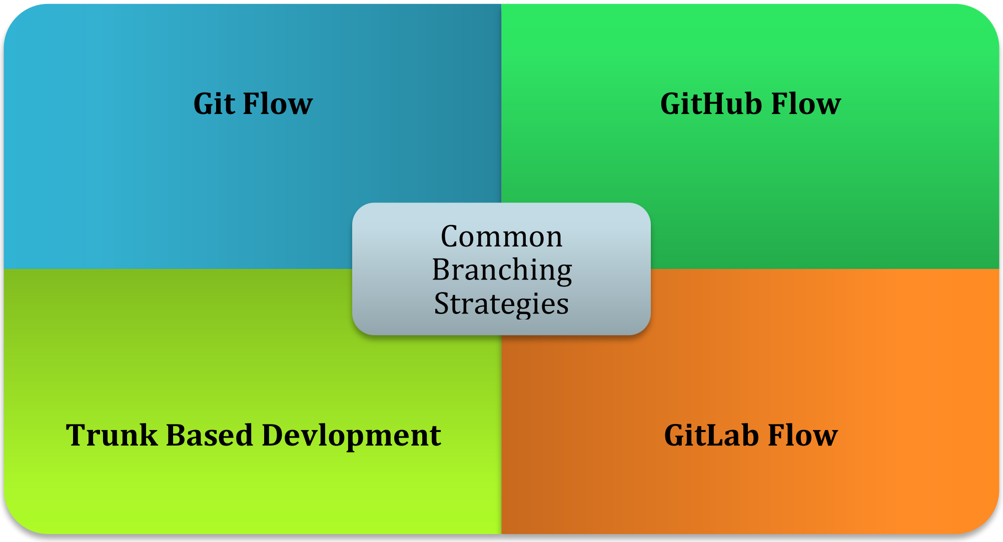
Types of Git Merge Conflicts

The general types of merge conflicts depend on when the issue appears. The conflicts happen either:

- **Before merging**, indicating there are local changes not up to date. The conflict error message appears before the merge starts to avoid issues.
- **During the merge**, indicating an overwrite issue. The error message appears and stops the merging process to avoid overwriting changes.

Git Branching Strategy:

When working with Git, a Git branching strategy (or version control branching strategy) is the model used so that your codebase evolves in a logical, consistent, and (a mostly) "easy to understand" way. The model provides the rules for how, when, and why branches are created and named. There are different branching strategies that allow for work on like-concepts to be organized and developed in parallel.

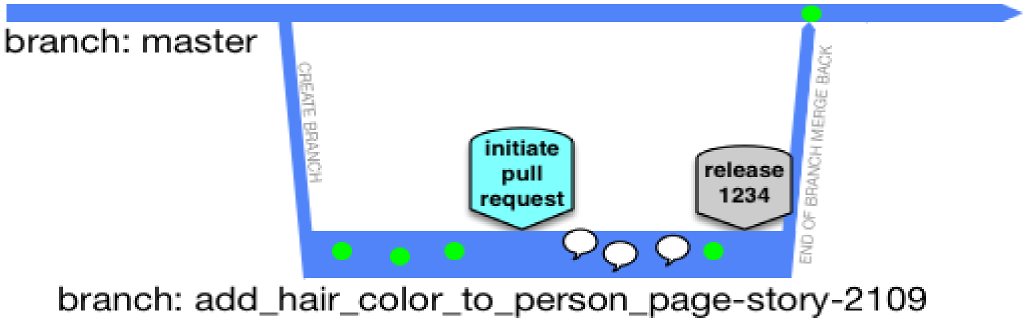


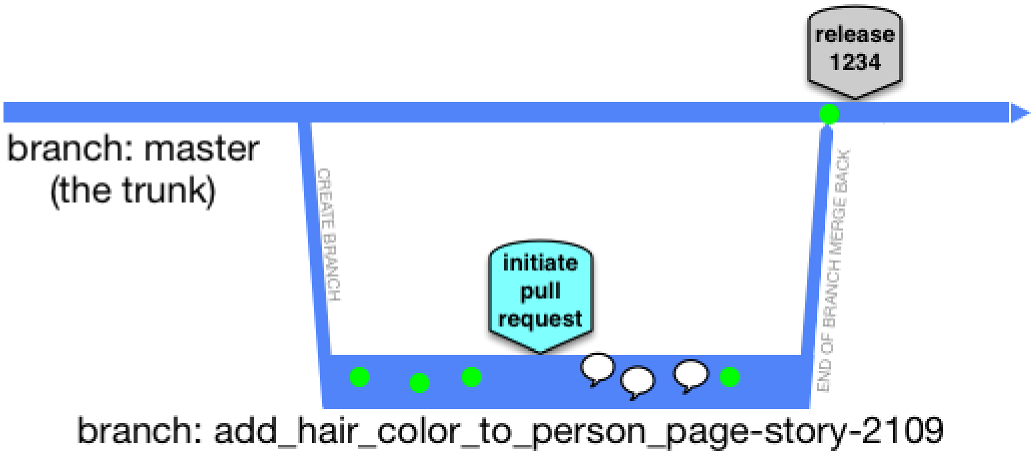
<u>The Top Five Main Git Branching Strategies</u>

Strategy	Overview

GitFlow	Created by Vincent Driessen, GitFlow works with two main branches – master and develop – over the lifetime of the project. It also uses three supporting branches: feature-*, hotfix-*, and release-*. It's the most complex model.
GitHub Flow	A simpler strategy promoted by Scott Chacon of GitHub mandates keeping a continuously deployable main branch. A feature branch is created to work on any feature or bug fix. Each feature branch must be finished and fully tested before being merged with the main branch.
Trunk-Based Development	Very similar to GitHub Flow except that Trunk-Based Development suggests deployment after production code is merged to the main branch to minimize chances for regression.
GitLab Flow	Created by GitLab, this strategy is like an extension of GitHub Flow with master and feature branches. However, it adds environment and release branches to better support SaaS and mobile projects.
OneFlow	Formulated by Adam Ruka, OneFlow proposes to be a simplification of GitFlow with the use of rebase options some consider controversial.

Already, we can extinguish the torch of one of our brave contestants. Trunk-Based Development didn't get with the Flow. It's very, very similar to GitHub Flow. The difference between GitHub Flow and Trunk-Based Development can be seen comparing the two images below:



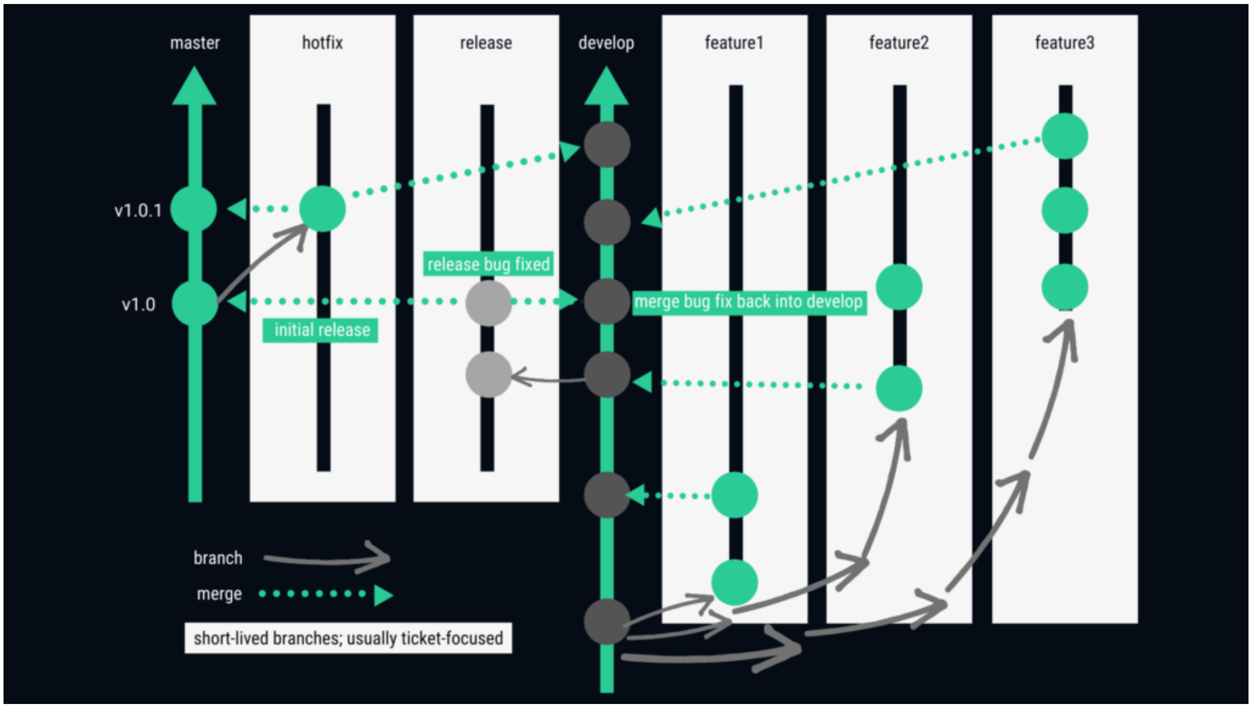

release 1234
branch: master (the trunk)
CREATE BRANCH
initiate pull request
END OF BRANCH MERGE BACK
branch: add_hair_color_to_person_page-story-2109

Comparing Version Control Strategies:

Let's move on to a comparison of the basic strengths and weaknesses of the remaining 4 Git branching strategy options. Each is designed for different scenarios. What's best for a startup isn't always going to work well for a large enterprise.

	GitFlow	GitHub Flow	GitLab Flow	OneFlow
Types of Projects	Enterprise	Startups	All	Enterprise
Branch Overhead	Heavy	Light	Variable	Light
Work Process Complexity	Complex	Simple	Simple	Medium
Merge Difficulty	Hard	Easy	Easy	Easy
Delivery Support	Releases	Continuous	Both	Releases
History	Convoluted	Clean	Clean	Variable
Feedback	Slow	Fast	Fast	Fast

There's a lot of red flags for GitFlow. Let's take a closer look at the GitFlow process:



It’s not so complex that you can’t understand the process, but it’s not easy on the eye, either.

GitFlow, though ubiquitous, has some limitations and issues that constrain its effectiveness. It’s perhaps the first clearly defined version control strategy to have been created. Functionally, it doesn’t work well for Continuous Delivery/Deploy or Multi-Environment projects. It works well for open-source and established/enterprise software projects so you can keep close control over any changes. It’s a useful strategy for large teams having many junior developers with close oversight by senior developers.

Advantages and Disadvantages:

Advantages:

- Good distributed model as each developer gets a local repository with a full history of commits which makes git fast compared to other VCs.
- Branching capabilities and merging are easy (as they are cheap), good data integrity.
- They are free and open-source we can easily download the source code and performs changes to it. They can handle larger projects efficiently.

- The push/pull operations are faster with a simple They save time and developers can fetch and create pull requests without switching.
- Data redundancy and replications. Add ons can be written in many languages.
- They have good and faster network performance and superior disk utilization and they think about its data like a sequence of snapshots.
- The object model is very simple and minimizes push/pull data transfers.

Disadvantages:

- GIT requires technical excellence and it is slower on windows. They have tedious command lines to input and don't track renames.
- They have poor GUI and usability. And also, they take a lot of resources which slows down the performance.
- GIT doesn't support checking out sub-trees. For each project, the central service would need to be set up for multiple package repositories.
- It lacks window support and doesn't track empty folders.
- GIT needs multiple branches to support parallel developments used by the developers.
- There is no built-in access control and doesn't support binary files.
- They do not provide access control mechanisms in case of security.
- The process of Packing is very expensive completely.