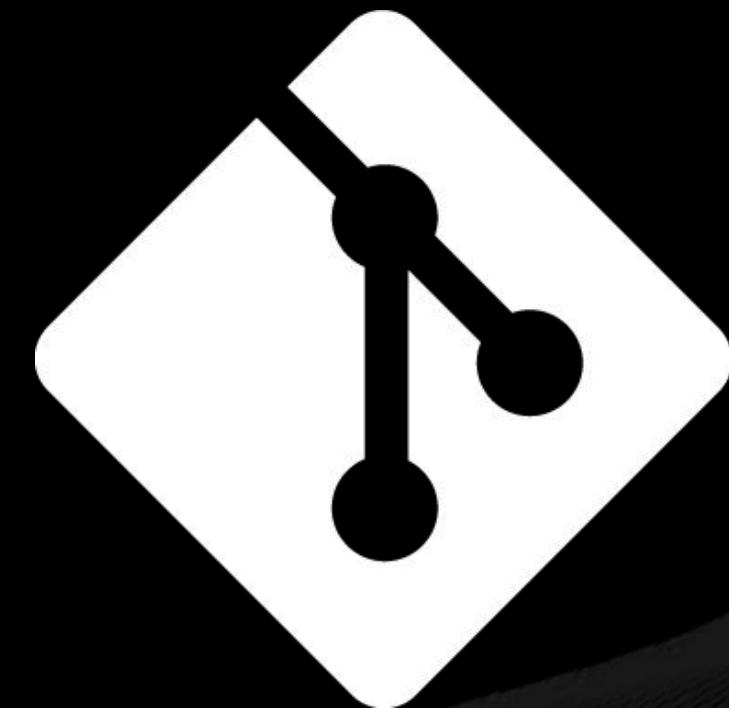


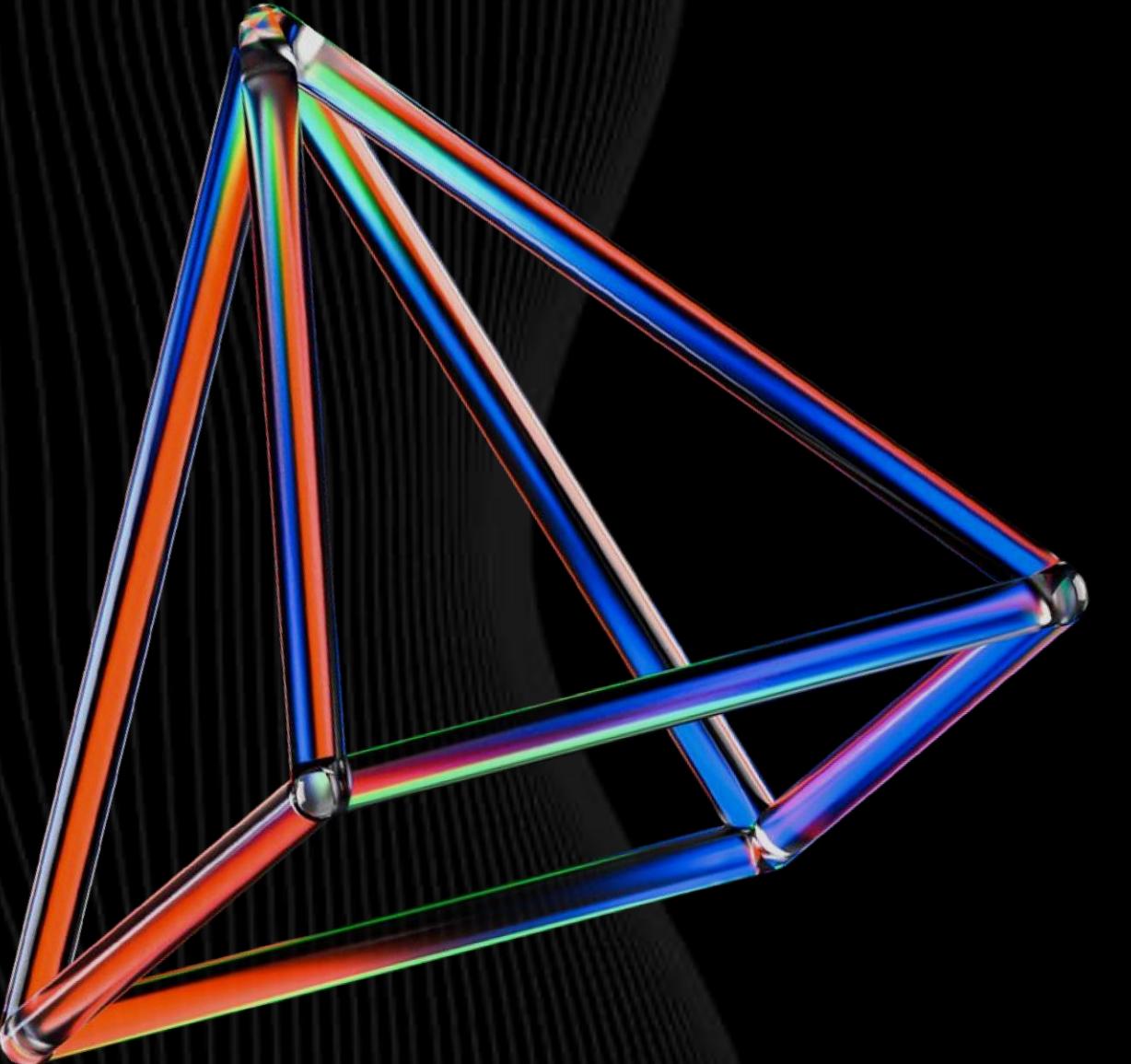
LABORATORIO DE PROGRAMACIÓN CIENTÍFICA PARA CIENCIA DE DATOS

CLASE 2: VCS



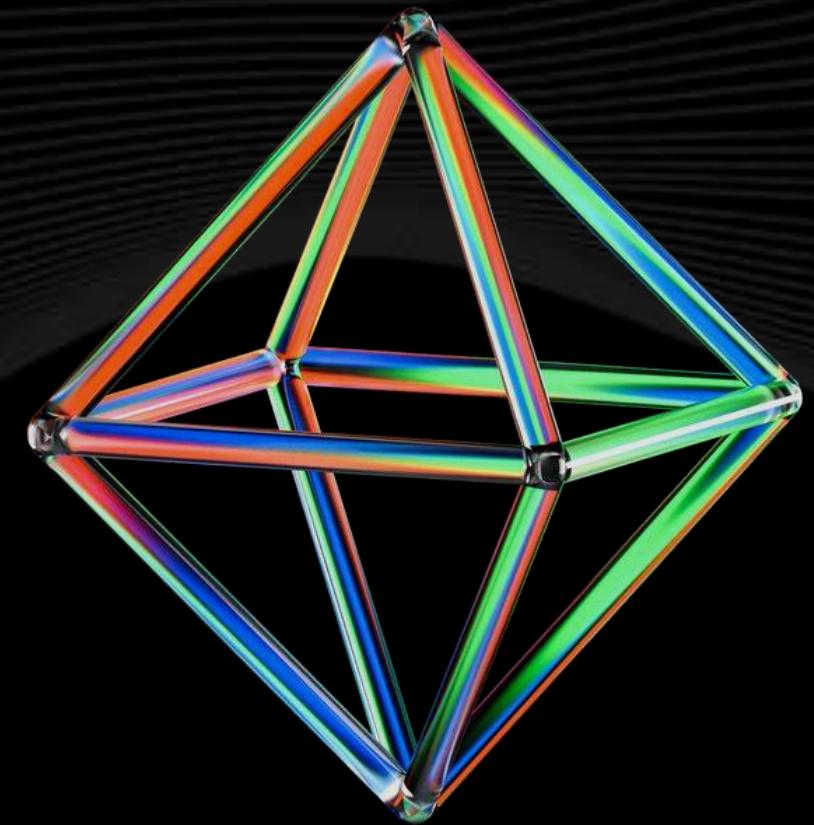
git

MDS7202-1 – PRIMAVERA 2025



{O}BJETIVO

- {01} VCS
- 02 Git
- {03} ¿Necesito VCS?
- 04 Referencias



Veamos un ejemplo

Pensemos en la Tesis...



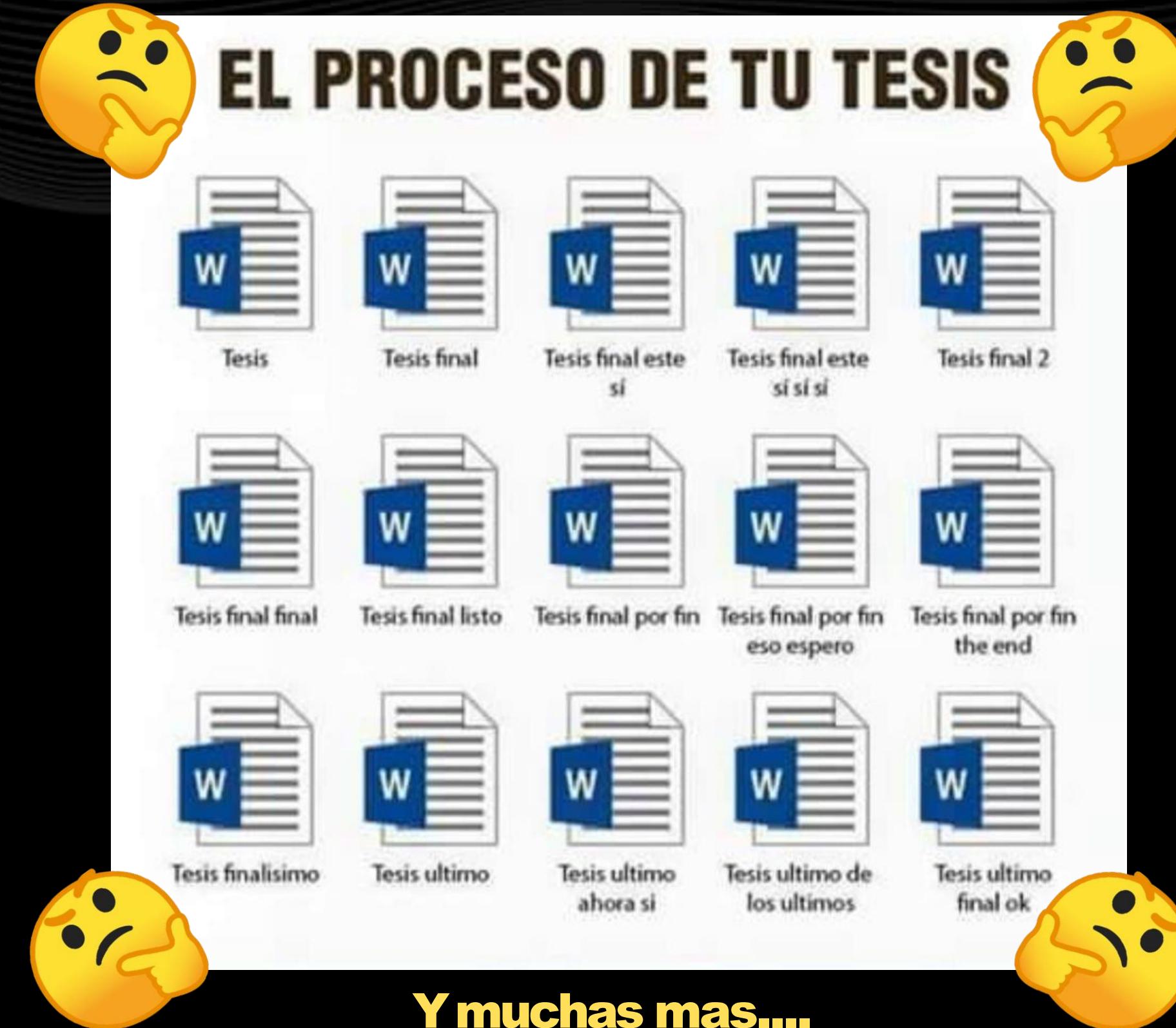
Pensemos en la Tesis...

¿Cuál era mi
último
archivo?

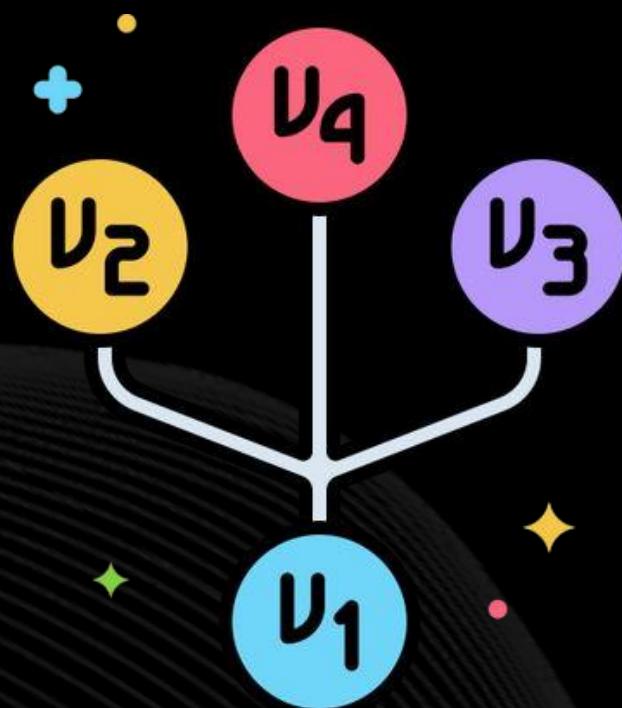
¿Que podría
hacer para
ordenar esto?

¿Qué
diferencia
tiene con la
versión
anterior?
¿Cuál era la
versión
anterior?

¿Quién ***** borró mi
archivo?!!!



Sistema de Control de Versiones (VCS)



¿Qué es?

- Un control de versiones es un **sistema que registra los cambios realizados** en un archivo o conjunto de archivos a lo largo del tiempo.

¿Qué nos permite hacer?

Te permite ver **snapshots** antiguas de un proyecto, llevar un registro de por qué se hicieron ciertos cambios, trabajar en ramas paralelas de desarrollo y mucho más. Cuando se trabaja con otras personas, es una herramienta indispensable para ver lo que han cambiado los demás, así como para resolver conflictos en el desarrollo concurrente.

Los VCS nos permiten responder

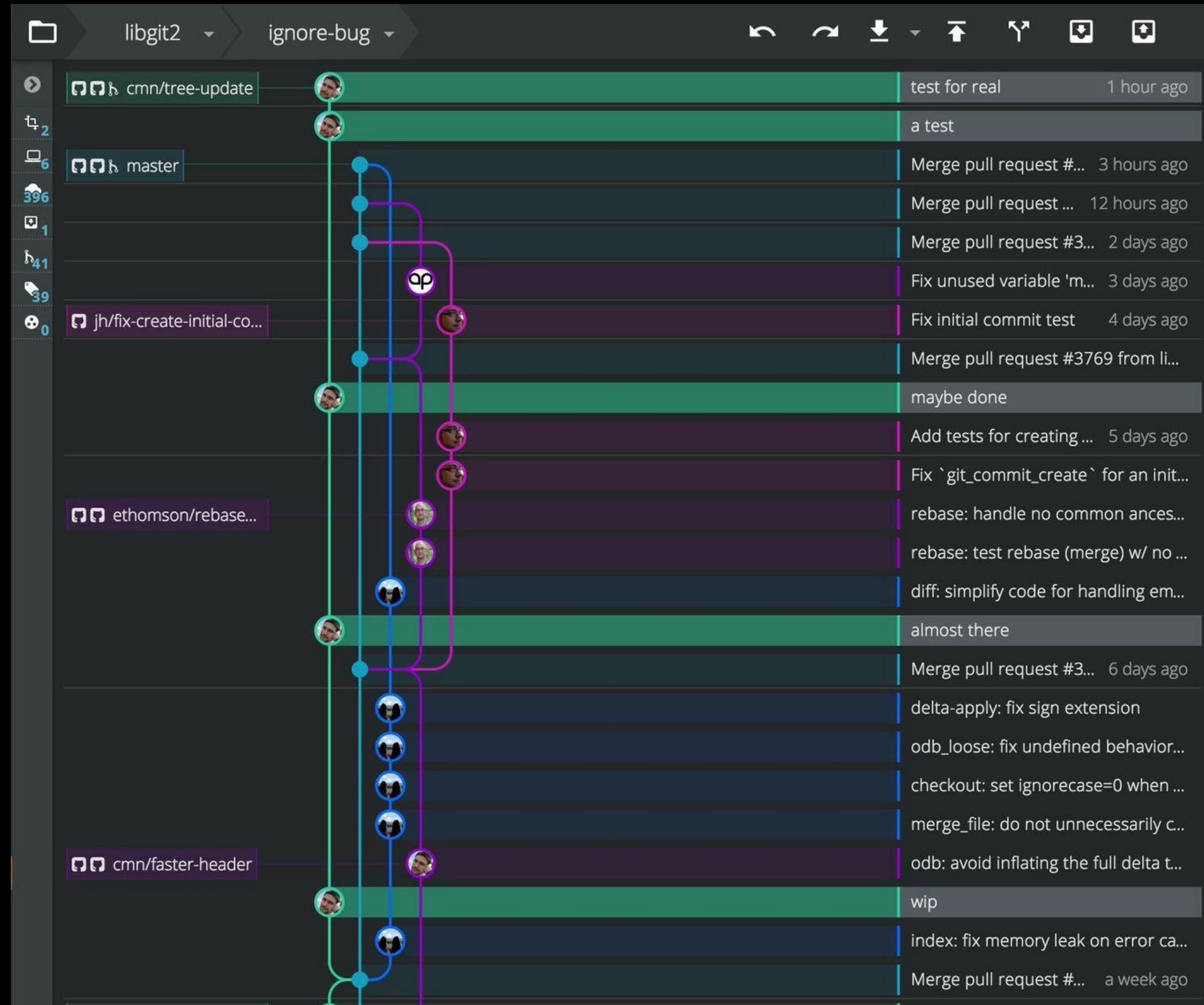
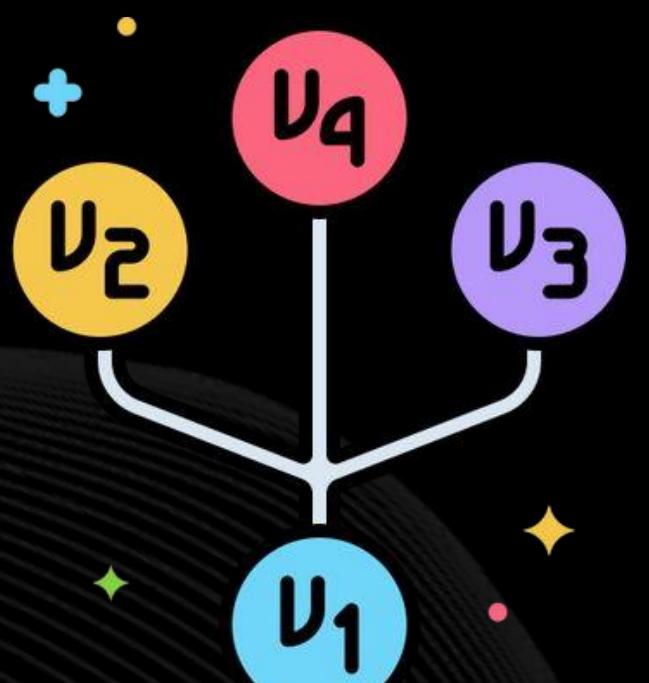


¿Quién
escribió este
módulo?

En las últimas 1000
revisiones, ¿cuándo/por
qué dejó de funcionar
una prueba unitaria
concreta?

¿Cuándo se editó esta línea
concreta de este archivo
concreto? ¿Quién lo hizo?
¿Por qué se editó?

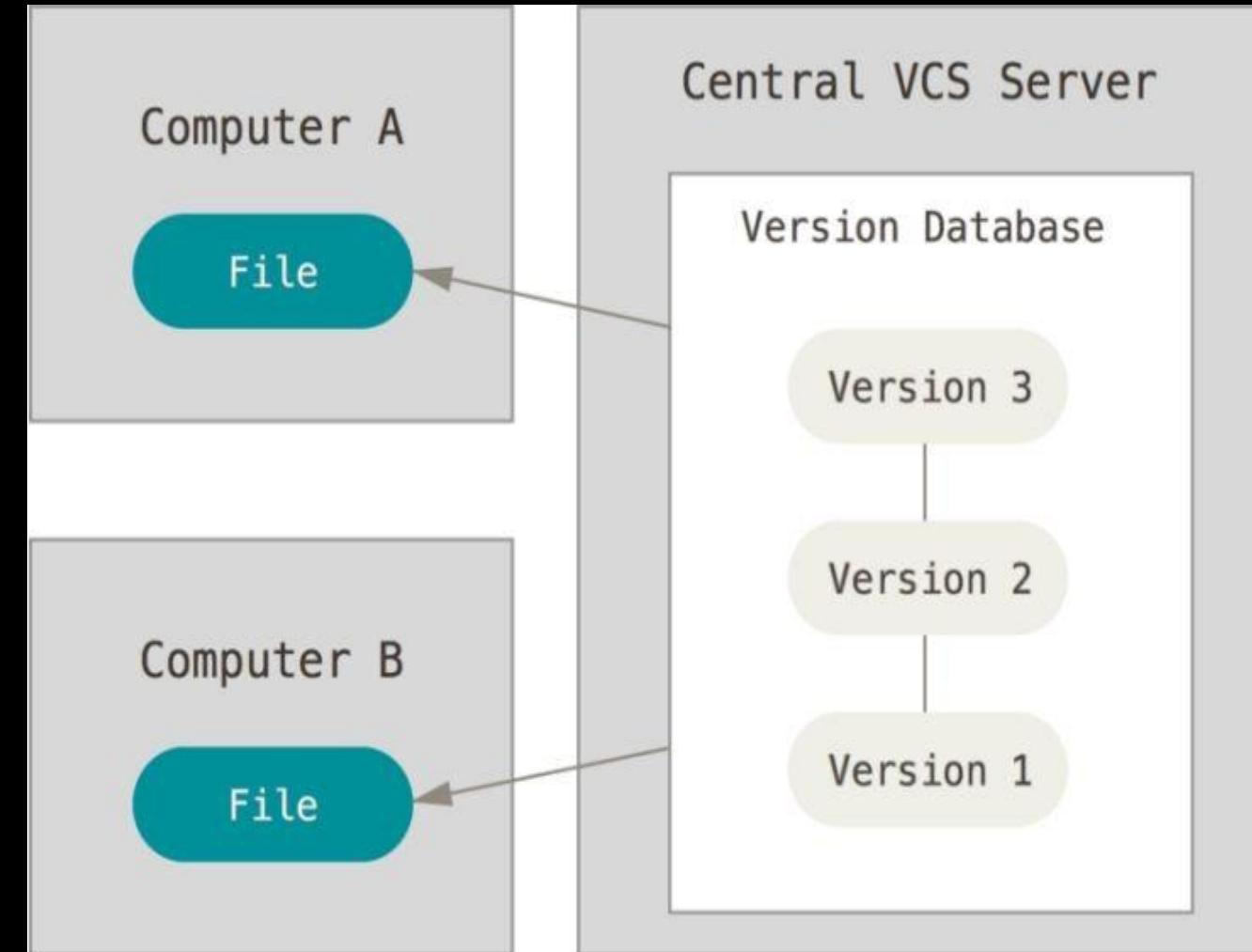
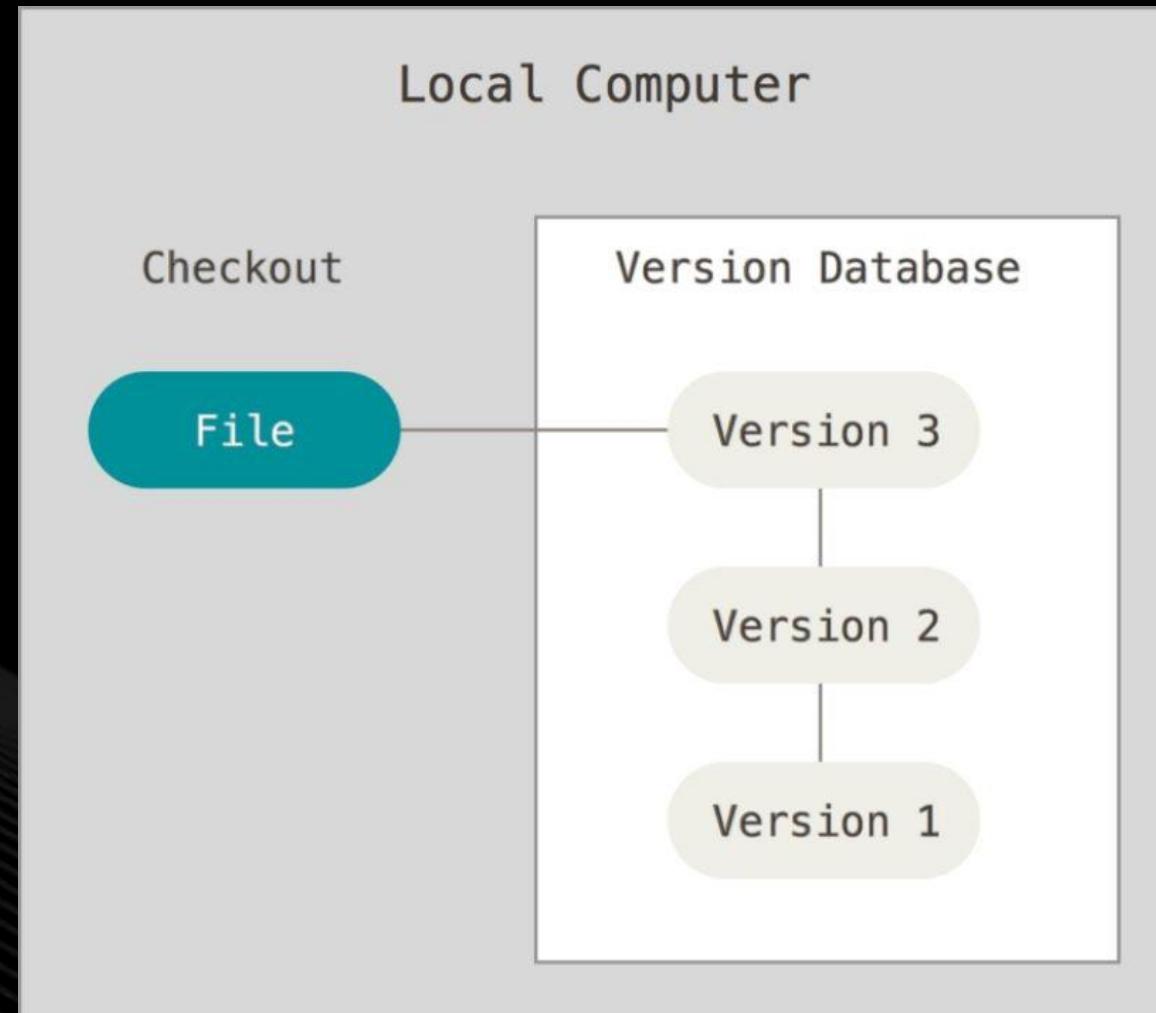
Sistema de Control de Versiones (VCS)



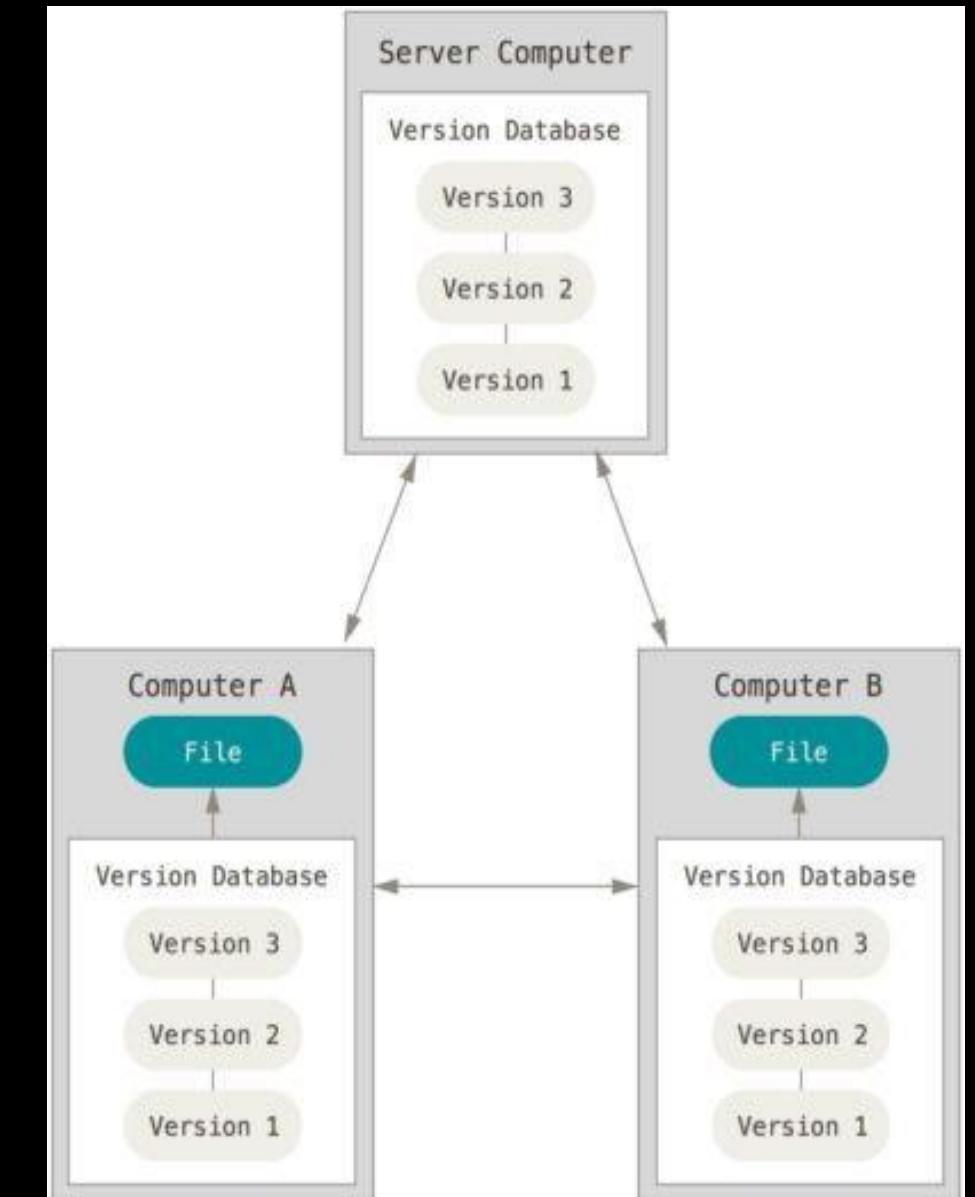
¿Qué Sistemas de versionamiento existen?



VCS Locales

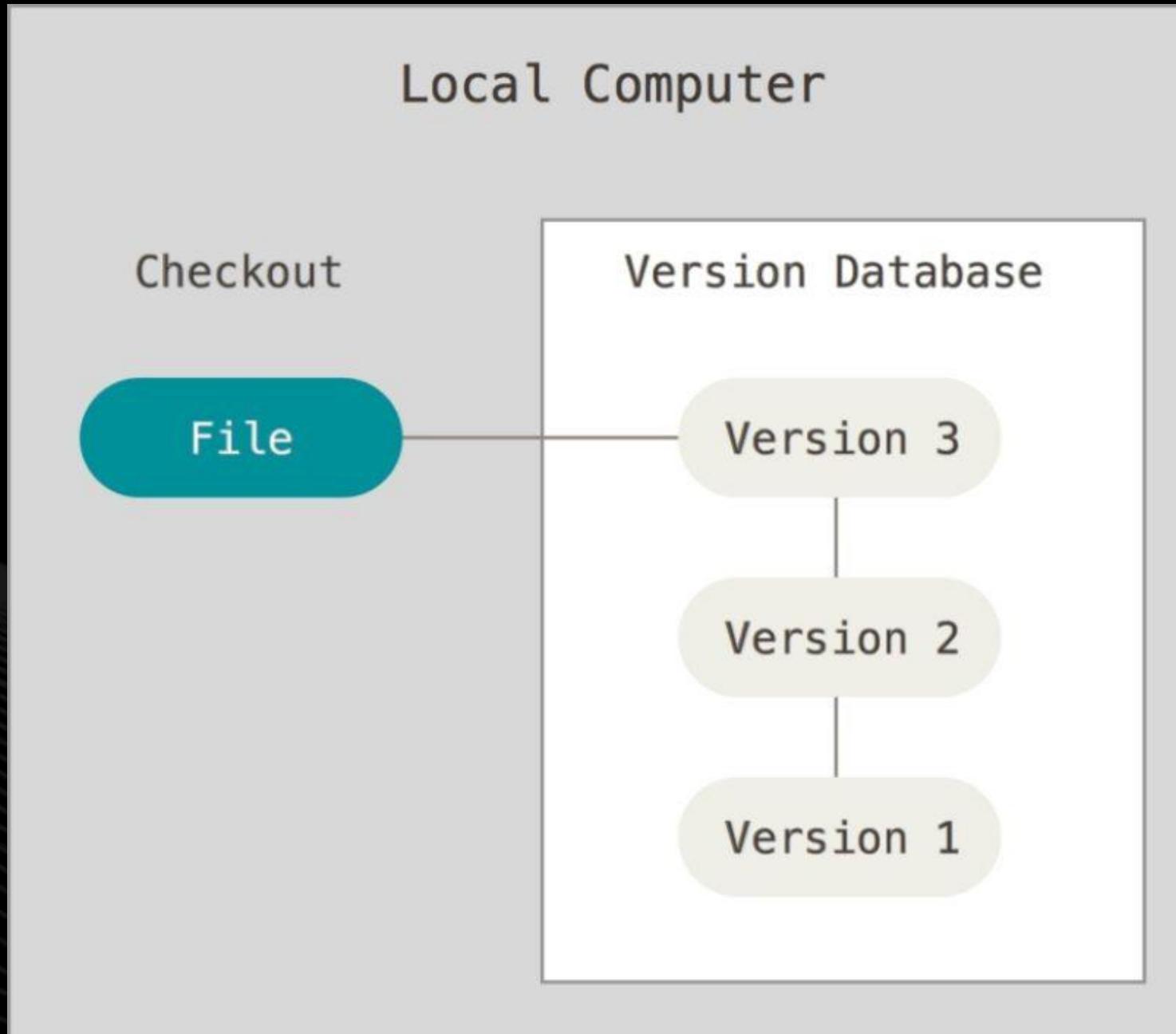


VCS Centralizados



VCS Distribuidos

VCS Locales



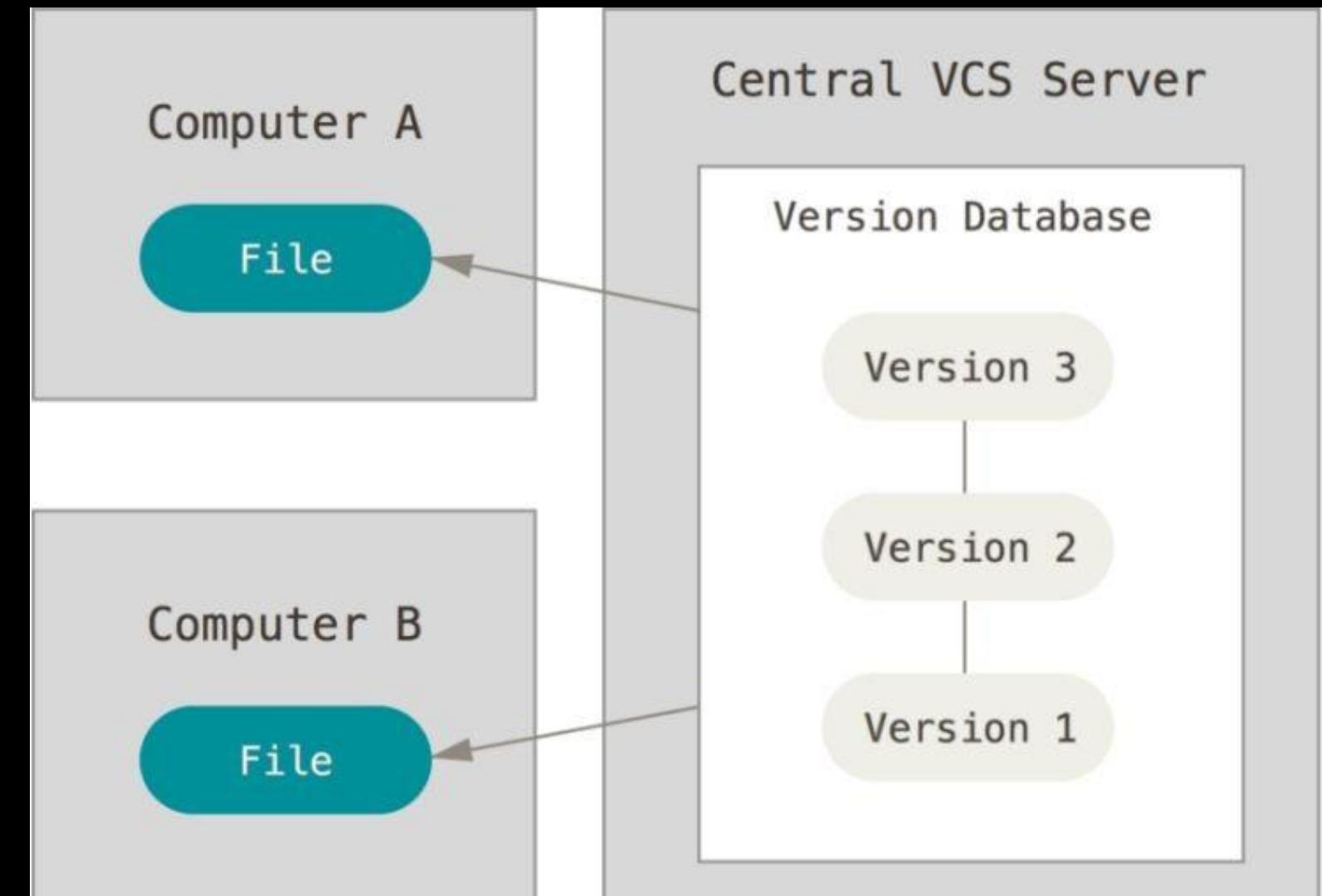
- Basado en generar copias de los archivos en directorios locales.
- Utiliza una base de datos que lleva el registro de los cambios realizados
- Sencillo
- Propenso a errores.

¿Tiene Problemas?

- ¿Qué sucede si se corrompe o me roban el computador?
- ¿Qué hago si quiero trabajar colaborativamente?

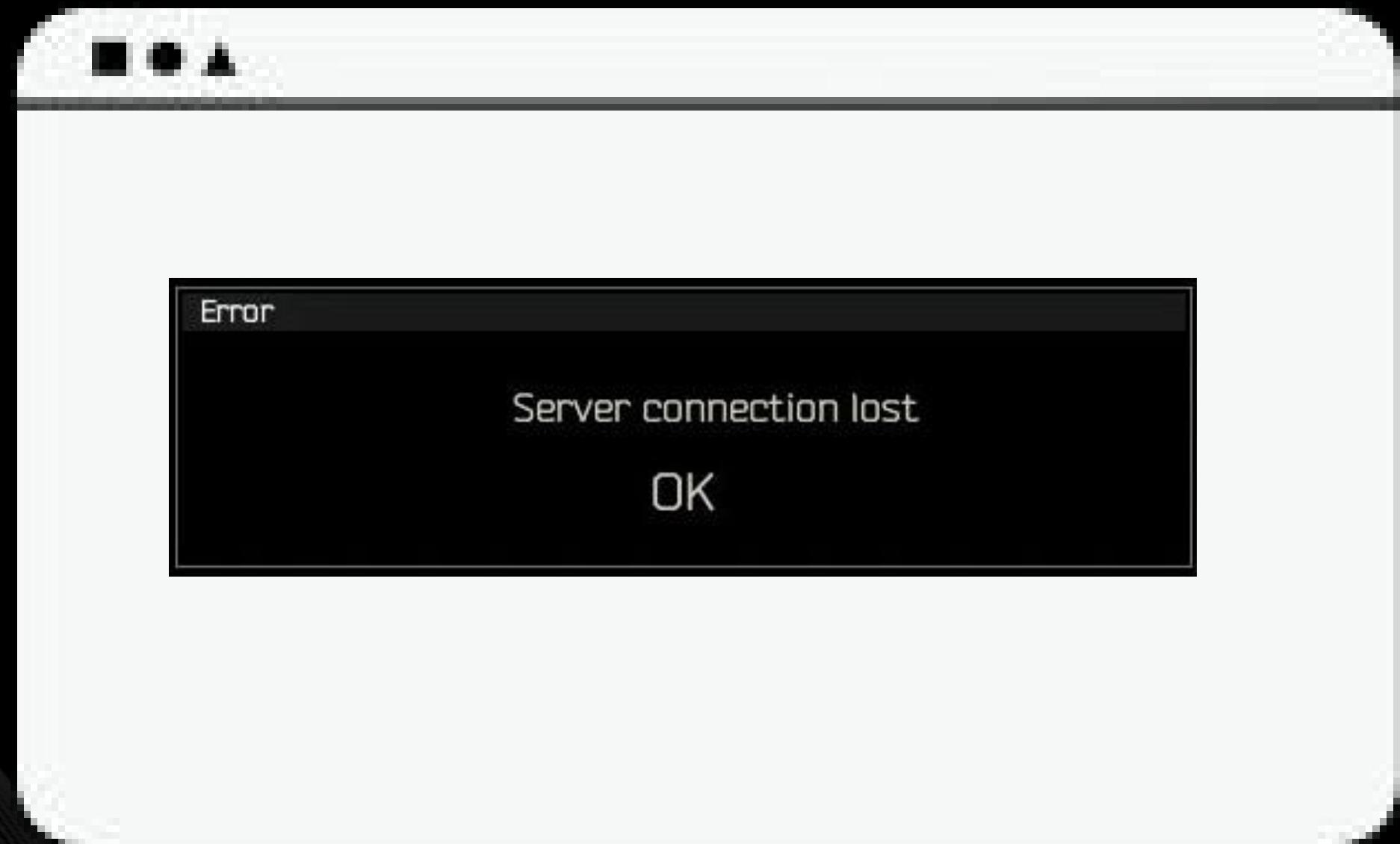
VCS Centralizados

- Los archivos se guardan en un **único** servidor.
- El servidor que contiene todos los archivos versionados y varios clientes que descargan los archivos desde ese lugar central.
- Permite conocer qué procesos están desarrollando compañeros.
- Los administradores controlan qué puede hacer cada uno.
- Mas simples de manejar, ya que no manejan una base de datos como el local.



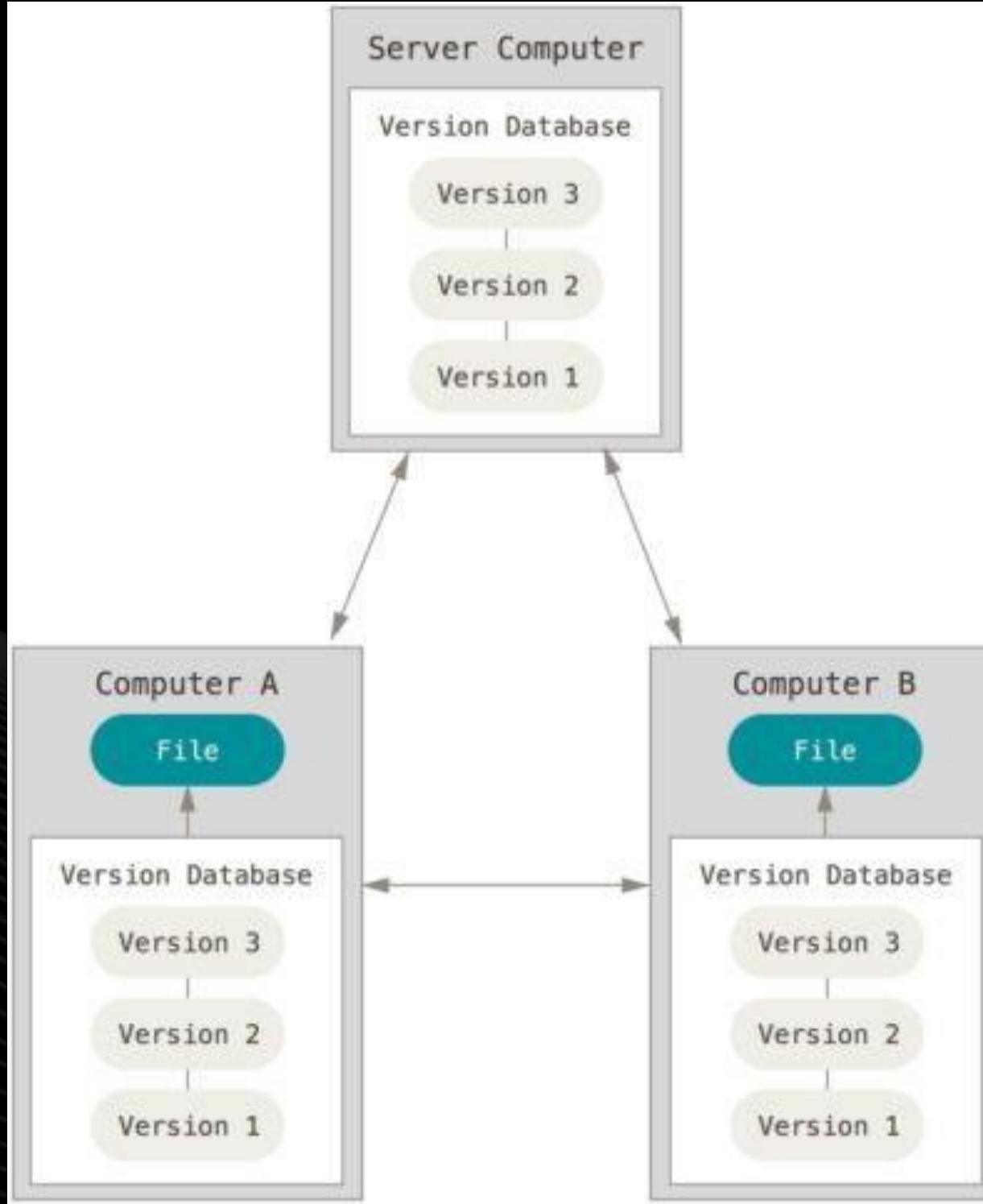
¿Qué problemas tiene?

Poseen un punto único de fallo: El servidor centralizado.



Si el disco duro falla y no existen copias de seguridad, el proyecto puede perderse. Solo se salvan las copias locales 😢

VCS Distribuidos

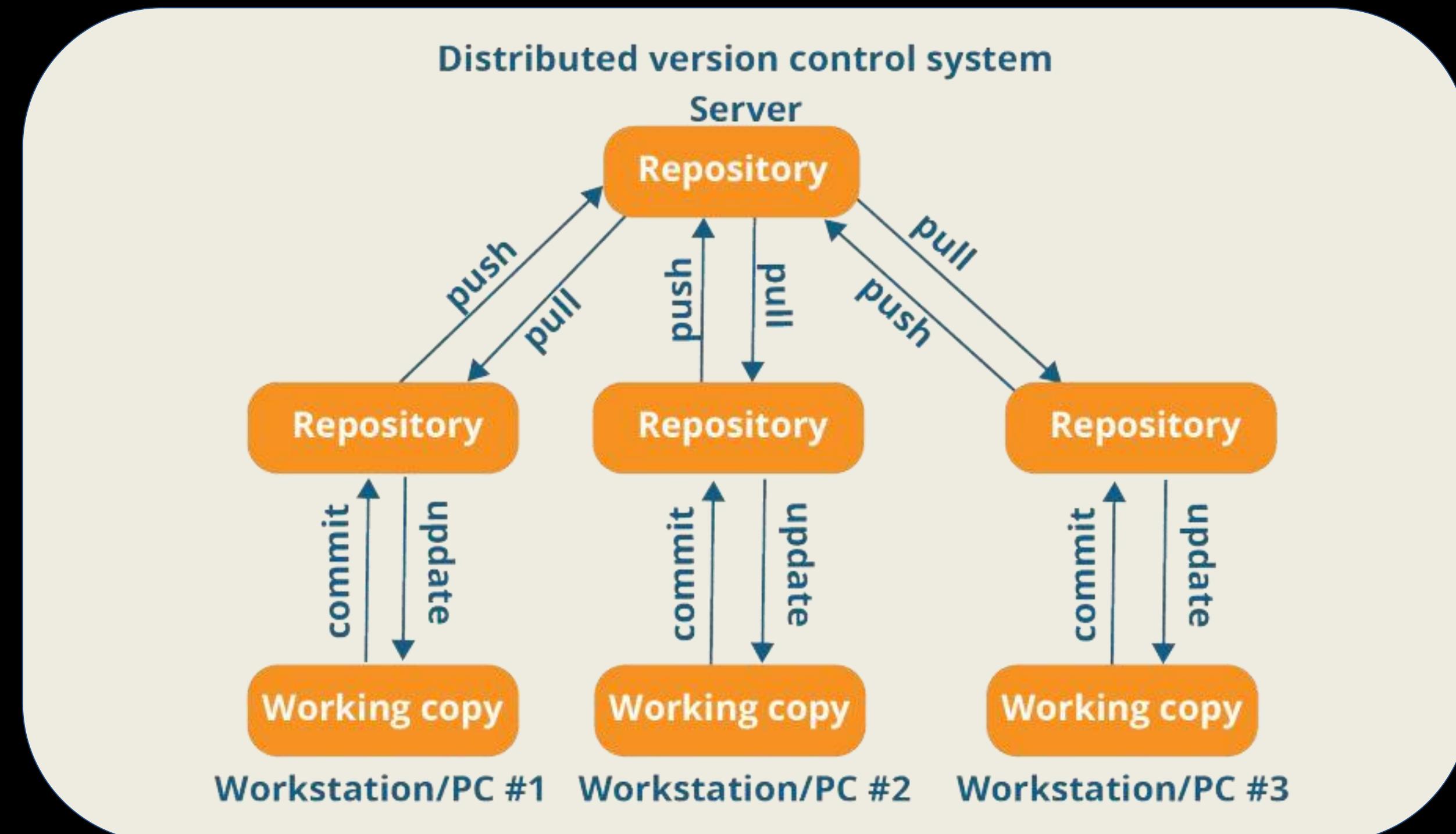


- Los clientes no solo pueden descargar la última copia instantánea de los archivos, sino que se replica completamente el repositorio.
- Cada clon es realmente una **copia completa** de todos los datos.
- Permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados

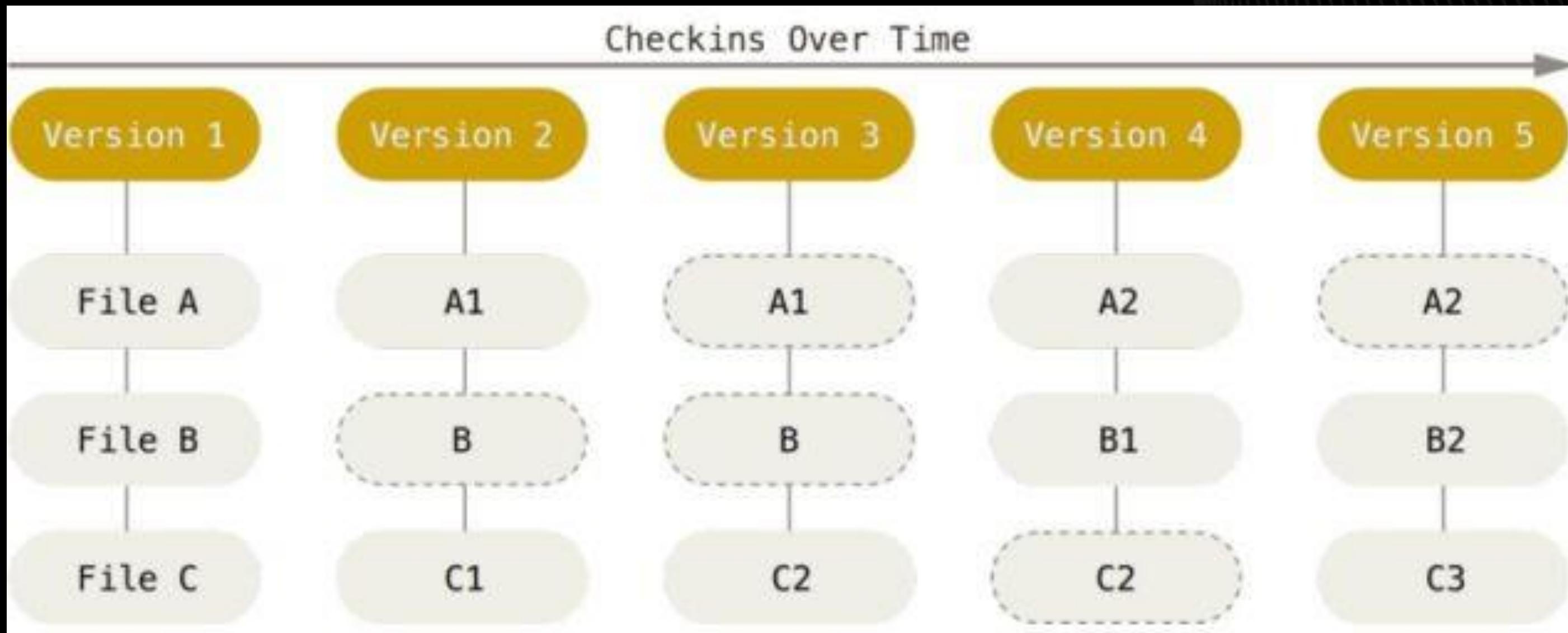
y... ¿GIT que tiene que ver
con todo esto?



Git es un DVCS!



Git es un DVCS!



¿Donde se hace el seguimiento?

Un **repositorio** es un conjunto de carpetas y archivos en el que se almacena el código de un proyecto más el historial de cambios de cada archivo. Git administra los registros a través de archivos en una carpeta oculta llamada **.git**.

 .git	17-03-2021 10:52	Carpeta de archivos
 docker	17-03-2021 10:52	Carpeta de archivos
 docs	17-03-2021 10:52	Carpeta de archivos
 src	17-03-2021 10:52	Carpeta de archivos
 .coveragerc	17-03-2021 10:52	Archivo COVERAG...
 .gitignore	17-03-2021 10:52	Archivo de origen ...
 .travis.yml	17-03-2021 10:52	Archivo de origen ...
 boilerplate.ini	17-03-2021 10:52	Opciones de confi...
 INSTALL.rst	17-03-2021 10:52	Archivo RST
 LICENSE	17-03-2021 10:52	Archivo
 Makefile	17-03-2021 10:52	Archivo
 MANIFEST.in	17-03-2021 10:52	Archivo IN
 PKG-INFO	17-03-2021 10:52	Archivo
 README.rst	17-03-2021 10:52	Archivo RST
 requirements.txt	17-03-2021 10:52	Documento de tex...

Un ejemplo de un repositorio típico de un proyecto basado en python (Fuente: <https://github.com/fabiommendes/python-boilerplate>)

¿Problemas con GIT?

La interfaz de Git no es muy intuitiva y aprender git puede significar una gran dificultad. Por esto, la gente tiende a aprenderse los comandos de memoria, como si se trataran de un hechizo mágico que solucionara sus problema... **Avada Kadabra!!!!**

Pero... si bien la interfaz no es intuitiva y no posee una bonita interfaz, Git es una herramienta muy potente que puede mejorar considerablemente el desarrollo de nuestro proyecto.

```
$ git pull
remote: Counting objects: 216, done.
remote: Total 78 (delta 23), reused 34 (delta 9)
Receiving objects: 100% (78/78), 45.34 KiB, done.
Hey! You know what any of that means? You don't? How about this:
Resolving deltas: 100% (23/23), completed with 23 local objects.
From github.com:phreaky/beammeup
 * [new branch] DoSomeMore    -> origin/DoSomeMore
   76ef09a..4f7865d ShakeIt.m    -> origin/Shakeit.m
 + eac56a3..bf6c107 master      -> origin/master
Haven't got a clue, have you? Go on, try something else...
```

```
$ git diff
diff --git a/src/ShakeIt.m b/src/ShakeIt.m
index 13faa6e..060bc45 100755
--- a/src/ShakeIt.m
+++ b/src/ShakeIt.m
@@ -21,6 +21,7 @@
 // Get MM
 - (CMMotionManager *)motionManager {
     AppDelegate *appDelegate = [UIApplication sharedApplication].delegate;
+ // I changed something here
     return appDelegate.motionManager;
 }
```

Didn't understand any of that did you? Go on, be honest...

```
$ git stfu
```

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



¿Github es lo mismo que Git?

En simples palabras: **NO**

¿En qué se diferencian?



- Git es un sistema de control de versiones distribuido.
- Es una herramienta de línea de comando.
- Crea un repositorio local para trackear los cambios locales.
- Es un software.



- Es un servicio de alojamiento basado en web.
- Entrega una interfaz gráfica.
- Es un espacio para subir una copia de los repositorios de Git.
- Entrega funcionalidades de Git como VCS, añadiendo otras features.

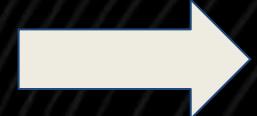
Modelo de Datos de Git

```
1100100100110010011  
00001100110000101001  
1001001100100011001  
0000110010010011001  
10001000100011001001  
  
ETIC_MATERIAL_IS_RES  
LC12N125C12N23C12P19
```

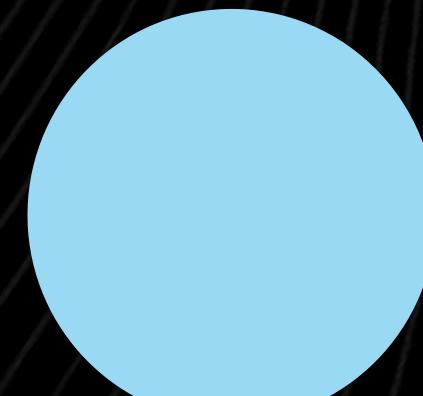
Modelo de Datos de Git

Git modela la historia de una colección de archivos y carpetas dentro de un directorio de nivel superior como una serie de **snapshots**.

```
<root> (tree)
|
+- foo (tree)
|   |
|   + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```



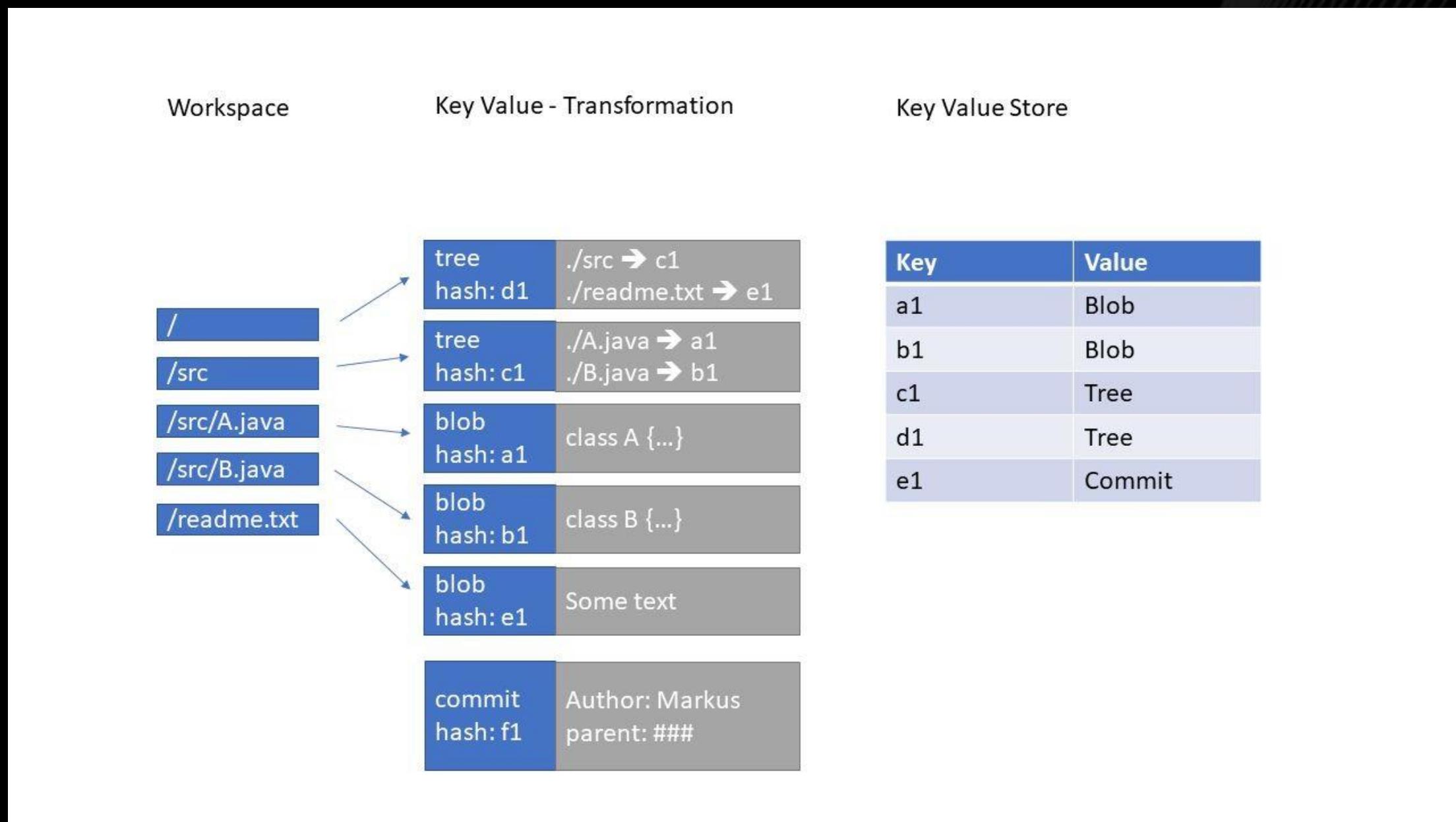
Snapshot



47e1ab

Modelo de Datos de Git

El núcleo de git es una estructura de datos **key-value**. En él, son guardadas cada una de las rutas y archivos del repositorio.



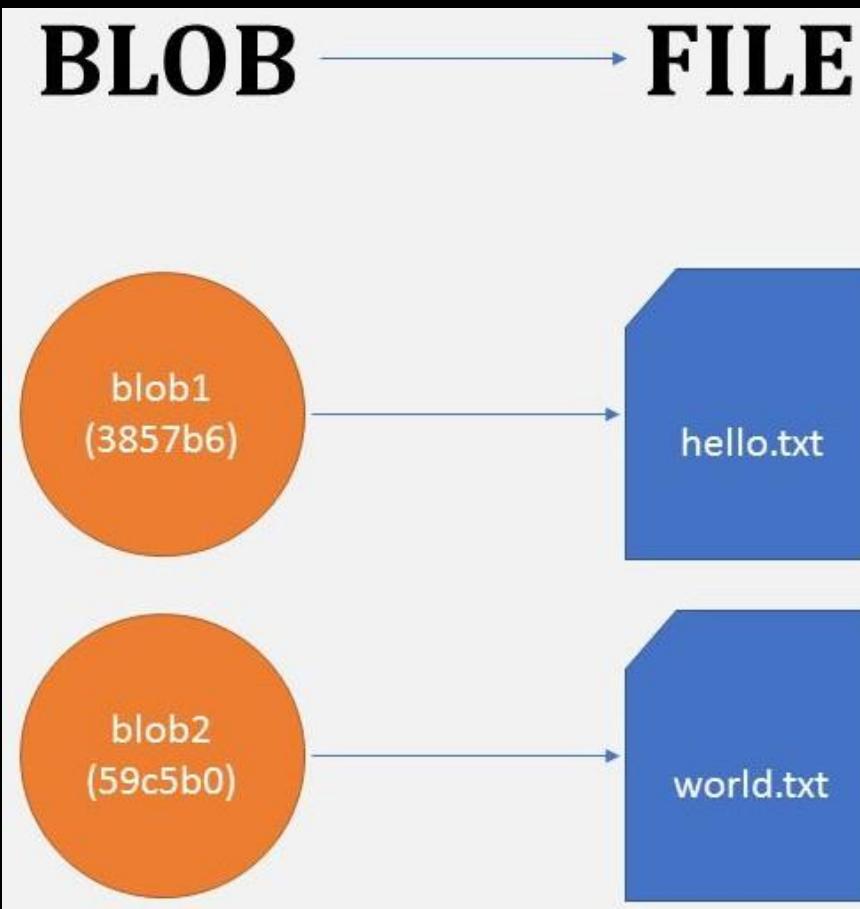
Modelo de Datos de Git

El núcleo de git es una estructura de datos key-value. En él, son guardadas cada una de las rutas y archivos del repositorio.

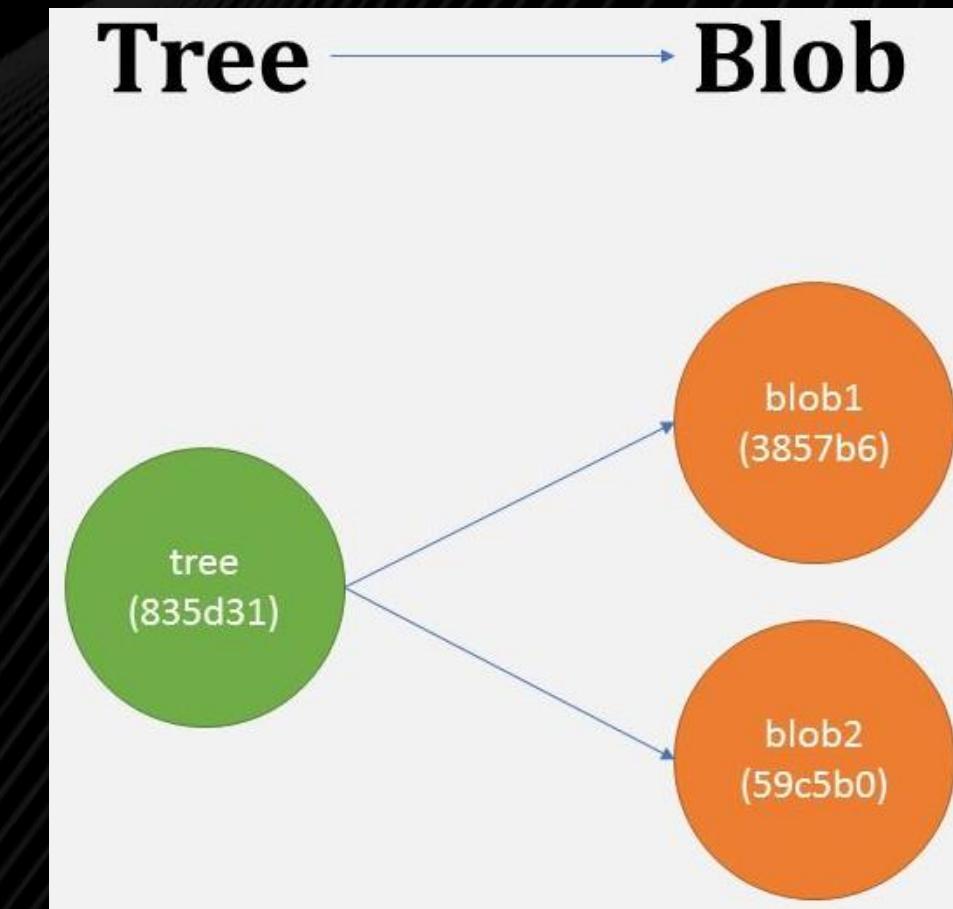
```
[imezadelajara@imezadelajaras-MacBook-Pro local2remote % cd .git
[imezadelajara@imezadelajaras-MacBook-Pro .git % ls
COMMIT_EDITMSG  config          hooks          info          objects
HEAD           description      index          logs          refs
```

Modelo de Datos de Git

- Todo el contenido de un archivo se almacena en una “cosa” llamada **blob**.
- Cuando un archivo es modificado, un nuevo blob almacenará el archivo completo con todos los nuevos cambios.
- Blob es una de las unidades básicas de almacenamiento en Git. Otras unidades/tipos de objetos son Commit y Tree.

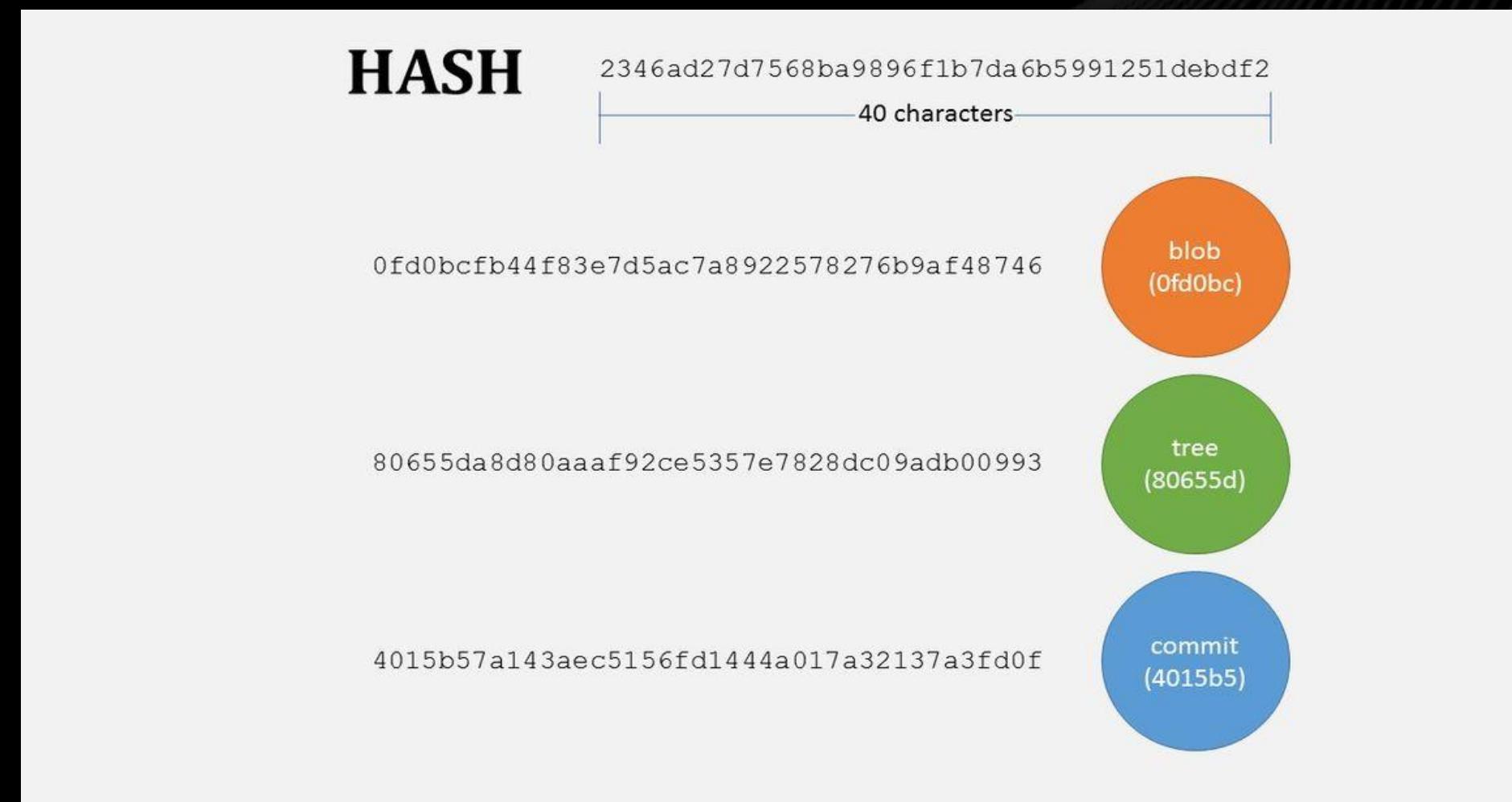


- Los archivos siempre se almacenan en algún directorio o carpeta.
- Las carpetas también pueden contener más directorios.
- Un **árbol** en git representa directorios para blobs y más árboles.
- Siempre hay un árbol en la raíz, apuntando al árbol que contiene cosas.

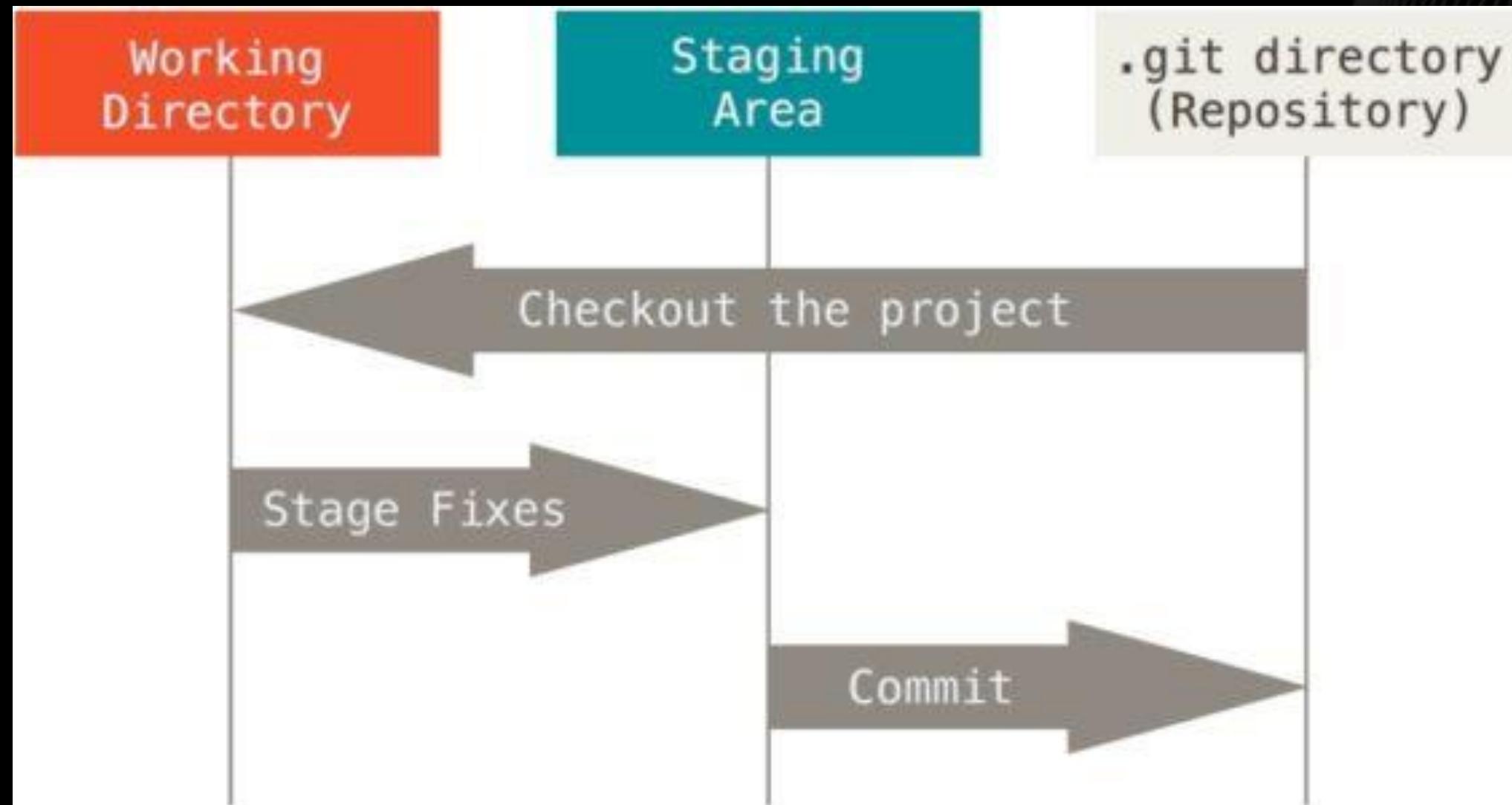


Modelo de Datos de Git

- Un **Hash** puede apuntar a un blob, un commit o un árbol. Tiene 40 caracteres, pero sólo unos pocos suelen ser suficientes para identificar un commit.
- En Git se utiliza el hash **SHA1**.
- Cada elemento posee un hash único, ya que son elementos que nos sirven para referenciar a un elemento de interés.



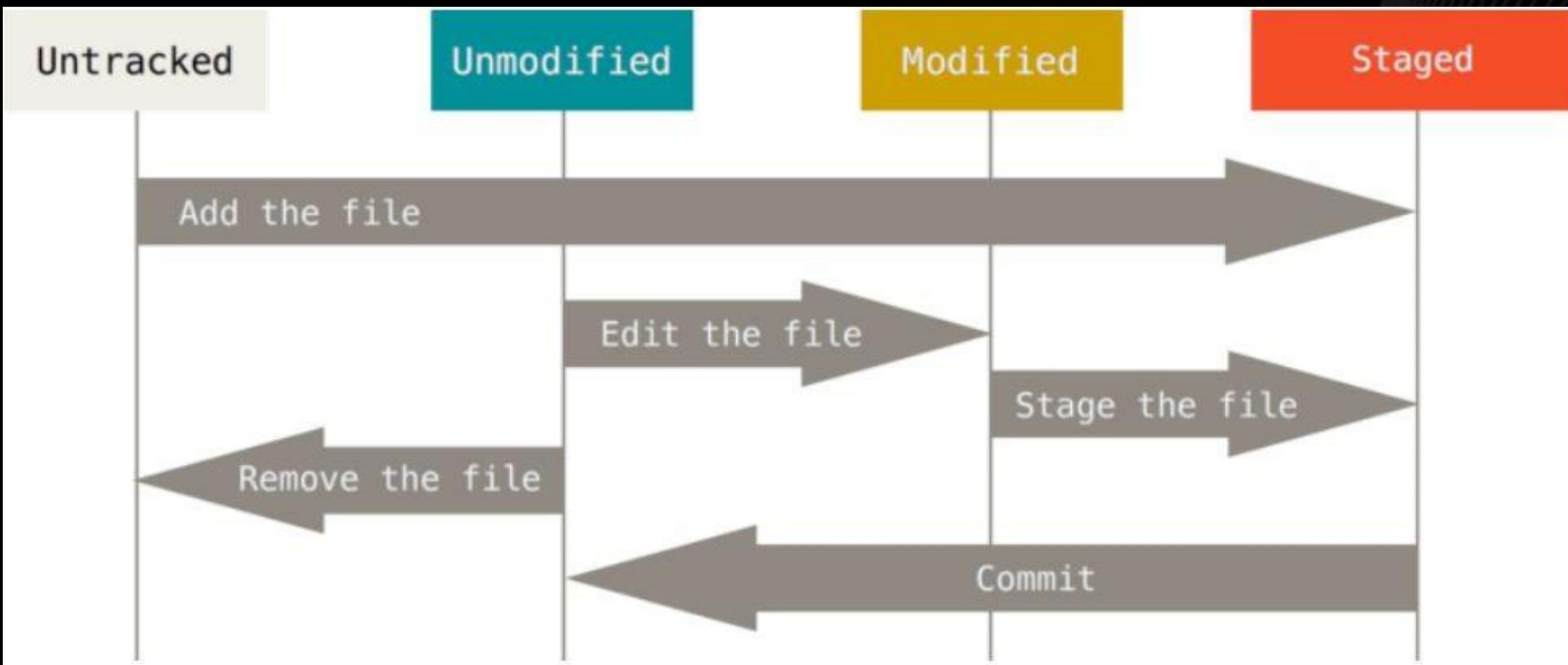
Los Tres Estados



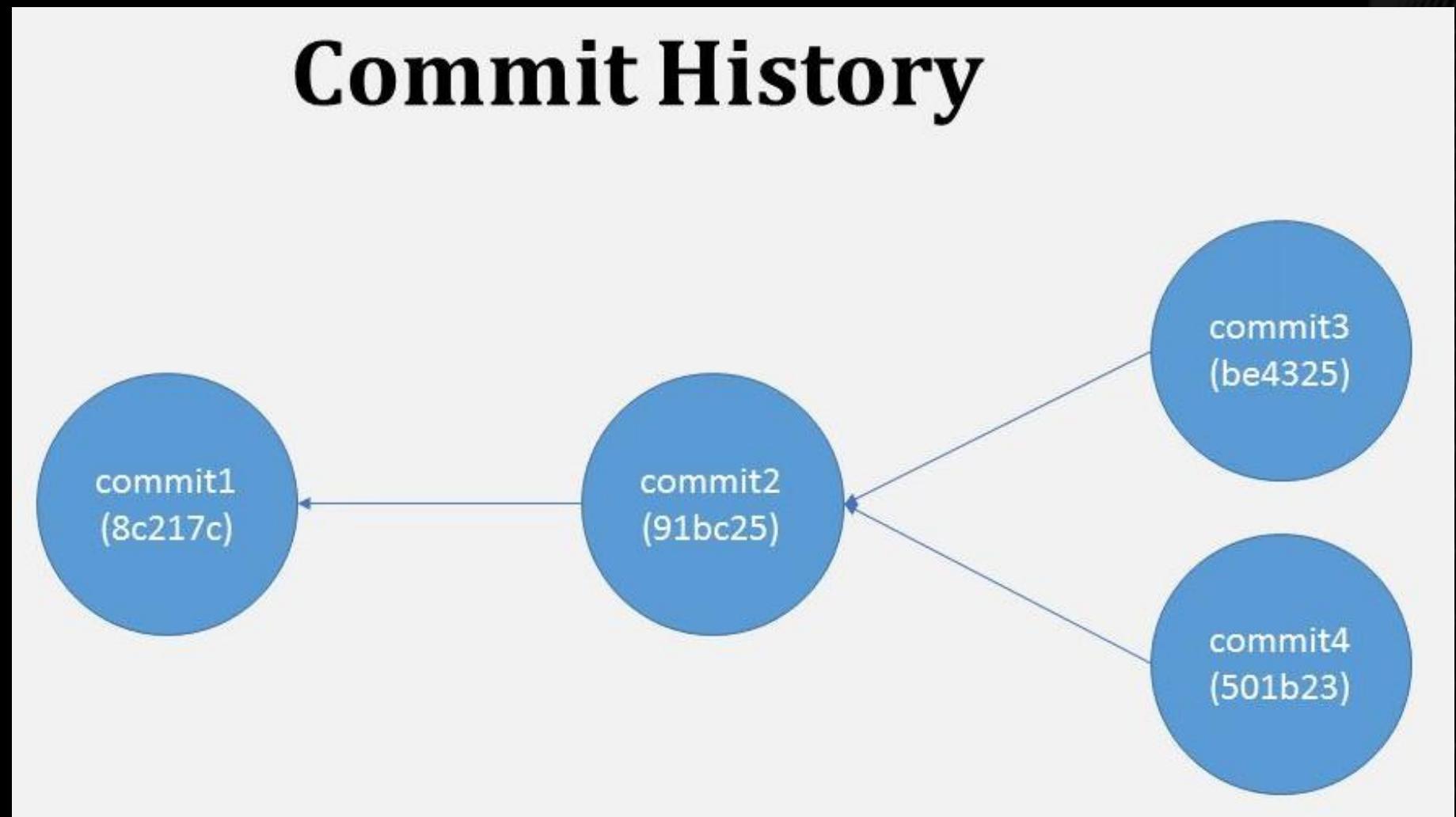
Staging Area
La staging area consiste en el conjunto de archivos modificados que aún no han sido aprobados para formar parte del repositorio ni de su registro de cambios, pero que están a un paso de serlos.

Realizar un **commit** permite capturar una instantánea de los archivos staged del repositorio y agregarlos a su historial de cambios.

Ciclo de vida del estado de los archivos



Modelo de Datos de Git



La idea general de git es ir guardando snapshots/capturas (commits) en el historial que registran y consolidan los cambios.

Primeros pasos con Git



Instalación

La instalación de git la pueden hacer a través de las siguientes formas:

- **Windows:** Instalador que encuentran en la página oficial.
- **MacOS:**

```
>- brew install git
```

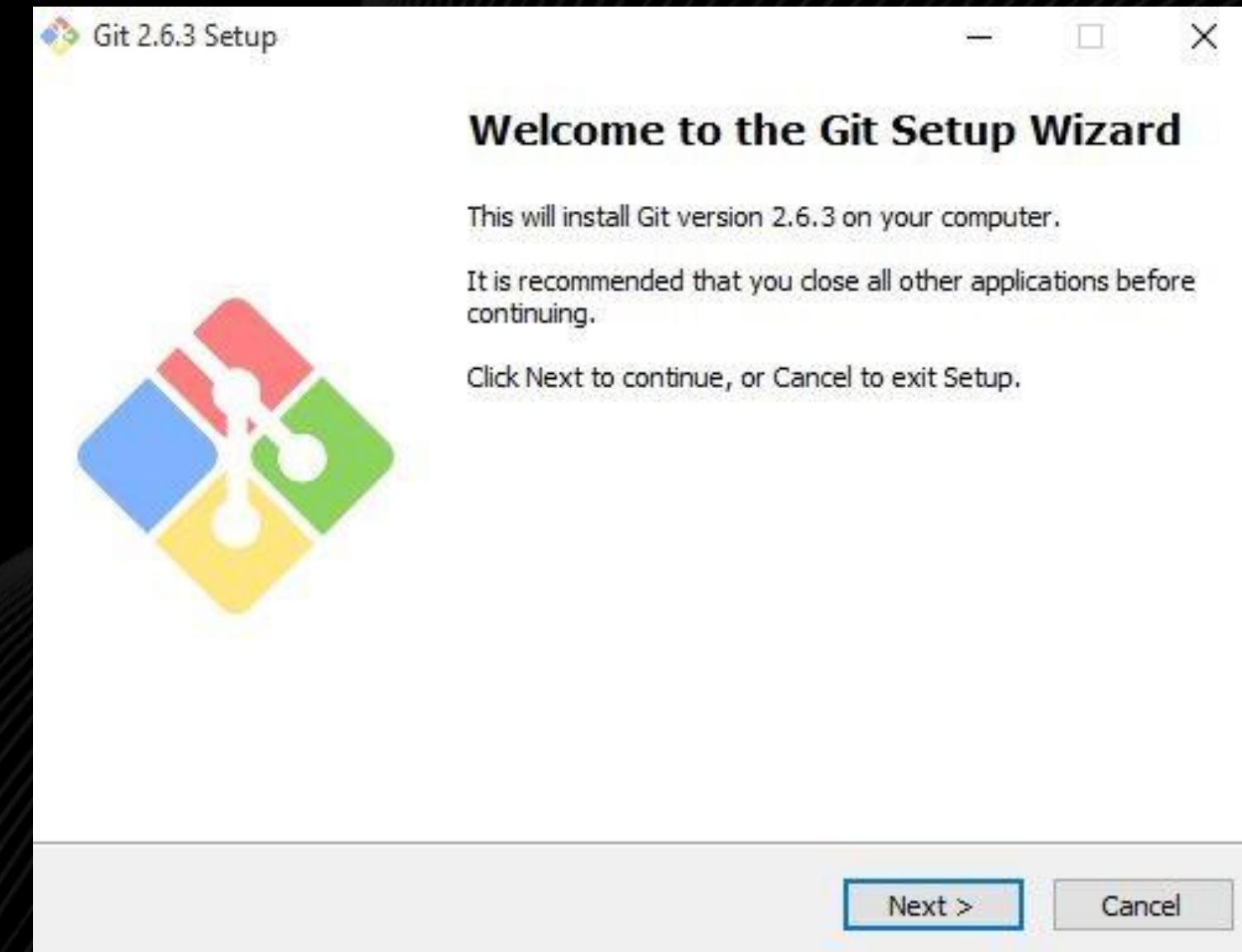
- **Linux (ubuntu):**

```
>- sudo apt-get update  
    sudo apt-get install git
```

Una vez realizada, es necesario configurar la identidad de quien lo va a utilizar.

Esto se hace por medio de:

```
>- $ git config --global user.name "Nombre-Usuario"  
    $ git config --global user.email mail@ejemplo.xyz
```



¿Formas de Usar Git?

Hay dos formas de utilizar git:

- Desde la consola de comandos.
- Desde alguna aplicación con interfaz visual (como [Github desktop](#), [Gitkraken](#) o incluso a través de plugins de su editor de código favorito).

Si bien las interfaces gráficas permiten manejar las propiedades de Git de una forma mas intuitiva, estas aplicaciones no poseen todas las features que entrega la línea de comando. Por otro lado, si manejas la línea de comando, sabrás utilizar alguna interfaz gráfica pero no así al revés.

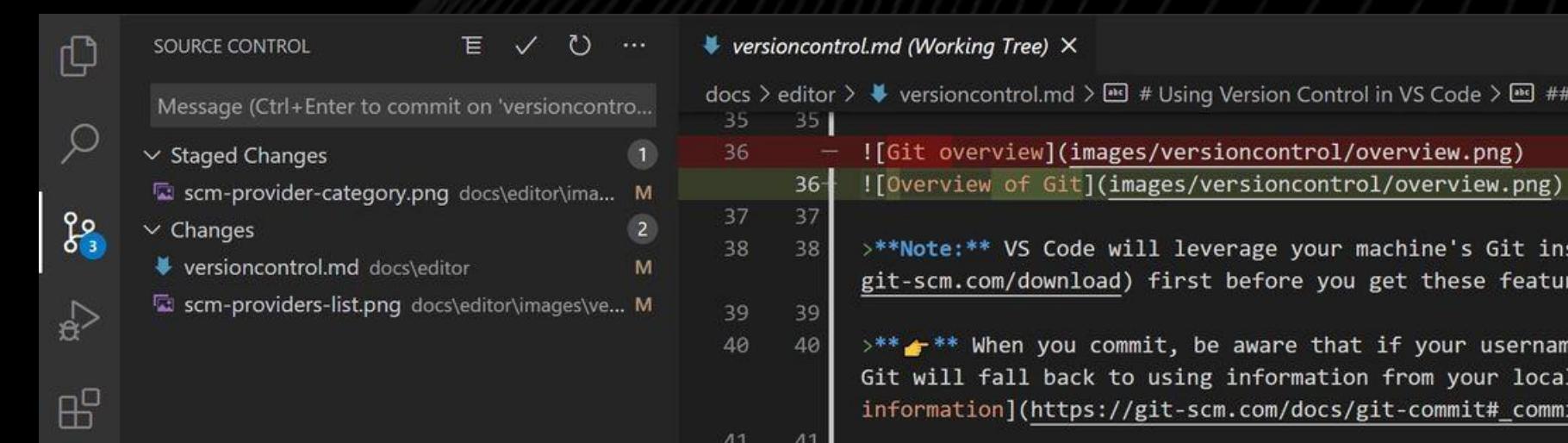
```
jnk@JUAN-GUANA MINGW64 ~/Documents/primerospasosgit (main)
$ git add index.html

jnk@JUAN-GUANA MINGW64 ~/Documents/primerospasosgit (main)
$ git commit -m "formulario agregado"
[main e2793ec] formulario agregado
 1 file changed, 18 insertions(+)

jnk@JUAN-GUANA MINGW64 ~/Documents/primerospasosgit (main)
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 529 bytes | 529.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/JuanGuana/primerospasosgit.git
 b016906..e2793ec main -> main

jnk@JUAN-GUANA MINGW64 ~/Documents/primerospasosgit (main)
$ |
```

Línea de Comandos



Interfaz Gráfica

Demo: nuestros primeros pasos

Disclaimer : Desde aquí en adelante hablaremos solo de nuestro repositorio local

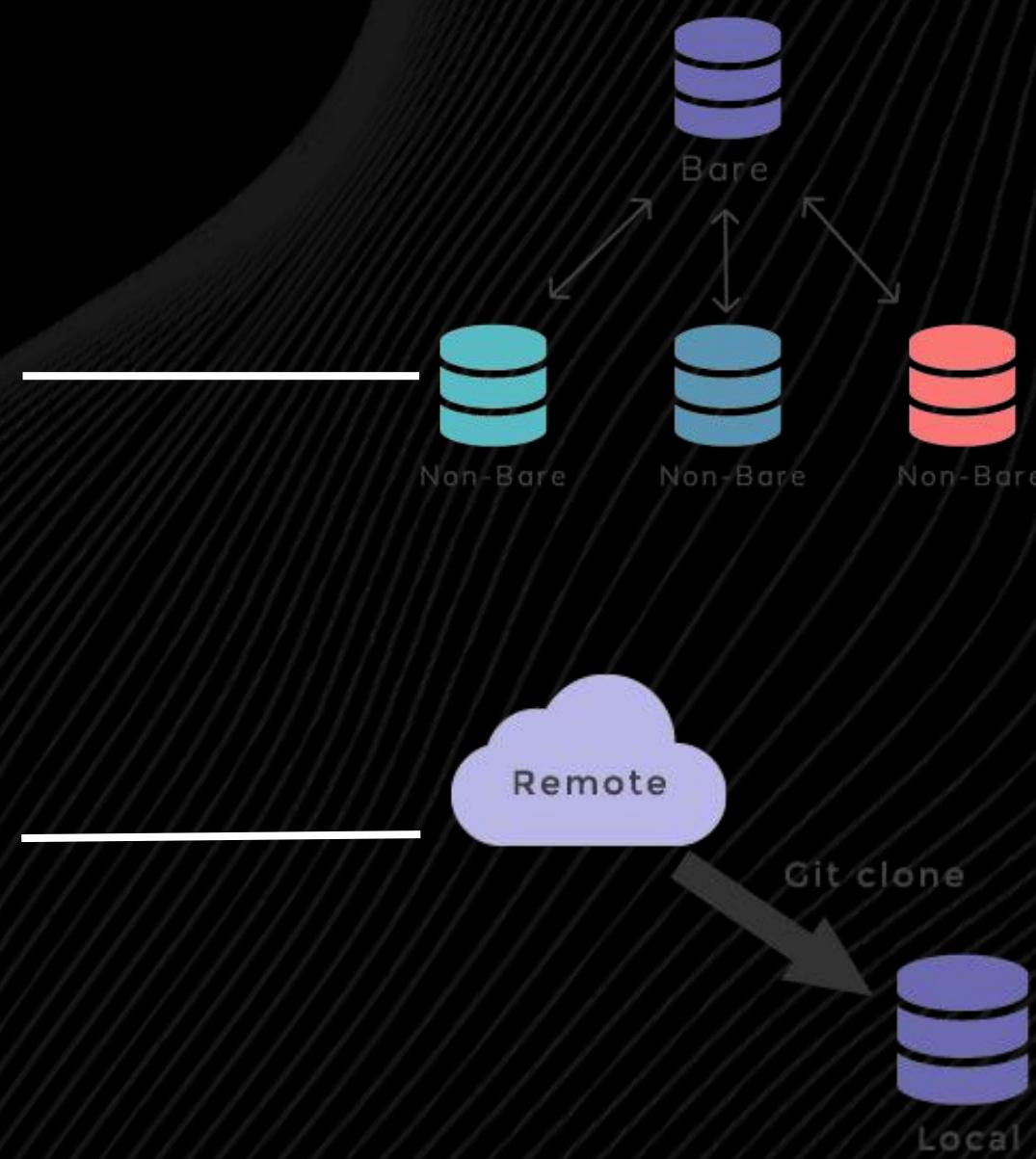
- >- • Para inicializar un repositorio desde cero, usamos:

```
>> git init nombre-del-repo
```

- Para clonar un repositorio (es decir, copiar un repositorio a tu almacenamiento local desde algún servidor github, gitlab, bitbucket u otro servidor de git):

```
>> git clone /ruta/al/repo.git
```

- Para inicializar un repositorio en algún servicio de repositorios remotos, como github.



Demo: nuestros primeros pasos

```
>- git status
```

Una operación simple de entender es el comando status. Este permite tener una idea del estado actual del repositorio. Este se ejecuta por medio del comando :

```
>>git status
```

Al clonar el repositorio, por defecto veremos el último commit. Por ejemplo, para el proyecto ejemplo_MDS7202 que tiene ya varios archivos:

```
To https://github.com/pbadillatorrealba/ejemplo_MDS7202.git
  fe4c5fc..52793f8  main -> main
(base) PS C:\Users\pablo\Desktop\ejemplo_MDS7202> git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Demo: nuestros primeros pasos

```
>- git add
```

Para que un archivo que ha sido modificado (modified) o untracked pase a ser parte de la staging area, debe estar marcado como staged. Esto puede ser realizado a través de git add:

Caso de un archivo:

```
>>git add path/to/file.py
```

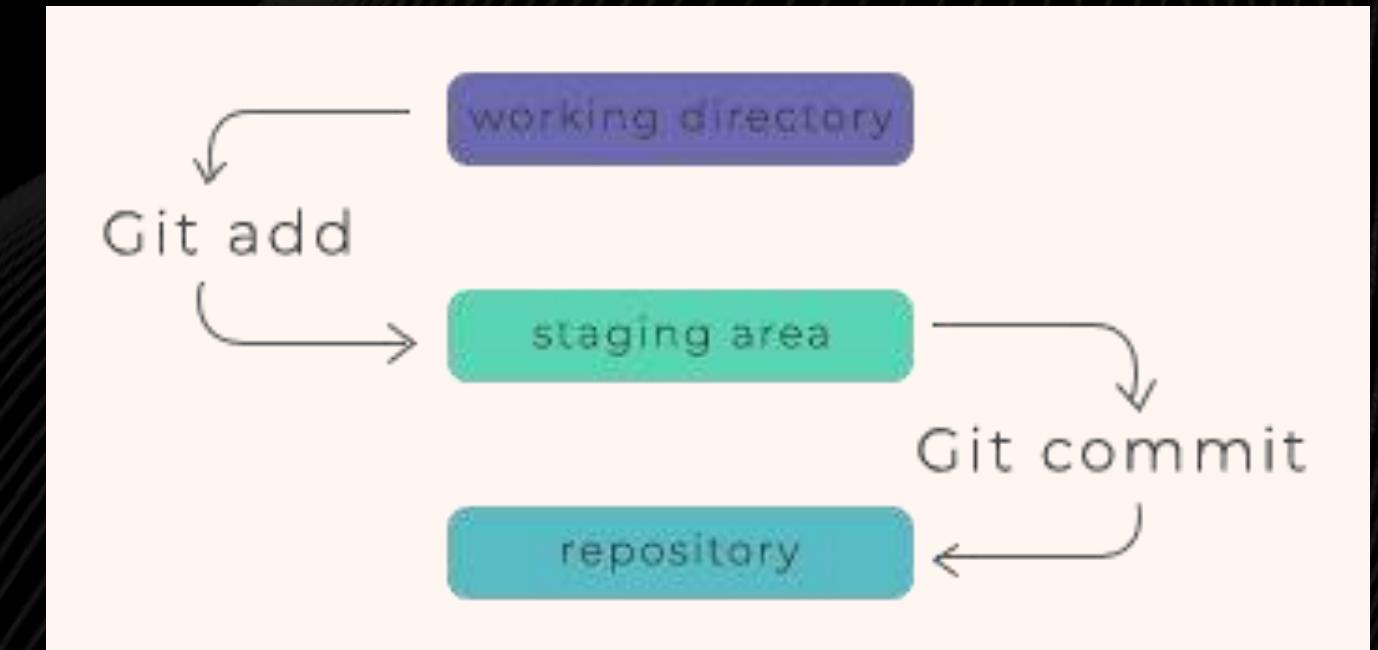
Caso de una carpeta:

```
>>git add path/*
```

Caso de todos los cambios:

```
>>git add --all
```

¿Qué pasa si agrego un archivo al staging area y después lo modiflico? 😅😅



Demo: nuestros primeros pasos

>- git commit

Realizar un commit permite capturar una instantánea de los archivos staged del repositorio y agregarlos a su historial de cambios.

Para ello utilizó:

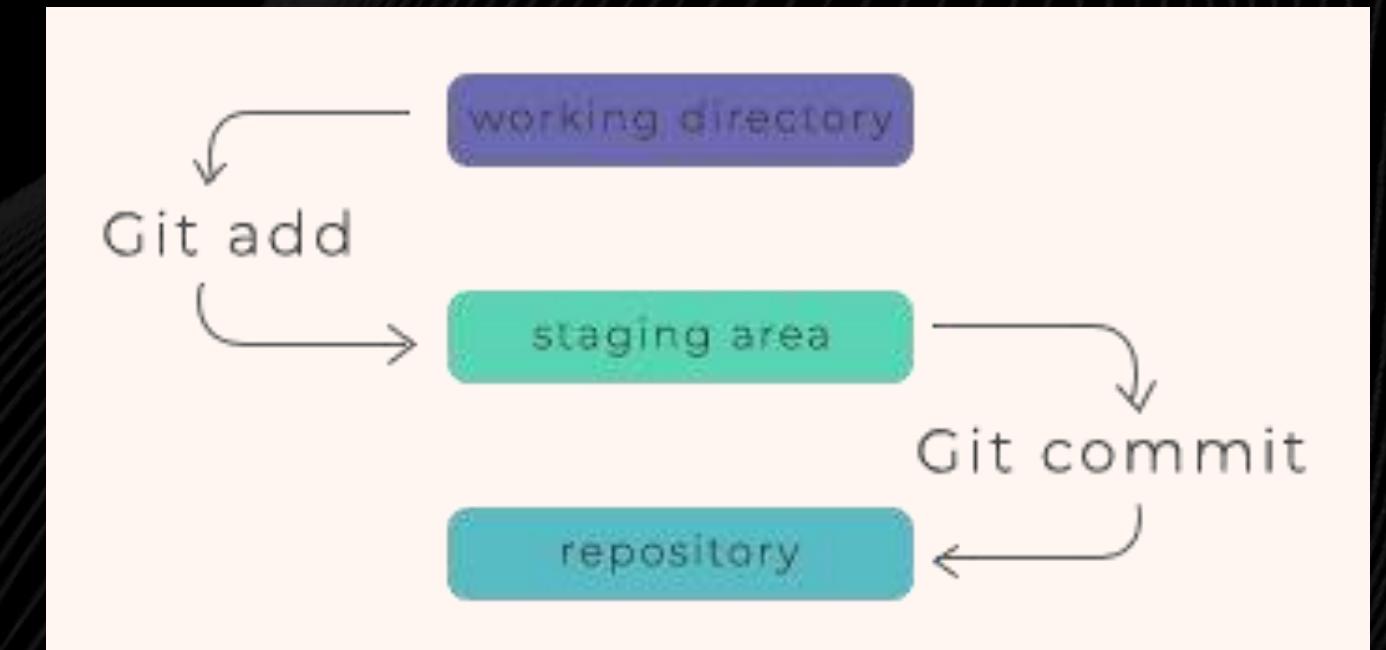
>>git commit

Esto abrirá un archivo en su editor de texto predeterminado donde deberán dejar un mensaje de que cambios hicimos y porque.

Si desean dejar una descripción pequeña de los cambios, simplemente podemos agregar el parámetro **-m**

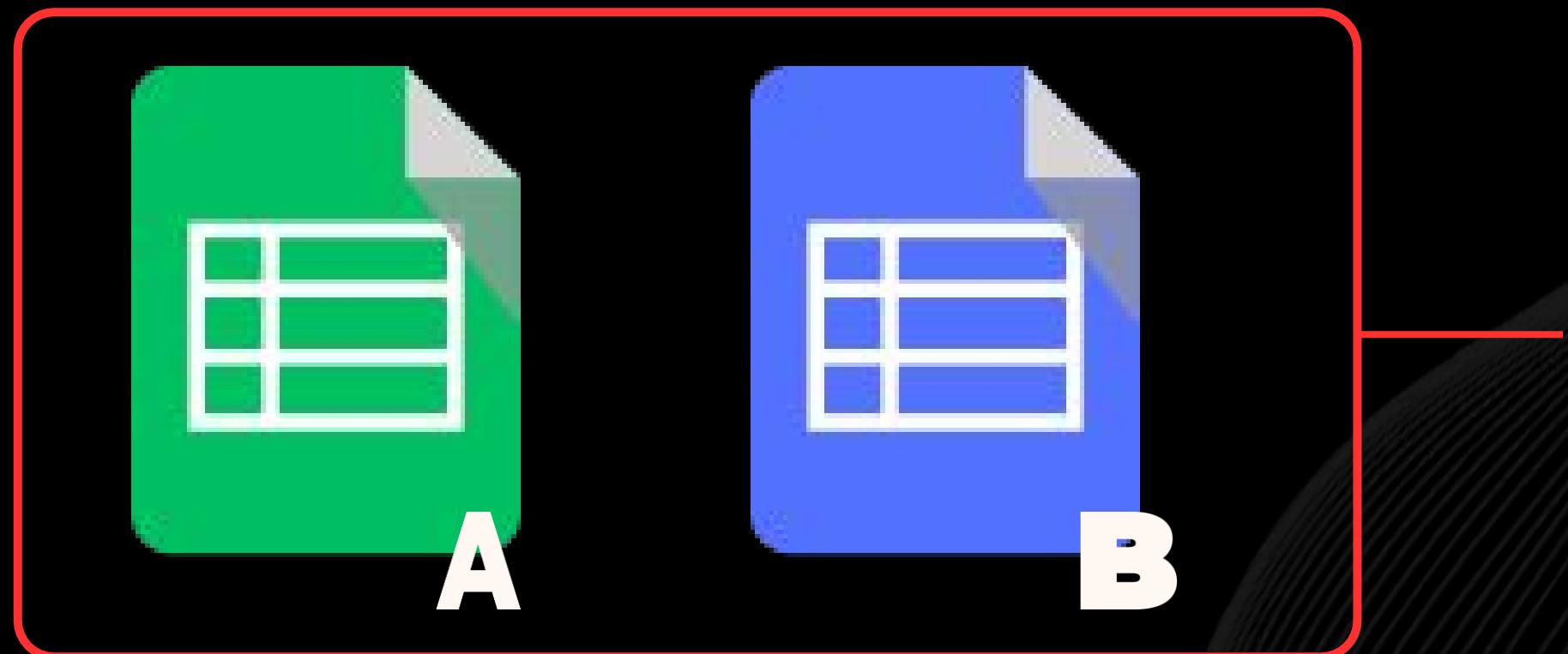
>>git commit -m "Mi mensaje aquí"

Hacer un commit hará que los archivos staged pasen a ser unmodified.



¿El commit perfecto existe?

- Probablemente No, pero si existen buenas practicas para realizar un commit.

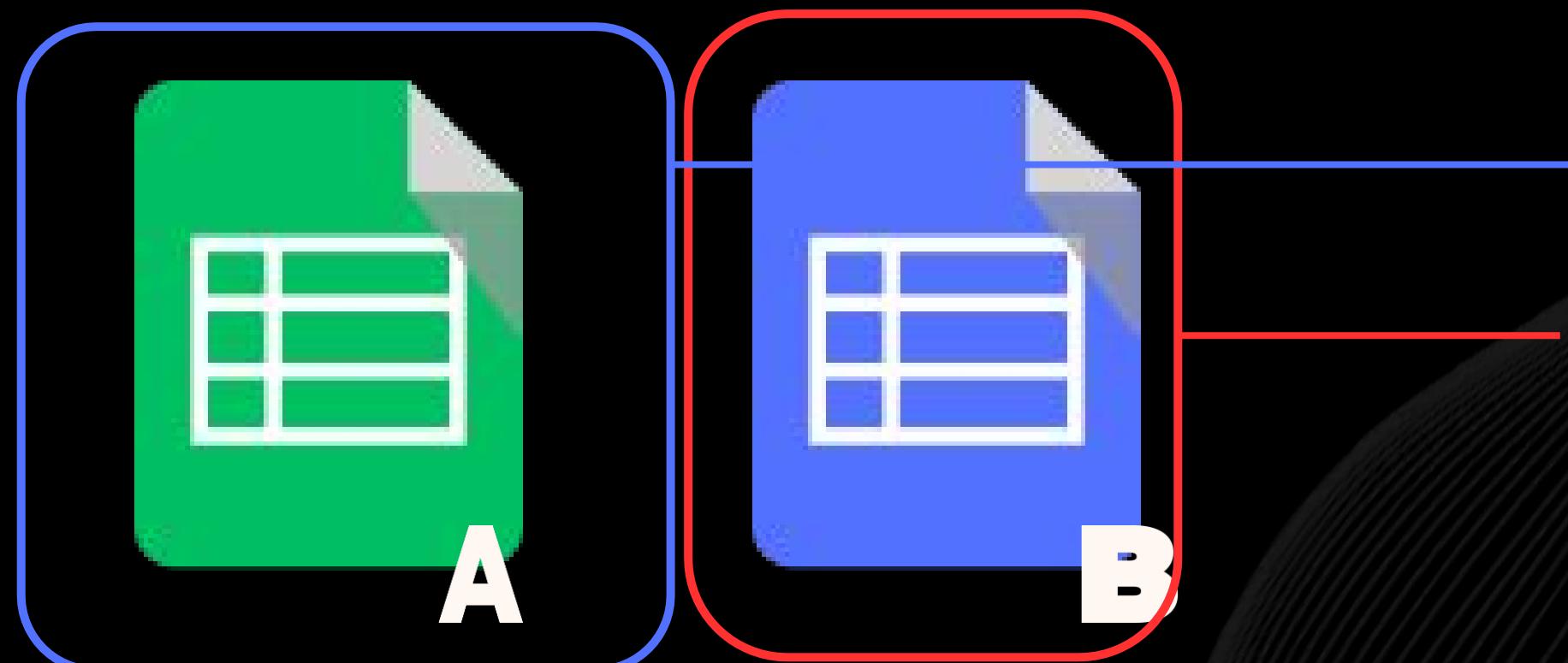


`git add .`
`git commit -m 'the best commit'`

¿Qué sucede si realizamos un commit de dos archivos diferentes?

¿El commit perfecto existe?

- Probablemente No, pero si existen buenas practicas para realizar un commit.



```
git add file A  
git commit -m 'the best commit 1'  
git add file B  
git commit -m 'the best commit 2'
```

¿El commit perfecto existe?

En el caso ideal:

- Utilizar **git add** de acuerdo a la funcionalidad que tienen el archivo que quieren respaldar.
- Los **commits deben ser concisos**.
- Si es necesario utilizar el **body** para generar una explicación más detallada.

Ejemplo de mal uso de commits

	COMMENT	DATE
O	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
O	ENABLED CONFIG FILE PARSING	9 HOURS AGO
O	MISC BUGFIXES	5 HOURS AGO
O	CODE ADDITIONS/EDITS	4 HOURS AGO
O	MORE CODE	4 HOURS AGO
O	HERE HAVE CODE	4 HOURS AGO
O	AAAAAAA	3 HOURS AGO
O	ADKFJSLKDFJSOKLFJ	3 HOURS AGO
O	MY HANDS ARE TYPING WORDS	2 HOURS AGO
O	HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Comandos útiles

git diff

Una herramienta potente en el sistema Git es el comando diff, este permite obtener una visualización con la cual es posible observar las diferencias entre conjuntos de archivos en el repositorio.

Nota: *git diff* devuelve la diferencia entre el directorio de trabajo y el staging area

```
(ds) PS C:\Users\pablo\Desktop\ejemplo_MDS7202> git diff
diff --git a/main.py b/main.py
index c526340..60c1425 100644
--- a/main.py
+++ b/main.py
@@ -1,4 +1,5 @@
 from src.some_func import print_hola

 if __name__ == "__main__":
-    print_hola()
 \ No newline at end of file
+    print_hola()
+    print("¿Qué tal?")
```

En este caso, a y b indican la primera y segunda versión del archivo mostrado. Las líneas con - indican líneas borradas, mientras que + indica líneas agregadas. La sección demarcada con @@ contiene coordenadas linea inicio, numero de líneas.

Comandos útiles

>- git log

Para observar el registro de cambios se utiliza git log, al ejecutar este comando se aprecia la estructura de un commit: este consta de un autor, momento de realización de la modificación y un hash identificador del cambio.

```
(ds) PS C:\Users\pablo\Desktop\ejemplo_MDS7202> git log
commit 369ad5b781e995bfcf7b0bdd8c0235bbaa79ad7d (HEAD -> main)
Author: Pablo Badilla <pablo.badilla@ug.uchile.cl>
Date:   Wed Mar 17 12:21:26 2021 -0300

    Class new class

commit fe4c5fc703429d830fa7f51965a2607f89fb905c (origin/main, origin/HEAD)
Author: Pablo Badilla <pablo.badilla@ug.uchile.cl>
Date:   Wed Mar 17 11:28:30 2021 -0300

    Initial commit
```

¿Puedo omitir archivos automáticamente?

En algunos proyectos, como producción de archivos en LaTex, se generan archivos colaterales como logs y pdfs. Es posible ignorar este tipo de archivos en el repositorio. Para ello se genera un archivo `.gitignore` en la carpeta raíz listando archivos y carpetas no deseados en cada línea.



160 lines (131 sloc) | 3.01 KB

```
1 # Byte-compiled / optimized / DLL files
2 __pycache__
3 *.py[cod]
4 *$py.class
5
6 # C extensions
7 *.so
8
9 # Distribution / packaging
10 .Python
11 build/
12 develop-eggs/
13 dist/
14 downloads/
15 eggs/
16 .eggs/
17 lib/
18 lib64/
19 parts/
20 sdist/
21 var/
22 wheels/
23 share/python-wheels/
24 *.egg-info/
25 .installed.cfg
26 *.egg
27 MANIFEST
28
29 # PyInstaller
30 # Usually these files are written by a python script from a template
31 # before PyInstaller builds the exe, so as to inject date/other infos into it.
32 *.manifest
33 *.spec
34
35 # Installer logs
36 pin-log.txt
```

Raw Blame ⌂ ⌂

Deshacer Cambios



Eliminar del área de montaje

¿Qué ocurre si se agrega accidentalmente un archivo al área de montaje?

>>git reset folder/file

Deshacer cambios de un archivo:

Si por otra parte, un archivo folder/file ya fue consolidado y se desean revertir los cambios al commit base es posible usar el comando

>>git checkout -- folder/file

Volver a una versión (commit) anterior de un archivo:

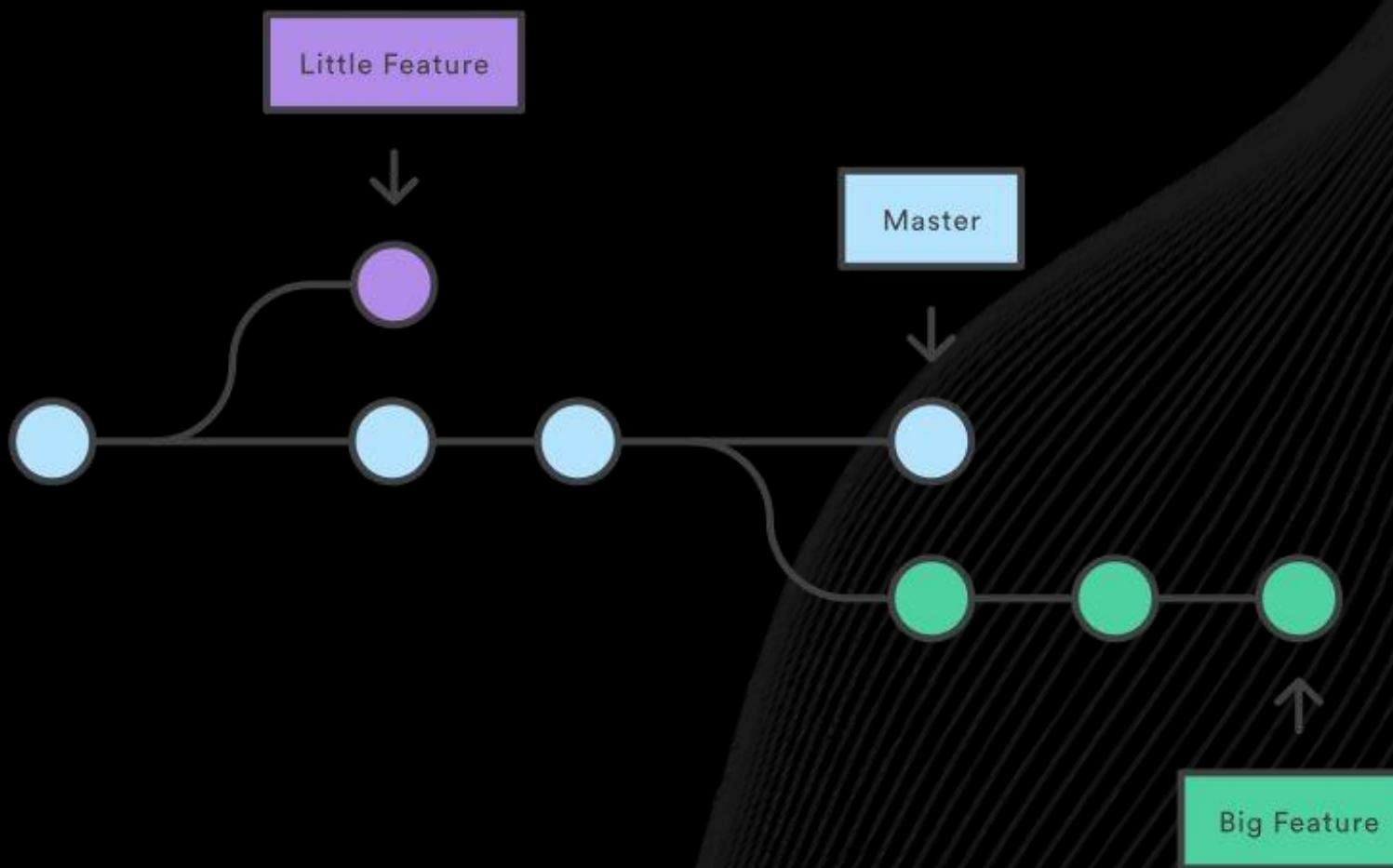
>>git checkout commit-hash folder/file

Y si quieres retornar todos los archivos a un commit anterior, puedes usar:

>>git checkout commit-hash

Branches

- Una rama o branch es un mecanismo que nos permite trabajar en un ambiente independiente del código principal.
- **Al crear una branch, se "replican" los elementos de la rama desde donde se originó.** Todos los commits que hagamos se harán en la branch que creamos. Luego, podemos juntar los cambios a través de un **merge**.



¿Que nos permite esto?

¿Existe alguna regla para las ramas?

Type	Description
hotfix	para solucionar rápidamente problemas críticos, generalmente con una solución temporal
bugfix	para corregir un error
feature	para añadir, eliminar o modificar una característica
test	para experimentar algo que no es un problema
wip	para un trabajo en progreso

¿Como uso las ramas?

- » Para ver las ramas disponibles en el repositorio.

```
>>git branch
```

- » Crear ramas: El siguiente comando permite crear una rama con nombre new-branch.

```
>>git branch new-branch
```

- » Cambiar de ramas: Para cambiar a la nueva rama, hacemos uso de

```
>>git checkout new-branch
```

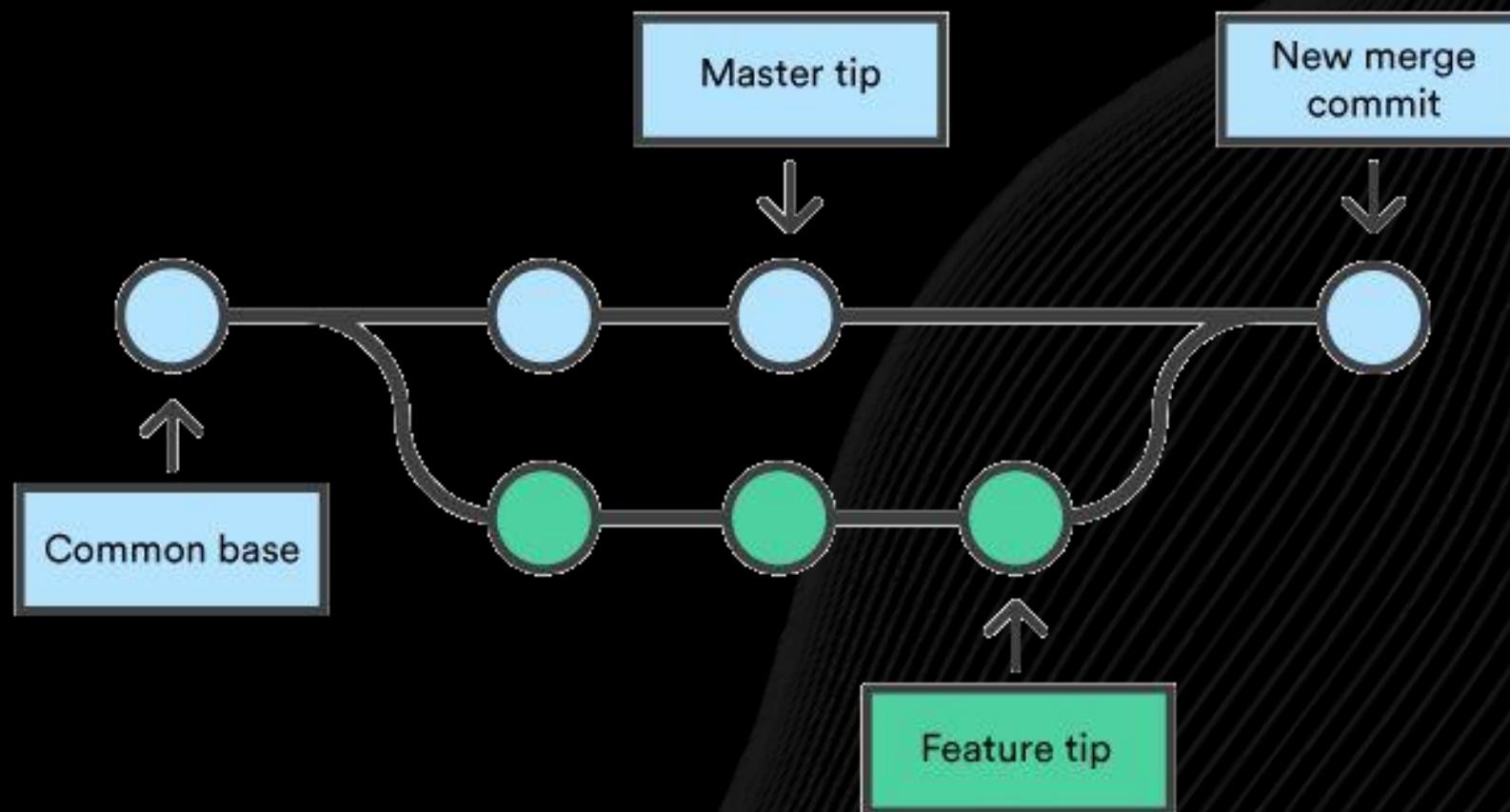
- » Un atajo para crear una nueva rama y acceder inmediatamente es el comando:

```
>>git checkout -b new-branch
```

Merge

- La gran ventaja de trabajar con ramas viene de unir posteriormente los resultados y registros, esto se denomina **merging**. Para unir dos ramas se utiliza el comando:

```
>>git merge source_branch destination_branch
```



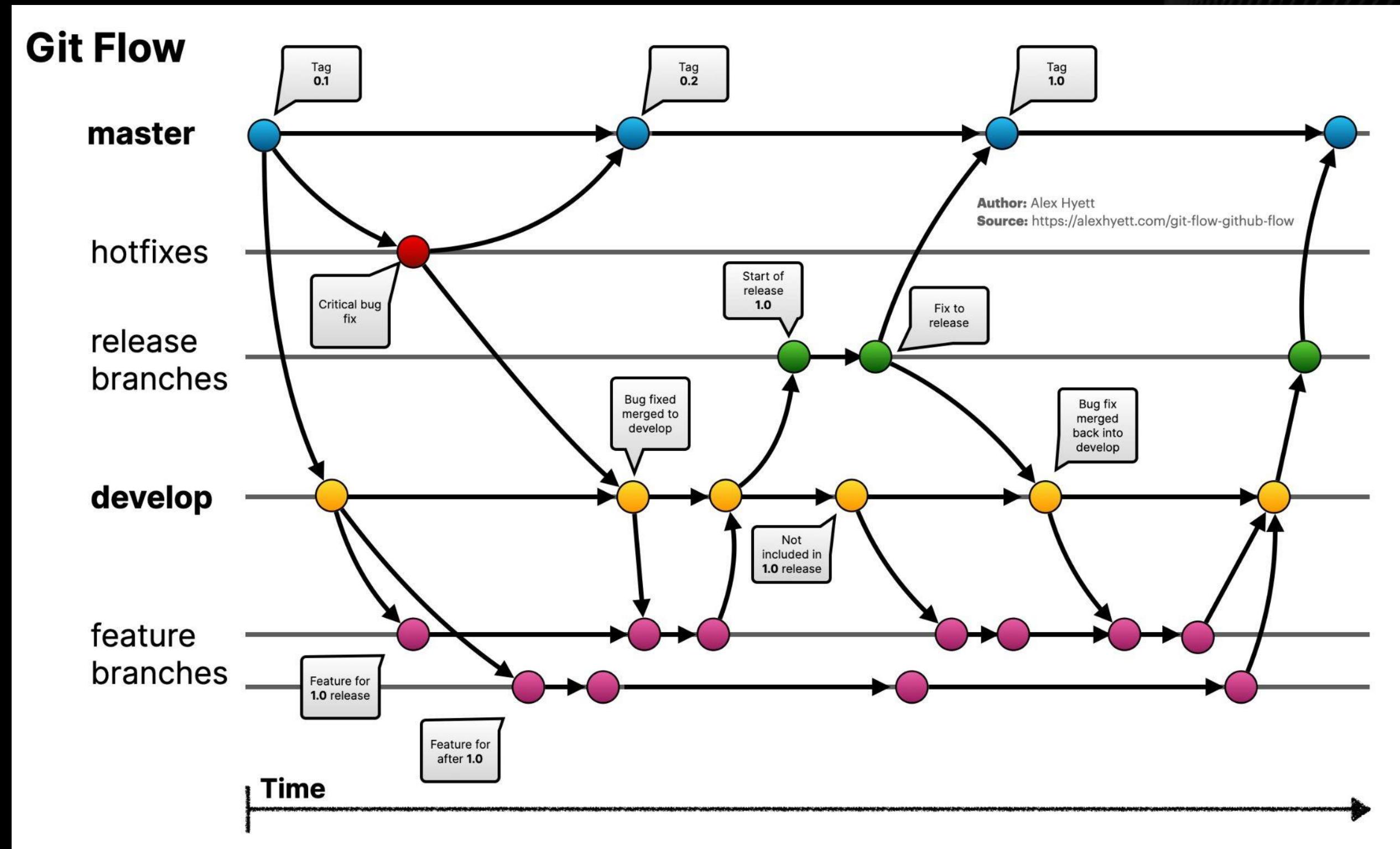
¿Podría generar
problema esto?

Merge

- » Es posible que existan colisiones en el trabajo realizado dentro de distintas ramas. Estas ocurren cuando 2 personas trabajan en las mismas líneas de código y luego intentan unir sus cambios. Estas colisiones se denominan **conflictos**.
- » **Comúnmente uno deberá resolver estos conflictos a mano y luego subir los cambios.**
- » En el siguiente ejemplo se muestra que es lo que ocurre cuando git detecta conflicto que no pudo resolver por cuenta propia:

```
1  class NewClass:  
2      Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes | Start Live Share Session  
3      <<<<< HEAD (Current Change)  
4          a = 0  
5          b = a * 0  
6          c = 1  
7            
8          b = 2  
9          c = 3  
10         >>>>> clase_app (Incoming Change)
```

Modelo de Datos de Git



Repositorios Remotos



- El comando `git clone ruta` permite nombrar el repositorio clonado por medio de `git clone ruta nombre_clon`.

La ventaja de este sistema de almacenamiento de rutas remotas, es que permite sincronizar el trabajo tanto en una máquina local, como en colaboraciones por medio de internet.

➢ Pull

La comunicación entre repositorios se hace por medio de instrucciones especiales, una de ellas es pull.

`git pull` permite obtener el registro de cambios de un repositorio remoto, e incluso, una rama de tal repositorio. La sintaxis es:

```
>>git pull repo_remoto rama
```

Con este comando, se toma toda la información del repositorio `repo_remoto` y se une directamente con el repositorio actual (merging).

➢ Push

Por otra parte, es posible enviar cambios a un repositorio remoto, esto se hace por medio de `git push`.

```
>>git push repo_remoto rama
```

¿Pull Request (PR)?

updated calendar and readme #4

Merged Mezosky merged 1 commit into main from update_calendar now

Conversation 0 Commits 1 Checks 0 Files changed 2

sebatinoco commented 3 hours ago

Updating calendar and readme for current iteration (Fall 2024)

updated calendar and readme b35182a

sebatinoco requested a review from Mezosky 3 hours ago

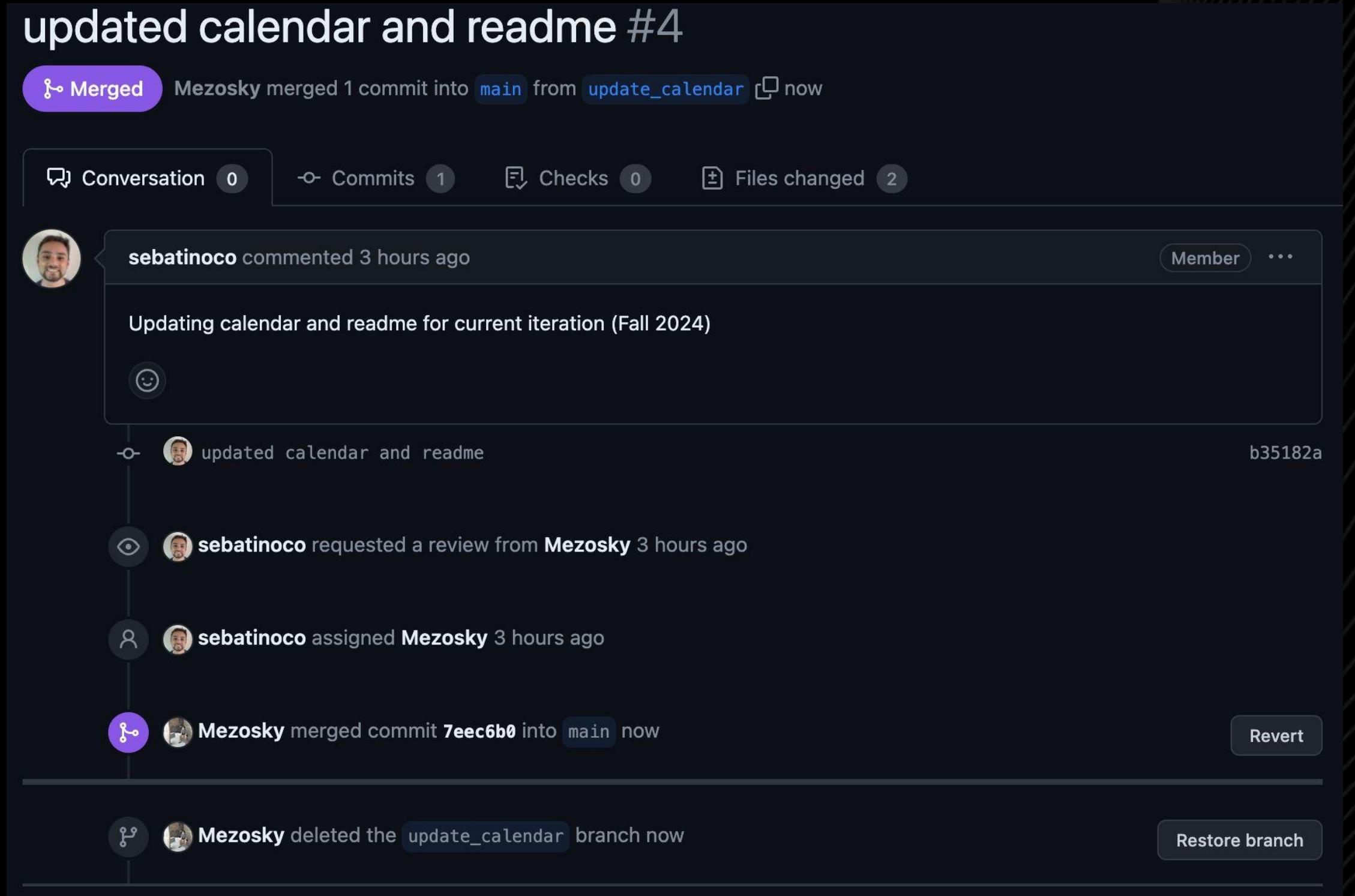
sebatinoco assigned Mezosky 3 hours ago

Mezosky merged commit 7eec6b0 into main now

Revert

Mezosky deleted the update_calendar branch now

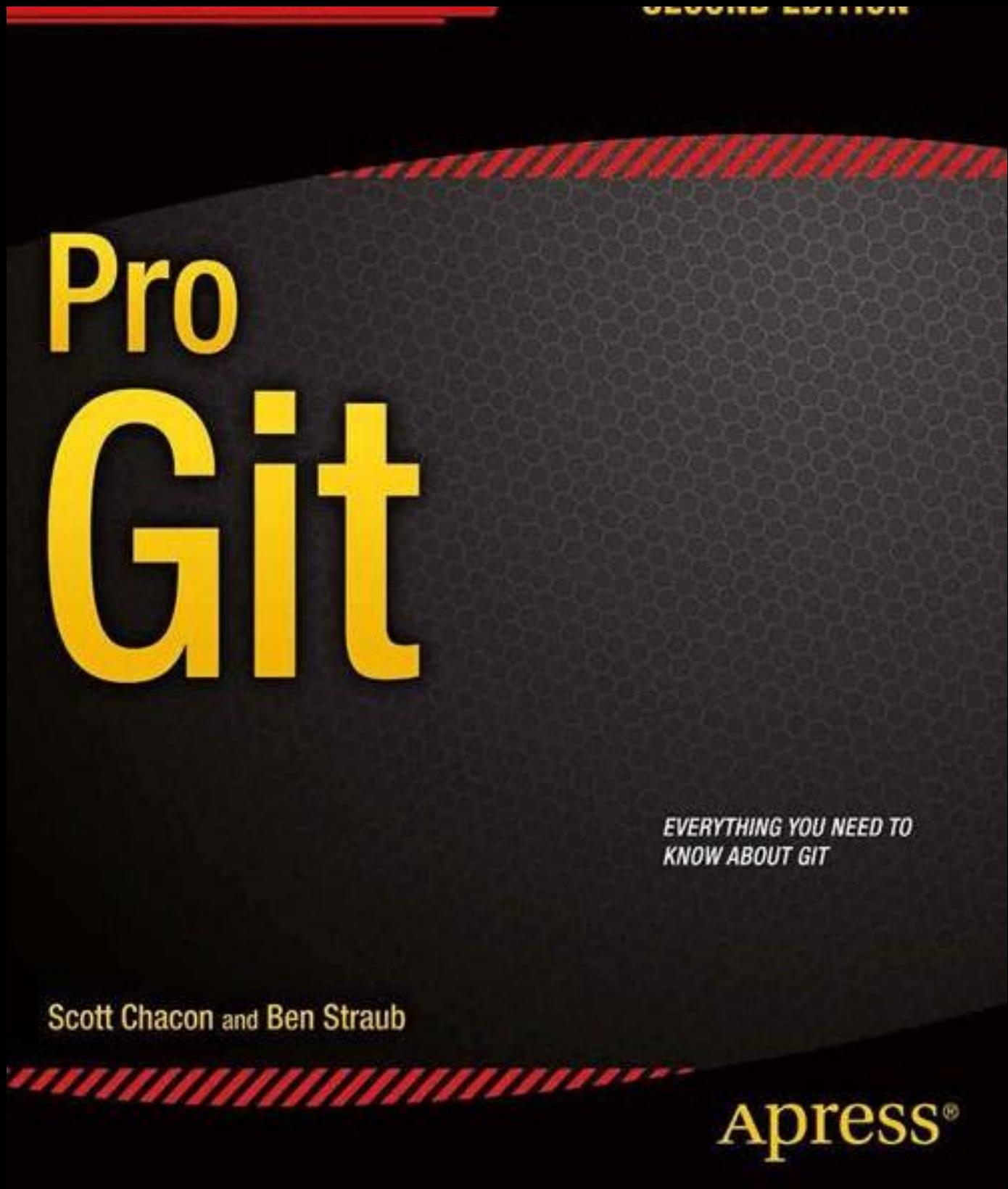
Restore branch



¿Necesito VCS?



Referencias





Git Cheat Sheet

GIT BASICS

<code>git init <directory></code>	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
<code>git clone <repo></code>	Clone repo located at <repo> onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
<code>git config user.name <name></code>	Define author name to be used for all commits in current repo. Devs commonly use <code>--global</code> flag to set config options for current user.
<code>git add <directory></code>	Stage all changes in <directory> for the next commit. Replace <directory> with a <file> to change a specific file.
<code>git commit -m "message"</code>	Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
<code>git status</code>	List which files are staged, unstaged, and untracked.
<code>git log</code>	Display the entire commit history using the default format. For customization see additional options.
<code>git diff</code>	Show unstaged changes between your index and working directory.

UNDOING CHANGES

<code>git revert <commit></code>	Create new commit that undoes all of the changes made in <commit>, then apply it to the current branch.
<code>git reset <file></code>	Remove <file> from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
<code>git clean -n</code>	Shows which files would be removed from working directory. Use the <code>-f</code> flag in place of the <code>-n</code> flag to execute the clean.

REWRITING GIT HISTORY

<code>git commit --amend</code>	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
<code>git rebase <base></code>	Rebase the current branch onto <base>. <base> can be a commit ID, branch name, a tag, or a relative reference to HEAD.
<code>git reflog</code>	Show a log of changes to the local repository's HEAD. Add <code>--relative-date</code> flag to show date info or <code>--all</code> to show all refs.

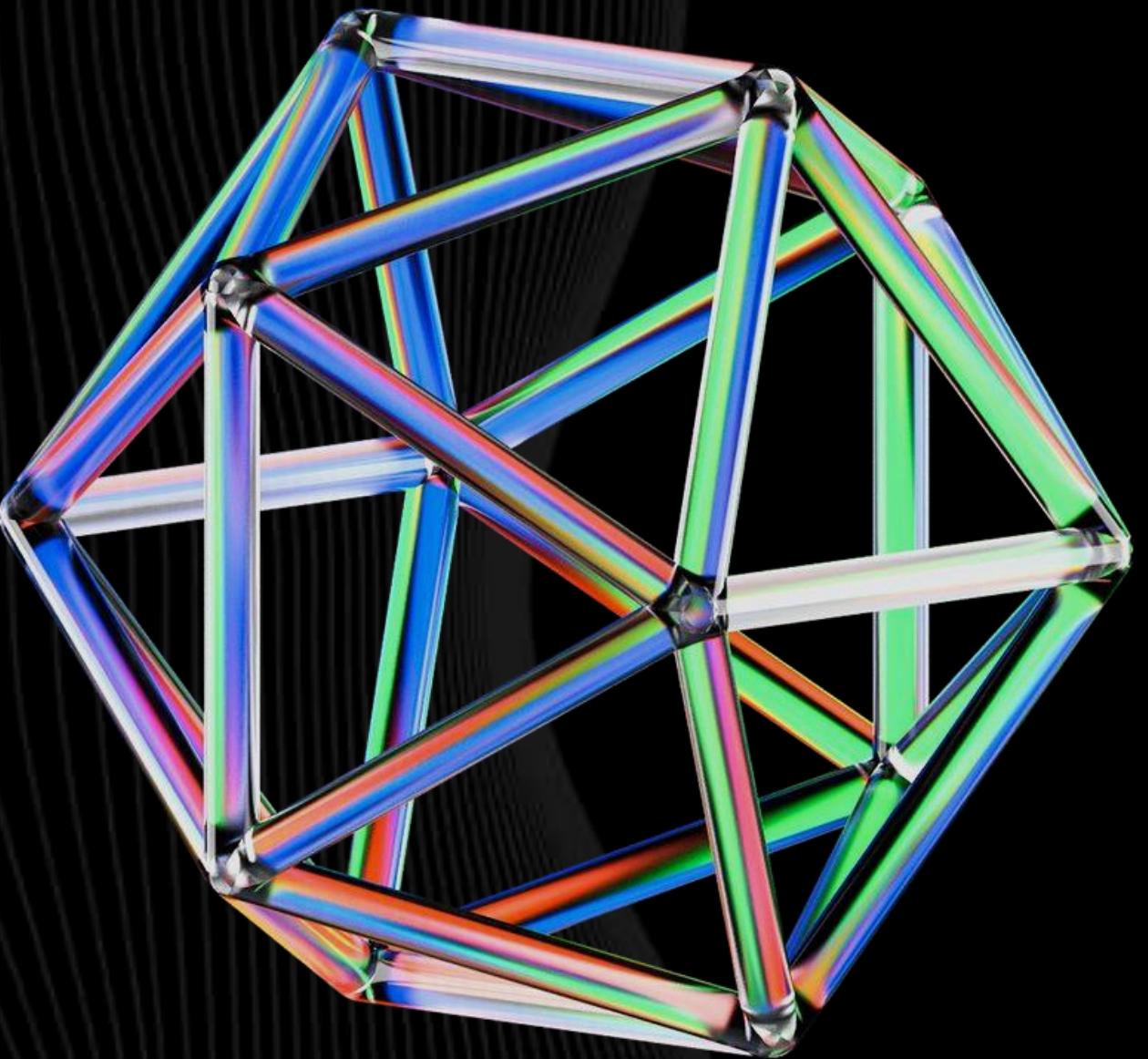
GIT BRANCHES

<code>git branch</code>	List all of the branches in your repo. Add a <branch> argument to create a new branch with the name <branch>.
<code>git checkout -b <branch></code>	Create and check out a new branch named <branch>. Drop the <code>-b</code> flag to checkout an existing branch.
<code>git merge <branch></code>	Merge <branch> into the current branch.

REMOTE REPOSITORIES

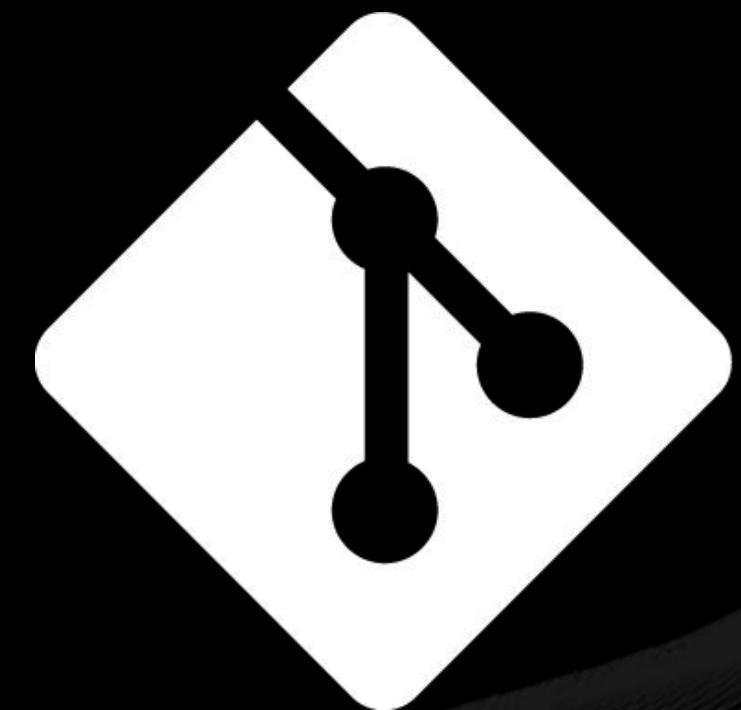
<code>git remote add <name> <url></code>	Create a new connection to a remote repo. After adding a remote, you can use <name> as a shortcut for <url> in other commands.
<code>git fetch <remote> <branch></code>	Fetches a specific <branch>, from the repo. Leave off <branch> to fetch all remote refs.
<code>git pull <remote></code>	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
<code>git push <remote> <branch></code>	Push the branch to <remote>, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

GRACIAS



LABORATORIO DE PROGRAMACIÓN CIENTÍFICA PARA CIENCIA DE DATOS

CLASE 2: VCS



git

MDS7202-1 – PRIMAVERA 2025