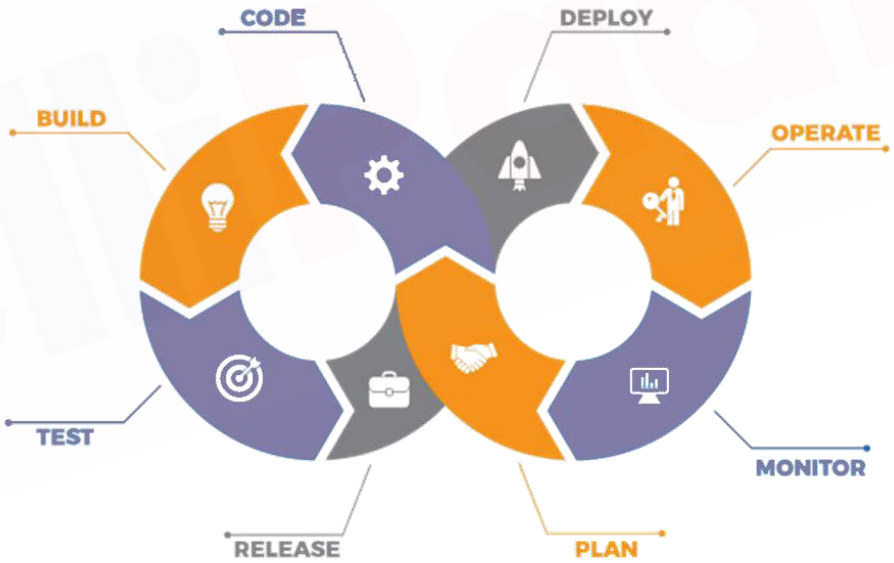
Version Control with GIT



Agenda

01 WHAT IS VERSION CONTROL?

02 TYPES OF VERSION CONTROL SYSTEM

03 INTRODUCTION TO GIT

04 GIT LIFECYCLE

05 COMMON GIT COMMANDS

06 MERGING IN GIT

07 RESOLVING MERGE CONFLICTS

08 GIT WORKFLOW

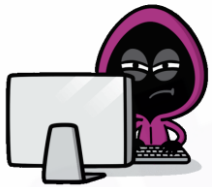
What is Version Control?

What is Version Control?

Version control is a system that records/manages changes to documents, computer programs etc over time. It helps us tracking changes when multiple people work on the same project

Problems before Version Control

Imagine, Developer A creates a software, and starts a company with this software.

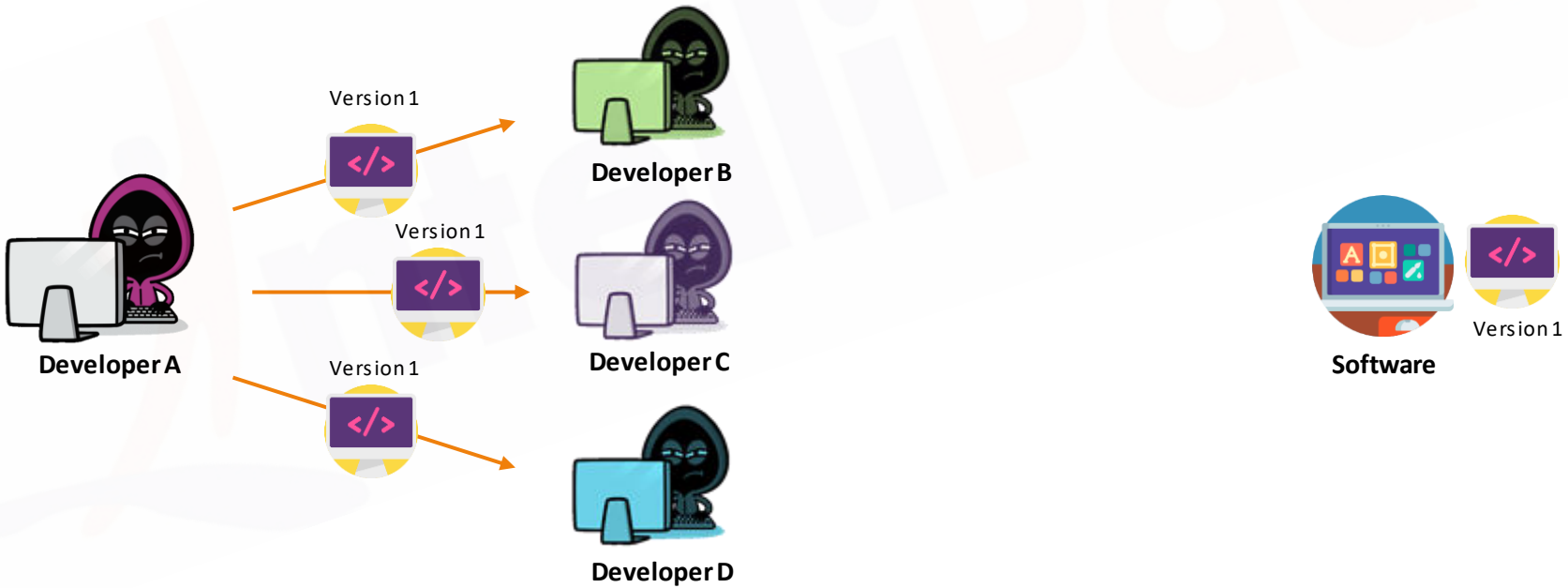


Developer A

Software

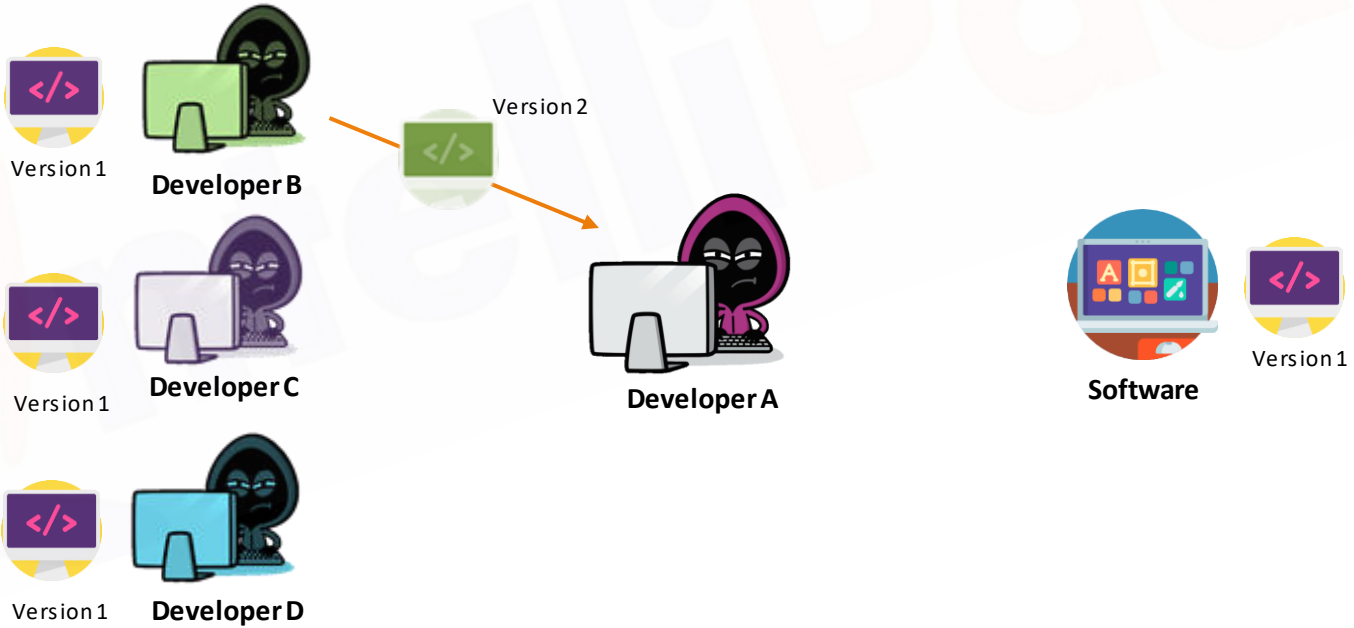
Problems before Version Control

As the company grows, Developer A hires more people to enhance the features of this software. Developer A shares the source code copy with each one of them to work on



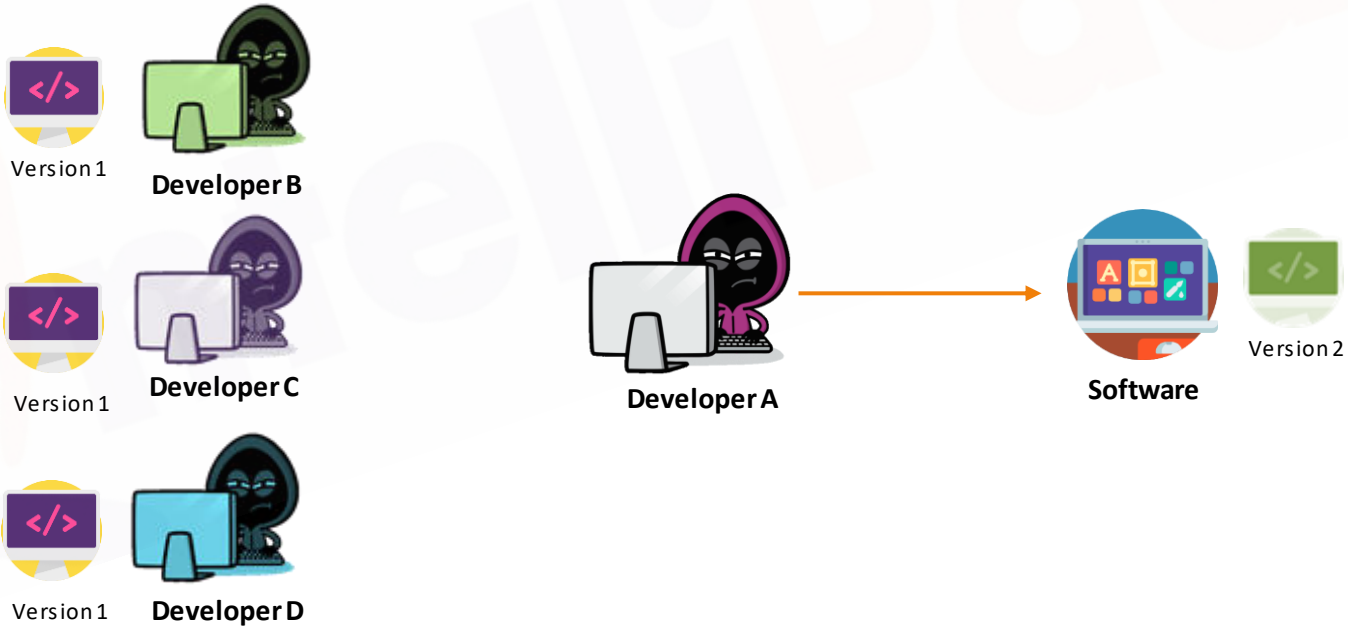
Problems before Version Control

Developer B, enhances the software with a feature and submits it to Developer A



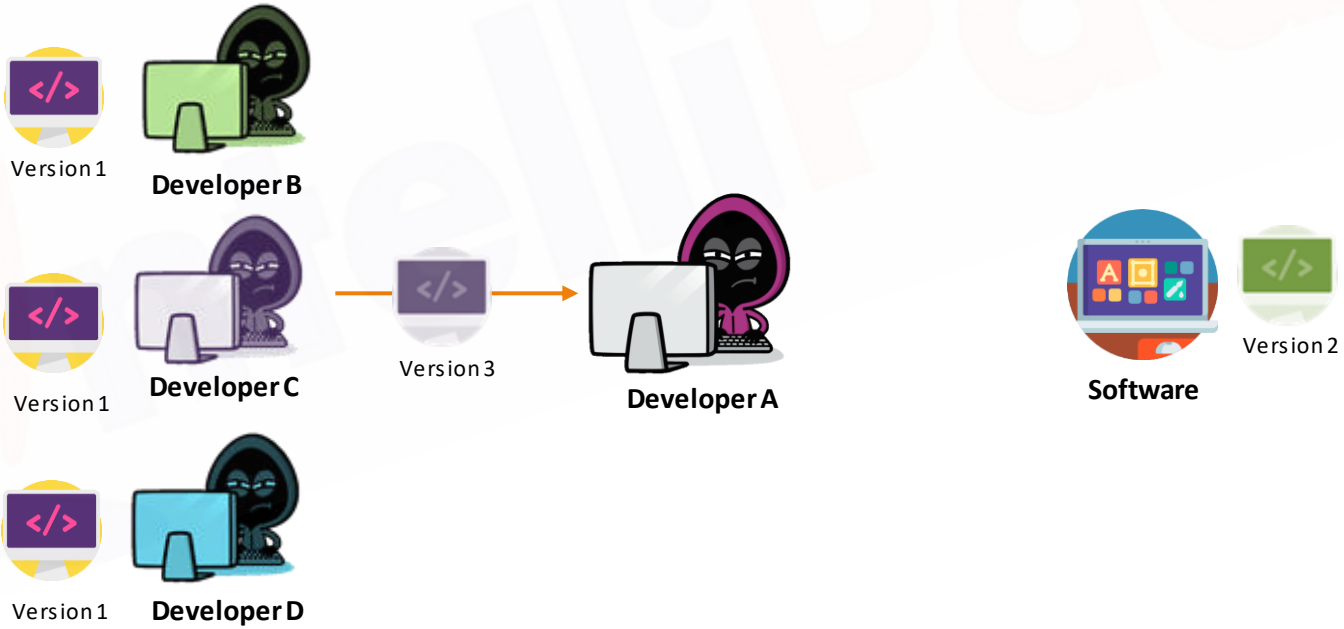
Problems before Version Control

Developer A, verifies the changes, and if all looks well, simply replaces the code of the main software



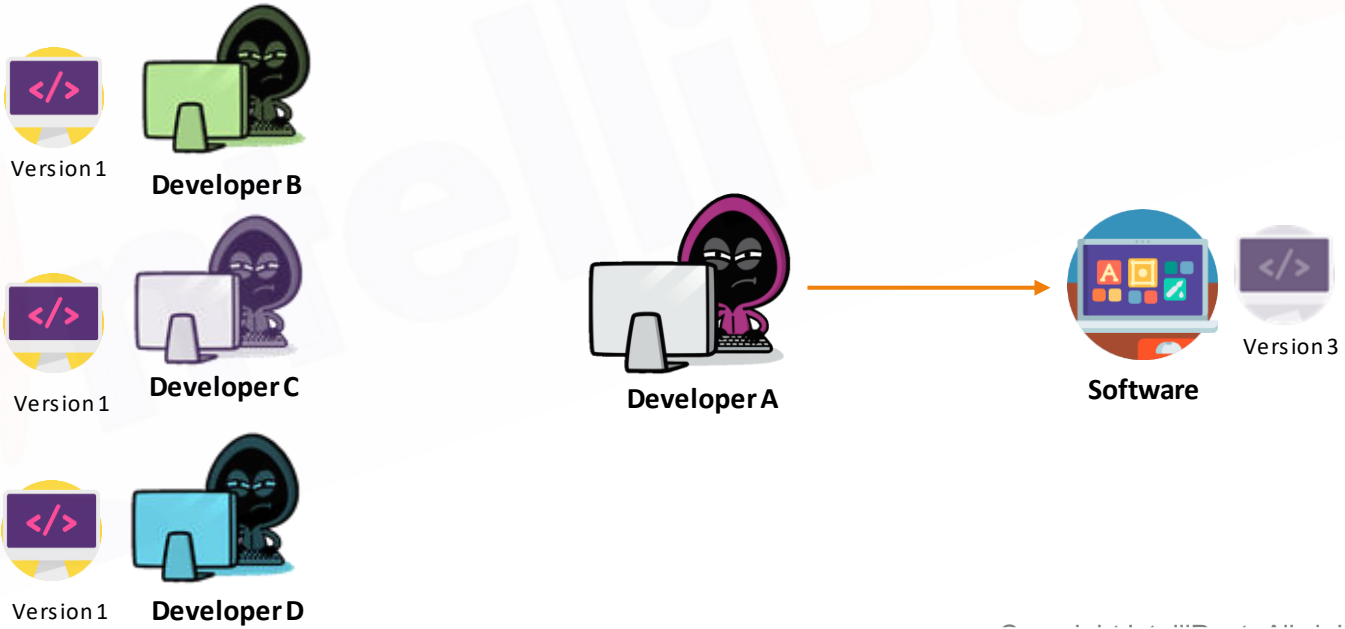
Problems before Version Control

Now, the problem starts here, Developer C also finished his work, and submits the changes to Developer A. But, Developer C worked on the code of Version 1.



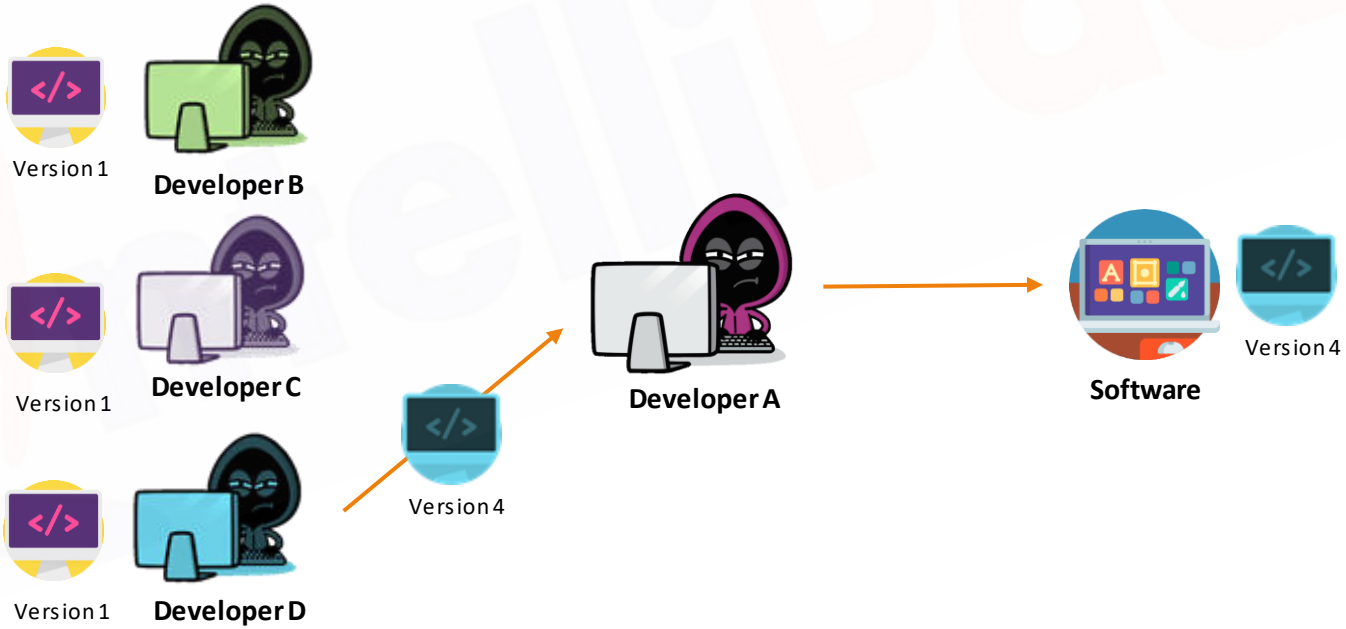
Problems before Version Control

Developer A verifies the features, takes the code changes and manually integrates them with Version 2 code



Problems before Version Control

Similarly when Developer C is done with his work, submits the work to Developer A. Developer A verifies it, manually integrates the changes with Version 3



Problems before Version Control

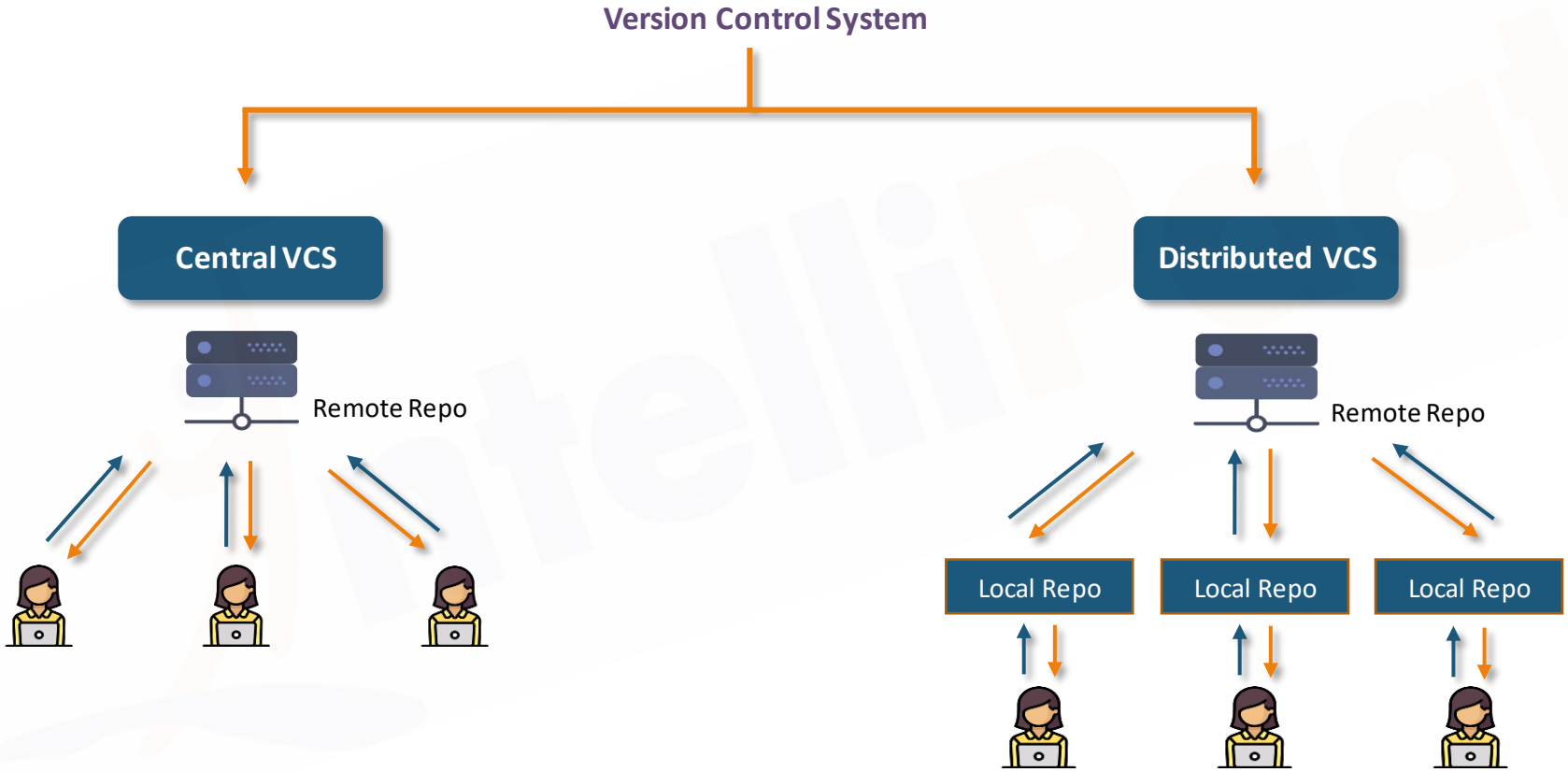
- Versioning was Manual
- Team Collaboration was a time consuming and hectic task
- No easy access to previous versions
- Multiple Version took a lot of space

Advantages of Version Control

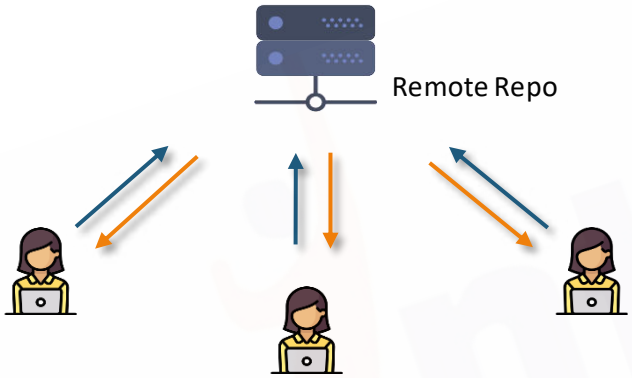
- Versioning is Automatic
- Team Collaboration is simple
- Easy Access to previous Versions
- Only modified code is stored across different versions, hence saves storage

Types of Version Control System

Types of Version Control System



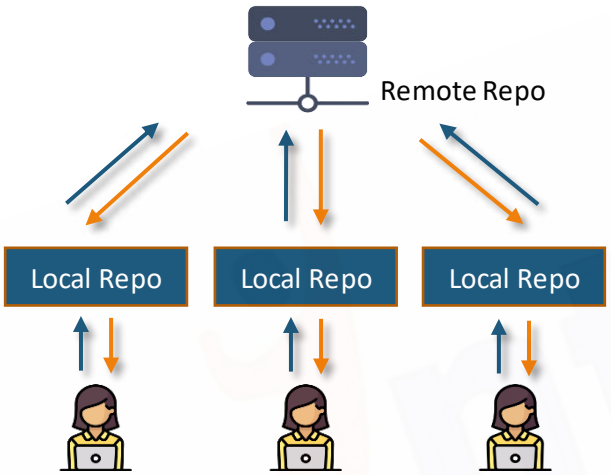
Centralized Version Control System



Centralized VCS

- Centralized Version Control System has one single copy of code in the central server
- Developers will have to “commit” their changes in the code to this central server
- “Committing” a change simply means recording the change in the central system

Distributed Version Control System



Distributed VCS

- In Distributed VCS, one does not necessarily rely on a central server to store all the versions of a project's file
- Every developer "clones" a copy of the main repository on their local system
- This also copies, all the past versions of the code on the local system too
- Therefore, the developer need not be connected to the internet to work on the code

Difference between DVCS and CVCS

Distributed VCS

- Everything except pushing and pulling can be done without Internet Connection
- Every Developer has full version history on local hard drive
- Committing and retrieving action is faster since data is on local drive
- Not Good for storing large files which are binary in nature, this would increase the repo size at every commit
- If a project has a lot of commits, downloading them may take a lot of time

Centralized VCS

- Needs a dedicated internet connection for every operation
- Developers just have the working copy and no version history on their local drive
- Committing and retrieving action is slower since it happens on the internet
- Good for storing large files, since version history is not downloaded
- Not dependent on the number of commits

Examples of CVCS

What is SVN?

- Apache Subversion is a software versioning and revision control system distributed as open source under the Apache License
- It is based on Centralized Version Control Architecture
- The development started in 2000, and this version finally became available in 2004
- It is still constantly being developed by a small but active open source community

Disadvantages of SVN

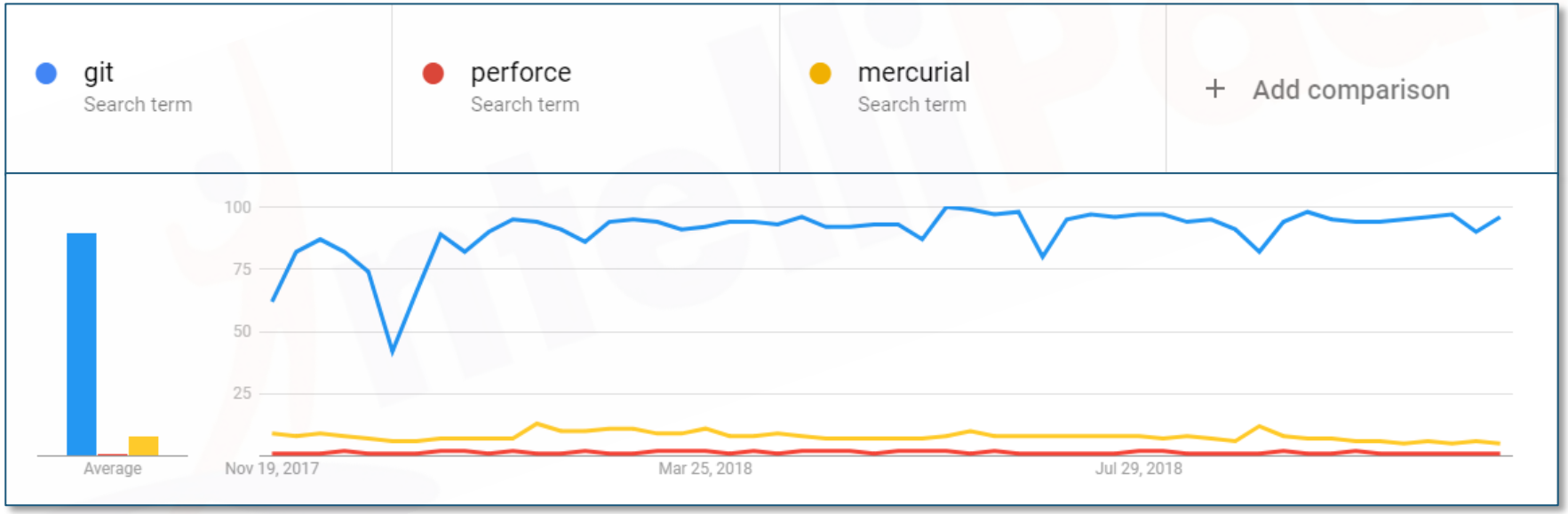
- Constantly needs an Internet Connection for any operation
- Version History is not downloaded or maintained on the local system
- Slower than DVCS, since requires internet for every operation
- Conflicts have to be resolved manually

Examples of DVCS

Introduction to Git

Why Git?

Git is the most popular tool among all the DVCS tools.



What is Git?

Git is a version-control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source-code management in software development, but it can be used to keep track of changes in any set of files.

Git Lifecycle

Git Lifecycle

Following are the lifecycle stages of files in Git

Working Directory

Staging Area

Commit

Git Lifecycle

Working Directory

Staging Area

Commit

- The place where your project resides in your local disk
- This project may or may not be tracked by git
- In either case, the directory is called the working directory
- The project can be tracked by git, by using the command *git init*
- By doing *git init*, it automatically creates a hidden .git folder

Git Lifecycle

Working Directory

Staging Area

Commit

- Once we are in the working directory, we have to specify which files are to be tracked by git
- We do not specify all files to be tracked in git, because some files could be temporary data which is being generated while execution
- To add files in the staging area, we use the command *git add*

Git Lifecycle

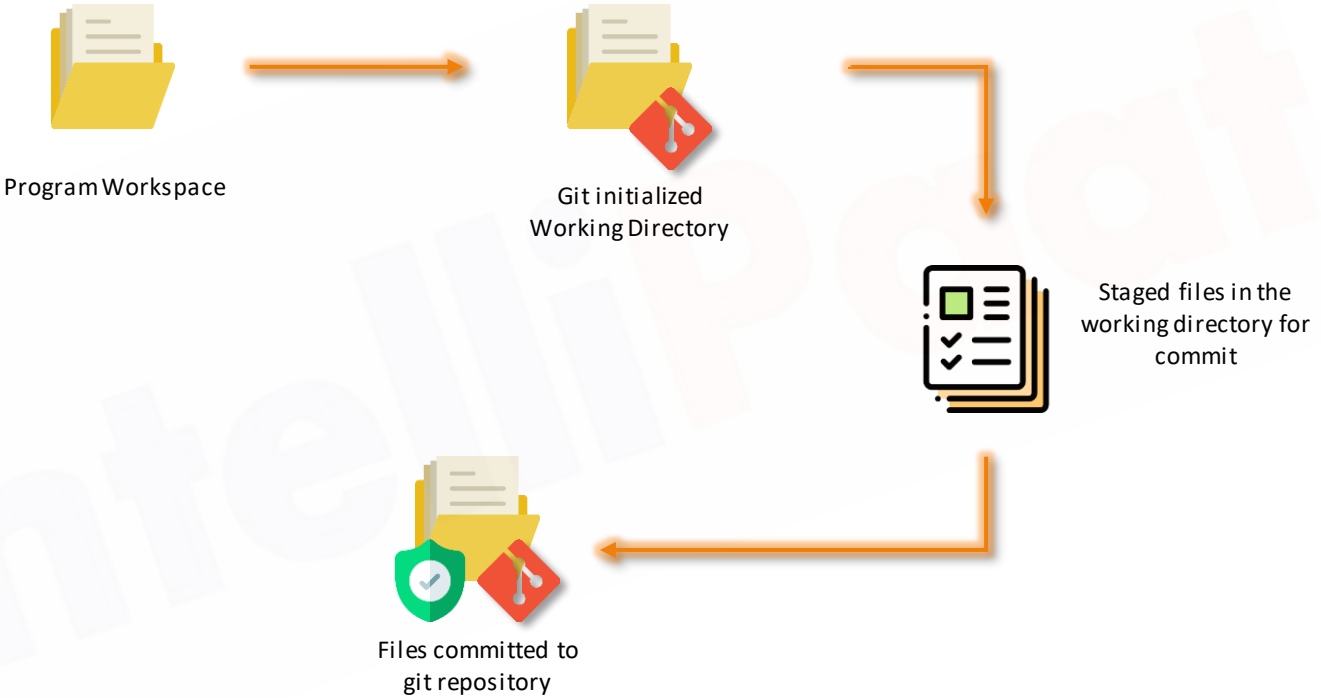
Working Directory

Staging Area

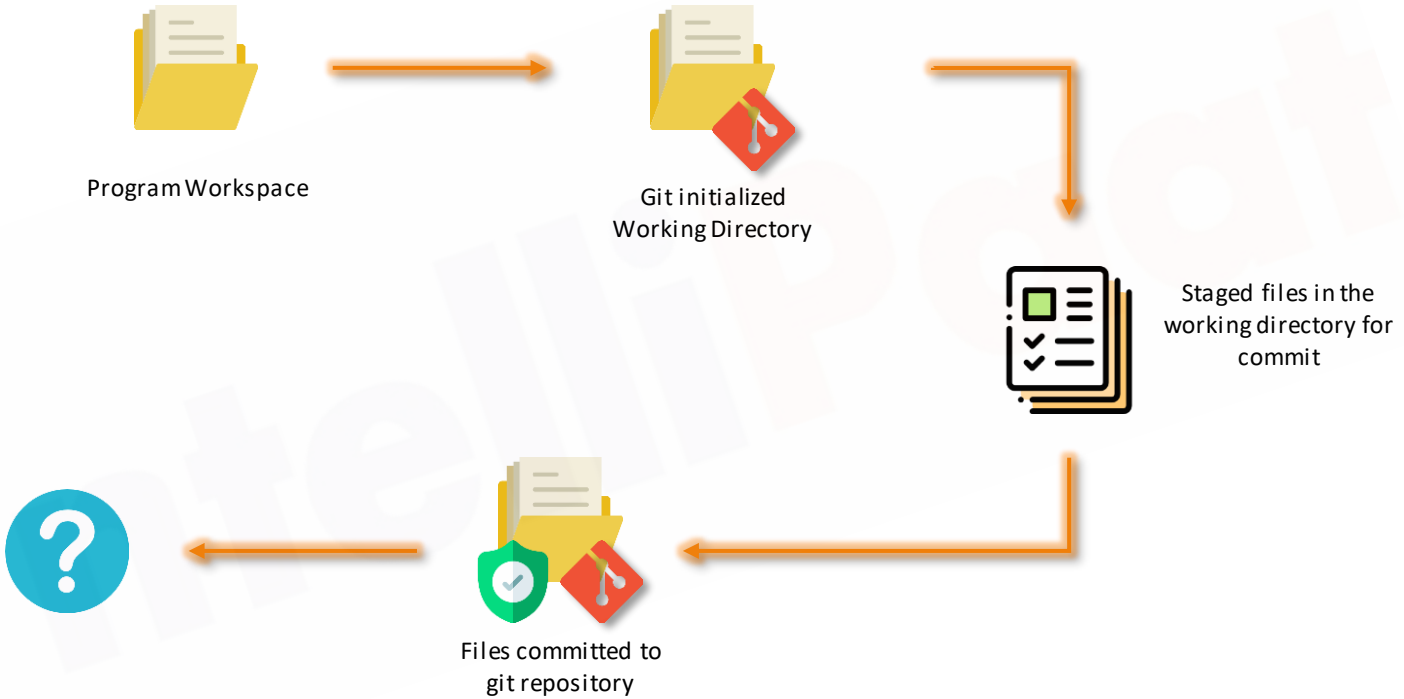
Commit

- Once the files are selected and are ready in the staging area, they can now be saved in repository
- Saving a file in the repository of git is known as doing a commit
- When we commit a repository in git, the commit is identified by a commit id
- The command for initializing this process is *git commit –m "message"*

Git Lifecycle

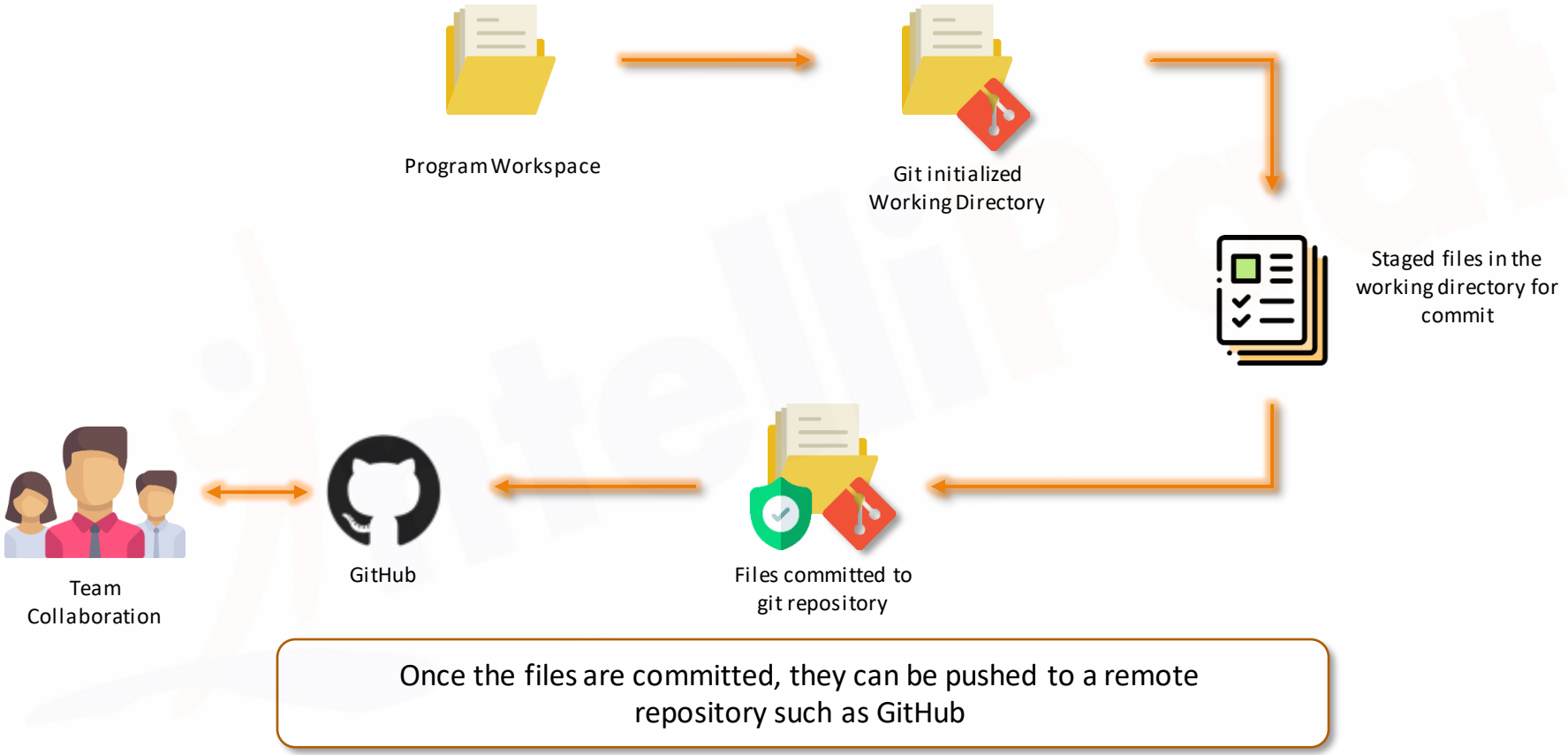


Git Lifecycle



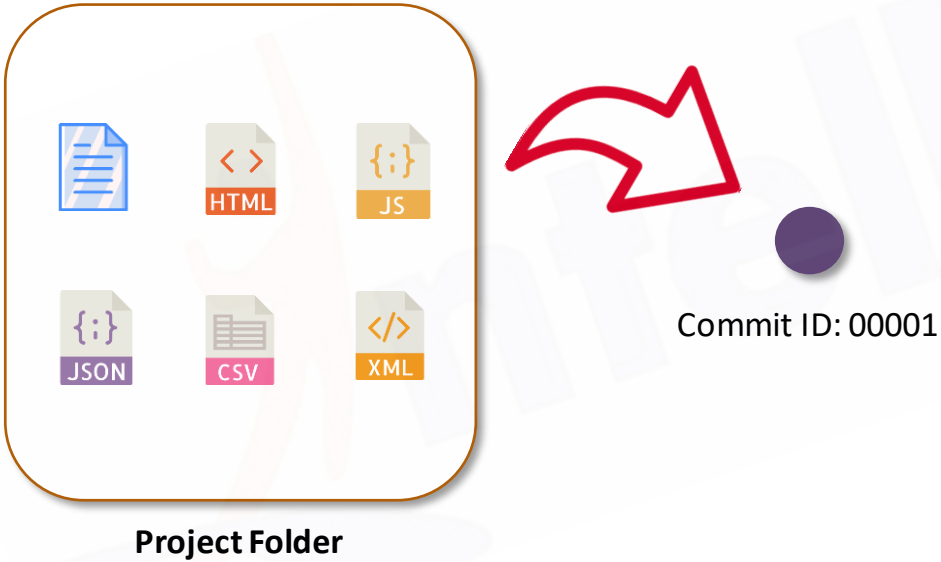
How do we collaborate with the team?

Git Lifecycle



How does Git work?

Any project which is saved on git, is saved using a commit. The commit is identified using a commit ID.



How does Git work?

When we edit the project or add any new functionality, the new code is again committed to git, a new commit ID is assigned to this modified project. The older code is stored by git, and will be accessible by it's assigned Commit ID

Project Folder

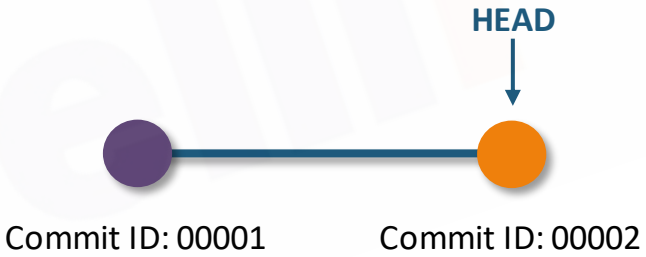
Commit ID: 00002

Commit ID: 00001

How does Git work?

All these commits are bound to a **branch.** Any new commits made will be added to this branch. A branch always points to the latest commit. The pointer to the latest commit is known as **HEAD**

Project Folder



How does Git work?

The default branch in a git repository is called the Master Branch

Project Folder

How does Git work?

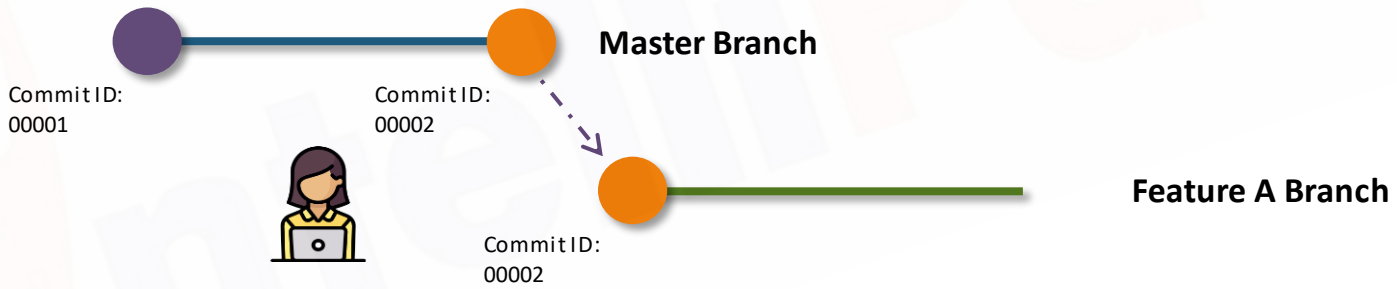
The default branch in a git repository is called the Master Branch

Project Folder

But, why do we need a branch?

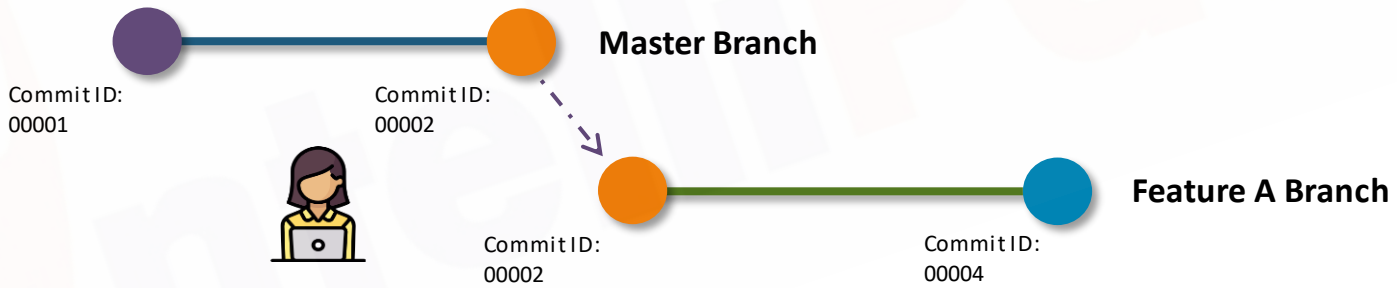
How does Git work?

Say, a developer has been assigned enhance this code by adding Feature A. The code is assigned to this developer in a separate branch “Feature A”. This is done, so that master contains only the code which is finished, finalized and is on production



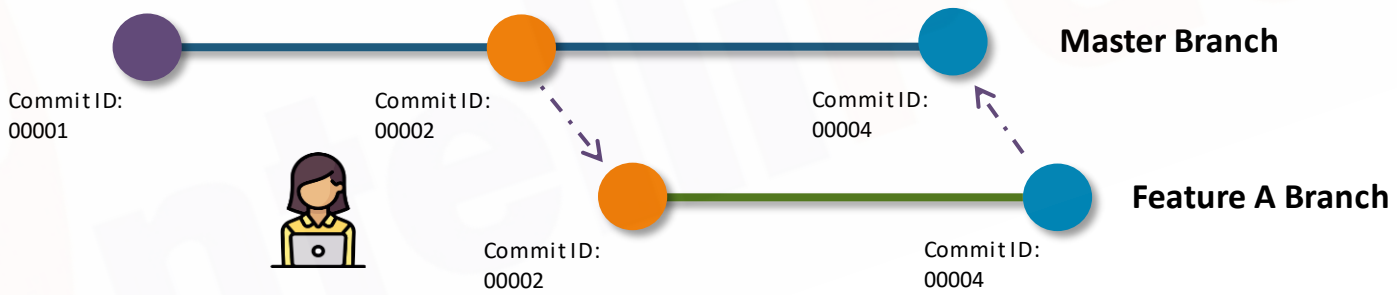
How does Git work?

Therefore, no matter how many commits are made by this developer on Feature A branch, it will not affect the Master Branch.



How does Git work?

Once the code is finished, tested and ready we can merge the Feature A branch, with the master branch and now the code is available on the production servers as well



Common Git Commands

Common Git Commands

You can do the following tasks, when working with git. Let us explore the commands related to each of these tasks

Creating Repository

Making Changes

Parallel Development

Syncing Repositories

Common Git Commands – git init

- Creating Repository
- Making Changes
- Syncing Repositories
- Parallel Development

You can create a repository using the command git init. Navigate to your project folder and enter the command git init to initialize a git repository for your project on the local system

```
ubuntu@ip-172-31-33-5:~/project$ ls
1.txt  2.txt
ubuntu@ip-172-31-33-5:~/project$ git init
Initialized empty Git repository in /home/ubuntu/project/.git/
ubuntu@ip-172-31-33-5:~/project$
```

Common Git Commands – git status


Creating Repository


Making Changes


Syncing Repositories


Parallel Development

Once the directory has been initialized you can check the status of the files, whether they are being tracked by git or not, using the command **git status**

```
ubuntu@ip-172-31-33-5:~/project$ ls
1.txt  2.txt
ubuntu@ip-172-31-33-5:~/project$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        1.txt
        2.txt

nothing added to commit but untracked files present (use "git add" to track)
ubuntu@ip-172-31-33-5:~/project$
```

Common Git Commands – git add

- Creating Repository
- Making Changes
- Syncing Repositories
- Parallel Development

Since no files are being tracked right now, let us now stage these files. For that, enter the command **git add.** If we want to track all the files in the project folder, we can type the command,

git add .

```
ubuntu@ip-172-31-33-5:~/project$ ls
1.txt  2.txt
ubuntu@ip-172-31-33-5:~/project$ git add .
ubuntu@ip-172-31-33-5:~/project$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   1.txt
        new file:   2.txt

ubuntu@ip-172-31-33-5:~/project$
```

Common Git Commands – git commit


Creating Repository


Making Changes


Syncing Repositories


Parallel Development

Once the files or changes have been staged, we are ready to commit them in our repository. We can commit the files using the command **git commit –m “custom message”**

```
ubuntu@ip-172-31-33-5:~/project$ ls
1.txt  2.txt
ubuntu@ip-172-31-33-5:~/project$ git commit -m "First Commit"
 2 files changed, 2 insertions(+)
 create mode 100644 1.txt
 create mode 100644 2.txt
```

Common Git Commands – git remote


Creating Repository


Making Changes


Syncing Repositories


Parallel Development

Once everything is ready on our local, we can start pushing our changes to the remote repository. Copy your repository link and paste it in the command

git remote add origin “<URL to repository>”

```
ubuntu@ip-172-31-33-5:~/project$ git remote add origin "https://github.com/devop
s-intellipaat/devops.git"
ubuntu@ip-172-31-33-5:~/project$
```

Common Git Commands – git push


Creating Repository


Making Changes


Syncing Repositories


Parallel Development

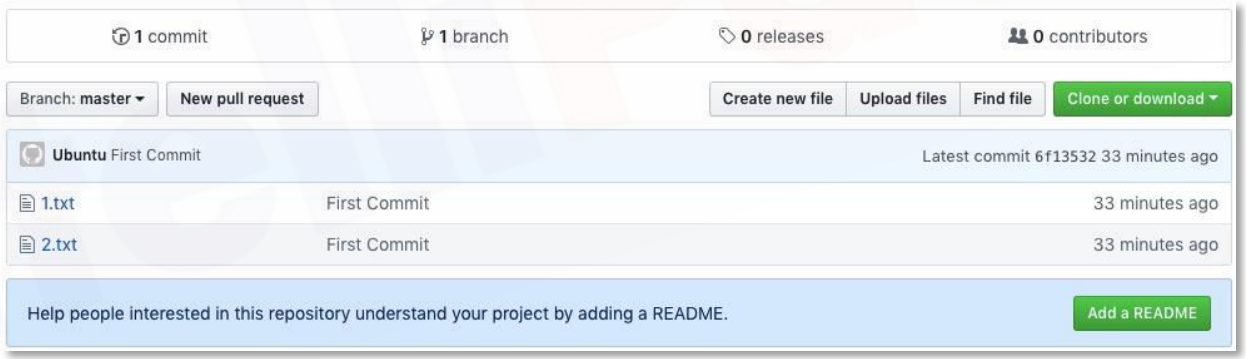
To push the changes to your repository, enter the command
git push origin <branch-name> and hit enter. In our case the branch is master, hence
git push origin master
This command will then prompt for username and password, enter the values and hit enter.

```
ubuntu@ip-172-31-33-5:~/project$ git push origin master
Username for 'https://github.com': devops-intellipaat
Password for 'https://devops-intellipaat@github.com':
Counting objects: 4, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 292 bytes | 292.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/devops-intellipaat/devops/pull/new/master
remote:
To https://github.com/devops-intellipaat/devops.git
 * [new branch]      master -> master
ubuntu@ip-172-31-33-5:~/project$
```

Common Git Commands – git push

- Creating Repository
- Making Changes
- Syncing Repositories
- Parallel Development

Your local repository is now synced with the remote repository on github



Common Git Commands – git clone


Creating Repository


Making Changes


Syncing Repositories


Parallel Development

Similarly, if we want to download the remote repository to our local system, we can use the command:

git clone <URL>

This command will create a folder with the repository name, and download all the contents of the repository inside this folder. In our example, repository contents were downloaded into the ”devops” folder.

```
ubuntu@ip-172-31-33-5:~$ git clone https://github.com/devops-intellipaat/devops.
git
Cloning into 'devops'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 0), reused 4 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
ubuntu@ip-172-31-33-5:~$ ls
devops  project
ubuntu@ip-172-31-33-5:~$
```

Common Git Commands – git pull

Creating Repository

Making Changes

Syncing Repositories

Parallel Development

The git pull command is also used for pulling the latest changes from the repository, unlike git clone, this command can only work inside an initialized git repository. This command is used when you are already working in the cloned repository, and want to pull the latest changes, that others might have pushed to the remote repository

git pull <URL of link>

```
ubuntu@ip-172-31-33-5:~/devops$ git pull https://github.com/devops-intellipaat/d
evops.git
From https://github.com/devops-intellipaat/devops
 * branch            HEAD       -> FETCH_HEAD
Already up to date.
ubuntu@ip-172-31-33-5:~/devops$
```

Common Git Commands – git branch

- Creating Repository
- Making Changes
- Syncing Repositories
- Parallel Development

Until now, we saw how you can work on git. But now imagine, multiple developers working on the same project or repository. To handle the workspace of multiple developers, we use branches. To create a branch from an existing branch, we type

git branch <name-of-new-branch>

Similarly, to delete a branch use the command

git branch –D <branch name>

```
ubuntu@ip-172-31-33-5:~$ cd devops
ubuntu@ip-172-31-33-5:~/devops$ git branch branch1
ubuntu@ip-172-31-33-5:~/devops$
```

Common Git Commands – git checkout

- Creating Repository
- Making Changes
- Syncing Repositories
- Parallel Development

To switch to the new branch, we type the command

git checkout <branch-name>

```
ubuntu@ip-172-31-33-5:~/devops$ git checkout branch1
Switched to branch 'branch1'
ubuntu@ip-172-31-33-5:~/devops$ ls
1.txt  2.txt
ubuntu@ip-172-31-33-5:~/devops$
```

Common Git Commands – git log

Want to check the log for every commit detail in your repository?
You can accomplish that using the command

git log

```
ubuntu@ip-172-31-33-5:~/devops$ git log
commit dd6974eda23d7644d9cb724a82ebd829c7717ac6 (HEAD -> branch1, master)
Author: Ubuntu <ubuntu@ip-172-31-33-5.us-east-2.compute.internal>
Date:   Fri Nov 23 06:21:41 2018 +0000

    adding test file

commit 6f135327baf101788b23e3053a75d828709f6bb7 (origin/master, origin/HEAD)
Author: Ubuntu <ubuntu@ip-172-31-33-5.us-east-2.compute.internal>
Date:   Fri Nov 23 05:00:03 2018 +0000

    First Commit
ubuntu@ip-172-31-33-5:~/devops$
```

Common Git Commands – git stash

- Creating Repository
- Making Changes
- Syncing Repositories
- Parallel Development

Want to save your work without committing the code? Git has got you covered. This can be helpful when you want to switch branches, but do not want to save your work to your git repository. To stash your staged files without committing just type in **git stash.** If you want to stash your untracked files as well, type **git stash –u**.

Once you are back and want to retrieve working, type in **git stash pop**

```
ubuntu@ip-172-31-33-5:~/devops$ ls
1.txt  2.txt  3.txt  4.txt
ubuntu@ip-172-31-33-5:~/devops$ git stash -u
Saved working directory and index state WIP on master: dd6974e adding test file
ubuntu@ip-172-31-33-5:~/devops$ ls
1.txt  2.txt  3.txt
ubuntu@ip-172-31-33-5:~/devops$ git stash pop
Already up to date!
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        4.txt

nothing added to commit but untracked files present (use "git add" to track)
Dropped refs/stash@{0} (7f106523effac55075b2d03387245c487a3de84f)
ubuntu@ip-172-31-33-5:~/devops$ ls
1.txt  2.txt  3.txt  4.txt
ubuntu@ip-172-31-33-5:~/devops$
```

Common Git Commands – git revert

This command helps you in reverting a commit, to a previous version

git revert <commit-id>

<commit-id> can be obtained from the output of **git log**

```
ubuntu@ip-172-31-33-5:~/devops$ git revert dd6974eda23d7644d9cb724a82ebd829c7717
ac6
[branch1 88c0d66] Revert "adding test file"
 Committer: Ubuntu <ubuntu@ip-172-31-33-5.us-east-2.compute.internal>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 1 deletion(-)
 delete mode 100644 3.txt
```

Common Git Commands – git diff

This command helps us in checking the differences between two versions of a file

git diff <commit-id of version x> <commit-id of version y>

<commit-id> can be obtained from the output of **git log**

```
ubuntu@ip-172-31-23-227:~/devopsIQ/devopsIQ$ git diff 4bdbc8b0d037553729e2e75e75
48bc84dcf19564 55d4c573efcd1f1ab70c2f926cb41f4c61d29d20
diff --git a/devopsIQ/index.html b/devopsIQ/index.html
index 87f0103..e4404e7 100644
--- a/devopsIQ/index.html
+++ b/devopsIQ/index.html
@@ -1,5 +1,5 @@
 <html>
-<title>Jenkins Final Website2</title>
+<title>Jenkins Final Website</title>^M
 <body background="images/1.jpg">
 </body>
 </html>
```

Merging Branches

Merging Branches

Once the developer has finished his code/feature on his branch, the code will have to be combined with the master branch. This can be done using two ways:

Git Merge

Git Rebase

Merging Branches – git merge

Git Merge

Git Rebase

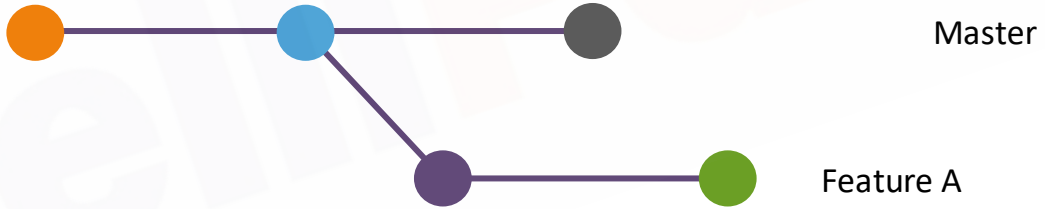
- If you want to apply changes from one branch to another branch, one can use merge command
- Should be used on remote branches, since history does not change
- Creates a new commit, which is a merger of the two branches
- Syntax: git merge <source-branch>

Merging Branches – git merge

Git Merge

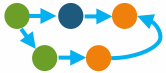
Git Rebase

Imagine, you have a Master branch and a Feature A branch. The developer has finished his/her work in the feature A branch and wants to merge his work in the master.



Merging Branches – git merge

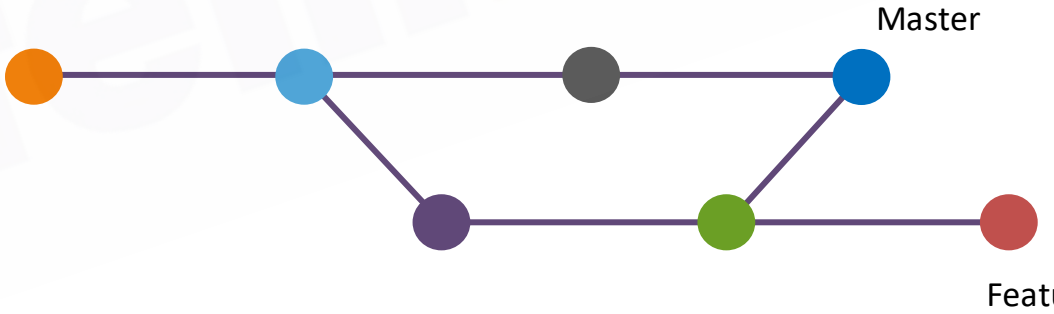
Git Merge



Git Rebase

If he is using **git merge**, a new commit will be created, which will have the changes of Feature A and Master branch combined.

Any new commits to the Feature branch will be isolated from the master branch



Merging Branches – git merge

Git Merge

Git Rebase

This command can be executed using the syntax

git merge <source-branch-name>

```
ubuntu@ip-172-31-33-5:~/devops$ ls
1.txt  2.txt  3.txt
ubuntu@ip-172-31-33-5:~/devops$ git status
On branch branch1
nothing to commit, working tree clean
ubuntu@ip-172-31-33-5:~/devops$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
ubuntu@ip-172-31-33-5:~/devops$ ls
1.txt  2.txt
ubuntu@ip-172-31-33-5:~/devops$ git merge branch1
Updating 6f13532..dd6974e
Fast-forward
 3.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 3.txt
ubuntu@ip-172-31-33-5:~/devops$ ls
1.txt  2.txt  3.txt
ubuntu@ip-172-31-33-5:~/devops$
```

Merging Branches – git merge

Git Merge

Git Rebase

The history of the branch will look something like this, if we are using **git merge**

```
ubuntu@ip-172-31-26-120:~/n$ git log --graph --pretty=oneline
*   d92f22eeb6bb7fefd1706b397abe804dc557ec88 (HEAD -> master) Merge branch
'
|\
| * aebc77927892bd1c74ffd9b3d9af7f3b763ee8da (test) 1st on test
* | b62c11b6a12e4c0431bf4ae7f9fe90f744d485b7 second on master
|/
* 071f9bd946e502d4643d2fc7e2dd7c26dea0eaf9 first commit in master
```

Merging Branches – git merge

Git Merge

Git Rebase

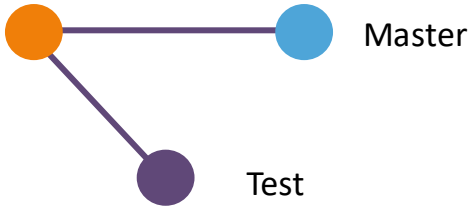
- This is an alternative to git merge command
- Should be used on local branches, since history does change and will be confusing for other team members
- Does not create any new commit, and results in a cleaner history
- The history is based on common commit of the two branches (base)
- The destination's branch commit is pulled from it's "base" and "rebased" on to the latest commit on the source branch

Merging Branches – git rebase

Git Merge

Git Rebase

- Imagine, you have a Master branch and a test branch(local branch)
- The developer has finished his/her work in the test branch
- But the master moved forward, while the code was being developed
- Code being developed is related to the new commit added in master



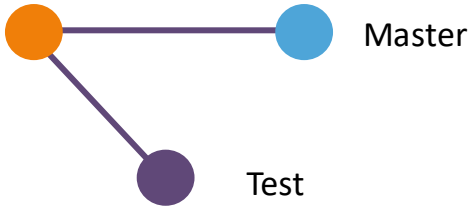
Merging Branches – git rebase

Git Merge

Git Rebase

- Therefore you want all the changes from master in feature.
- Since, it is a local branch, you would want a cleaner or linear history, you decide to use git rebase

Syntax: **git rebase <source branch>**



Merging Branches – git rebase

Git Merge

Git Rebase

This is how the output looks like:

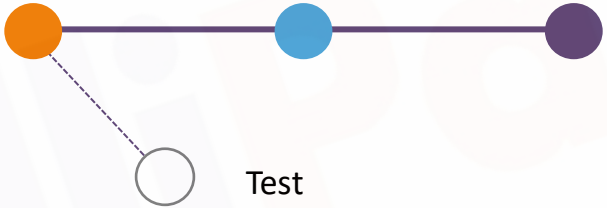
```
ubuntu@ip-172-31-26-120:~/n$ git checkout test
Switched to branch 'test'
ubuntu@ip-172-31-26-120:~/n$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: 1st in test
```

Merging Branches – git rebase

Git Merge

Git Rebase

This is how the commits look like, after a rebase. The commit was “rebased” from the first commit to the next commit



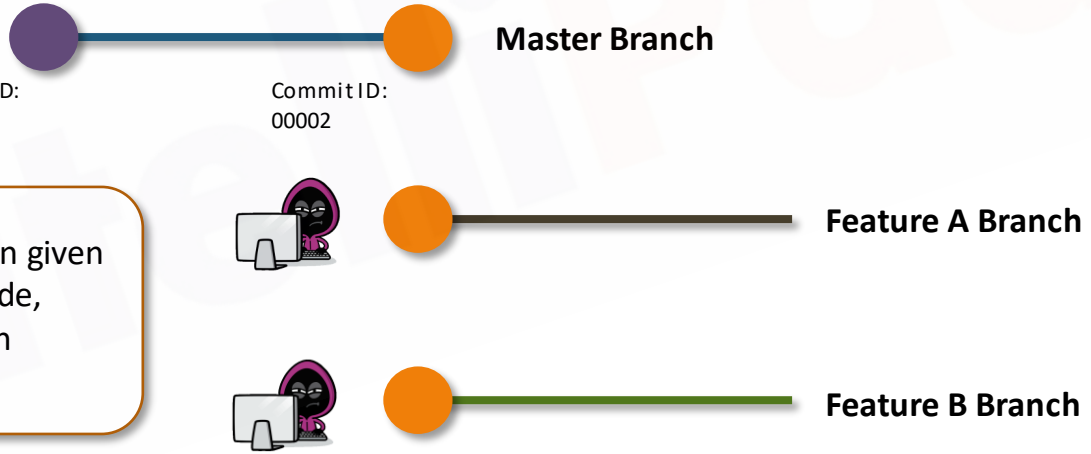
And looking at the history we can clearly see, it’s a clean linear history, without any branches

```
ubuntu@ip-172-31-26-120:~/n$ git log --graph --pretty=oneline
* 3885b20a7f8880acf4b7a785a638e95d1759dcf2 (HEAD -> test) 1st in test
* cce38fa142699171d08b08b27ed44f49052ac134 (master) 2nd in master
* 7d77f726ad1d0b64f6f20c2587560dc18123082d 1st in master
```

Merge Conflicts

Merge Conflicts

Merge conflicts occur when we try to merge two branches, which have the same file updated by two different developers. Let's understand it using a scenario:



Imagine, two developers have been given the task of adding features to a code, they both have to add their code in ***functions.c*** file

Merge Conflicts

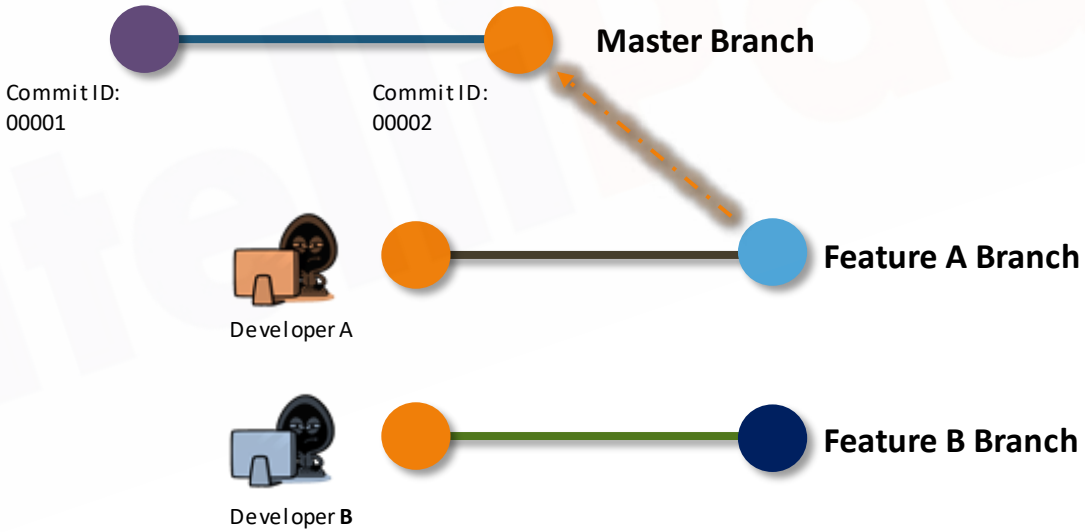
The functions.c file looks something like this as of now,

```
Main()
{
Function1()
 {
    //Initial Code
 }

}
```

function.c

Merge Conflicts

Developer A finished his code, and pushes the changes to the master branch



Merge Conflicts

```
Main()
{
Function1()
 {
    //Initial Code
 }
  Function2()
   {
      //Developer A Code
    }

}
```

function.c

After the **Developer A** changes his code and pushes it to master, the code on the **Master** branch looks something like this

```
Main()
{
Function1()
 {
    //Initial Code
 }
Function3()
   {
      //Developer B Code
   }

}
```

function.c

After the **Developer B** changes his code, the code on **Feature B** branch looks something like this

Merge Conflicts

Comparing the two code, we can see Feature A Branch is missing Developer A code. Therefore if we merge Feature A Branch with Master Branch, logically Developer A changes will disappear

Master Branch

```
Main()
{
Function1()
 {
    //Initial Code
 }
  Function2()
   {
      //Developer A Code
    }

}
```

function.c

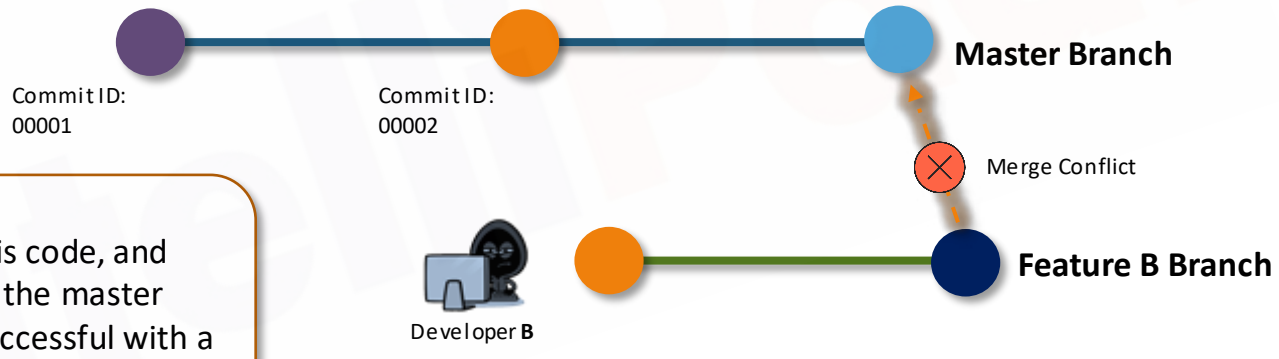
Feature A Branch

```
Main()
{
Function1()
 {
    //Initial Code
 }
Function3()
   {
      //Developer B Code
   }

}
```

function.c

Merge Conflicts

To solve this, git has a fail safe. If the Master branch has been moved forward in commits, compared to the branch which is being merged, it creates a conflict.



Developer B finished his code, and pushes the changes to the master branch. The merge is unsuccessful with a conflict message

Hands-on – Simulating a Merge Conflict

Merge Conflicts

This is the message, you will get when you merge a branch, which has a conflicting file

```
ubuntu@ip-172-31-26-120:~/dev1/devops$ git merge dev2
Auto-merging feature.c
CONFLICT (content): Merge conflict in feature.c
Automatic merge failed; fix conflicts and then commit the result.
```

Merge Conflict Message

How to resolve Merge Conflicts?

How to resolve Merge Conflicts?

Once we have identified, there is a merge conflict we should go ahead and use the command

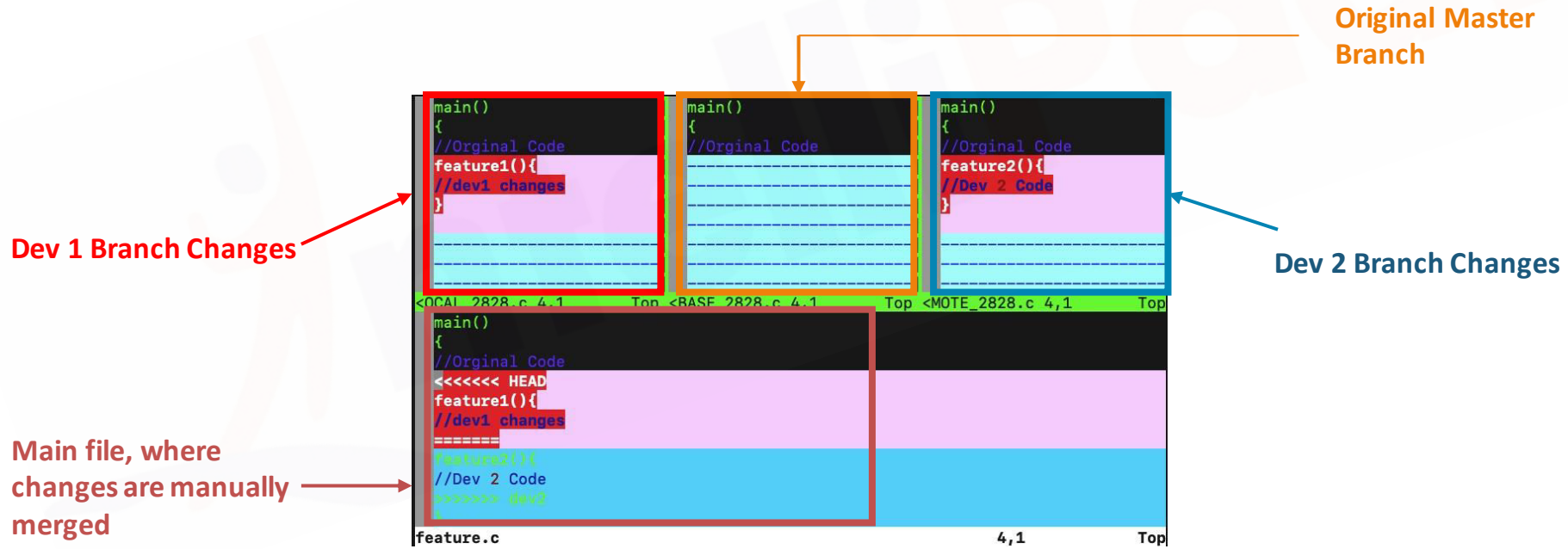
git mergetool

```
ubuntu@ip-172-31-26-120:~/dev1/devops$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
tortoisemerge emerge vimdiff
Merging:
feature.c

Normal merge conflict for 'feature.c':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (vimdiff):
```

Hit enter after this prompt, and then you should enter the merge tool

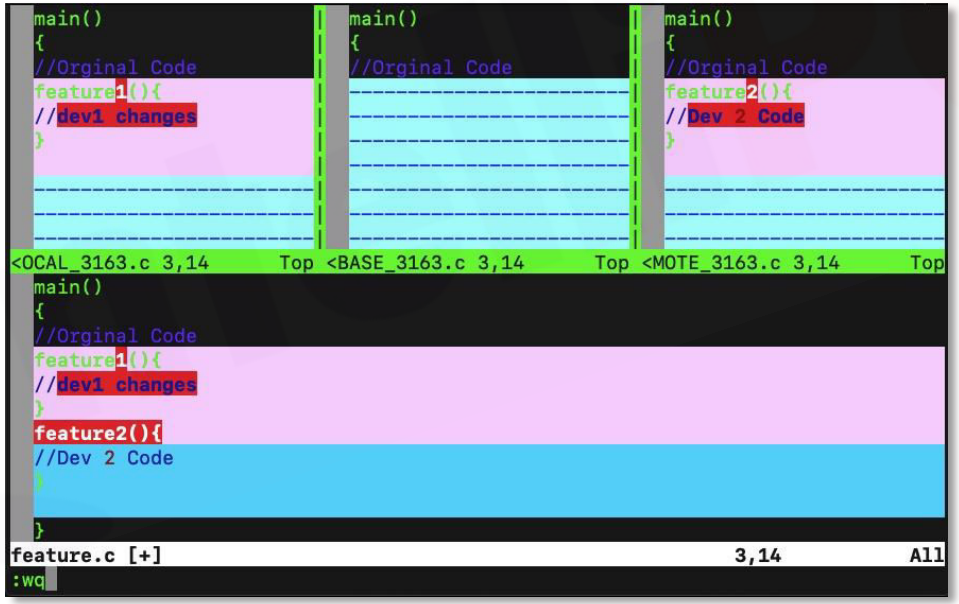
How to resolve Merge Conflicts?

The merge tool looks something like this, the top leftmost column is for Dev1 Branch changes, the centre column is for the original code i.e the master's code before any commits, and the right most column are the dev 2 branch changes. The area below these columns is where we make the changes, this is the place where we have the merged code.



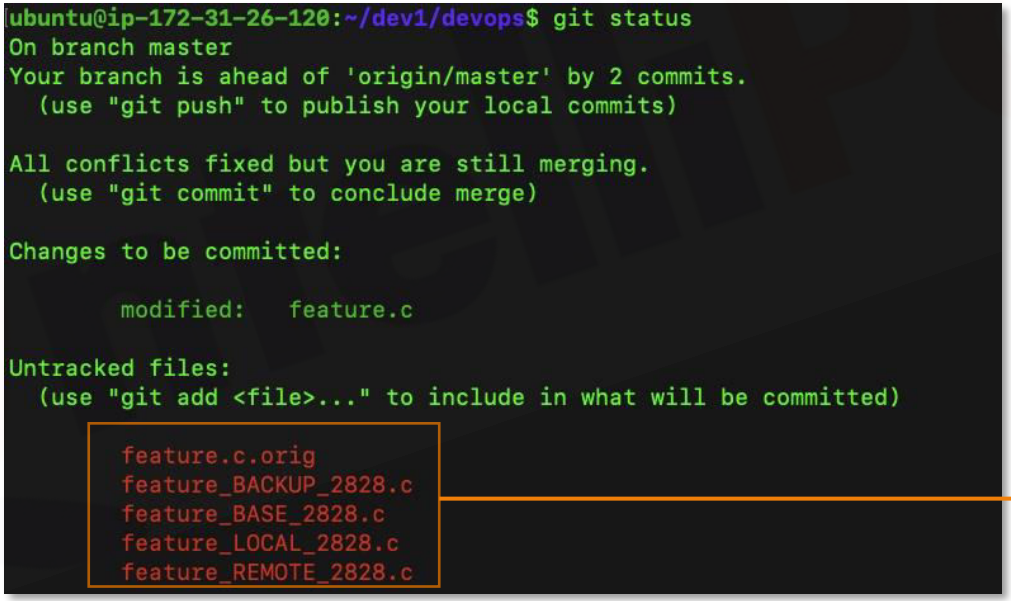
How to resolve Merge Conflicts?

Once you have resolved the changes, save the file using "**:wq**", vim command for save and exit. Do the same for all the files



How to resolve Merge Conflicts?

After this step, see the status of you local repository you can see the file in conflict has been modified successfully and is merged with master. This modified file can now be committed to the master



There will be some other files which have been created, these files are a copy of the original files which have been changed, you can delete them, if not needed.

How to resolve Merge Conflicts?

Finally commit your changes, to the branch and then push it to the remote repository

```
ubuntu@ip-172-31-26-120:~/dev1/devops$ git commit -m "merged feature"
[master f0ecdbd] merged feature
 Committer: Ubuntu <ubuntu@ip-172-31-26-120.us-east-2.compute.internal>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 1 insertion(+)
 create mode 100644 feature.c
```

Collaboration in GitHub

Collaboration in GitHub

Collaboration in GitHub is very important aspect of Software Development. It enables developers to parallelly work on the same project

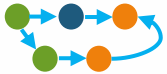
Process of Collaboration in GitHub

1. Add collaborators to your repository

2. Protect the branches, which need restricted access

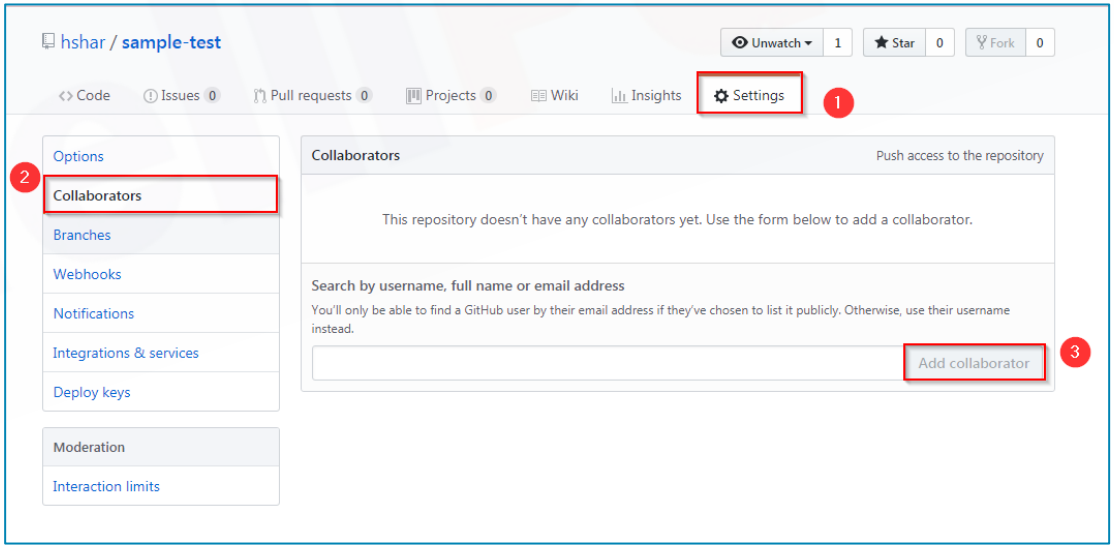
Process of Collaboration in GitHub

Collaborators



Protecting Branches

Collaborators are the people who will be working on your project. To add contributors to your repository, use the steps in the following image:



Process of Collaboration in GitHub

Collaborators

Protecting Branches

Using this feature, you can restrict access on how commits are made to the branches you specify

Hands-on: Collaborating in GitHub

Hands-on

1. Add a Collaborator in your GitHub repository, by creating an alternate account
2. Protect the Master branch, from getting changes directly pushed on it
3. Push changes to the repository, in a feature branch
4. Create a Pull Request from the feature branch to the Master Branch
5. From the owner's account approve the pull request

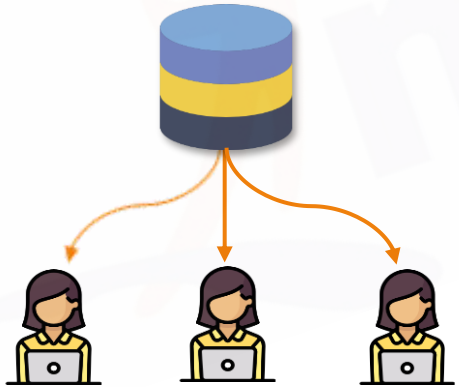
Git Workflow

Git Workflow

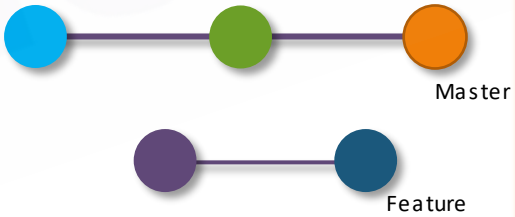
A Git Workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner. Git workflows encourage users to leverage Git effectively and consistently.

There are the three popular workflows which are accepted and are followed by various tech companies:

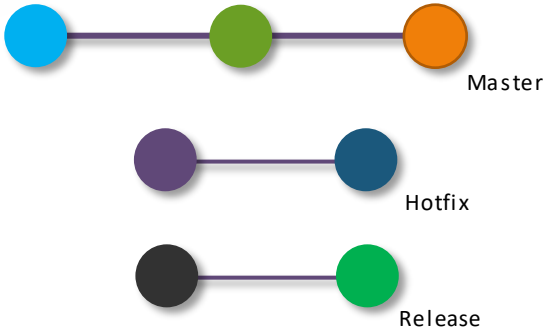
Centralized Workflow



Feature Branching



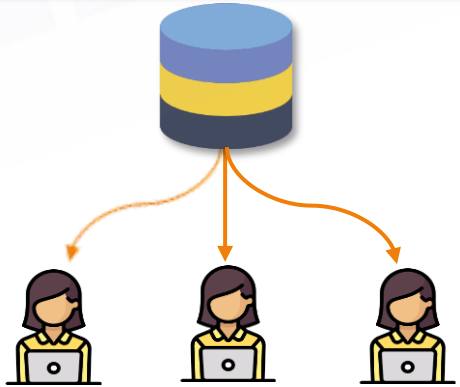
GitFlow Workflow



Git Workflow

- Centralized Workflow
- Feature Branching
- GitFlow Workflow

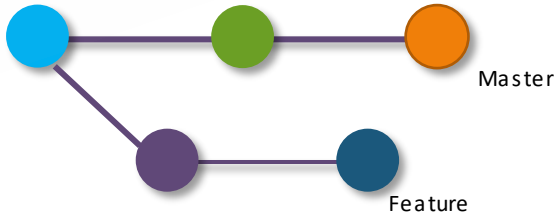
- This workflow doesn't require any other branch other than master
- All the changes are directly made on the master, and finally merged on the remote master, once work is finished
- Before pushing changes, the master is rebased with the remote commits
- Results in a clean, linear history



Git Workflow

- Centralized Workflow
- Feature Branching
- GitFlow Workflow

- Master only contains the production ready code
- Any Development work, is converted into a feature branch
- There can be numerous feature branches, depending on the application's development plan
- Once the feature is complete, the feature branch is merged with the master



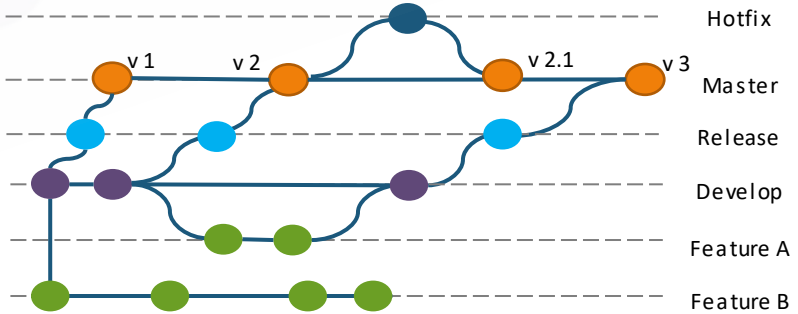
Git Workflow

Centralized Workflow

Feature Branching

GitFlow Workflow

- The Feature branches are never merged directly with master
- Once the features are ready, commit is merged with Develop
- When there are enough commits on Develop, we merge the Develop branch with Release branch, only read me files or License files are added after this commit on Release
- Any quick fixes which are required, are done on the Hotfix branch, this branch can directly be merged with Master



Forking in GitHub

What is Forking?

A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. Most commonly, forks are used to either suggest changes to someone else's project or to use someone else's project as a starting point for your own idea.

Forking in GitHub

- Forking is a GitHub concept, it has nothing to do with Git software
- It is used to copy someone else's repository to your own repo in GitHub
- The changes made to a forked repository are not reflected in the parent repository
- If one wants to suggest any change to the parent repository from the forked repository

Quiz

Quiz

1. Which of the following is a characteristics of a Centralized Version Control System?

A. Local Operations are fast

B. Version History is available on the local storage as well

C. Version History is not available on the local storage as well

D. None of these

Quiz

1. Which of the following is a characteristics of a Centralized Version Control System?

A. Local Operations are fast

B. Version History is available on the local storage as well

C. Version History is not available on the local storage as well

D. None of these

Quiz

2. Which of the following should git rebase be used on?

A. Master Branch

B. Remote Branch

C. Local Branch

D. None of these

Quiz

2. Which of the following should git rebase be used on?

A. Master Branch

B. Remote Branch

C. Local Branch

D. None of these

Quiz

3. Which of the following workflow should be used, if we want to deploy multiple features at once?

A. Centralized Workflow

B. Feature Workflow

C. GitFlow Workflow

D. None of these

Quiz

3. Which of the following workflow should be used, if we want to deploy multiple features at once?

A. Centralized Workflow

B. Feature Workflow

C. GitFlow Workflow

D. None of these

Quiz

4. Forking helps us in _____

A. Cloning the Repository

B. Copying a Repository to your GitHub account

C. Suggesting Changes to the present repository

D. None of these

Quiz

4. Forking helps us in _____

A. Cloning the Repository

B. Copying a Repository to your GitHub account

C. Suggesting Changes to the present repository

D. None of these

Quiz

5. Merge Conflicts occur when,

A. Two developers simultaneously commit to a repository

B. Two or more developers try to merge on local system

C. Two or more developers try to merge to the remote repository

D. Two developers who worked on the same file, try to merge on the branch

Quiz

5. Merge Conflicts occur when,

A. Two developers simultaneously commit to a repository

B. Two or more developers try to merge on local system

C. Two or more developers try to merge to the remote repository

D. Two developers who worked on the same file, try to merge on the branch

Thank You