Git and version control

Hector de la Torre Perez
Northern Illinois University

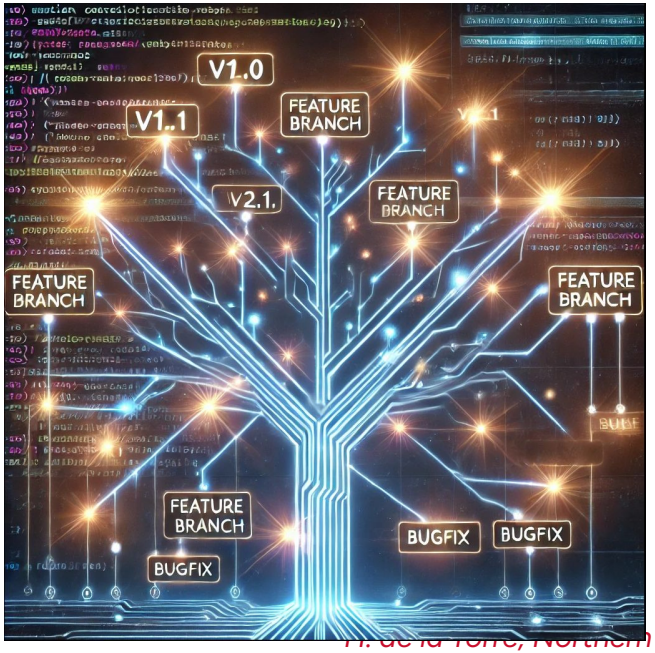
Northern Illinois University

What are we talking about?

Some (Or all?) of you have already used (sometimes knowingly, sometimes unknowingly) version control software or features in the past. I'm going to aim at telling you some things about it that you may not know and then provide some tip and tricks on the most well known version control software (git).

But what is version control?

Version control is the practice of **controlling**, **organizing**, and **tracking** different versions in history of computer files; primarily source code text files, but generally any type of file.



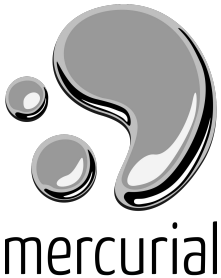
Why do we want version control

The main idea behind version control is that when developing code is extremely useful to keep a history of the process (the changes in the code). A system like that has plenty of advantages:

- You can go back and check what the code looked like at some point in the past
- You can recover earlier versions if necessary
- You can test things easily without fear of screwing things up
- You can keep multiple versions
- If sharing of the earlier versions is done in a smart and organized way, it allows for efficient collaboration between multiple developers
    - One could argue most of the 'improvements' in version control comes from the globalization and increased complexity of development projects

What is and what isn't version control sw

These are code (or general) repositories. They may have internally some version of version control (that sometimes use software control software or dedicated code) but they are not version control sw

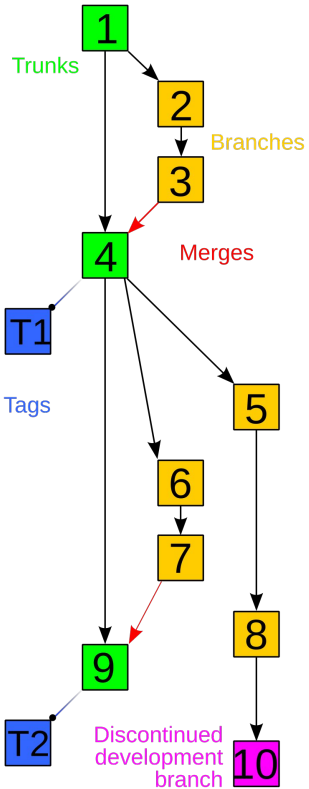


Piper

Same deal with IDEs. They 'can' have version control, but they are not version control sw

Basic elements of version control

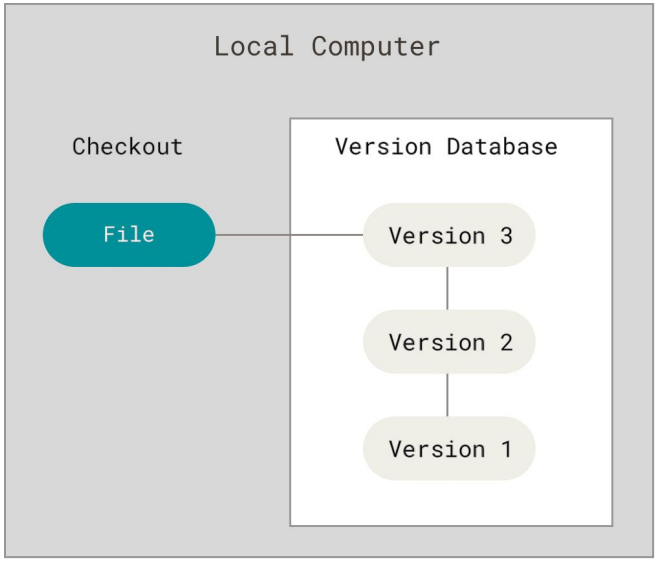
Many depend on the specific software, but some of them are fairly general



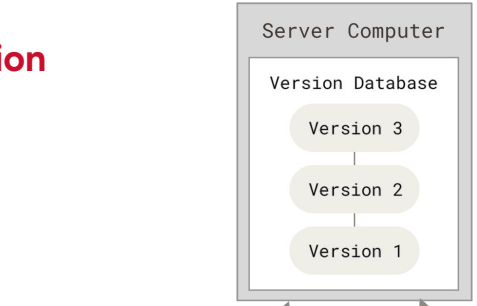
- **Commit:** Also known as changeset it's a single set of changes to the code (together with associated metadata) that defines a particular instance, or revision, of the code. It's the unit in version control
- **Branch:** Different paths of changes that can develop with it's separate sets of commits
- **Head:** The latest commit for a specific Branch
- **Merges:** The combination of changes from different branches into a common path
- **Trunk:** The central line of development that is not a Branch
- **Tag:** A label that defines a specific snapshot of code
- **Repository:** The data structure that contains the data and metadata of the version control system itself. For example all previous versions or differences with respect to the current version
- **Working copy:** It's the local copy of a specific version of the code from the repository at a specific point in time

We'll talk about how this looks in git explicitly in a bit, but these things go beyond specific git implementation

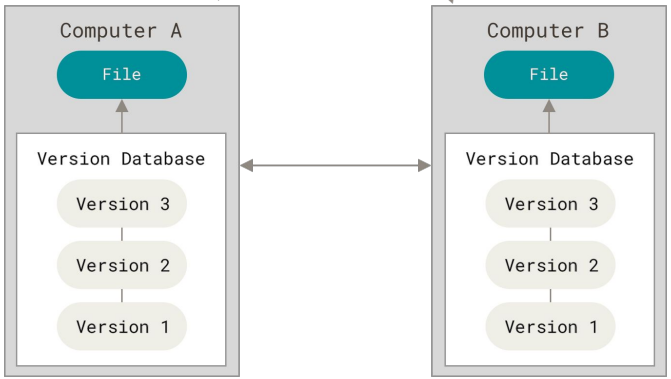
Types of version control



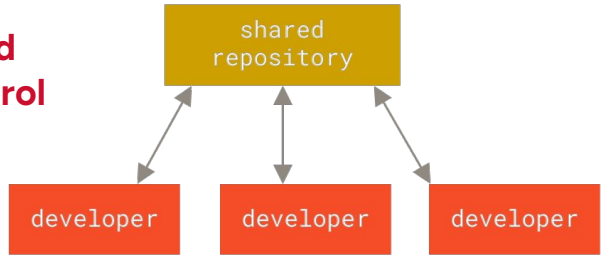
Local version control



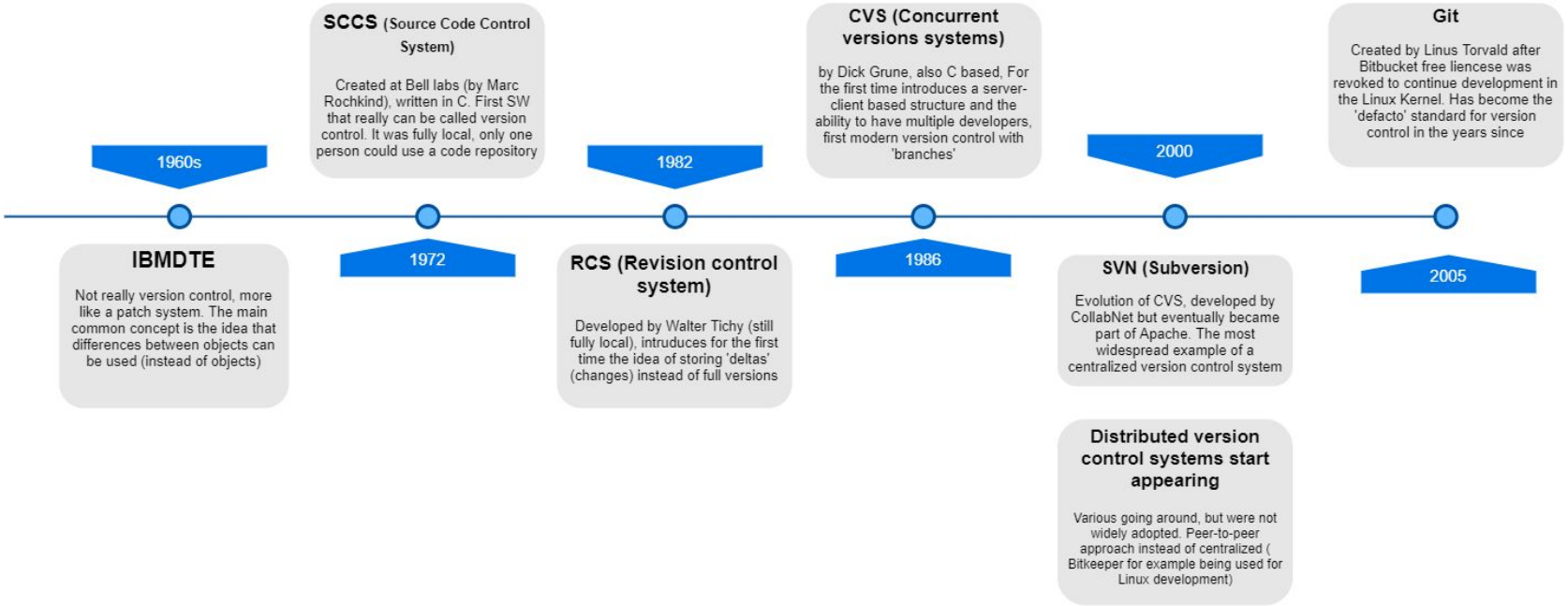
Distributed version control



Centralized version control



Brief history of version control



And thus enter git

The most widely version control software nowadays. About 93% of market share (Already 70% in 2015) with SVN having most of the rest. Fully free, fully open source. Legally supported by the software freedom conservancy

Developed by Linus Torvald to support the development of the Linux Kernel when Bitbucket decided to stop providing free licenses to open source projects

Four main guiding principles:

1. Patching should take less than 3 seconds for the linux repository
2. Support a distributed system with multiple developers
3. Do exactly the opposite that CVS did
4. Include safeguards against data corruption

Everyone in HEP uses git. Mostly everyone in industry uses git (google internal codebase one of the only exceptions)

Avoiding common misconceptions

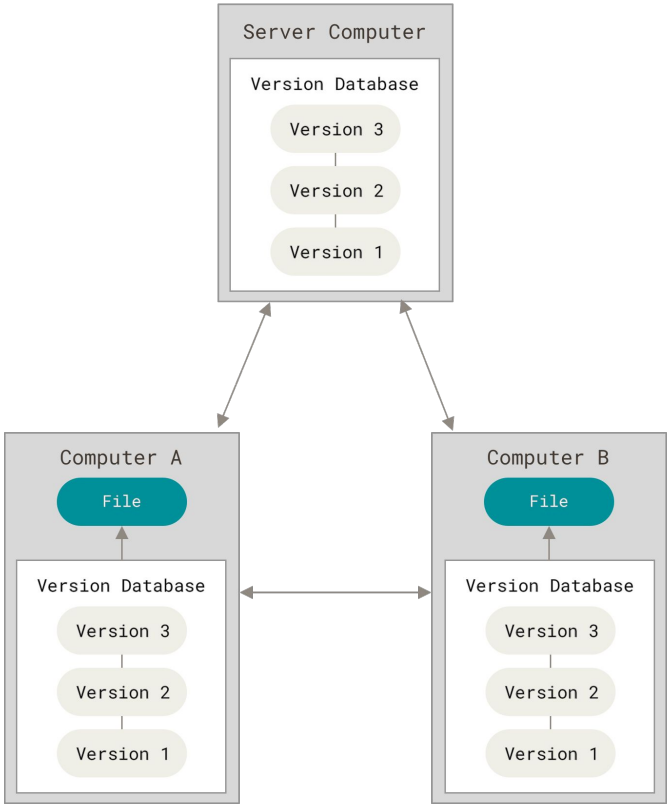
Has nothing to do with

or

Gitlab and github are code storage platforms. ‘Cloning’ something from one of them simply takes the repository stored them and makes a local copy in your plattform but that’s it. You can then sever the connection (we’ll see what that means later) and forget about them forever. You can create repositories on your own without cloning it from anywhere!

And it goes even further…. code storage platforms are not even necessary to work with multiple developers in git… although in practice they are used that way

Git is a fully distributed system



All repositories are exactly equivalent from a technical point of view. I could set up a system that connects directly with another developer and allow us to execute git commands between ourselves without a designated 'main repository'

In most practical cases though git is used 'sort of' like a centralized system. There is only one place (the remote repository, for example hosted in gitlab or github) where people connect, set updates and get updates, but there is no real client-server differentiation.

Most of the time in git you are interacting with your local repository!

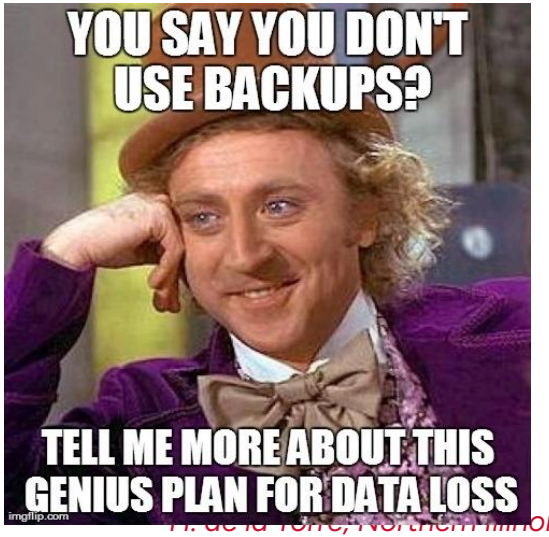
Three ways to use git

For the same project

A student writing their thesis or dissertation (with LateX):

You start writing in your laptop, keeping track of the changes you make using git	*Fully local repository, single developer, useful to keep track of changes and recover previous versions !*
You realize that having backups of your thesis is **the best idea ever**	*You add a remote repository in your preferred code repository and keep synchronizing it regularly with your local repository*
You want to share your work with your supervisor and let them do modifications	*You both have access to the remote repository. You are able to control the level of access that your supervisor has*

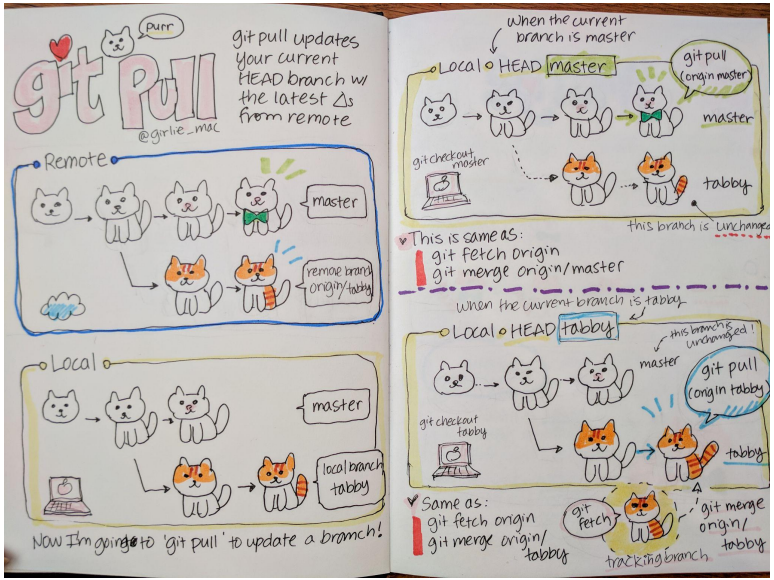
PSA intermission



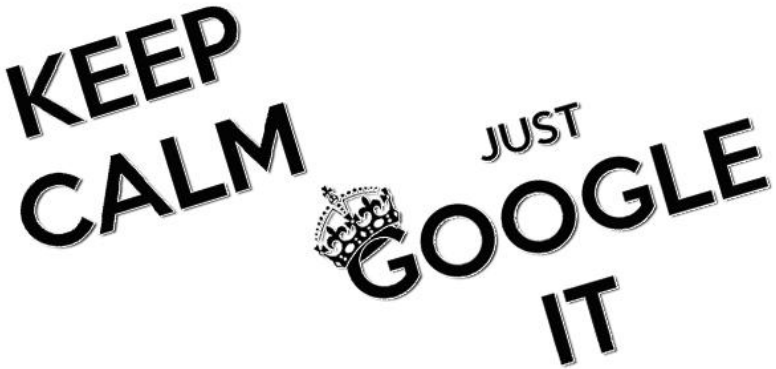
I've personally known several cases of significant data loss in my career (including partially completed thesis) because of insufficient backups. You don't have to use remote git repositories, you can use whatever you want, but keep backups of everything that you don't want to lose!

A disclaimer

I'm not intending to give you a full description of all of the possible commands and workflows possible on git, but rather describe the most useful ones. There will be bias towards what I typically use and there are many ways of doing similar things in git.



There are a lot of references and cheat sheets out there ! Use them



You don’t need friends (or internet) to git

Assuming you have git installed (which most unix machines will have) starting a repository is as easy as doing *git init*.

```
┌┌(htorre@khazad-dum)┌(1015/pts/0)┌(09:38pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git init
Initialized empty Git repository in /home/htorre/work/workingFolder/.git/
```

My current folder (workingFolder) is now version controlled with a repository located in .git!

Lets explore the repository with the most useful and common git command… git status

```
└┌(%:~/work/workingFolder)┌- git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.py
        file2.py

nothing added to commit but untracked files present (use "git add" to track)
```

Status will tell you a lot of how the state of the git repository. In this case it’s telling me that I have two files in my working folder copy but they are ‘untracked’

More basic commands

Untracked files are files that are not in the repository. They are in your current folder, but they are unknown by your repository. The first command that we use to add files to the repository (We'll use it later to warn git that we have made changes to files) is **git add.**

```
┌┌(htorre@khazad-dum)┌(1019/pts/0)┌(10:27pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git add file1.py
┌┌(htorre@khazad-dum)┌(1020/pts/0)┌(10:27pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git add file2.py
┌┌(htorre@khazad-dum)┌(1021/pts/0)┌(10:27pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.py
        new file:   file2.py
```

What we have done now is to **stage** some changes, in this case the fact that there are two new files to be added. This is simply preparing changes, is not actually operation on the repository yet!

More basic commands

The second command that we need to add files is a vital one. **git commit.** commit will create a commit and write it in the repository. Each commit has a message associated with it. You can create it directly using the -m “message” option. If you do git commit without -m a text editor will pop up and you can write whatever message you can. Configuring which editor is system dependent (a usual way is doing git config --global core.editor "editor").

```
└┌(%:~/work/workingFolder)┌- git commit -m 'First commit, adding files'
[main (root-commit) 46edca7] First commit, adding files
 2 files changed, 5 insertions(+)
 create mode 100644 file1.py
 create mode 100644 file2.py
┌┌(htorre@khazad-dum)┌(1035/pts/0)┌(10:53pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main
nothing to commit, working tree clean
┌┌(htorre@khazad-dum)┌(1036/pts/0)┌(10:54pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git log
commit 46edca7be53198d22543c6d924110bcad5a86bab (HEAD -> main)
Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
Date:   Sun Feb 2 22:53:44 2025 -0600

    First commit, adding files
┌┌(htorre@khazad-dum)┌(1037/pts/0)┌(10:54pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌-
```

You can check the message from previous commits using **git log**

Basic workflow

```
┌┌(htorre@khazad-dum)┌(1038/pts/0)┌(10:58pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- vim file1.py
┌┌(htorre@khazad-dum)┌(1039/pts/0)┌(10:58pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- vim file2.py
┌┌(htorre@khazad-dum)┌(1040/pts/0)┌(10:59pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.py
        modified:   file2.py

no changes added to commit (use "git add" and/or "git commit -a")
┌┌(htorre@khazad-dum)┌(1041/pts/0)┌(10:59pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git add file*
┌┌(htorre@khazad-dum)┌(1042/pts/0)┌(10:59pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.py
        modified:   file2.py

┌┌(htorre@khazad-dum)┌(1043/pts/0)┌(10:59pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git commit -m 'I made some changes'
[main d7950f2] I made some changes
 2 files changed, 2 insertions(+)
┌┌(htorre@khazad-dum)┌(1044/pts/0)┌(11:04pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git log
commit d7950f2e895244b93aa167109f90439161ea2a3d (HEAD -> main)
Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
Date:   Sun Feb 2 23:04:13 2025 -0600

    I made some changes

commit 46edca7be53198d22543c6d924110bcad5a86bab
Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
Date:   Sun Feb 2 22:53:44 2025 -0600

    First commit, adding files
┌┌(htorre@khazad-dum)┌(1045/pts/0)┌(11:04pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌-
```

You already know everything you need for the most basics of workflows. You make changes, you stage them (with add) you commit them and that's it!. Notice that every commit has a very long identifier. This is the git hash. Is a unique identifier produced using a complex algorithm (SHA-1) that takes elements of the commit as inputs. It's a cryptographic hash function (Designed by the NSA! in the 90s)

Exploring the history

Using the commit hash, it is possible to recover the changes you did in every commit. For this we use **git diff.**

```
┌┌(htorre@khazad-dum)┌(1056/pts/0)┌(11:25pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git log
commit d7950f2e895244b93aa167109f90439161ea2a3d (HEAD -> main)
Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
Date:   Sun Feb 2 23:04:13 2025 -0600

    I made some changes

commit 46edca7be53198d22543c6d924110bcad5a86bab
Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
Date:   Sun Feb 2 22:53:44 2025 -0600

    First commit, adding files
┌┌(htorre@khazad-dum)┌(1057/pts/0)┌(11:26pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git diff 46edca7be53198d22543c6d924110bcad5a86bab d7950f2e895244b93aa167109f90439161ea2a3d
diff --git a/file1.py b/file1.py
index 45579f0..8e4140a 100644
--- a/file1.py
+++ b/file1.py
@@ -1 +1,2 @@
 print("This is a simple python file")
+print("I made a change")
diff --git a/file2.py b/file2.py
index 2895583..ca90b99 100644
--- a/file2.py
+++ b/file2.py
@@ -2,3 +2,4 @@ a = 1
 b = 2

 print("The numbers are " + str(a) + " and " + str(b))
+print("I made another change")
```

Uncommited/Unstaged changes

```
└┌(%:~/work/workingFolder)┌- git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.py

no changes added to commit (use "git add" and/or "git commit -a")
┌┌(htorre@khazad-dum)┌(1075/pts/0)┌(11:39pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git diff file1.py
diff --git a/file1.py b/file1.py
index 8e4140a..fd2d121 100644
--- a/file1.py
+++ b/file1.py
@@ -1,2 +1,3 @@
 print("This is a simple python file")
 print("I made a change")
+print("I made another change")
```

diff can also be used to see the changes between your working copy and the latest commits before staging them

git restore (git restore --staged) can be used to discard changes before committing them. Note that restore is not reversible ! Discarded changes are lost forever

```
└┌(%:~/work/workingFolder)┌- git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.py

no changes added to commit (use "git add" and/or "git commit -a")
┌┌(htorre@khazad-dum)┌(1077/pts/0)┌(11:39pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git restore file1.py
┌┌(htorre@khazad-dum)┌(1078/pts/0)┌(11:39pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main
nothing to commit, working tree clean
┌┌(htorre@khazad-dum)┌(1079/pts/0)┌(11:40pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌-
```

But what about committed changes

There are many commands and workflows that would allow you to ‘return’ to a previous commit, with various levels of messing with the commit history. In my opinion a good rule of thumb for anything git is **‘Never mess with the timeline’**. Instead, create a new commit that represents a return to the previous code version.

```
└┌(%:~/work/workingFolder)┌- git revert --no-commit d7950f2e895244b93aa167109f90439161ea2a3d..HEAD
┌┌(htorre@khazad-dum)┌(1086/pts/0)┌(11:44pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main
You are currently reverting commit a7f92bd.
  (all conflicts fixed: run "git revert --continue")
  (use "git revert --skip" to skip this patch)
  (use "git revert --abort" to cancel the revert operation)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.py

┌┌(htorre@khazad-dum)┌(1087/pts/0)┌(11:44pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git commit -m 'I returned to the previous change'
[main f083b58] I returned to the previous change
 1 file changed, 1 deletion(-)
┌┌(htorre@khazad-dum)┌(1088/pts/0)┌(11:45pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main
nothing to commit, working tree clean
┌┌(htorre@khazad-dum)┌(1089/pts/0)┌(11:45pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git log
commit f083b5832b9750527a3bbfaaad2b6fae1214fa44 (HEAD -> main)
Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
Date:   Sun Feb 2 23:45:05 2025 -0600

    I returned to the previous change

commit a7f92bd3784ee1d73aa19b6c9bec809def181007
Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
Date:   Sun Feb 2 23:43:16 2025 -0600

    A second change, yuhu

commit d7950f2e895244b93aa167109f90439161ea2a3d
Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
Date:   Sun Feb 2 23:04:13 2025 -0600

    I made some changes

commit 46edca7be53198d22543c6d924110bcad5a86bab
Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
Date:   Sun Feb 2 22:53:44 2025 -0600

    First commit, adding files
┌┌(htorre@khazad-dum)┌(1090/pts/0)┌(11:46pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌-
```

It’s not a very common operation, but my preference is to use git revert --no-commit hash..HEAD.

This creates a new staged version (that you then commit) that is identical to the indicated commit.

```
└┌(%:~/work/workingFolder)┌- git checkout 46edca7be53198d22543c6d924110bcad5a86bab
Note: switching to '46edca7be53198d22543c6d924110bcad5a86bab'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 46edca7 First commit, adding files
┌┌(htorre@khazad-dum)┌(1092/pts/0)┌(11:47pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git status
HEAD detached at 46edca7
nothing to commit, working tree clean
┌┌(htorre@khazad-dum)┌(1093/pts/0)┌(11:48pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌- git switch -
Previous HEAD position was 46edca7 First commit, adding files
Switched to branch 'main'
┌┌(htorre@khazad-dum)┌(1094/pts/0)┌(11:48pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌-
```

You can also use git checkout to ‘explore’ an older commit, and git switch - to return to your HEAD

How git stores information

```
└┌(%:~/work/workingFolder)┌- ll .git
total 40
drwxr-xr-x  4 htorre htorre 4096 Feb  2 21:38 refs/
drwxr-xr-x  2 htorre htorre 4096 Feb  2 21:38 info/
drwxr-xr-x  2 htorre htorre 4096 Feb  2 21:38 hooks/
-rw-r--r--  1 htorre htorre   73 Feb  2 21:38 description
-rw-r--r--  1 htorre htorre   92 Feb  2 21:38 config
drwxr-xr-x  3 htorre htorre 4096 Feb  2 22:53 logs/
-rw-r--r--  1 htorre htorre   34 Feb  2 23:45 COMMIT_EDITMSG
drwxr-xr-x 16 htorre htorre 4096 Feb  2 23:45 objects/
-rw-r--r--  1 htorre htorre  209 Feb  2 23:48 index
-rw-r--r--  1 htorre htorre   21 Feb  2 23:48 HEAD
┌┌(htorre@khazad-dum)┌(1101/pts/0)┌(11:54pm:02/02/25)┌-
└┌(%:~/work/workingFolder)┌-
```

Note: your config file is also here… which we will come back to later

A technical intermission

You repository lives in the .git folder wherever you did git init. The ‘actual’ objects live in the objects folder. There are four types of objects (all of them have a hash associated to it)

- Previous file versions (blobs)
- folder structure (tree)
- commits (including metadata)
- tags (human readable labels for specific commits)

Interestingly, git will store full files in the repository. It uses deltas (diffs) for compression when useful (automatically) but it’s always possible to reconstruct the whole codebase from a single commit.

Branches

Branches are separate development lines that will have it's own commit history. Changes made in one branch don't affect the trunk and viceversa. In reality, the 'definition' of trunk is philosophical in git, not technical. The 'trunk' (usually called main branch) is just another branch. Most of the times you create branches from main but you can create branches from anywhere

```
┌(htorre@khazad-dum)┌(1007/pts/0)┌(09:31am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git checkout -b 'development-branch'


Switched to a new branch 'development-branch'
```

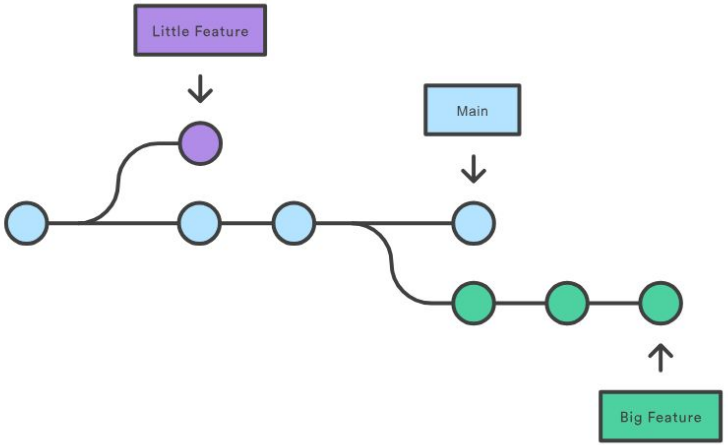
```
┌(htorre@khazad-dum)┌(1047/pts/0)┌(09:39am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git checkout -b 'development_another_branch'
Switched to a new branch 'development_another_branch'
```

```
┌(htorre@khazad-dum)┌(1036/pts/0)┌(09:37am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git checkout main
Switched to branch 'main'
```

```
┌(htorre@khazad-dum)┌(1057/pts/0)┌(09:44am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git branch --list
  development-branch
* development_another_branch
  main
```

git checkout to switch between existing branches. -b 'nameOfBranch' to create new branches. git branch --list to see the list of branches

Branches



Branches are useful for many things. Testing things without messing with master. developing things in a multi-developer environment (to avoid everyone working in main at the same time!). I personally use it for single developer projects too as I find it cleaner

```
┌┌(htorre@khazad-dum)┌(1054/pts/0)┌(09:39am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git log --graph --branches --all
* commit df151900cd7ded688fb79eea5b13c2f4cdb87523 (HEAD -> development_another_branch)
| Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
| Date:   Mon Feb 3 09:39:53 2025 -0600
|
|     This is a change in another branch
|
| * commit b53f6c5a6eb2a9ea1ab1ec04749ad17b8bcb9b1b (development-branch)
| | Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
| | Date:   Mon Feb 3 09:36:57 2025 -0600
| |
| |     Change in file 2
| |
| * commit d6a53b7b6a1412424fdb7c2e5bbb63c01516a16c
|/  Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
|   Date:   Mon Feb 3 09:33:16 2025 -0600
|
|       I added a change to file1.py
|
* commit f083b5832b9750527a3bbfaaad2b6fae1214fa44 (main)
| Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
| Date:   Sun Feb 2 23:45:05 2025 -0600
|
|     I returned to the previous change
|
* commit a7f92bd3784ee1d73aa19b6c9bec809def181007
| Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
| Date:   Sun Feb 2 23:43:16 2025 -0600
|
|     A second change, yuhu
|
* commit d7950f2e895244b93aa167109f90439161ea2a3d
| Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
| Date:   Sun Feb 2 23:04:13 2025 -0600
|
|     I made some changes
|
* commit 46edca7be53198d22543c6d924110bcad5a86bab
  Author: Hector de la Torre Perez <hector.de.la.torre.perez@cern.ch>
  Date:   Sun Feb 2 22:53:44 2025 -0600

      First commit, adding files
```

Merges

After development happens in a branch, the common next step is to merge it back into main. As you may guess, this is done using the git merge command.

```
└┌(%:~/work/workingFolder)┌- git merge development-branch
Updating f083b58..b53f6c5
Fast-forward
 file1.py | 1 +
 file2.py | 1 +
 2 files changed, 2 insertions(+)
┌┌(htorre@khazad-dum)┌(1062/pts/0)┌(10:01am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌-
```

This will integrate the changes into main

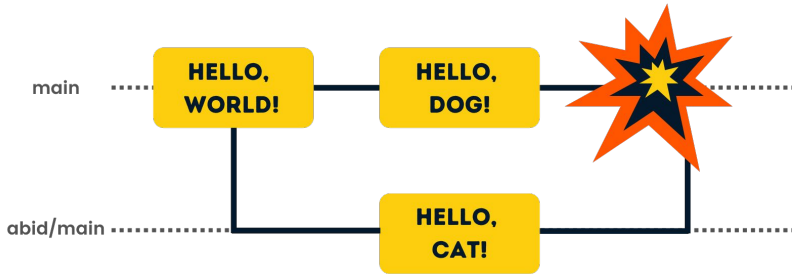
In multi-developer (or sometimes in single developer if the workflow is complicated) development, sometimes the changes that you are integrating in the target branch may conflict with other changes that someone else has merged (or edit directly) there. What happens then?

Conflicts

Note: *You can use diff between branches to check differences between branches*

```
┌┌(htorre@khazad-dum)┌(1073/pts/0)┌(10:12am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git diff main development_another_branch
diff --git a/file1.py b/file1.py
index 627e210..8e4140a 100644
--- a/file1.py
+++ b/file1.py
@@ -1,3 +1,2 @@
 print("This is a simple python file")
 print("I made a change")
-print("This change is very nice")
diff --git a/file2.py b/file2.py
index cae7916..f7bd091 100644
--- a/file2.py
+++ b/file2.py
@@ -1,6 +1,6 @@
-a = 7
-b = 6
+a = 3
+b = 4

 print("The numbers are " + str(a) + " and " + str(b))
 print("I made another change")
-print("I made some other changes to file 2")
┌┌(htorre@khazad-dum)┌(1074/pts/0)┌(10:12am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git merge development_another_branch
Auto-merging file2.py
CONFLICT (content): Merge conflict in file2.py
Automatic merge failed; fix conflicts and then commit the result.
┌┌(htorre@khazad-dum)┌(1075/pts/0)┌(10:12am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌-
```

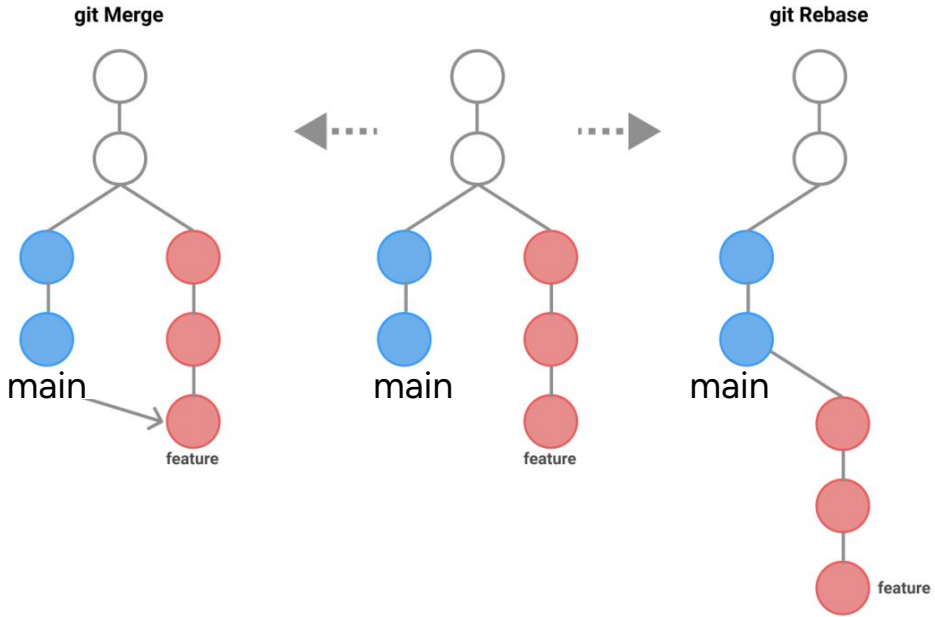


Conflicts need to be fixed by hand. You need to open the file (which will contain both versions with some useful metadata) and edit to your desired version. Then you just commit that version

Not all parallel development leads to conflict. If the changes from one merge are compatible with the changes from another merge then they will not create conflict (Changes to different part of the code or changes to different files for example)

Merge and rebase

There is a similar command to merge called rebase. They also integrate the content of one branch into a different branch. The difference is mostly on how the 'history' is organized. merge will merge the commit history as it is (sequentially). Rebase is saying 'put this branch as the new base of my current branch'

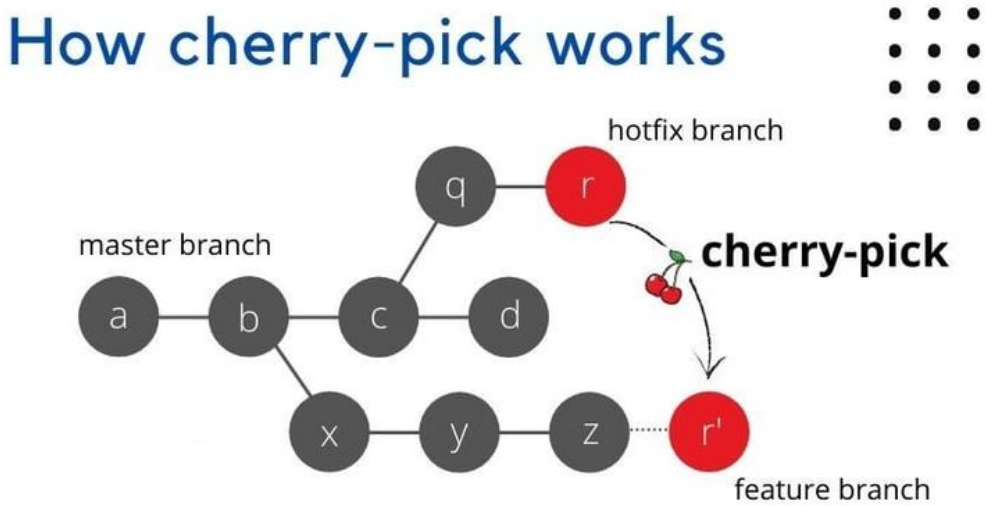


Canonically you use merge to merge into main and rebase to update development branches but in practice rebasing can create a lot of conflicts (and you are messing with the timeline)

Important: Your branch doesn't know about other branches being updated. To avoid conflicts make sure to update your development branch or just create new development branches!

Cherry pick

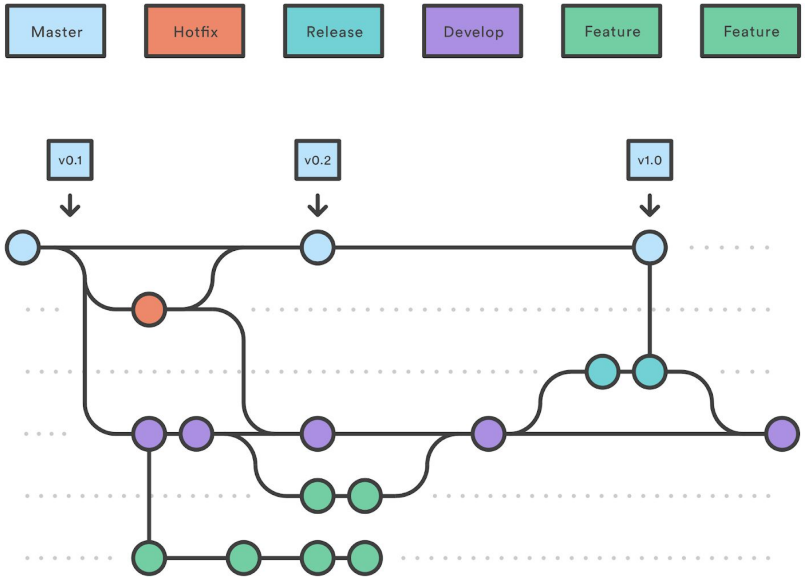
In some cases merge and rebase can lead to a lot of conflicts (especially rebase if you want to somehow integrate a main branch that has undergone lots of developments) because they are trying to merge a lot of commits. git cherry-pick <commit> allows you to integrate only ONE commit into your branch. It's canonically used to apply bug fixes found while in a different branch for example (i.e. you want the fix applied to main, but not the rest of the branch, at least yet) but sometimes is the best way to merge things 'carefully' without going crazy.



Tags

Tags are very similar to branches from the technical point of view, but they are meant to be used differently. They are particular versions that are brought ‘out’ of the current branch (usually main) , given a specific label and left like that. They are commonly used to define stable versions or specific versions that you want identify easily for some other reason

```
┌┌(htorre@khazad-dum)┌(1082/pts/0)┌(10:36am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git tag -a v1.1 -m 'Version 1.1 to give to the user!'
┌┌(htorre@khazad-dum)┌(1083/pts/0)┌(10:37am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git branch -list
error: did you mean `--list` (with two dashes)?
┌┌(htorre@khazad-dum)┌(1084/pts/0)┌(10:37am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git branch --list
  development-branch
  development_another_branch
* main
┌┌(htorre@khazad-dum)┌(1085/pts/0)┌(10:37am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git tag --list
v1.1
┌┌(htorre@khazad-dum)┌(1086/pts/0)┌(10:37am:02/03/25)┌-
└┌(%:~/work/workingFolder)┌-
```

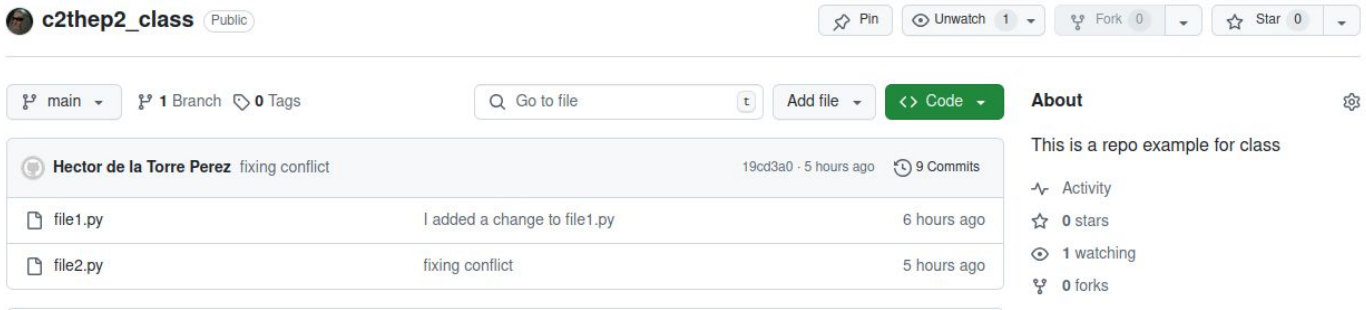


Establishing connections

When you clone a repository (from github for example) you are also establishing a connection between the remote repository and your local repository. Because we already have a repository, we can directly connect it to an empty repository that I created in github. We configure connections by using **git remote**

```
┌┌(htorre@khazad-dum)┌(1006/pts/0)┌(04:18pm:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git remote add origin https://github.com/hdltorre/c2thep2_class.git
┌┌(htorre@khazad-dum)┌(1007/pts/0)┌(04:19pm:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git push -u origin main
Username for 'https://github.com': hdltorre
Password for 'https://hdltorre@github.com':
Enumerating objects: 27, done.
Counting objects: 100% (27/27), done.
Delta compression using up to 12 threads
Compressing objects: 100% (26/26), done.
Writing objects: 100% (27/27), 2.53 KiB | 863.00 KiB/s, done.
Total 27 (delta 7), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (7/7), done.
To https://github.com/hdltorre/c2thep2_class.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

My setup is now exactly the same as if I had clone an existing repo (using git clone)



Establishing connections

The remote configuration lives in the .git/config file. You can have more than one remote and use it in different ways ! One to get information from and one to put information in... which we are going to talk about in a second

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = https://github.com/hdltorre/c2thep2_class.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
    remote = origin
    merge = refs/heads/main
~
```

One note. when you create a remote connection, you are creating additional **remote tracking branches** that represent your knowledge of the remote repository. You are (in essence) creating a local mirror of the remote repository

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = https://:@gitlab.cern.ch:8443/htorre/athena.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[remote "upstream"]
    url = https://:@gitlab.cern.ch:8443/atlas/athena.git
    fetch = +refs/heads/*:refs/remotes/upstream/*
[extensions]
    worktreeConfig = true
~
```

Pulling and pushing

When you do git push, you are sending your local branch to the remote repository. Pushing can create conflicts! You'll need to fix the conflicts before being able to push

git pull is a combination of (**git fetch** and **git merge**). git fetch is simply saying 'update the remote tracking branches' while git merge is saying 'merge the r**emote tracking branch** into my local branch'

When you connect a specific branch with the remote (cloning, pulling or pushing), you establish a tracking relationship between the local and remote branches (this allows you to push and pull without specifying the branch and do other things like diffs between local and remote branches)

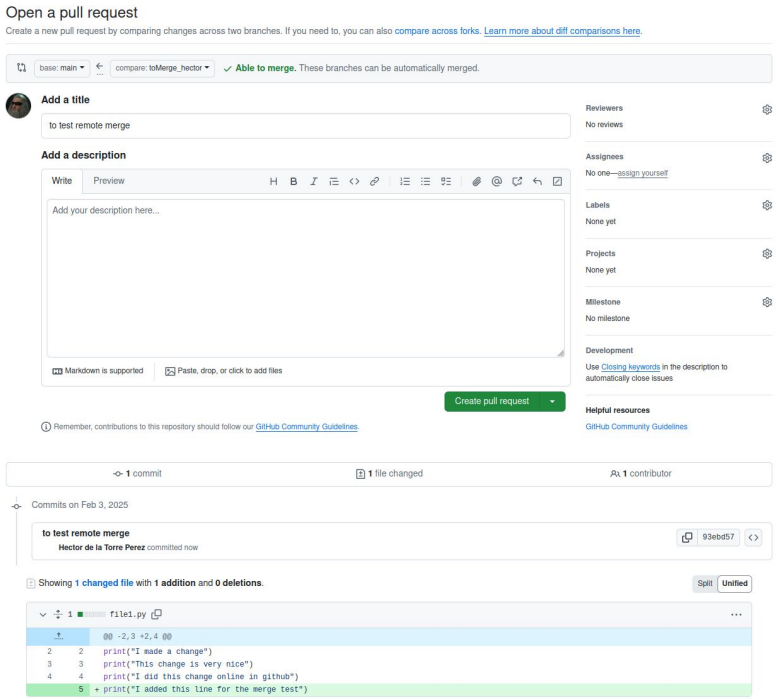
```
┌┌(htorre@khazad-dum)┌(1013/pts/0)┌(04:41pm:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
┌┌(htorre@khazad-dum)┌(1014/pts/0)┌(04:41pm:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- ll
total 8
-rw-r--r-- 1 htorre htorre 97 Feb  3 10:12 file1.py
-rw-r--r-- 1 htorre htorre 98 Feb  3 10:36 file2.py
┌┌(htorre@khazad-dum)┌(1015/pts/0)┌(04:42pm:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 996 bytes | 996.00 KiB/s, done.
From https://github.com/hdltorre/c2thep2_class
   19cd3a0..646e10f  main       -> origin/main
┌┌(htorre@khazad-dum)┌(1016/pts/0)┌(04:57pm:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
┌┌(htorre@khazad-dum)┌(1017/pts/0)┌(04:57pm:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git merge
Updating 19cd3a0..646e10f
Fast-forward
 file1.py | 1 +
 1 file changed, 1 insertion(+)
┌┌(htorre@khazad-dum)┌(1018/pts/0)┌(05:01pm:02/03/25)┌-
└┌(%:~/work/workingFolder)┌- git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
┌┌(htorre@khazad-dum)┌(1019/pts/0)┌(05:01pm:02/03/25)┌-
└┌(%:~/work/workingFolder)┌-
```

The remote merge and the simplest collaborative workflow

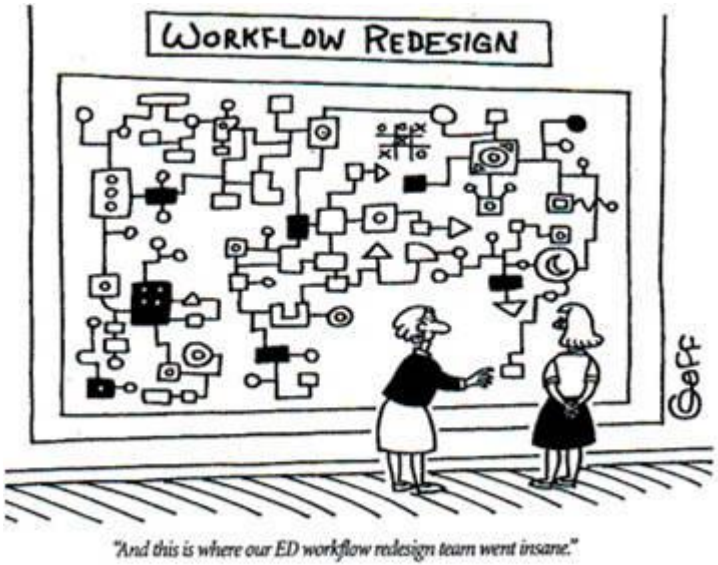


In many typical workflows, developers will work locally and then push the changes to a remote (shared) repository. However, when multiple developers are working on the same project, everyone pushing to main is a bad idea (conflicts galore !) . The most typical way is to push into a short-lived branch (that will be destroyed after) and do a merge in the remote repository. How this works depends a bit on the type of repository being used (github, gitlab, etc) and the project itself but it usually has a nice web interface

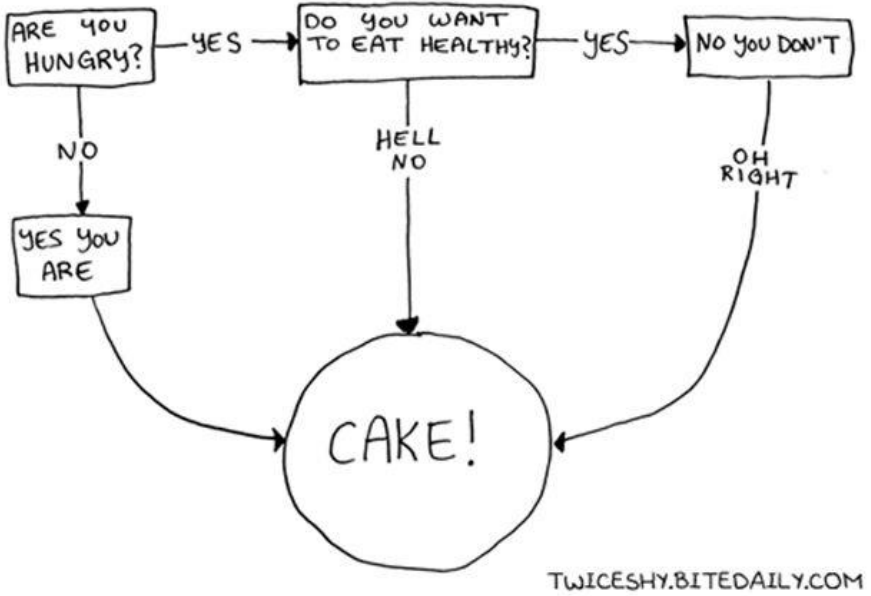
(Confusingly enough, github calls this pull request instead of merge request)

There is no one workflow fits all

You already know the basics… but the actual workflow depends on the project, on the experiment (or company), the people involved and their perspectives of what's 'more efficient'

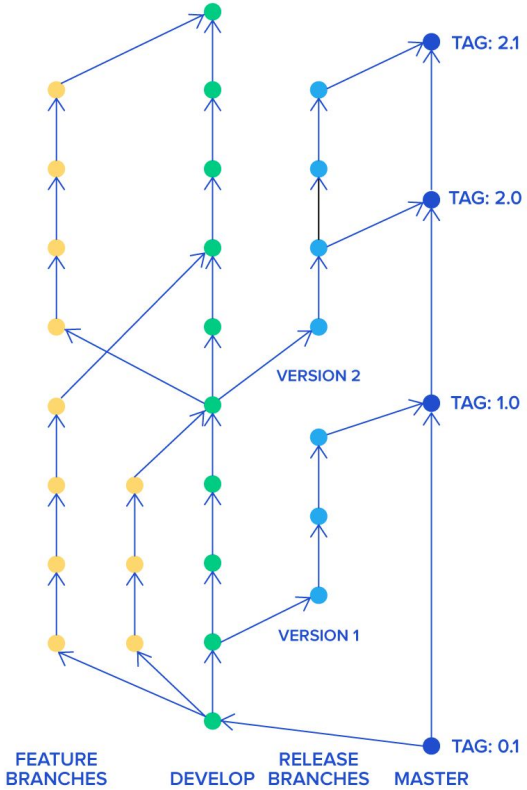


"And this is where our ED workflow redesign team went insane."



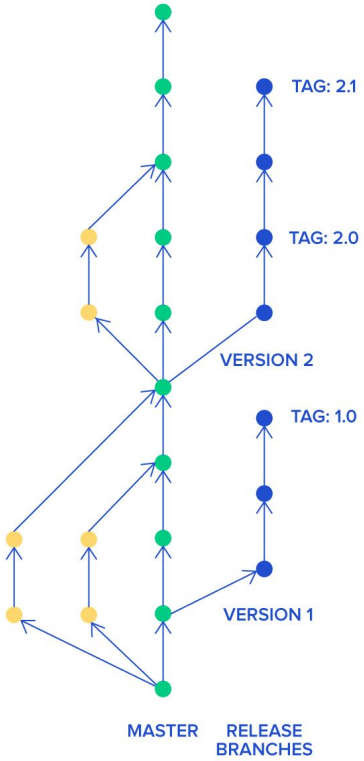
With git, there are two well known types… which are more or less different philosophies of development…

gitflow



Used to be the industry standard some years ago, a bit out of fashion right now. There is one main 'develop' branch from where everything flows. Feature branch from independent developers can be quite lengthy and the review process to merge into develop can be quite lengthy. Testing from deployment happens in release branches and only after some iterations it is release to the users in main

Trunk based workflow

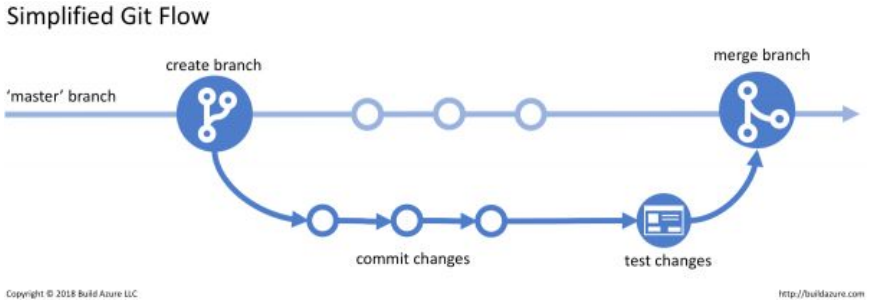


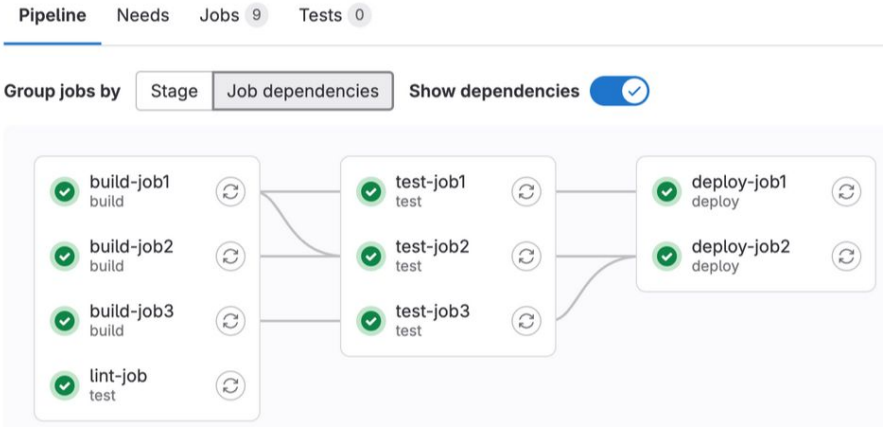
Fewer branches. Developing happens in very short branches that are quickly merged into main again. versions are directly obtain from master and are released and tagged after validation/bug fixes

Sort of the standard nowadays, much more ‘agile’... only really possible thanks to the popularization of continuous integration ! Testing and validation happens during the merge process between the feature branches and the main branch

Continuous integration/deployment

Agile workflows like the one before can only really exist thanks to continuous integration. Continuous integration is a set of tools that automatizes the testing of code as it's being integrated into the main branch. Same idea can be used for deployment when final tags are created





.gitlab-ci.yml 312 Bytes

```yaml
image: python:3.6

stages:
    - build
    - test

build:
    stage: build
    script:
        - echo "Building"
        - mkdir build
        - touch build/info.txt
    artifacts:
        paths:
            - build/

test:
    stage: test
    script:
        - echo "Testing"
        - test -f "build/info.txt"
```

We would need another class to talk about continuous integration. In gitlab (that I know better) it's based in a set of instructions in yml files that tells the system where and how to run the tests. They are usually done using docker containers (which I also won't talk about today)

What about reality?

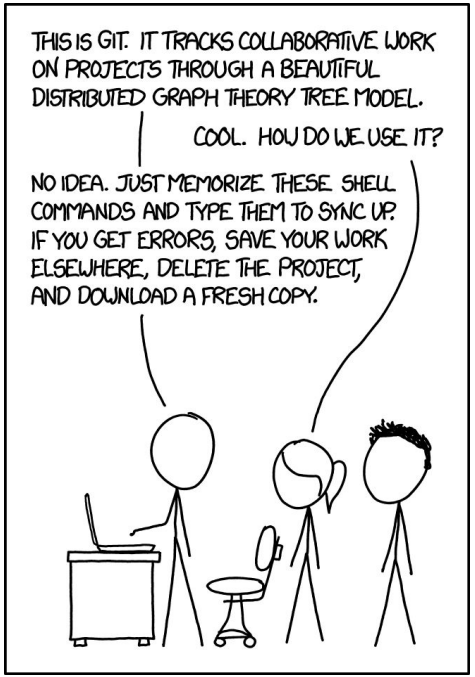
Reality in HEP is that most people follow a 'hammer meets nail' approach to git workflows. They will push directly to main if allowed and if not, they will merge short lived local branches. That's it. Continuous integration is an extremely useful tool to help you manage your own repositories if you have additional developers but it requires work ! With the exception of very complex repositories (ATLAS software for example) that usually have a lengthy review process to accept merges most of your git adventures will be dealing with a small number of developers in a simple repository

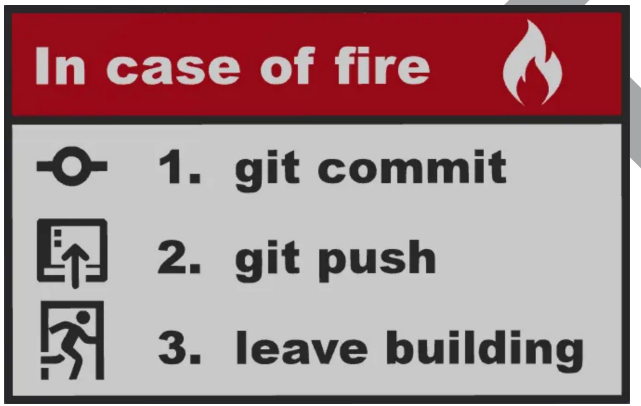
- I (personally) never push to main. Even if I'm working on my own repository. Branch + merge is just cleaner. It automatically translates to multiple developer scenarios and keeps a clean environment.
- You will very often hear the 'Let me know when you stop working' method to avoid conflicts. It's actually quite ok when the number of developers is small.
  - But always remember to fetch+merge/pull when you start working… or you won't be working on the last version
- If you run into troubles. Cherry picking can help you get out of it.

Everyone, and I mean everyone, has at some point in time completely messed up their repository (hopefully only locally) and was unable to fix it. It happens to all of us. Just nuke it and start from scratch (you can save your changes 'by copy-pasting' somewhere safe if you have them)

And that's it !

This is just scratching the surface of git usage. There are many 'tricks' that you will only learn to get you out of trouble, and everyone uses slightly different recipe to do the same thing.





ENJOY

THE REST OF THE COURSE