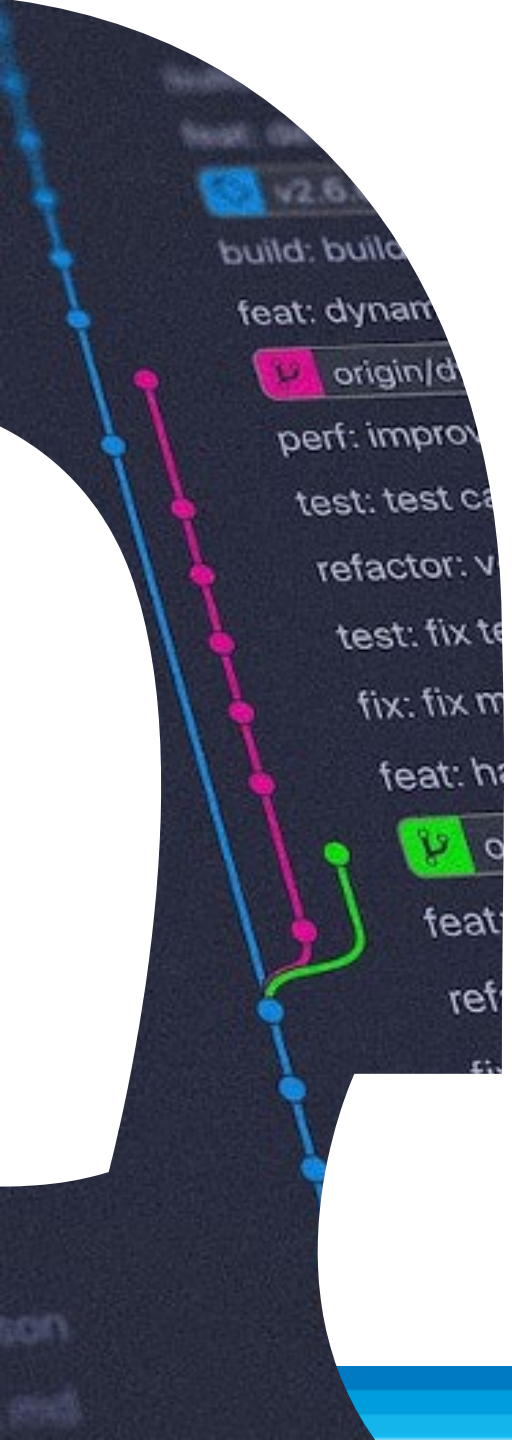


ULPGC
Universidad de
Las Palmas de
Gran Canaria

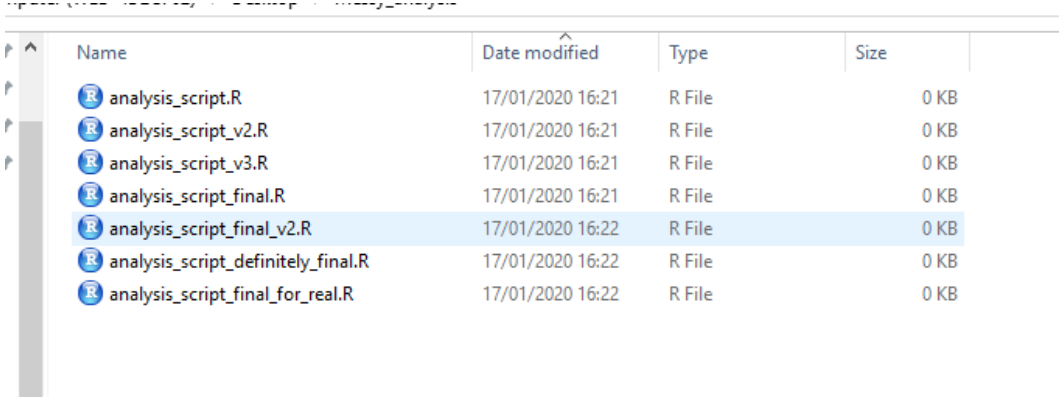
eII
ESCUELA DE INGENIERÍA
INFORMÁTICA

Control de Versiones

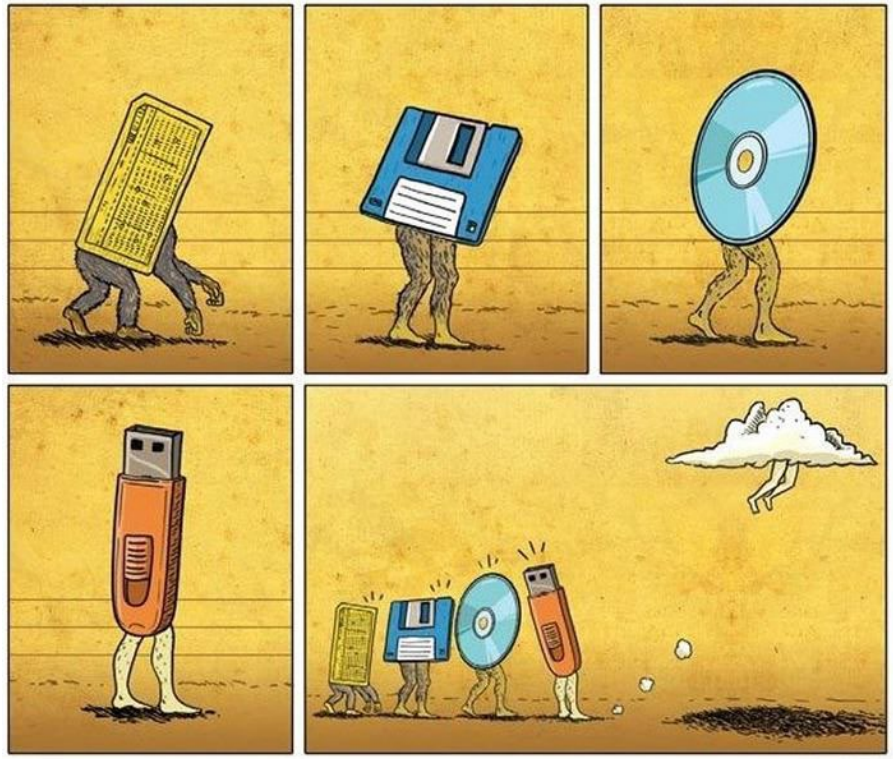
Nelson Monzón
Daniel Santana Cedrés
Jonay Suárez Ramírez

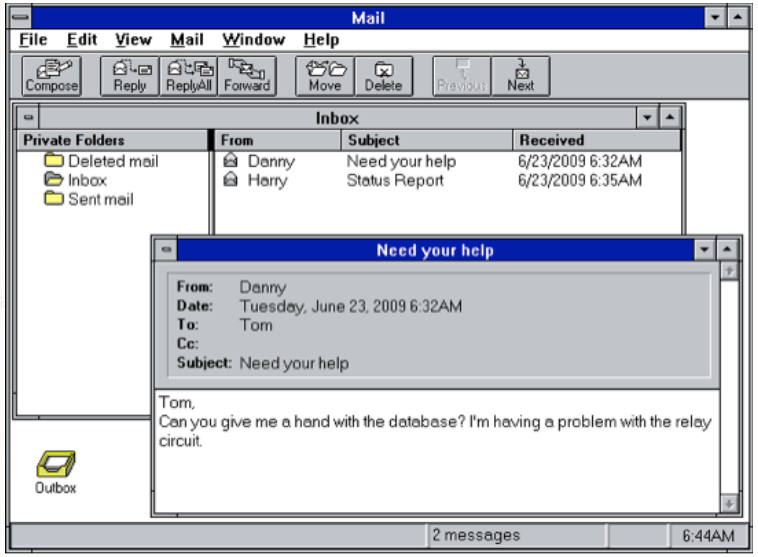
Producción de Software

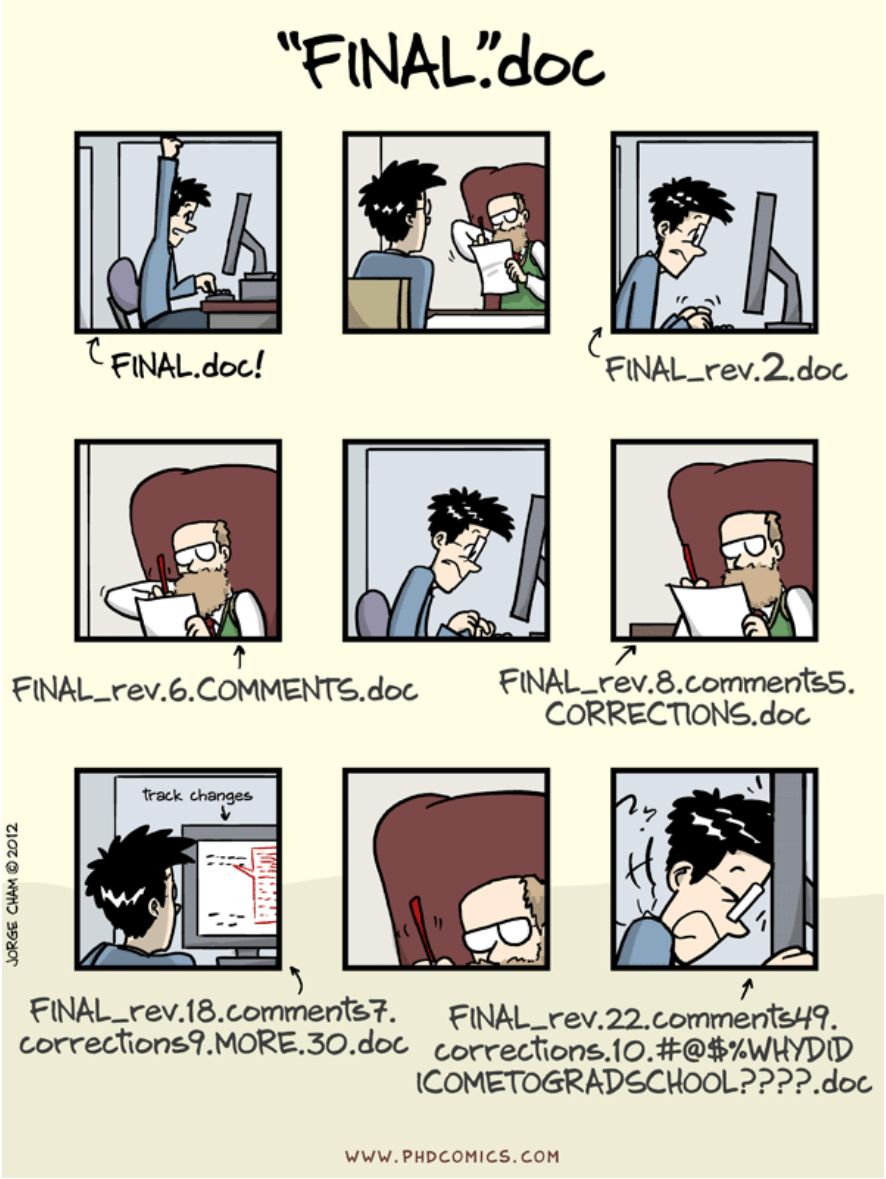
El mundo antes de los VCS…



Name	Date modified	Type	Size
analysis_script.R	17/01/2020 16:21	R File	0 KB
analysis_script_v2.R	17/01/2020 16:21	R File	0 KB
analysis_script_v3.R	17/01/2020 16:21	R File	0 KB
analysis_script_final.R	17/01/2020 16:21	R File	0 KB
analysis_script_final_v2.R	17/01/2020 16:22	R File	0 KB
analysis_script_definitely_final.R	17/01/2020 16:22	R File	0 KB
analysis_script_final_for_real.R	17/01/2020 16:22	R File	0 KB







Sistemas de control de versiones o VCS (*Version Control System*)

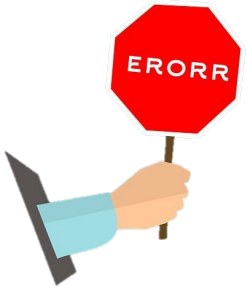
Se llama **Control de versiones** a la **gestión** de los diversos **cambios** que se realizan sobre los elementos de algún **producto** o una **configuración** del mismo.

Una **versión**, **revisión** o **edición** de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación.

Son un grupo de aplicaciones originalmente ideadas para gestionar ágilmente los cambios en el código fuente de los programas y poder revertirlos.

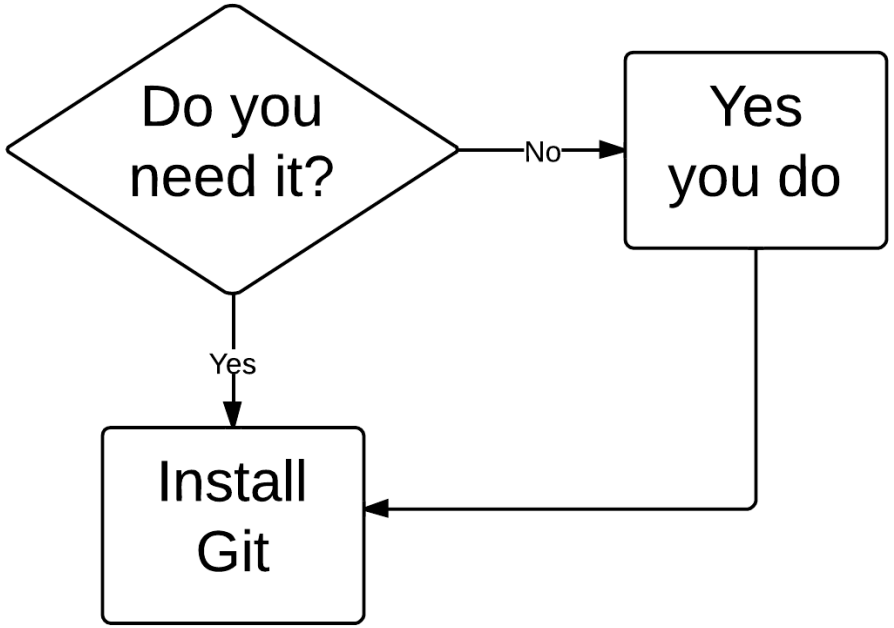
El ámbito se ha ampliado **a otras actividades (fuera de la informática)** que generan ficheros digitales:

- Documentos
- Ofertas
- Dibujos
- Esquemas
- Etc..



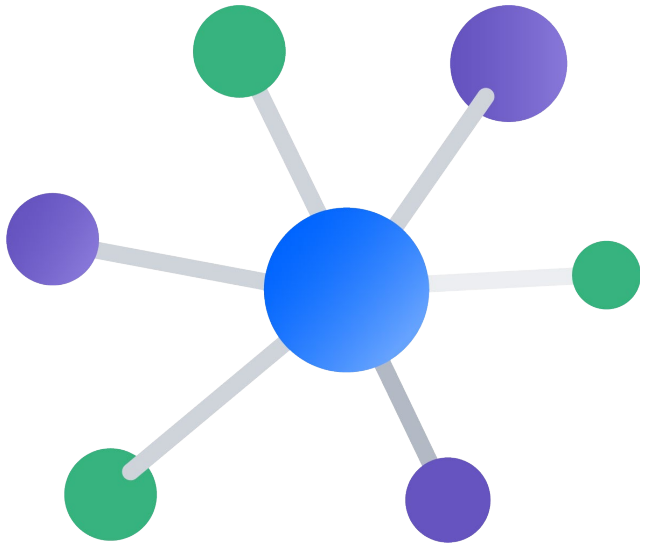
Desde hace no tantos años…

Version Control Flowchart

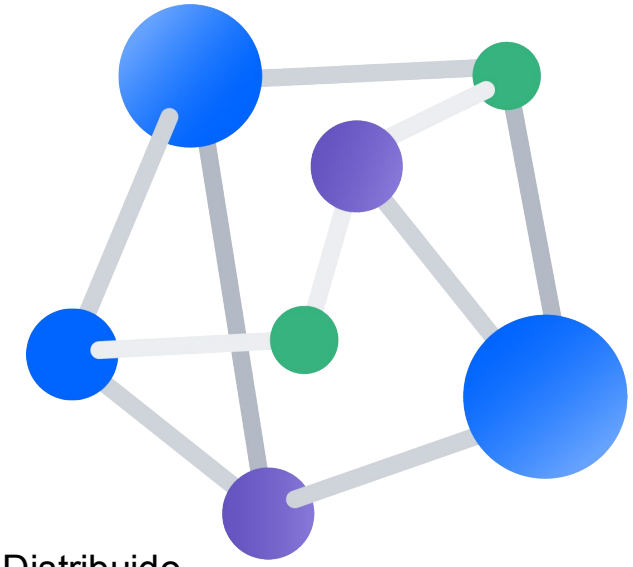


Tipos de VCS

Generación	Conexión	Operaciones	Concurrencia	Ejemplos
Primera	Ninguna	Un fichero cada vez	Cerrojos	RCS, SCCS
Segunda	Centralizada	Multi-fichero	Merge antes de commit	CVS, SourceSafe, Subversion, Team Fundation Server
Tercera	Distribuida	Conjunto de cambios	Commit antes de merge	Bazaar, Git, Mercurial

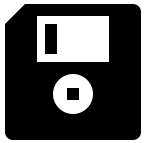


Modelo Centralizado



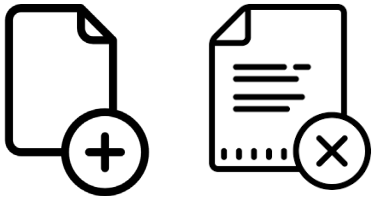
Modelo Distribuido

Principales características de los VCS



Almacenamiento

Mecanismo de almacenamiento de los elementos que deba guardar: archivos, documentación, esquemas, etc.



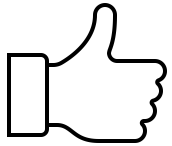
Cambios

Posibilidad de realizar cambios sobre los elementos almacenados: añadir, borrar, modificar, renombrar, mover, etc.

Histórico

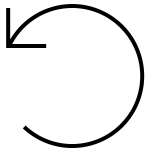
Registro histórico de las acciones realizadas con cada elemento o conjunto de elementos pudiendo volver o extraer un estado anterior del producto

¿Por qué necesito un VCS?



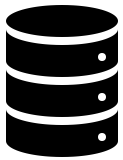
Resolver Conflictos

En la evolución de un proyecto software, es normal que diferentes desarrolladores lleven a cabo cambios sobre el mismo fichero. Ya que el VCS los monitoriza y ayuda a resolver los conflictos que surjan. Todas estas operaciones dejan un registro que ofrece información del histórico de la evolución del proyecto



Revertir Cambios

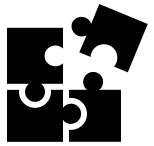
El sistema VCS mantiene un historial de la evolución de los cambios y el estado del código fuente en el sistema de archivos. Gracias a ello se pueden deshacer o revertir los cambios a un estado anterior conocido. Por ejemplo, si se detecta un fallo en la versión actual, se puede volver a una versión anterior estable



Copia de Seguridad Externa

El uso colaborativo del sistema de control de versiones requiere de una instancia remota para compartir entre desarrolladores, alojada en un sistema externo. Como consecuencia, se tiene una copia externa de todo el repositorio del proyecto, lo que permite tener un respaldo en caso de algún problema

Ventajas de usar VCS

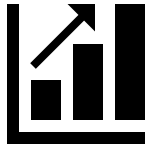


Integración

El VCS se puede integrar con herramientas de terceros. Por ejemplo, con el software de seguimiento de tareas Jira y la plataforma de documentación Confluence

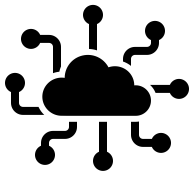
Comunicación

Al alojar un VCS normalmente se tienen herramientas de comunicación. Por ejemplo, es posible enviar comentarios a repos alojados en github, abrir issues, etc. También puede haber discusión interna entre desarrolladores por ejemplo a través de mecanismos de soporte a pull requests



Conocimiento, medición y responsabilidad

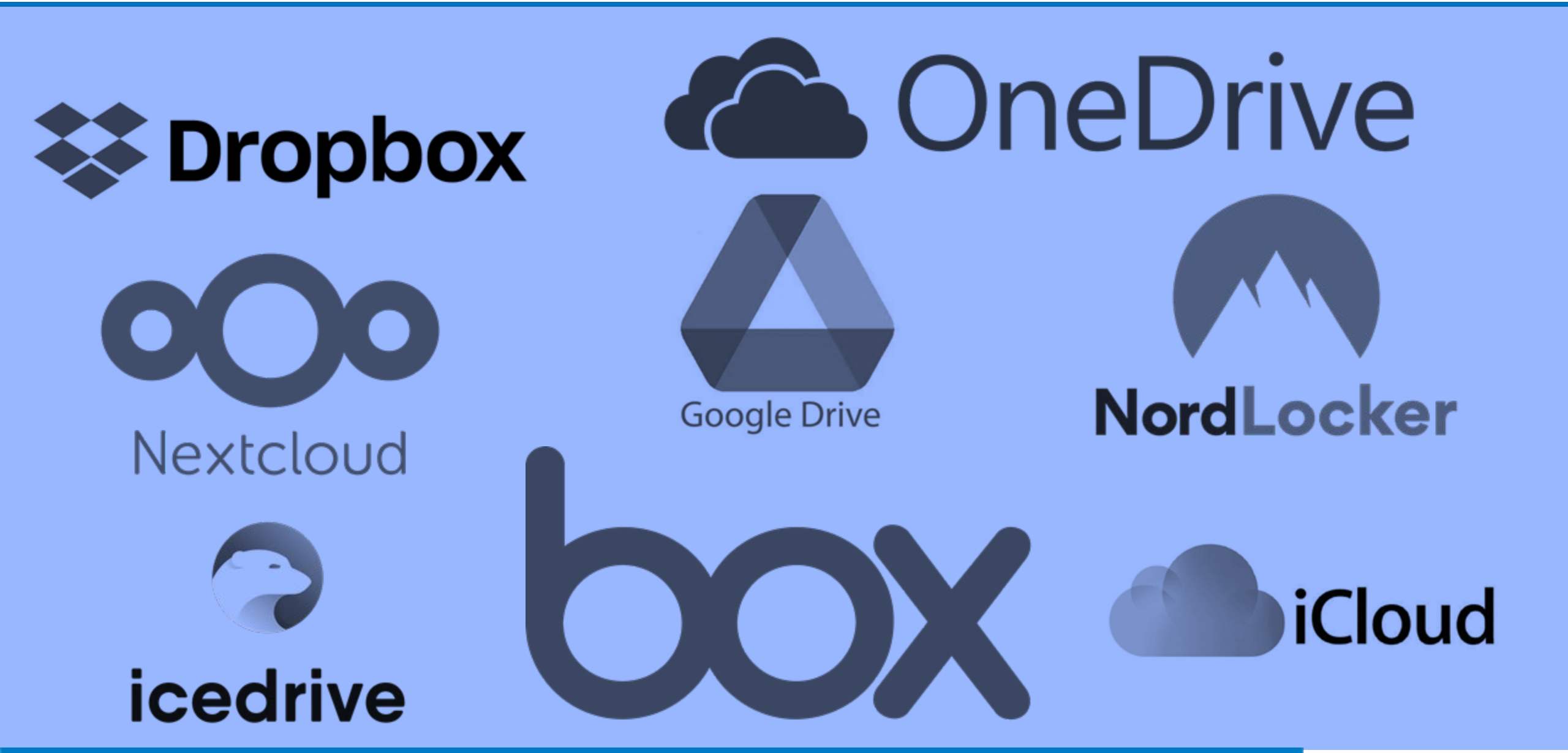
Se puede monitorizar y medir indicadores de rendimiento, pudiendo saber la velocidad y eficiencia en un proyecto. Además: funcionalidades implementadas, velocidad de creación de código, tiempo de desarrollo, fallos creados y arreglados, etc.



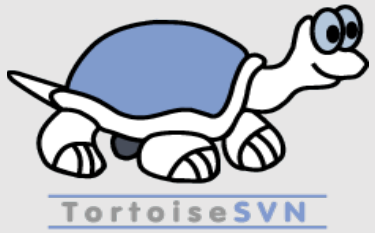
Automatización de CI o CD

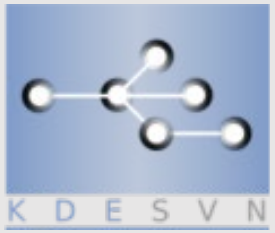
Flujos de trabajo de entrega continua. Es posible configurar extensiones para la revisión de código y el proceso de entrega. Además: compilaciones automáticas, pruebas e implementaciones (ficheros YAML)

¿Para qué **NO USAMOS** el control de versiones?

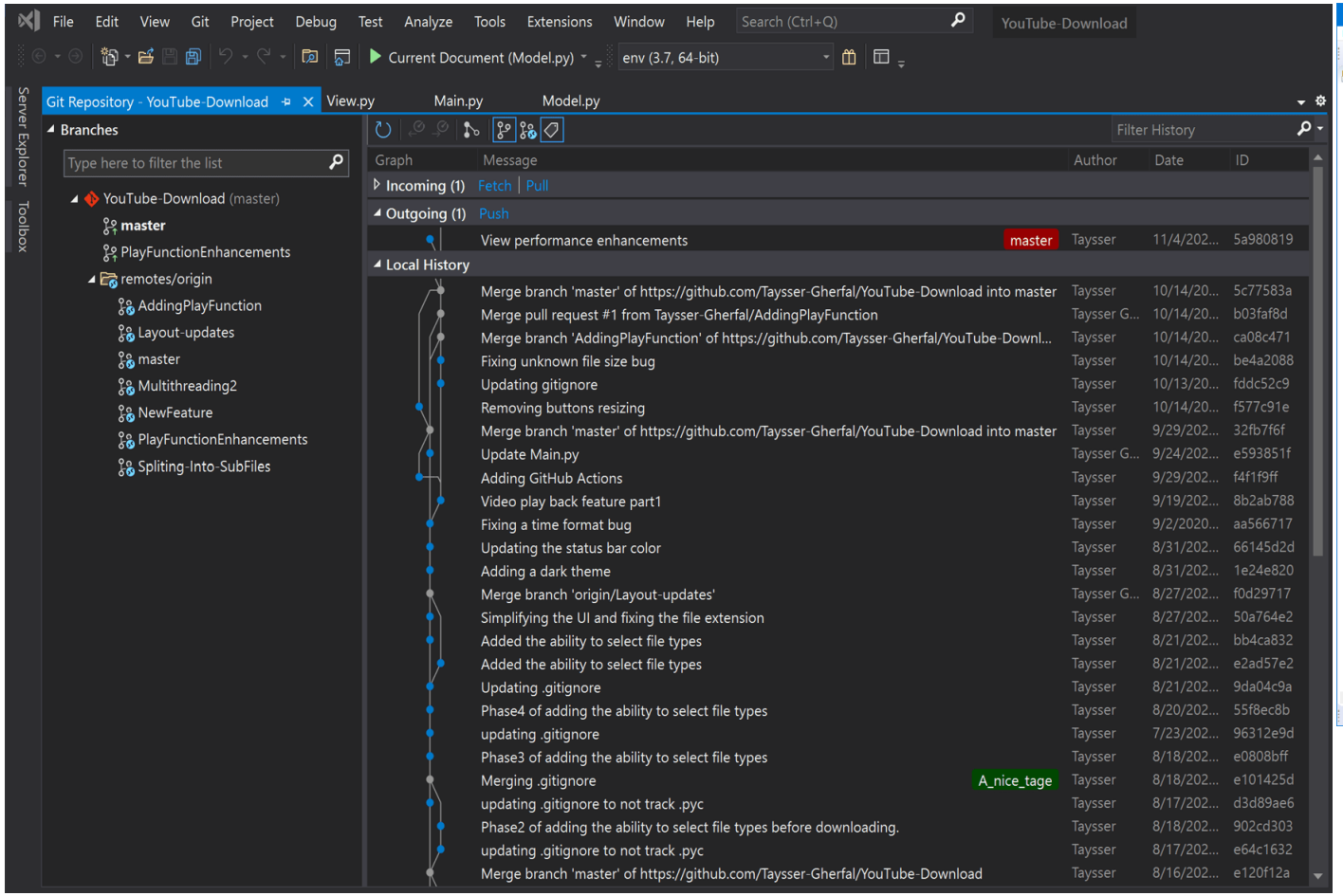


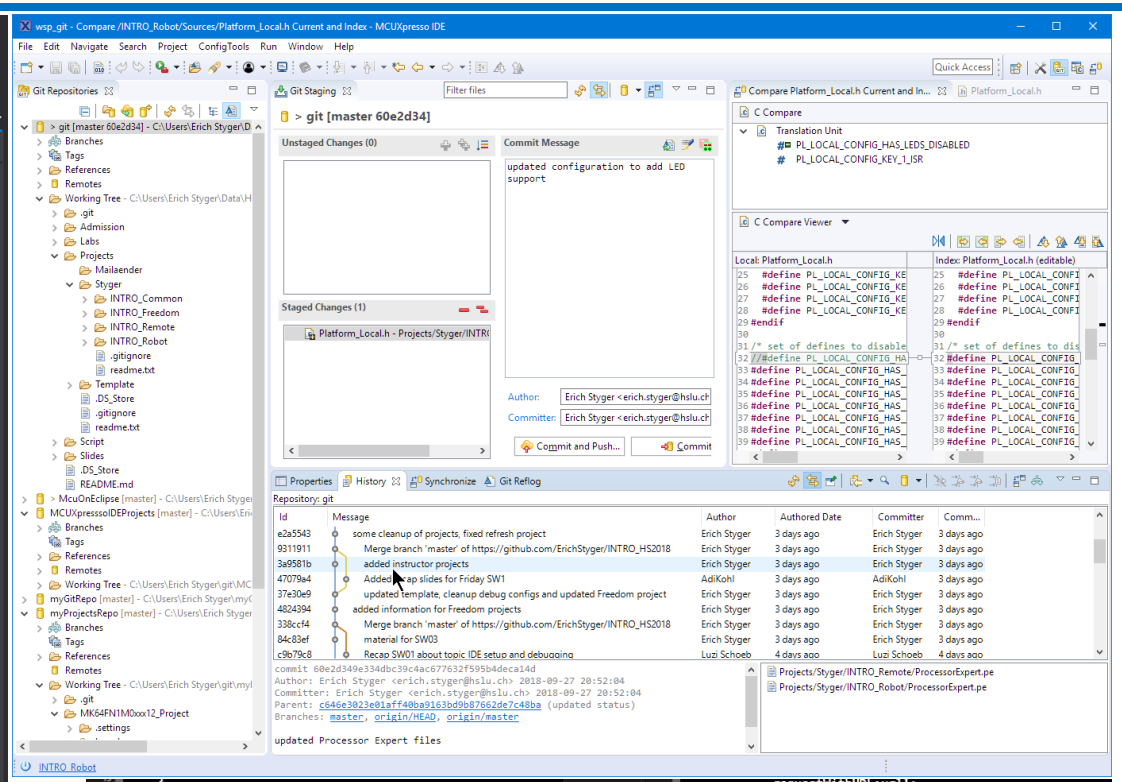
Ejemplos de dos CVS ampliamente utilizados & Clientes





Integración CVS – IDEs





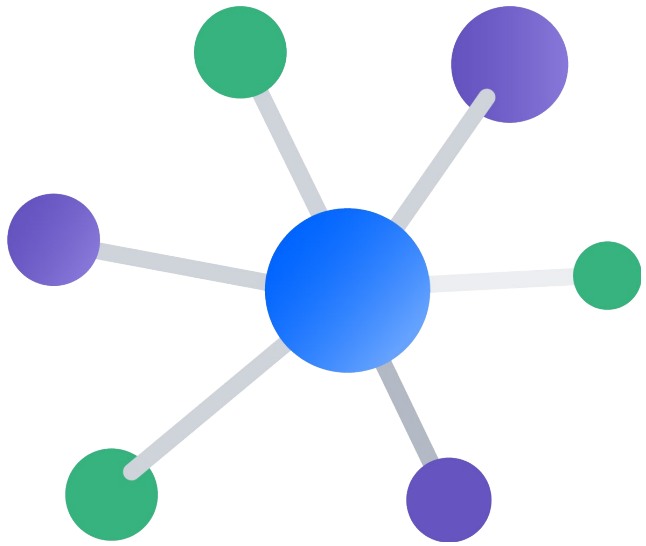
Visual Studio

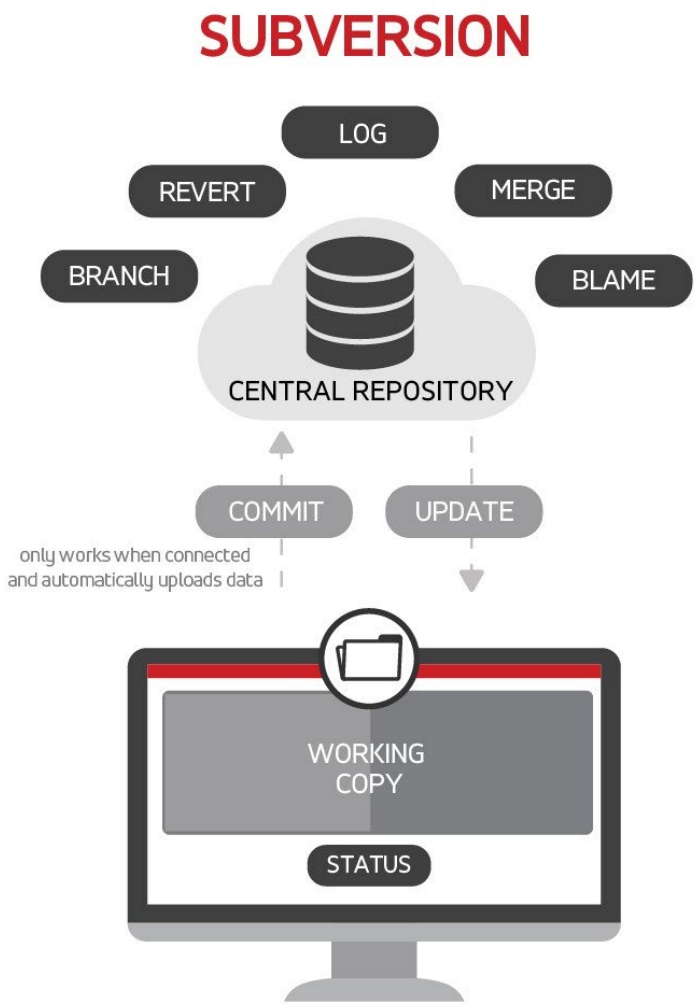
eclipse

XCODE

...

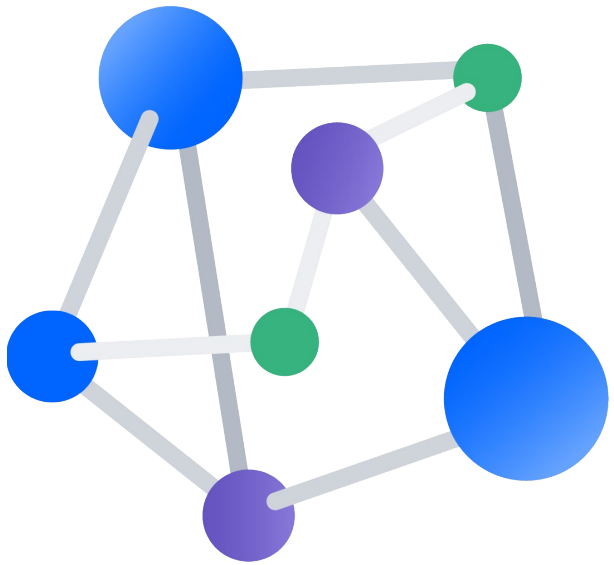
SVN - Subversion

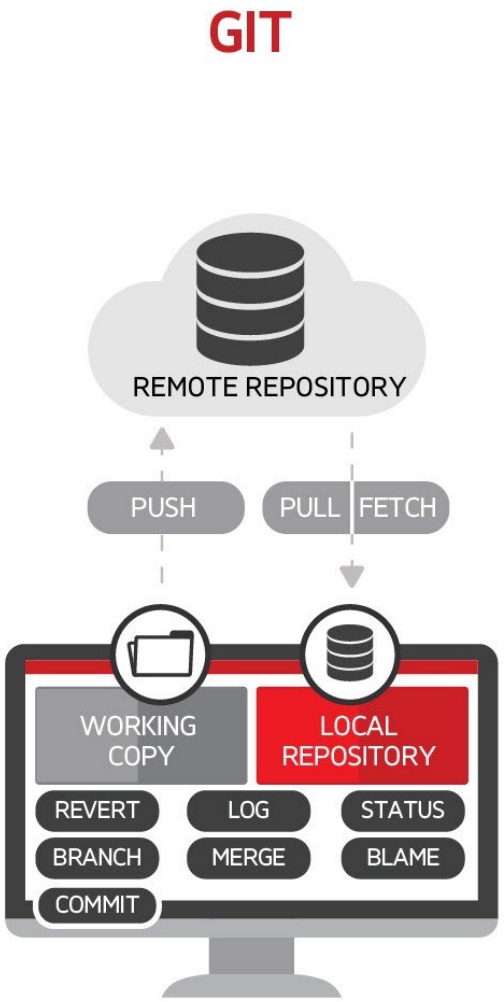




- Servidor remoto: todas las versiones de los ficheros.
- Copia local (working copy): directorio local de trabajo.
- Siempre información de tu repositorio.
- Rastreo de los ficheros ya que el historial es permanente (en git puede que el rastro se pierda - p.e. al hacer git rebase)
- Sabemos que todo está guardado en un único repositorio (git usa repos distribuidos)
- Control de acceso centralizado
- Permite bloqueo de archivos exclusivos (ejecutables que no se pueden fusionar p.e.)

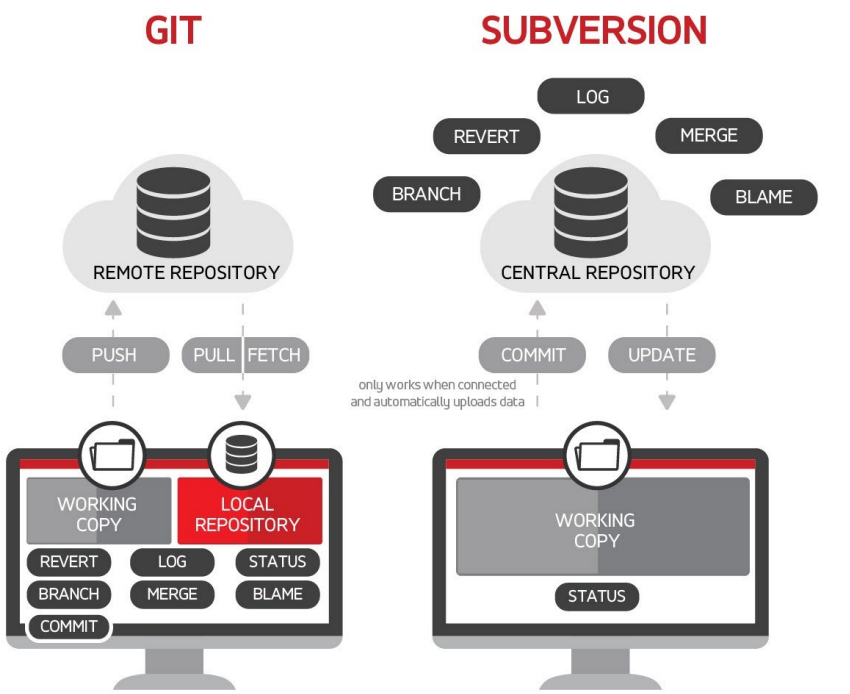
Git

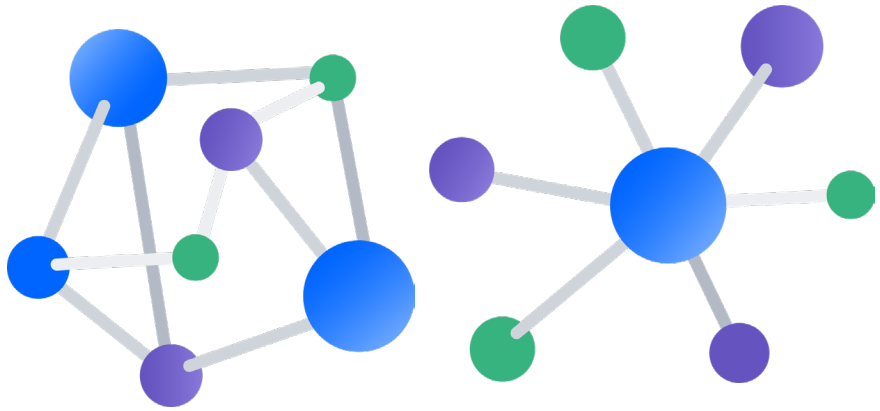




- Cada usuario tiene una copia completa del proyecto, pudiendo acceder al histórico rápido.
- Se puede interactuar con el repo local sin necesidad de conexión.
- Si alguna copia del repo se corrompe, sólo se pierde esa y no todo el repo remoto.
- En SVN se necesita red.
- Si en SVN se corrompe el repo, habrá que recuperar un backup y eso puede hacer perder info.
- Esto es dependiente de políticas de backup

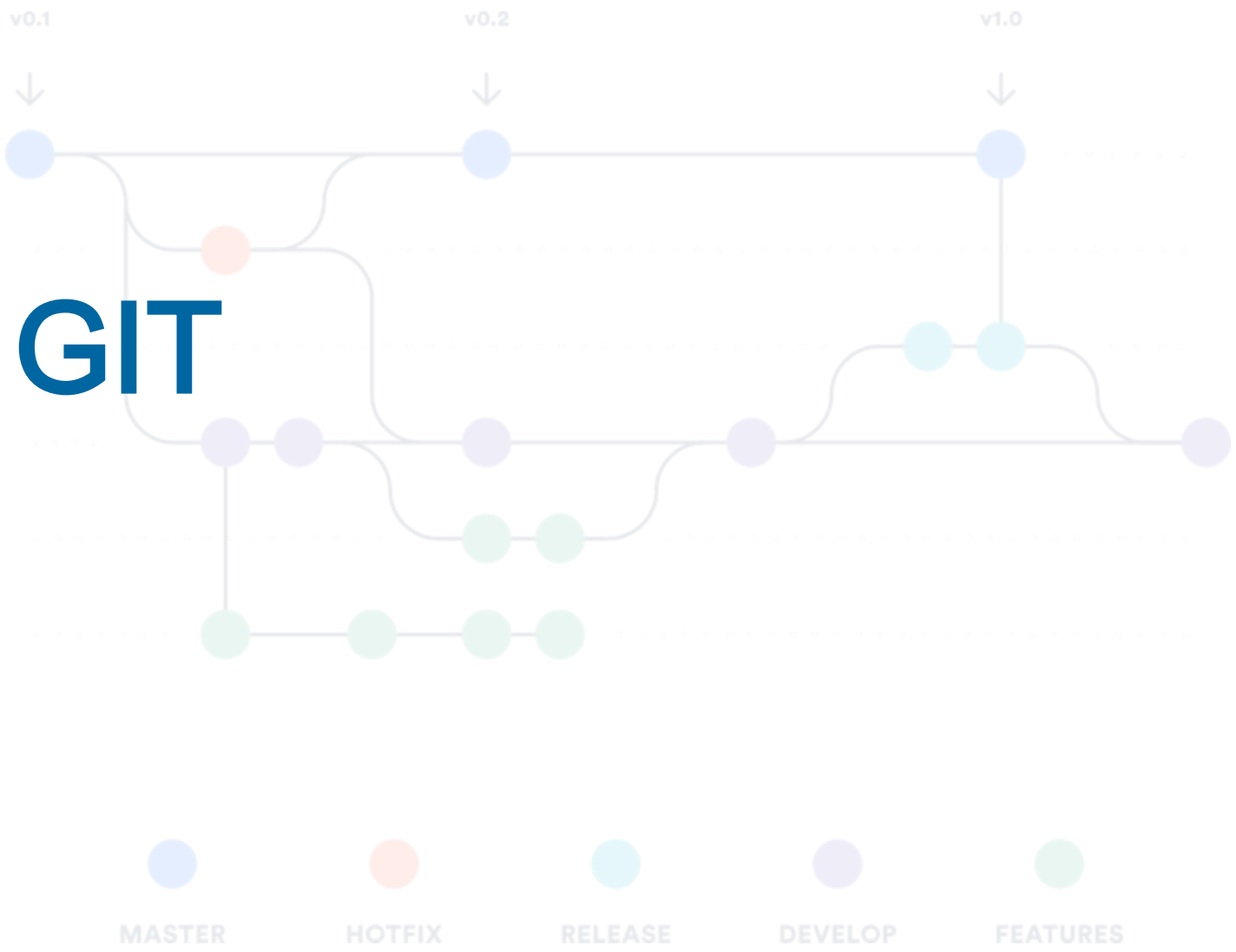
GIT vs SVN



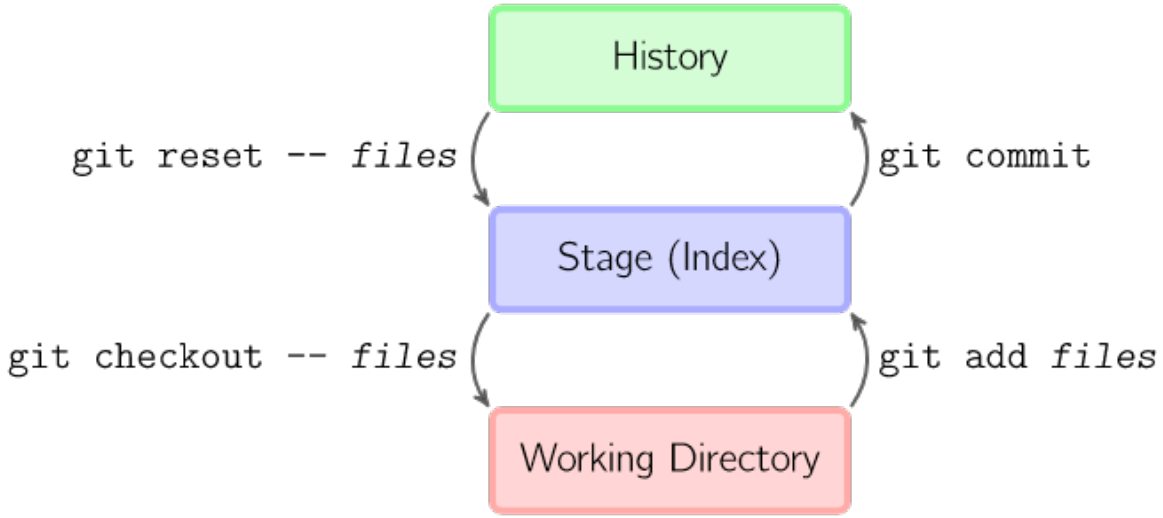


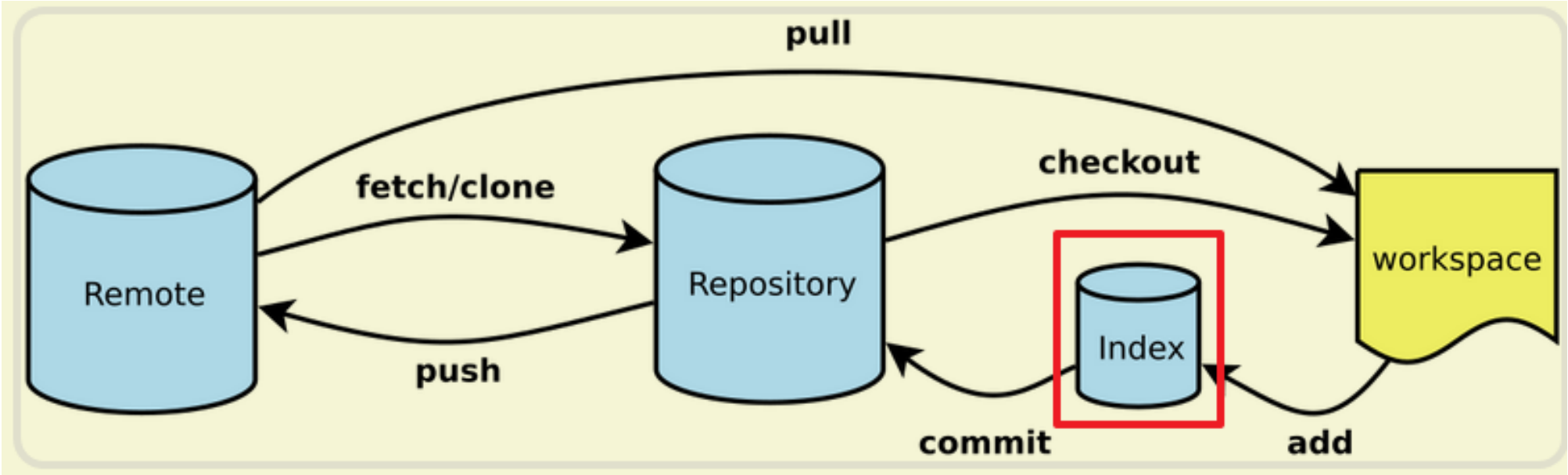
Característica	SVN	Git
Control de versiones	Centralizada	Distribuida
Repositorio	Un repo central donde se generan copias de trabajo	Copias locales del repo en las que se trabaja directamente
Autorización de acceso	Dependiendo de la ruta de acceso	Para la totalidad del directorio
Seguimiento de cambios	Basado en archivos	Basado en contenido
Historial de cambios	Solo en el repo completo, las copias de trabajo incluye la versión más reciente	Tanto el repo como las copias locales incluyen historial completo
Conectividad de red	Con cada acceso	Solo necesario para sincronización

Estructura de GIT

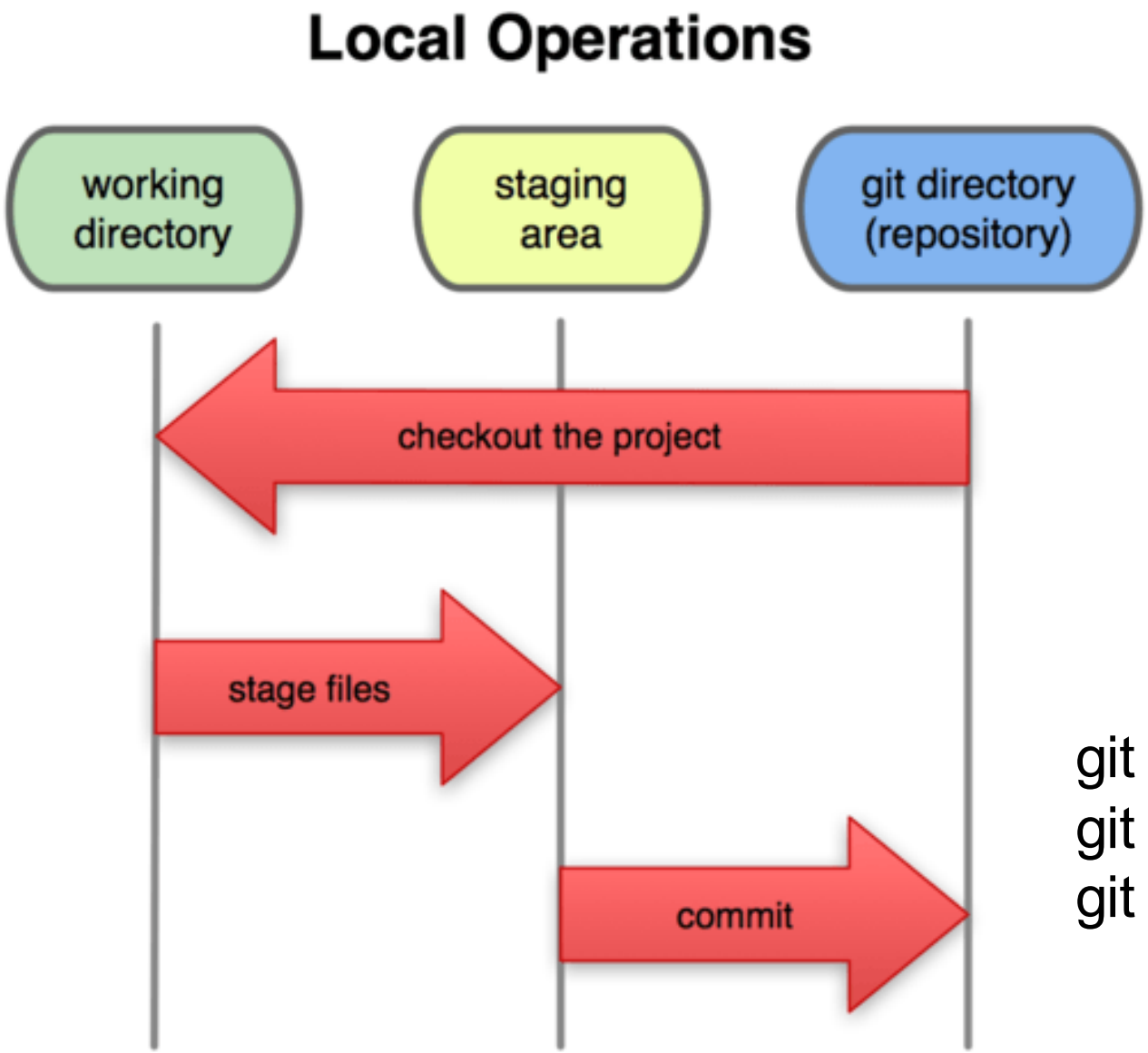


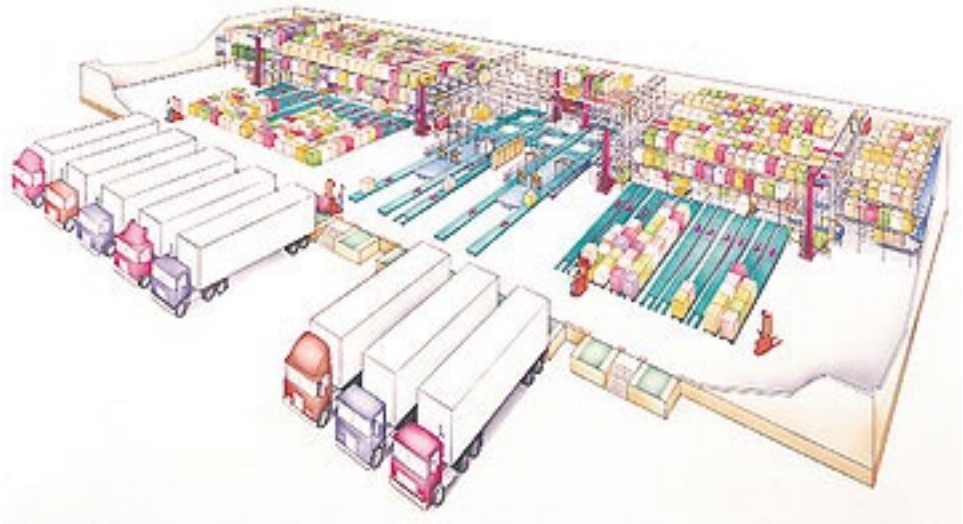
Stage


History
git reset -- files
git commit
Stage (Index)
git checkout -- files
git add files
Working Directory


pull
fetch/clone
checkout
Remote
Repository
workspace
Index
push
commit
add

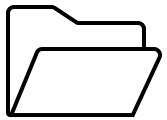
Estados GIT



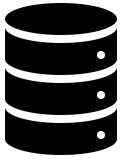


git clone [url]
git commit –m “[descriptive message]”
git push [alias] [brach]

Conceptos básicos



Working directory: sólo puede haber una versión en local



Repository: contiene todas las versiones y las relaciones entre ellas

HEAD

Puntero a la versión con la que estamos trabajando, la que tenemos en nuestro directorio de trabajo

Master

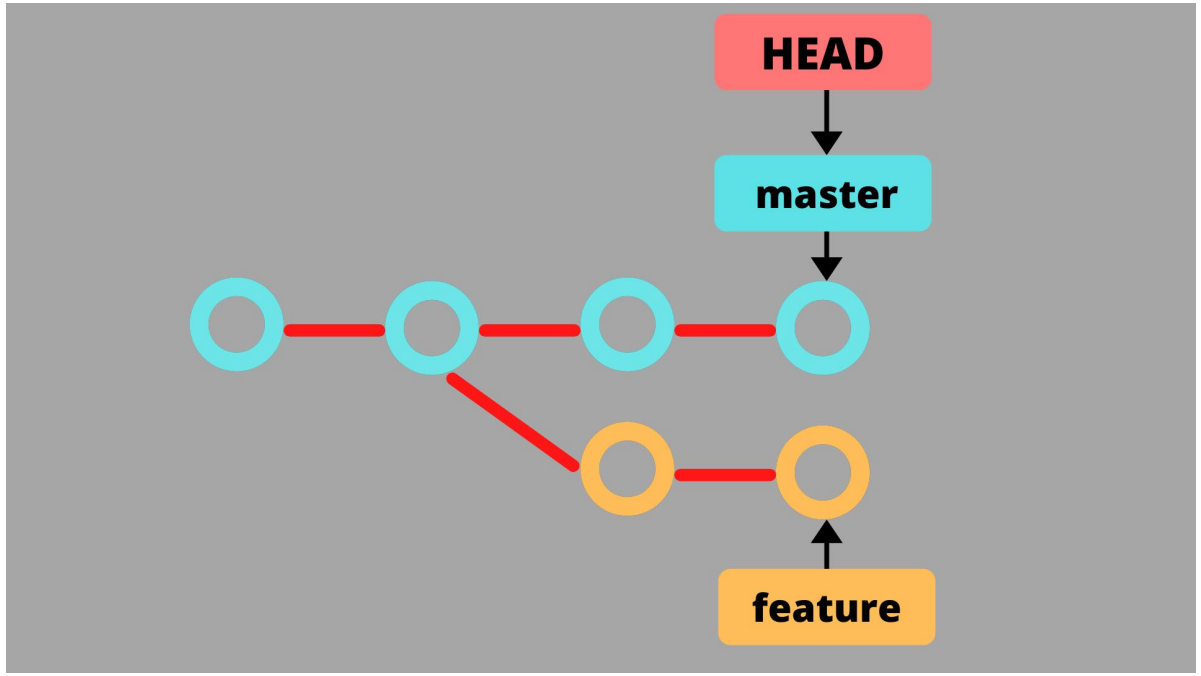
Es la rama principal sobre la que debe ir creciendo el desarrollo

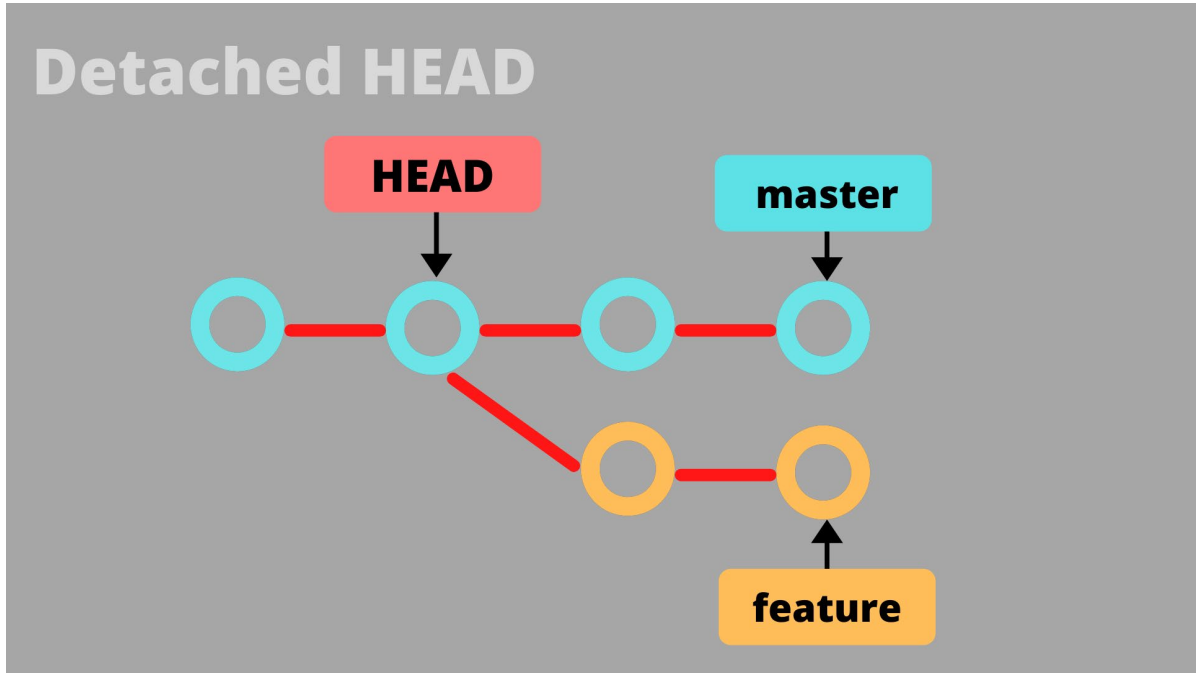
Branch

Son ramas secundarias sobre las que se crean nuevas funcionalidades

HEAD

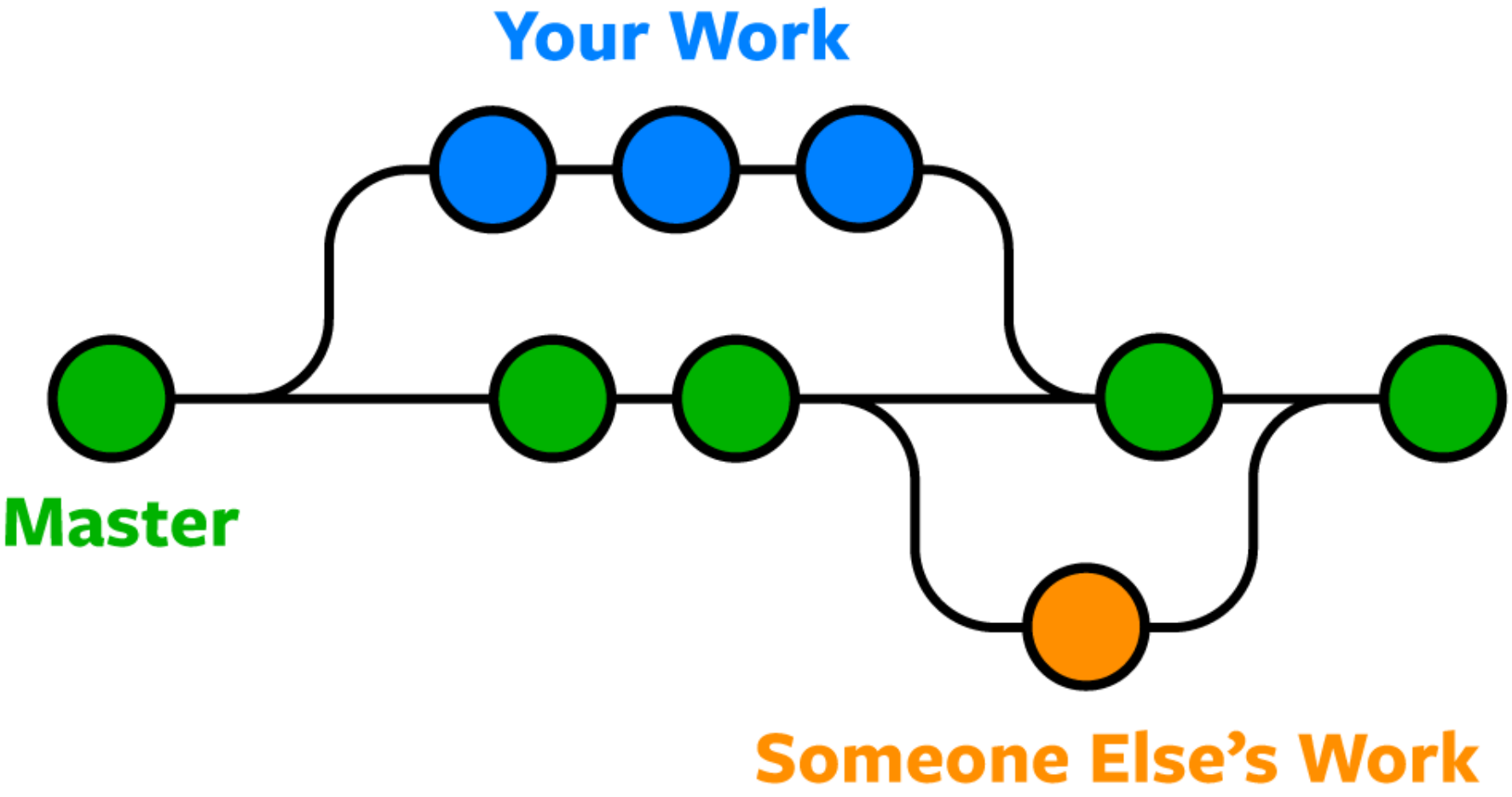
Es un puntero o referencia al commit más reciente de nuestra rama actual. Es el último estado commit de nuestro proyecto. Se puede considerar como nuestra rama de trabajo, no apunta directamente a un commit, sino a una rama. HEAD se actualiza automáticamente cuando se hacen nuevos commits.



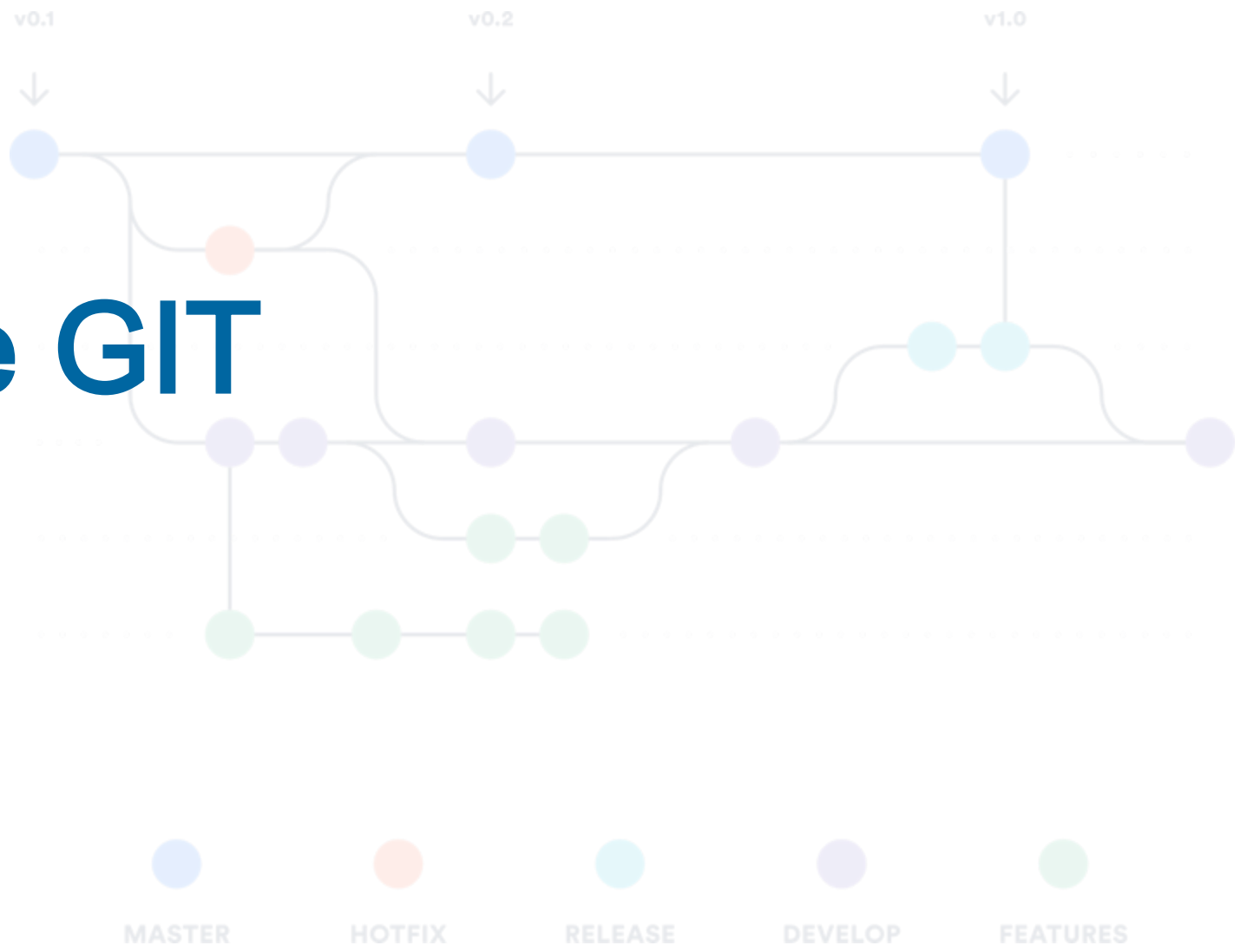


Normalmente HEAD referencia a la rama actual (izqda.) pero se puede mover a cualquier otro punto. Se denomina detached HEAD (dcha.) a cuando HEAD no apunta a la punta de una rama actual. Esto puede pasar cuando se hace un check-out de un commit, una tag o de una rama remota.

Master & Branches



Comandos de GIT



Términos básicos

- Repositorio
- Revisión (*version*)
- Etiqueta (*tag*)
- Línea base (*baseline*)
- Ramificar (*branch*)
- Desplegar (*chek-out*)
- Publicar o enviar (*commit*)
- Conflicto (*conflict*)
- Resolver (*resolve*)
- Cambio (*change*)
- Lista de cambios (*changelist*)
- Integración o fusión (*merge*)
- Integración inversa
- Actualización (*update*)
- Copia de trabajo (*workspace*)
- Congelar (*freeze*)

Guía mínima de comandos git

Comando	Función
git init	Inicializa un repositorio
git add	Agrega cambios a staging area
git status	Muestra el estado de los cambios
git reset HEAD	Elimina un cambio del staging area
git commit	Envía un cambio del staging al repositorio
git rm	Borra un cambio del repositorio
git checkout	Trae la última versión del cambio, crea y cambia de rama
git branch	Muestra las ramas
git merge	Fusiona las ramas
git remote add origin	Relacionar repo local a un repo remoto
git clone	Clonar repositorio remoto
git pull	Trae la última versión del repo remoto
git push	Envía los cambios del repo local al repo remoto

Un fichero bastante útil: el .gitignore

.gitignore

Es un fichero en el cual mediante patrones puedes indicar que archivos ignorar. Estos patrones se crean a partir de varios símbolos

Patrón	Explicación
**/logs	Doble asterisco se usa para ignorar directorios
**/logs/debug.log	Doble asterisco usado para ignorar ficheros basado en su nombre y el nombre de su directorio padre
*.log	Un asterisco representa cero o más caracteres
.log !important/.log trace.*	Los ficheros definidos después de ! Volverán a ignorar cualquier archivo previamente ignorado
/debug.log	La barra sólo identifica la raíz del repo
debug.log	Por defecto el nombre de fichero en cualquier directorio
debug?.log	? Identifica a sólo un carácter
…	Hay muchas más combinaciones basadas en sentencias similares a expresiones regulares

Inicialización del repositorio

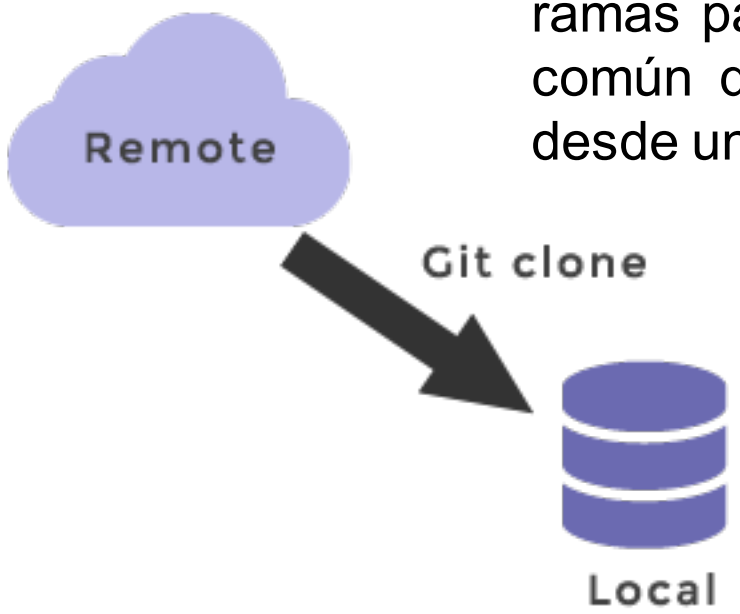
git init

```
$ cd project_folder
$ git init
```

Se puede inicializar un repositorio simplemente ejecutando el comando git init dentro del directorio que queremos usar para nuestro proyecto. Crea un subdirectorio de .git con todos los metadatos necesarios para el nuevo repo. También se crea un archivo HEAD que apunta al commit actual

git clone

```
git clone ssh://x_person@example.com/path/to/team-project.git
cd team-project
# You must work on this project
```



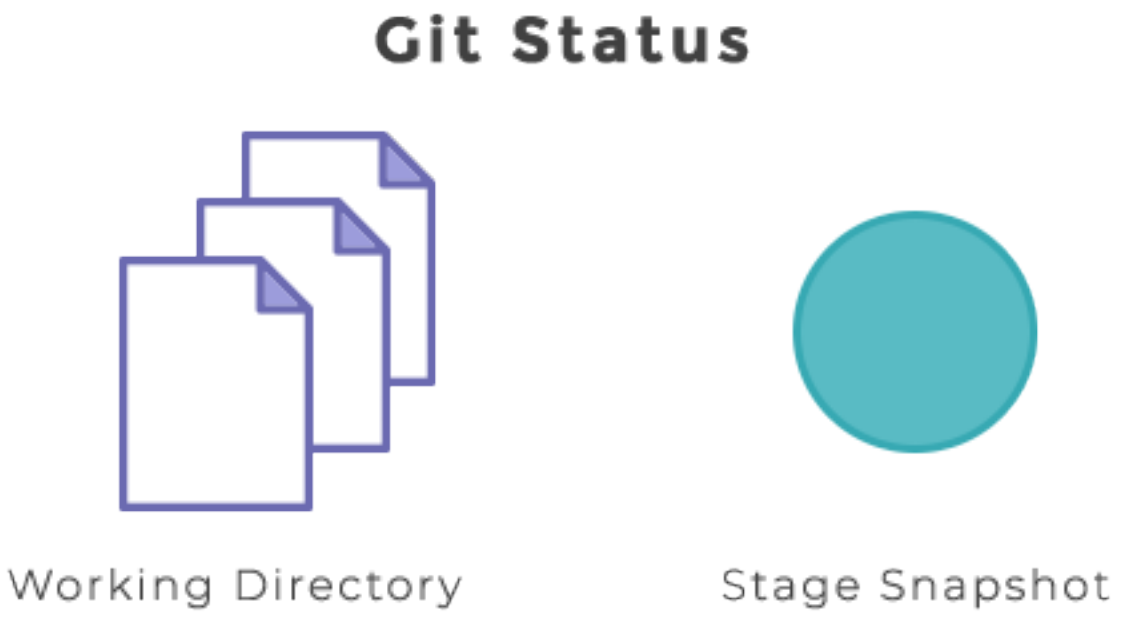
git clone crea un clon/copia de un repositorio existente en un nuevo directorio. También es usado para crear un seguimiento remoto de ramas para cada rama en el repositorio clonado. Es el comando más común que permite a los usuarios obtener una copia del desarrollo desde un repositorio central existente

Añadir contenido al repositorio

git status

```
$ git status
```

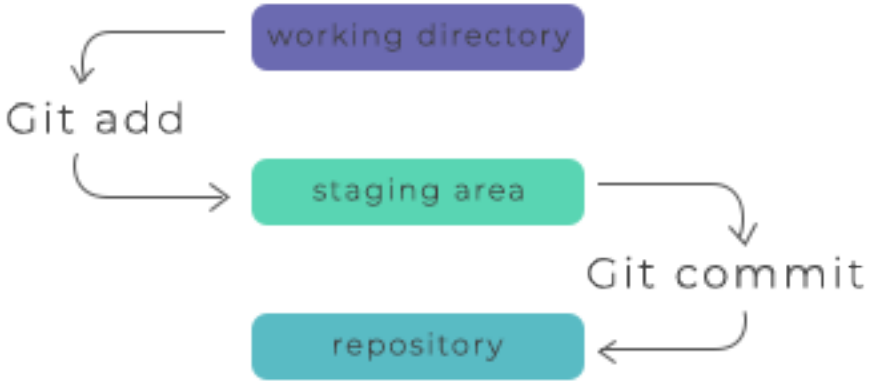
El comando git status se usa en un repo git para ver el estado actual, incluyendo el último commit, cualquier fichero modificado, y cualquier nuevo fichero que aún no es seguido por git.
La salida puede variar bastante, y a menudo incluye mensajes que ayudan al usuario a manejar su repositorio



git add

```
$ git add filename
```

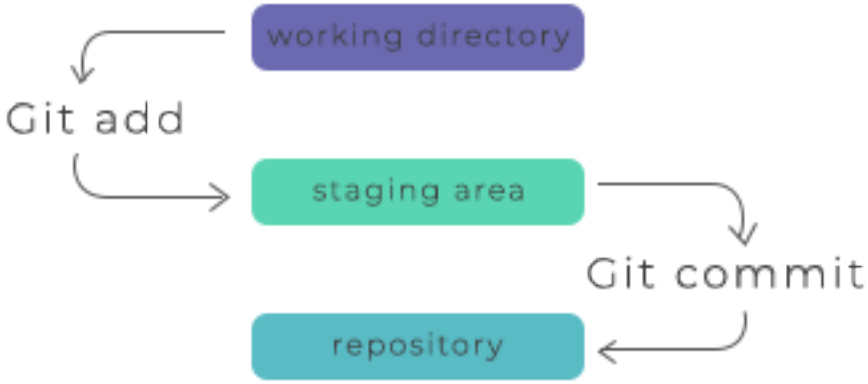
El comando git add se usa para añadir el nombre de fichero que aparece a la derecha al staging area. Una vez los cambios se añaden a esa área, se puede usar el comando git commit para guardar de forma permanente los cambios en el repo local



git commit

```
$ git commit -m "Commit message here"
```

Para hacer un commit se usa el comando git commit –m “message”. El flag –m indica que se está añadiendo un mensaje, y el texto entre comillas es dicho mensaje. Es normal que este mensaje sea algo informativo relativo al commit, si hemos revertido o fusionado algún código, los cambios que hemos hecho, etc.

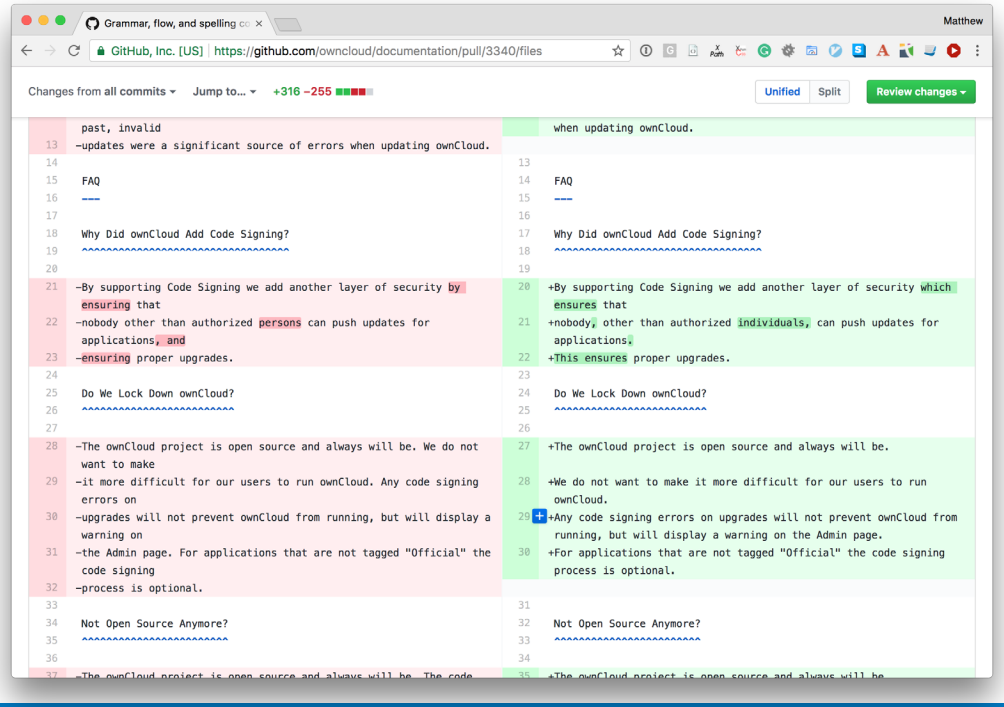


NOTA: git commit --amend se usa para editar los últimos commits, si te olvidas de formatearlo o de enviar un fichero a stage

Comparación y seguimiento de cambios en el repositorio

git diff

git diff es un comando multifunción, que se usa para comparar cambios que han sido “commiteados”. En particular, con la ayuda de este comando, se pueden usar dos conjuntos de datos de entrada y dar como salida los cambios entre ellos. Generalmente se usa en combinación con git status y git log para analizar el estado del repo



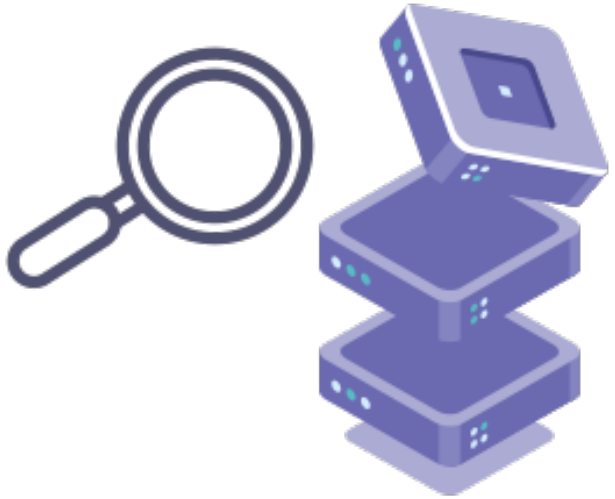
git log

El comando git log se usa para examinar el historial del repositorio y encontrar una versión en particular del proyecto. La salida del comando se puede personalizar:

Opción	Explicación
git log	Muestra todo el histórico
git log -n <limit>	Limita el número de commits
git log --oneline	Un commit por línea
git log --stat	Incluye los ficheros modificados y el número de líneas añadidas o borradas
git log -p	Muestra el diff completo de cada parche
git log --author="<pattern>"	Busca commits por autor (string o regex)
git log --grep="<pattern>"	Busca commits con un mensaje determinado (string o regex)
git log <since>..<until>	Muestra commits en un intervalo: commit ID, nombre de rama, HEAD, o cualesquiera otro tipo de referencia de versión
git log <file>	Muestra aquellos commits que incluyen un fichero concreto
git log --graph --decorate --oneline	Muestra una gráfica en texto basado en los commits al lado izquierdo de los mensajes de commit. El flag --decorate añade los nombres de las ramas o tags. El flag --oneline muestra información del commit en una sola línea

```
$ TZ=PST8PDT git log-compact --decorate --graph -n 17 v2.6.1
  === 2015-09-28 ===
* 22f698cb 19:19 jch (tag: v2.6.1) Git 2.6.1
*   3adc4ec7 19:16 jch Sync with v2.5.4
|\
| * 24358560 15:34 jch (tag: v2.5.4) Git 2.5.4
| *   11a458be 15:33 jch Sync with 2.4.10
| |\
| | * a2558fb8 15:30 jch (tag: v2.4.10) Git 2.4.10
| | *   6343e2f6 15:28 jch Sync with 2.3.10
| | |\
| | | * 18b58f70 15:26 jch (tag: v2.3.10, maint-2.3) Git 2.3.10
| | | *   92cdfd21 14:59 jch Merge branch 'jk/xdiff-memory-limits' into maint-2.3
| | | |\
| | | | * 83c4d380 14:58 jk  merge-file: enforce MAX_XDIFF_SIZE on incoming files
| | | | * dcd1742e 14:57 jk  xdiff: reject files larger than ~1GB
| | | | * 3efb9880 14:57 jk  react to errors in xdi_diff
| | | * |   f2df3104 14:46 jch Merge branch 'jk/transfer-limit-redirection' into maint-2.3
| | | |\ \
| | | | | | === 2015-09-25 ===
| | | | * | b2581164 15:32 bb  http: limit redirection depth
| | | | * | f4113cac 15:30 bb  http: limit redirection to protocol-whitelist
| | | | * | 5088d3b3 15:28 jk  transport: refactor protocol whitelist code
| | | | | |   === 2015-09-28 ===
| | | * | |   df37727a 14:33 jch Merge branch 'jk/transfer-limit-protocol' into maint-2.3
| | | |\ \ \
| | | | |/ /
| | | | | /
| | | | |/
| | | |/|
| | | | | === 2015-09-23 ===
| | | | * 33cfccbb 11:35 jk  submodule: allow only certain protocols for submodule fetches
```

git blame

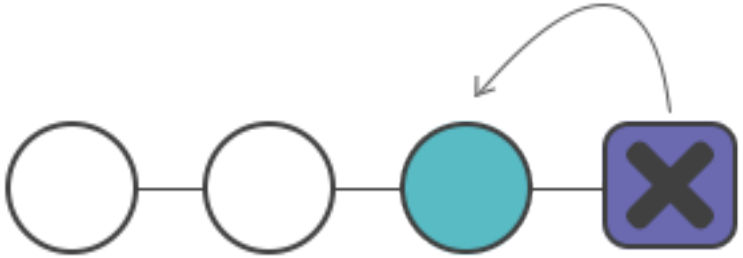


git blame es una herramienta flexible que permite varias opciones. La función más importante es que muestra los metadatos del autor para una línea particular dentro de un fichero. Se usa para explorar el histórico del fichero y descubrir quién fue el último autor que cambió una línea

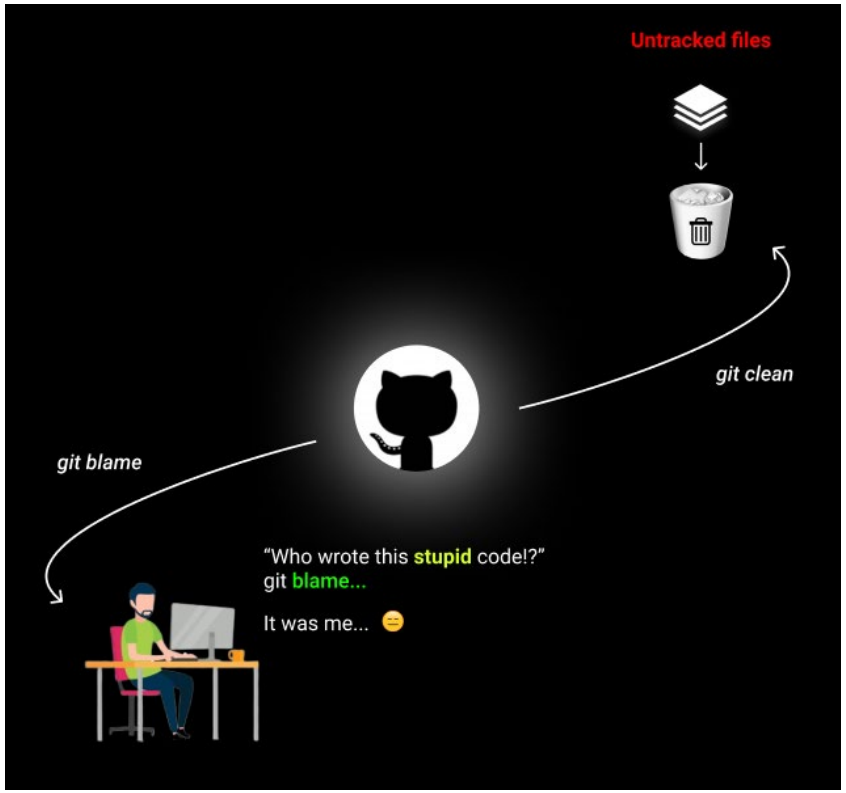
```
git log
commit 745nurcid82e4e5f3734c219d5a742b1c259926b2
Author: Bob Smith <bobsmith@3wdocs.com>
Date: Fri Apr 1 19:55:15 2019 +0000
Another commit to help git blame track the who, the what, and the when
commit eb06faedb1fdd159d62e4438fc8dbe9c9fe0728b
Author: Bob Smith <bobsmith@3wdocs.com>
Date: Fri Apr 1 19:53:23 2019 +0000
Creating the third commit, along with Nick and Robert, so that Nick can get git blame docs.
commit 990c2b6a84464fee153253dbf02e845a4db372bb
Merge: 82496ea 89feb84
Author: Tom Brown <tombrown@w3docs.com>
Date: Fri Apr 1 05:33:01 2019 +0000
Merged in tom-brown/git-blame-example/albert-so/readmemd-edited-online-with-bitbucket-1519865
README.md edited online with Bitbucket
commit 73nov64c885fe33d1182f2112885c2a64a4206ec
Author: Tom Brown <tombrown@w3docs.com>
Date: Fri Apr 1 00:54:03 2019 +0000
README.md edited online with Bitbucket
```

Revertir cambios en el repositorio

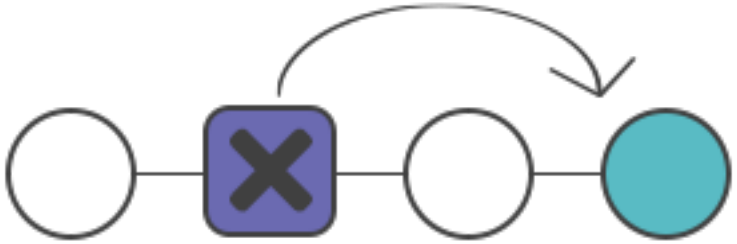
git clean

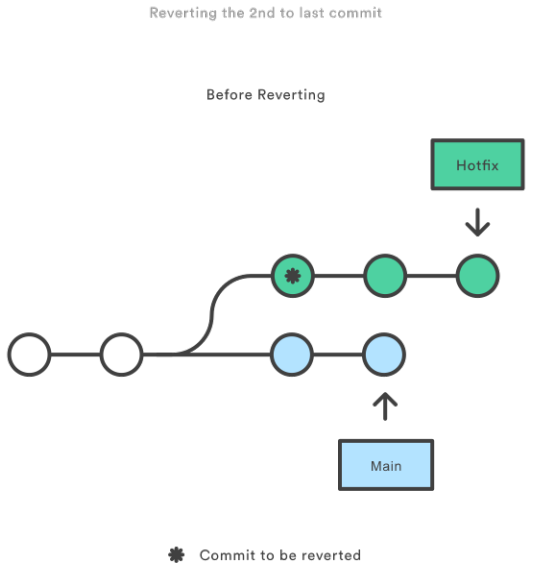


git clean es un comando de deshacer, que complementa a otros comandos como git reset y git checkout. Pero hay una diferencia fundamental, al contrario que esos dos últimos, git clean actúa sobre ficheros no ”trackeados”. Es decir, aquellos que han sido creados en el directorio de trabajo, pero que todavía no se han añadido en el staging area.

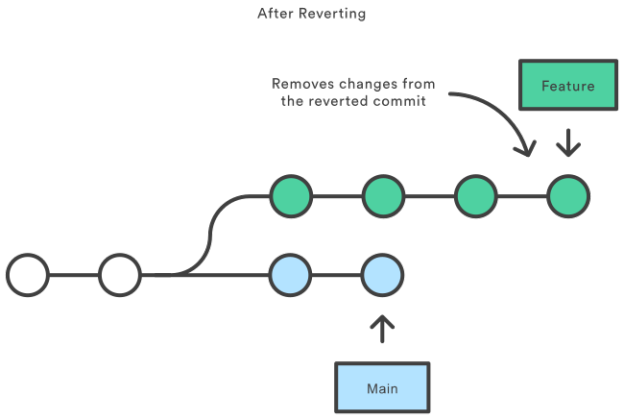


git revert



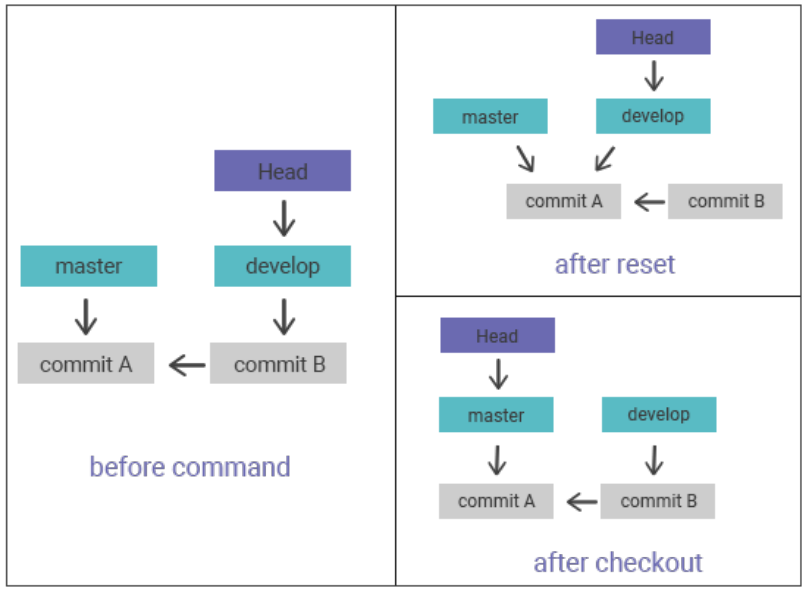


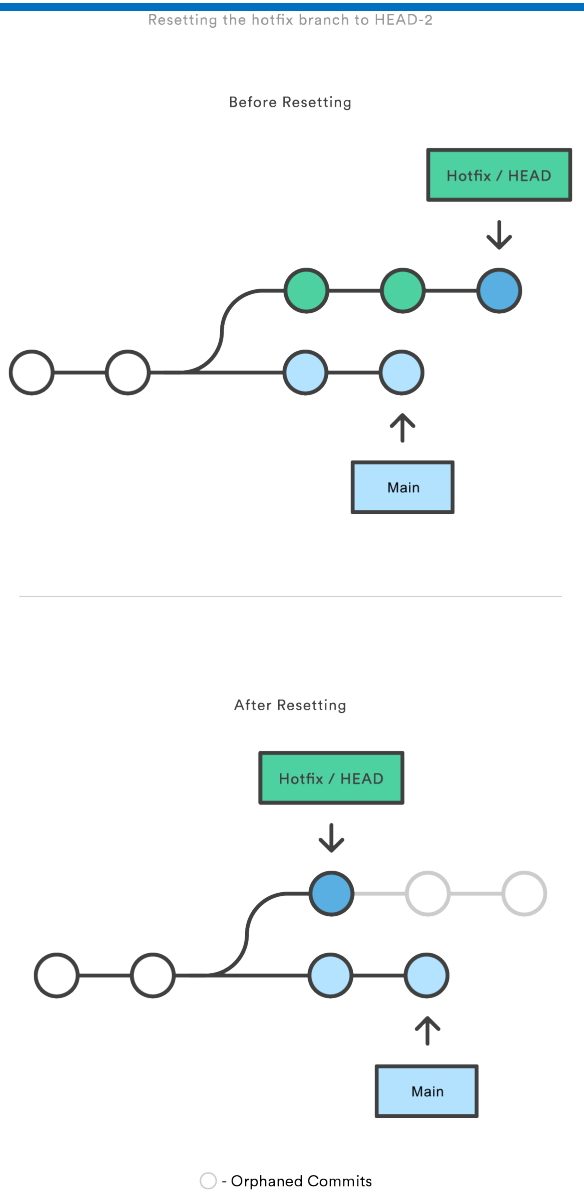
git revert es una operación de deshacer, pero no es la más apropiada. Revierte los cambios introducidos por el commit y añade un nuevo commit con el resultado del reverse. Esto no permite que git pierda el historial. git revert se usa para aplicar un commit inverso. Se puede usar para retroceder automáticamente y hacer correcciones.



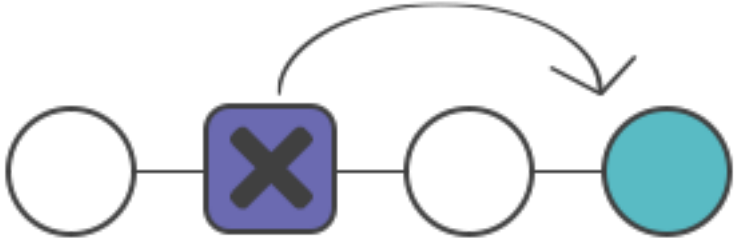
git reset

git reset es una herramienta para deshacer cambios. Tiene tres formas de invocación, en función de los denominados tres árboles de git (three trees of git). Estos sistemas incluyen HEAD (el histórico de commit), el índice de staging y el directorio de trabajo.

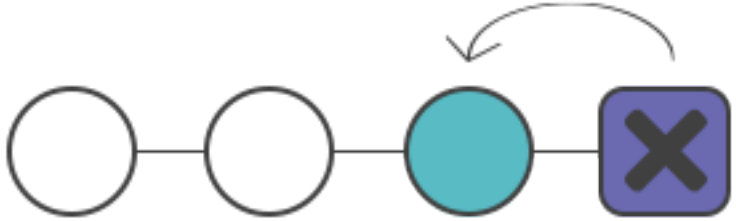




git revert vs git reset



revert

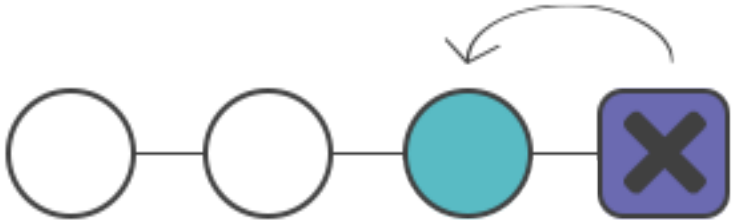


reset

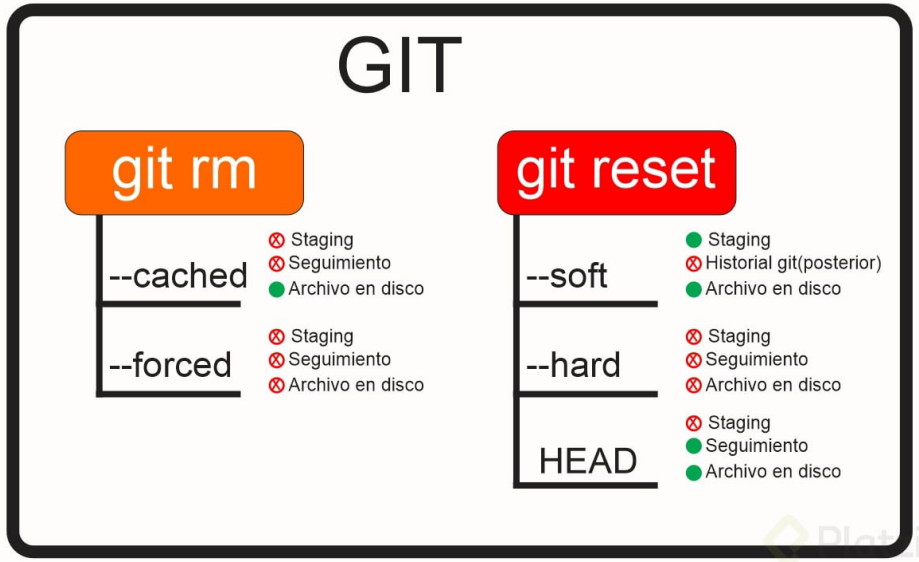
git reset deshace solamente un solo commit, mientras que git revert revierte al estado previo del proyecto, borrando commits que se han ejecutado satisfactoriamente.

Revertir se considera una operación segura para los commits que han sido publicados en el repositorio remoto.

git rm

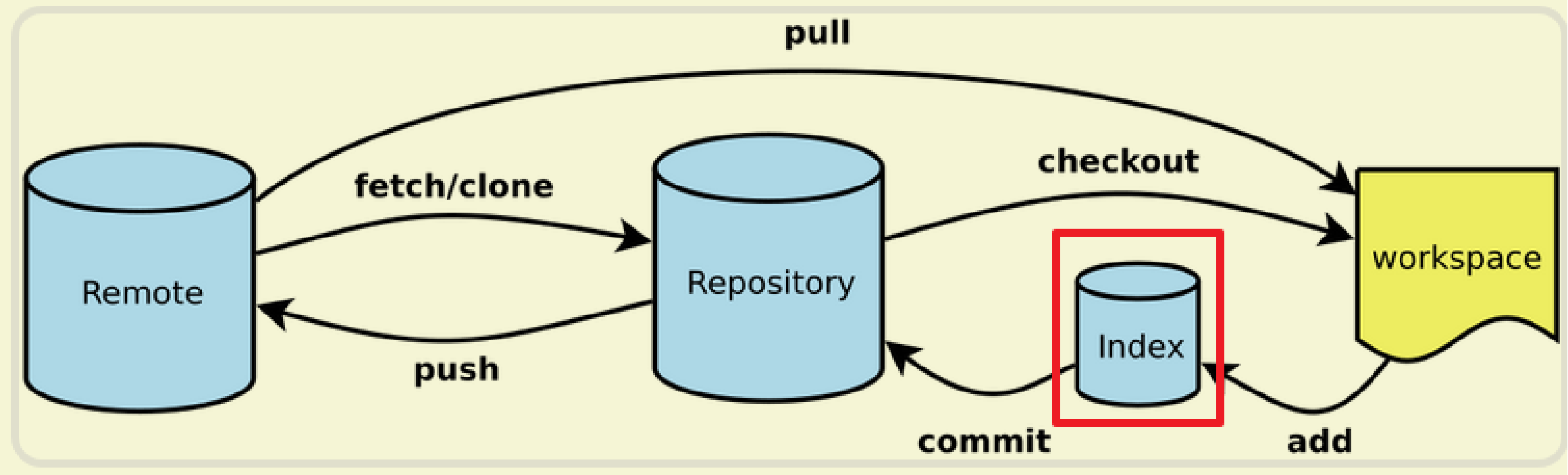


git rm elimina ficheros específicos o un grupo de ficheros de un repositorio git. La principal función de este comando es remover ficheros “trackeados” del index (staging). También puede ser usado para eliminar ficheros tanto del staging como del directorio de trabajo. Pero un fichero no puede ser eliminado solo del directorio de trabajo (ver git add). No sirve para eliminar ramas (ver git branch)

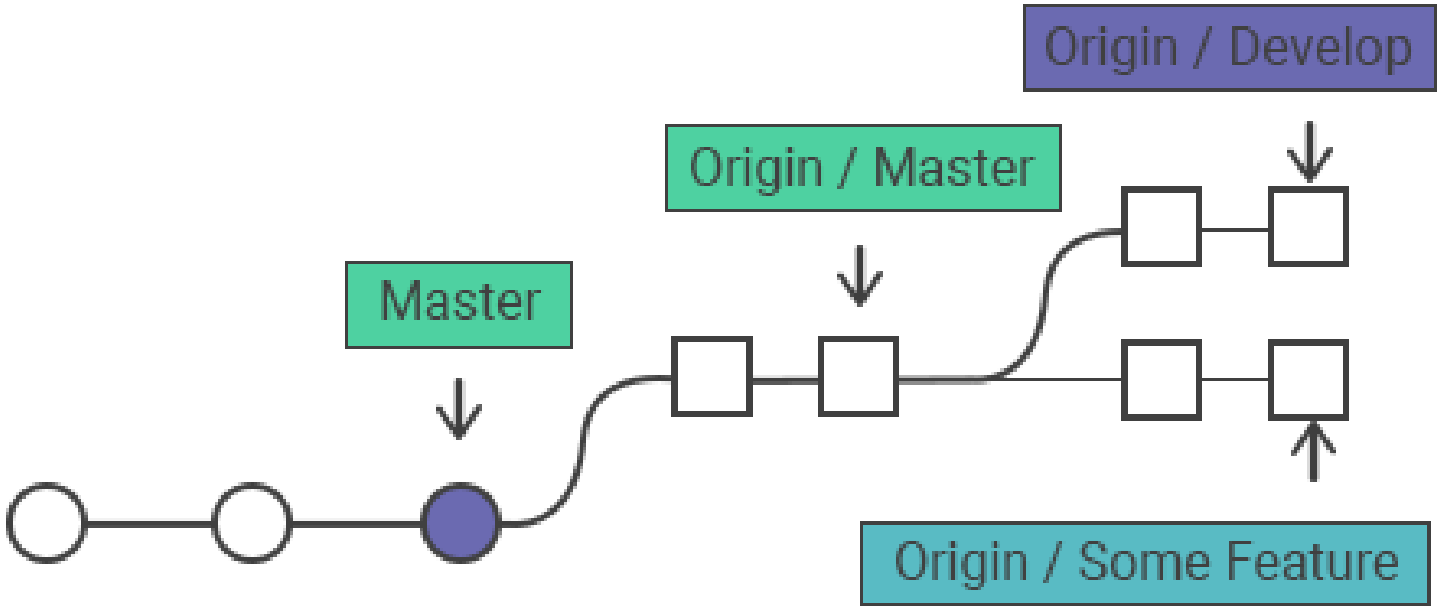


Actualización del repositorio remoto

Operaciones principales en Git

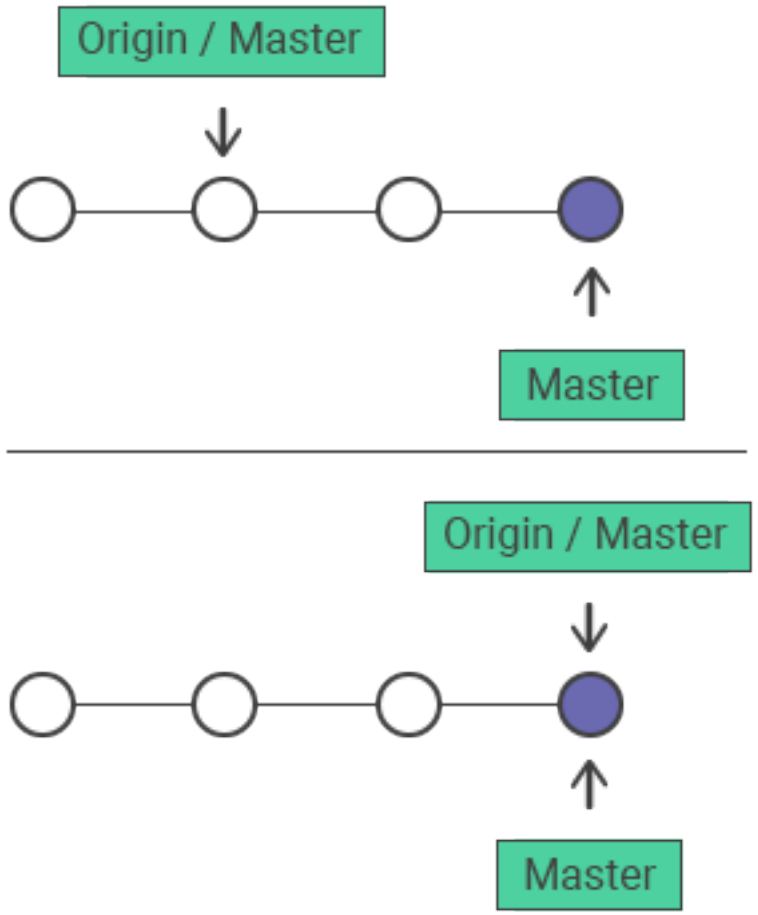


git fetch



git fetch se usa para descargar commits, ficheros y referencia de un **repositorio remoto** en el **repositorio local**. Se usa para ver en qué han estado trabajando otros miembros del equipo

git push



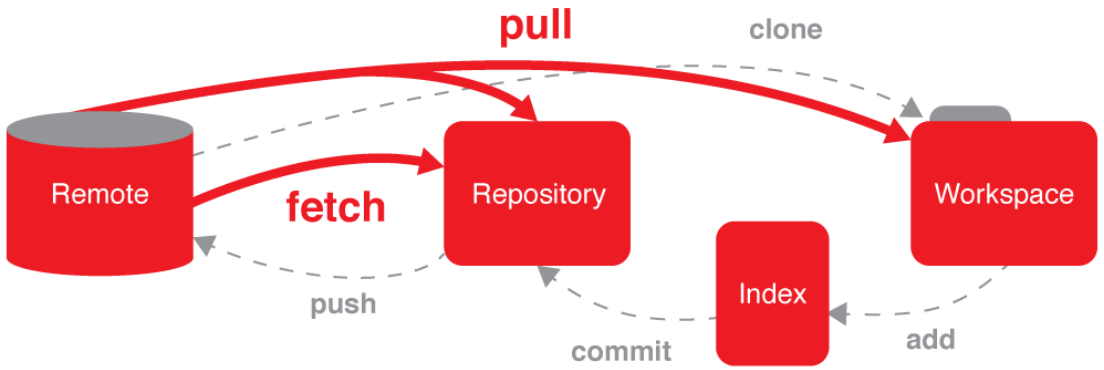
git push se usa para publicar los cambios locales en el repo central/remoto. Una vez hayas hecho cambios en tu repo local, puedes usarlo para compartir tus cambios con otros miembros del equipo. Está claramente involucrado en los procesos de sincronización.

Ej: la rama master local pasa al master del repo central y publica los cambios invocando git push origin master

git pull

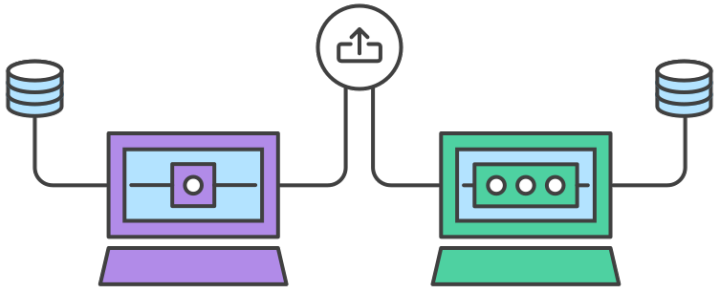
git pull obtiene y descarga el contenido del repositorio remoto e integra los cambios en el repositorio local. Este comando se llama como una combinación de git fetch seguido de git merge.

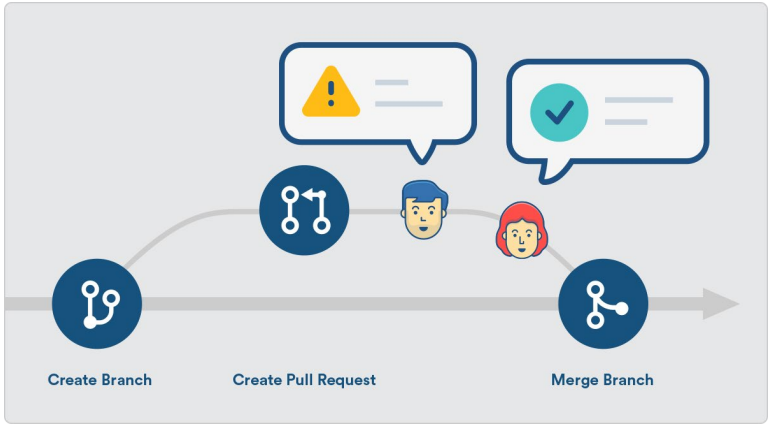
NOTA: téngase en cuenta que **fetch** trae los cambios al **repositorio local**, pero **pull** al **repositorio local** y además los trae al **directorio de trabajo**.

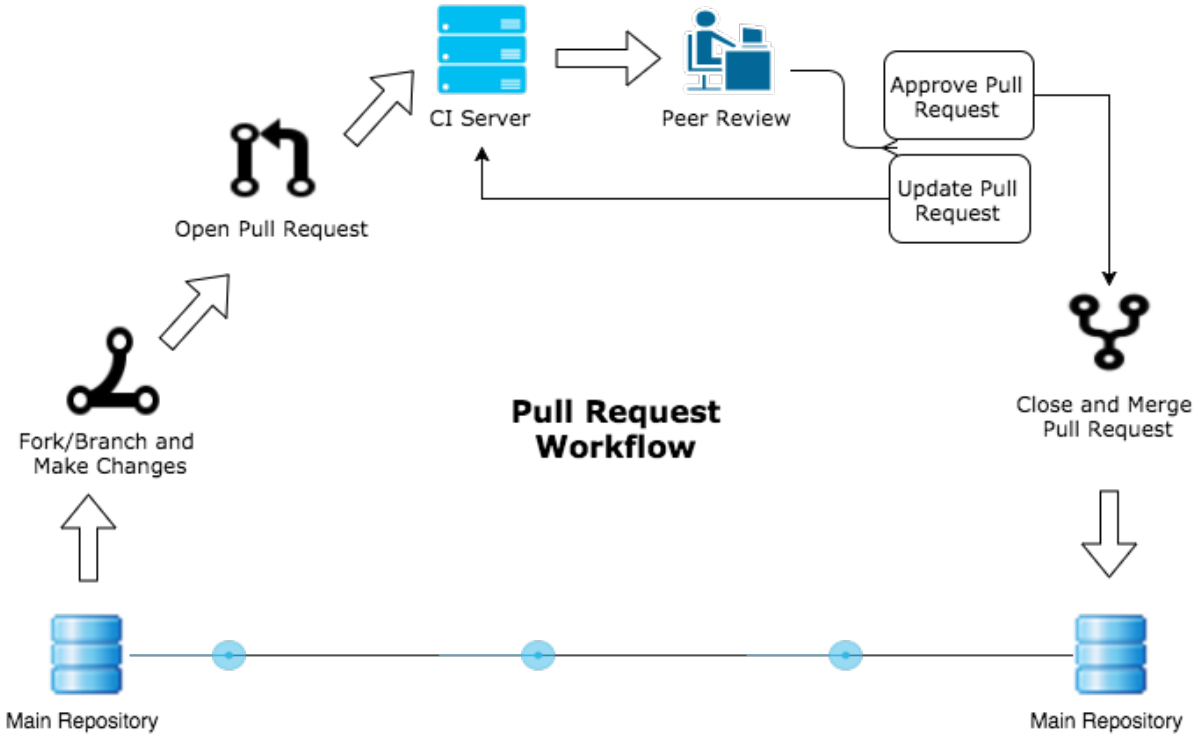


pull request

Un pull request permite hacer una solicitud de incorporación de cambios, es una funcionalidad que facilita la colaboración entre desarrolladores, p.e. en Bitbucket.
Cuando se hace un pull request solicitas que otro desarrollador incorpore una rama de tu repo al suyo.
Información a proporcionar: repo origen, rama origen, repo destino y rama destino.

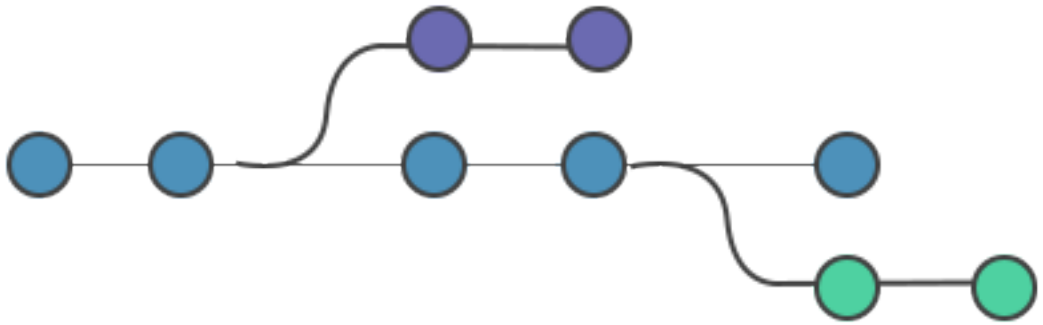






Ramificación en el repositorio

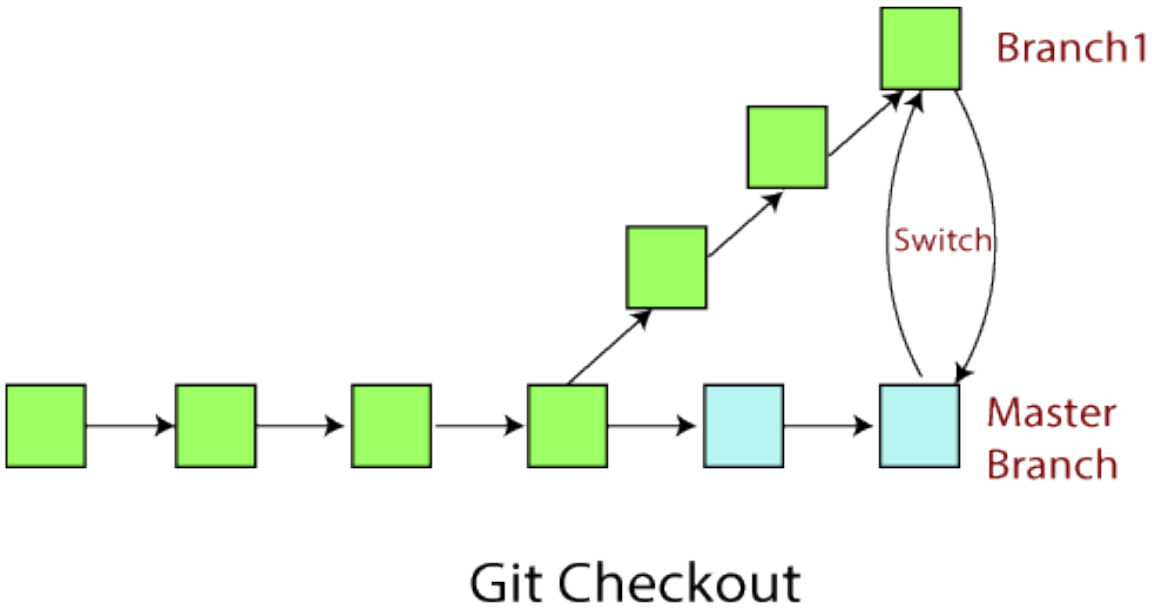
git branch



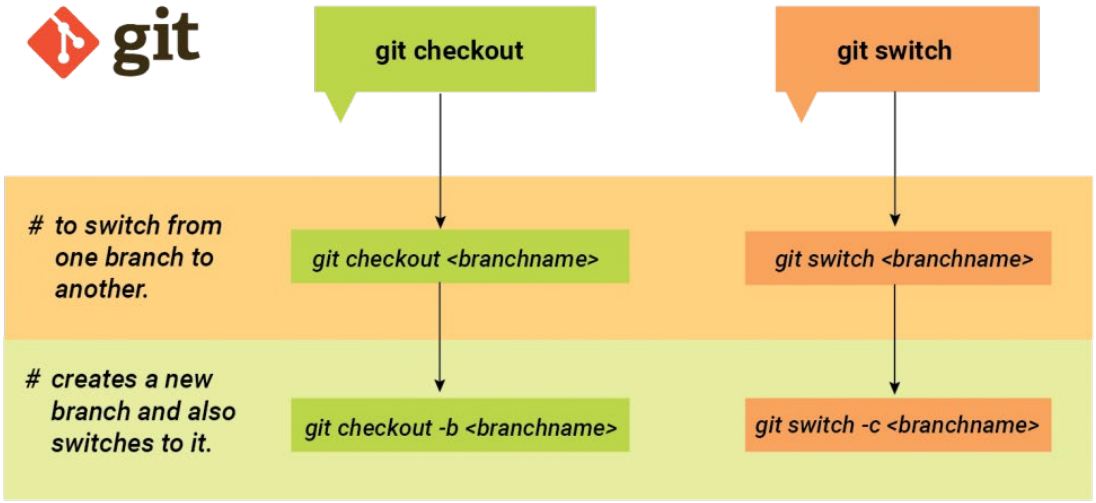
El comando git branch crea, lista y borra ramas. No permite el cambio entre ramas o volver a poner juntas de nuevo dos ramas que han sido bifurcadas. Por tanto, git branch está integrado con los comandos git checkout y git merge.

Comando	Descripción
git branch	Lista de todas las ramas del repositorio (igual que git branch --list)
git branch <branch>	Crea una nueva rama llamada <branch> pero hace el check out de la nueva rama
git branch -d <branch>	Borra una rama. Si hay cambios sin mergear, git no te permite borrarla
git branch -D <branch>	Fuerza el borrado de una rama, incluso si está sin mergear. Hacerlo sólo si se está seguro de quererla borrar de forma permanente
git branch -m <branch>	Mueve o renombra la rama actual a <branch>
git branch -a	Lista todas las ramas remotas

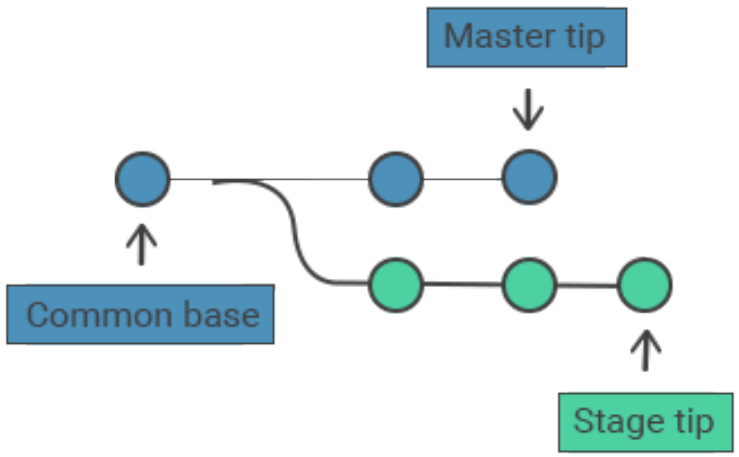
git checkout

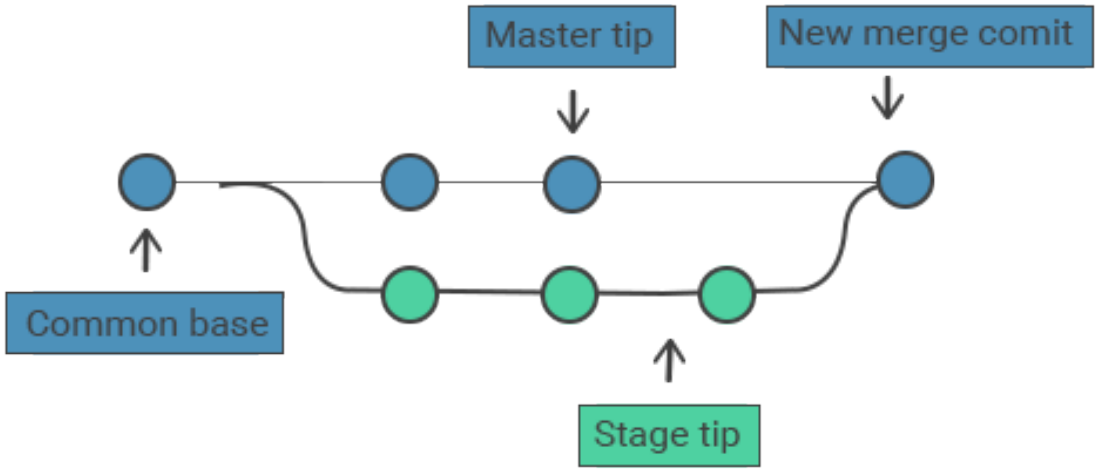


git checkout permite cambiar entre ramas o restaura el árbol de ficheros de trabajo. Opera sobre ficheros, commits y ramas. Permite cambiar entre múltiples características dentro de un mismo repositorio, como p.e. crear una rama y cambiar a ella



git merge



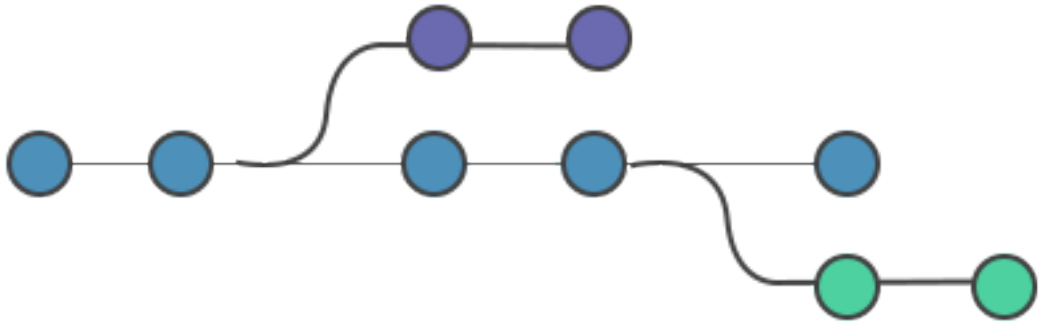


El uso primario de git merge es combinar dos ramas. También se usa para unir varios commits en un solo histórico.

En este ejemplo, git merge toma dos ramas y busca el punto común entre ambas. El commit de common base crea un nuevo commit que une los cambios de la secuencia de cada merge. Aquí tenemos dos ramas: master y stage. Debemos mergear la rama stage en la rama master.

Los commit de un merge son únicos, porque tienen dos commit padre. Git automáticamente combina las historias por separado cuando se crea un nuevo commit de merge. No combina los datos modificados en ambos históricos. Estos es lo que se llama un conflicto

merge conflicts



Los conflictos de merge suceden cuando múltiples autores editan el mismo contenido o cuando un desarrollador borra un fichero en el que otro estaba haciendo cambios. Para resolverlo, los desarrolladores trabajan en ramas independientes. El comando git merge es responsable de combinar esas ramas y de resolver los conflictos de edición.

Git actúa de forma automática en la mayoría de casos. En aquellos en los que no es posible, marca el fichero como en conflicto y detiene el proceso de merge. En estas ocasiones es necesario revisar en detalle el fichero y ver cómo y qué cambios integrar. Para facilitar el proceso, aparecen marcas para que sea fácilmente localizable.

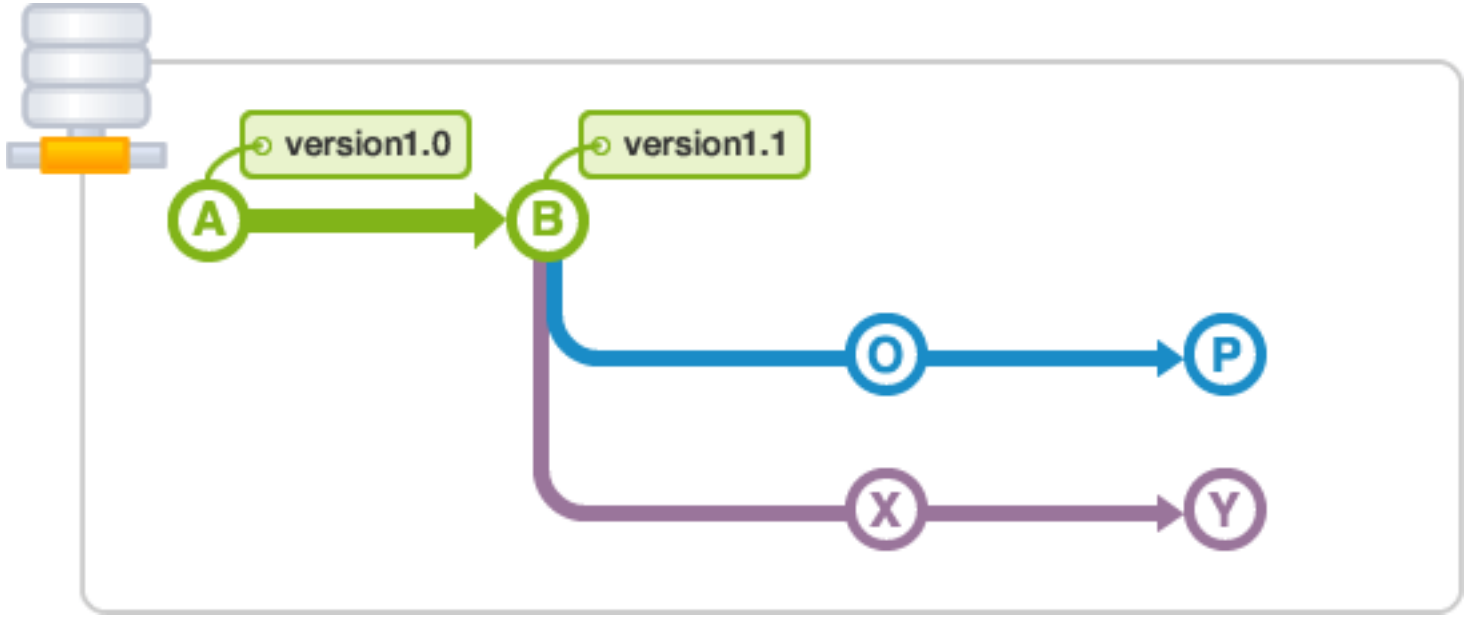
Una vez solucionado, se eliminan todas las marcas y conflictos, se ejecuta “git add .” y luego se hace un commit. Existen distintas estrategias de resolución (recursive, resolve, octopus, …)

```
<<<<<<< HEAD
=======
>>>>>>> branch_to_merge
```

Otros comandos

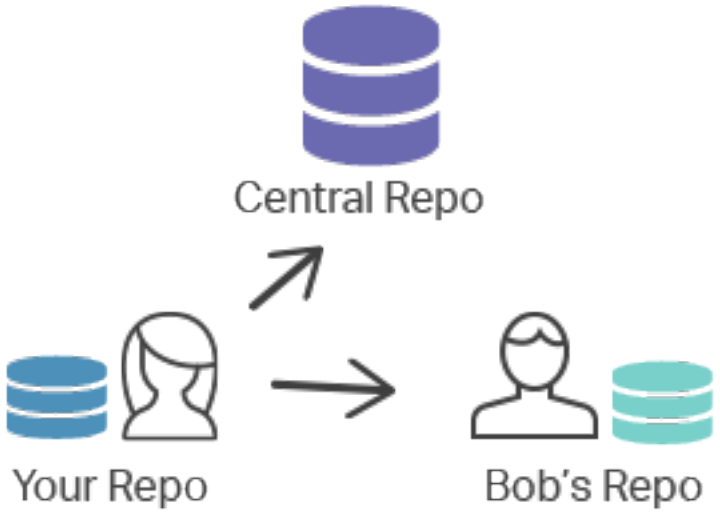
git tag

Los git tag son referencias que muestran puntos particulares en un histórico de git. La principal funcionalidad de este etiquetado es capturar un punto en en un histórico Git que marca una versión release. Los tags no cambian. Después de que se crea, **no tiene histórico de commits**



Se puede hacer un checkout de un tag y/o rama específicos:
git checkout tags / <tag> -b <branch>

git remote

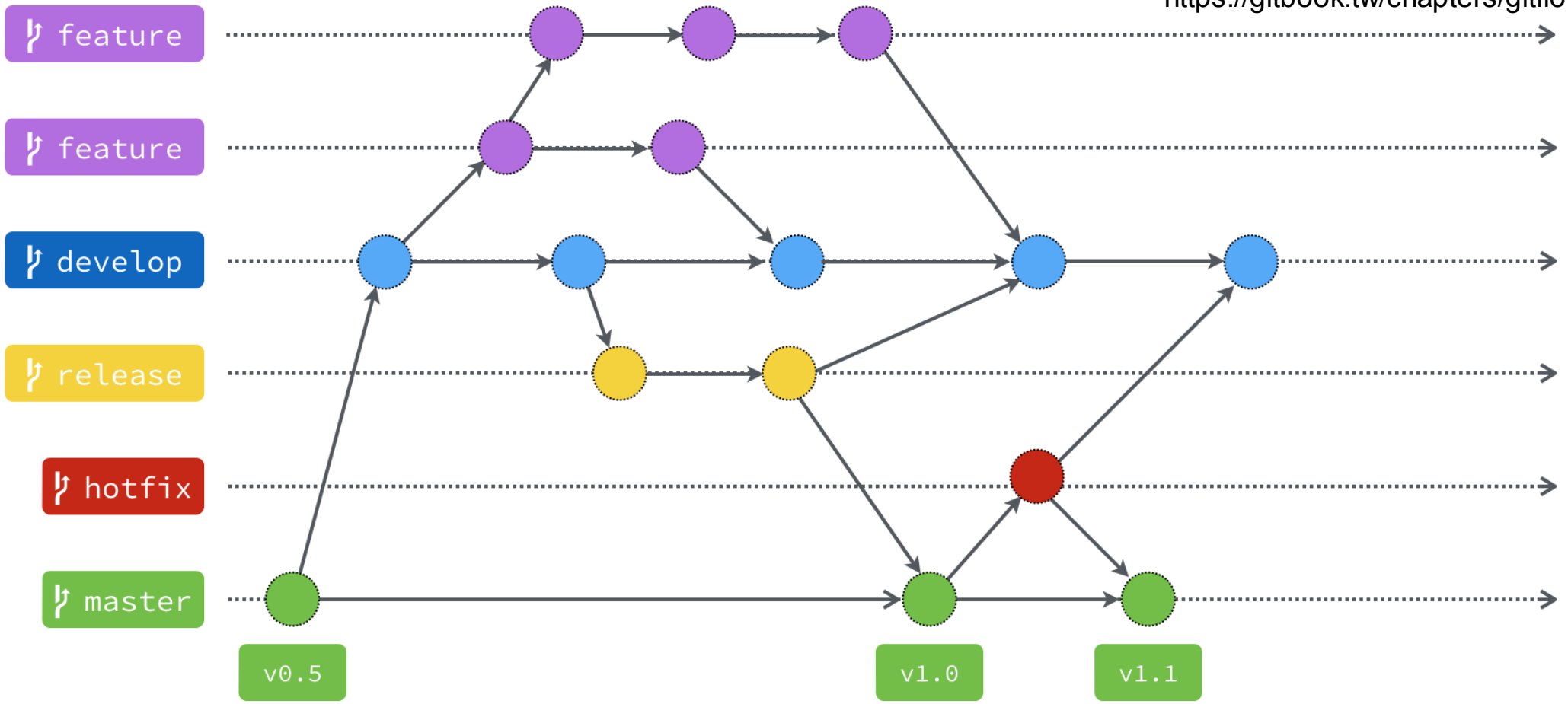


git remote está diseñado para crear, ver y eliminar conexiones a otros repositorios. Son como “favoritos” a otros repositorios, que vienen siendo identificadores sencillos para hacer referencias a urls que pueden no serlo

GitFlow

Git Flow

https://gitbook.tw/chapters/gitflow/why-need-git-flow



Git Flow es una estrategia de ramificación (branching model) para Git que ayuda a organizar el desarrollo de software de manera estructurada. Define un conjunto de reglas para gestionar el desarrollo.

Git Flow: Ramas Develop y Main

Git Flow es una estrategia de ramificación (branching model) para Git que ayuda a organizar el desarrollo de software de manera estructurada. Define un conjunto de reglas para gestionar el desarrollo.

En lugar de una sola rama principal, GitFlow utiliza dos ramas para registrar el historial del proyecto.

- La rama “Main” almacena el historial oficial de lanzamientos.
- La rama “Develop” sirve como una rama de integración para las características.

Para comenzar con GitFlow es crear una rama “Develop”.

```
$ git branch develop
$ git push -u origin develop
```

Git Flow: Rama Feature

- Creación una rama "feature“ sin la extensión "git-flow":
```
$ git checkout develop
$ git checkout -b feature_branch
```

- Creación una rama "feature“ con la extensión "git-flow":
```
$ git flow feature start feature_branch
```

Cuando se haya terminado el desarrollo de la “feature”, se debe fusionar el “feature_branch” en “develop”.

- Finalización una rama "feature“ sin la extensión "git-flow":
```
$ git checkout develop
$ git merge feature_branch
```

- Finalización una rama "feature“ con la extensión "git-flow":
```
$ git flow feature finish feature_branch
```

Conventional Commits

Conventional Commits

https://www.conventionalcommits.org/es/v1.0.0/

“Conventional Commits” define una convención sobre los mensajes de commits. Proporciona un conjunto sencillo de reglas para crear un historial de commits explícito. Esta convención encaja con SemVer (MAJOR.MINOR.PACH), al describir en los mensajes de los commits las funcionalidades Y cambios de ruptura hechos.

El mensaje del commit debe ser estructurado de la siguiente manera:

```
<tipo>[ámbito opcional]: <descripción>

[cuerpo opcional]

[nota(s) al pie opcional(es)]
```

Conventional Commits

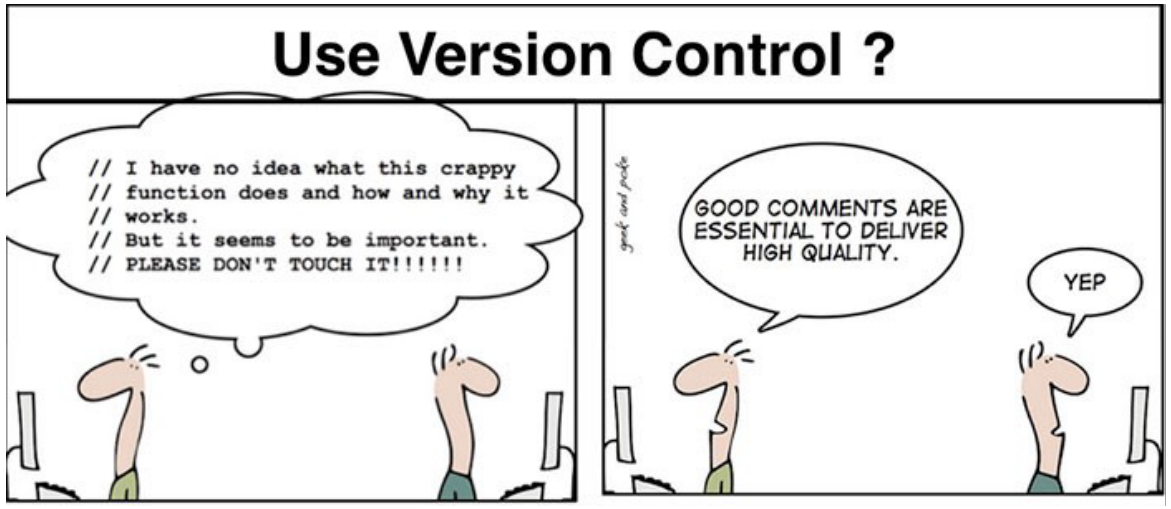
https://www.conventionalcommits.org/es/v1.0.0/

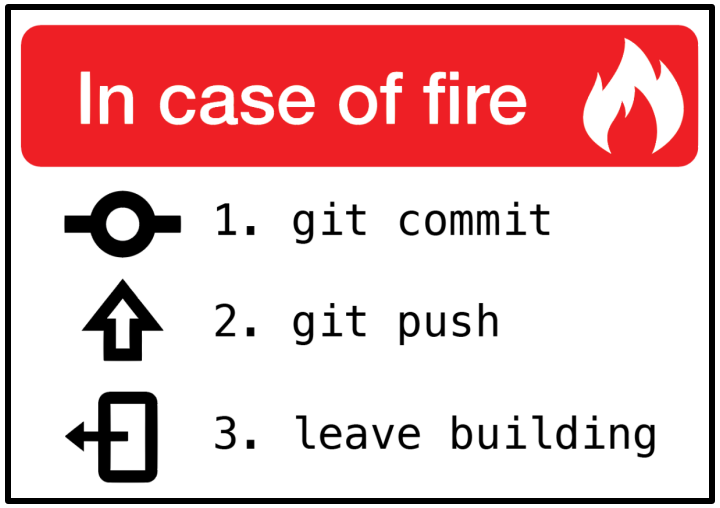
El commit contiene los siguientes elementos estructurales, para comunicar la intención a los consumidores de tu librería:

1. **fix:** un commit de *tipo* `fix` corrige un error en la base del código (se correlaciona con `PATCH` en el Versionado Semántico).
2. **feat:** un commit de *tipo* `feat` introduce una nueva funcionalidad en la base del código (se correlaciona con `MINOR` en el Versionado Semántico).
3. **BREAKING CHANGE:** un commit que contiene la nota al pie `BREAKING CHANGE:`, o que agrega un `!` después del tipo/ámbito, introduce un cambio de ruptura de API (se correlaciona con `MAJOR` en el Versionado Semántico). Un BREAKING CHANGE puede ser parte de commits de cualquier *tipo*.
4. *tipos* distintos a `fix:` y `feat:` están permitidos, por ejemplo **@commitlint/config-conventional** (basados en **la convención de Angular**) que recomienda `build:`, `chore:`, `ci:`, `docs:`, `style:`, `refactor:`, `perf:`, `test:`, y otros.
5. *notas al pie* distintas de `BREAKING CHANGE: <descripción>` pueden ser añadidas y siguen una convención similar al **formato git trailer**.

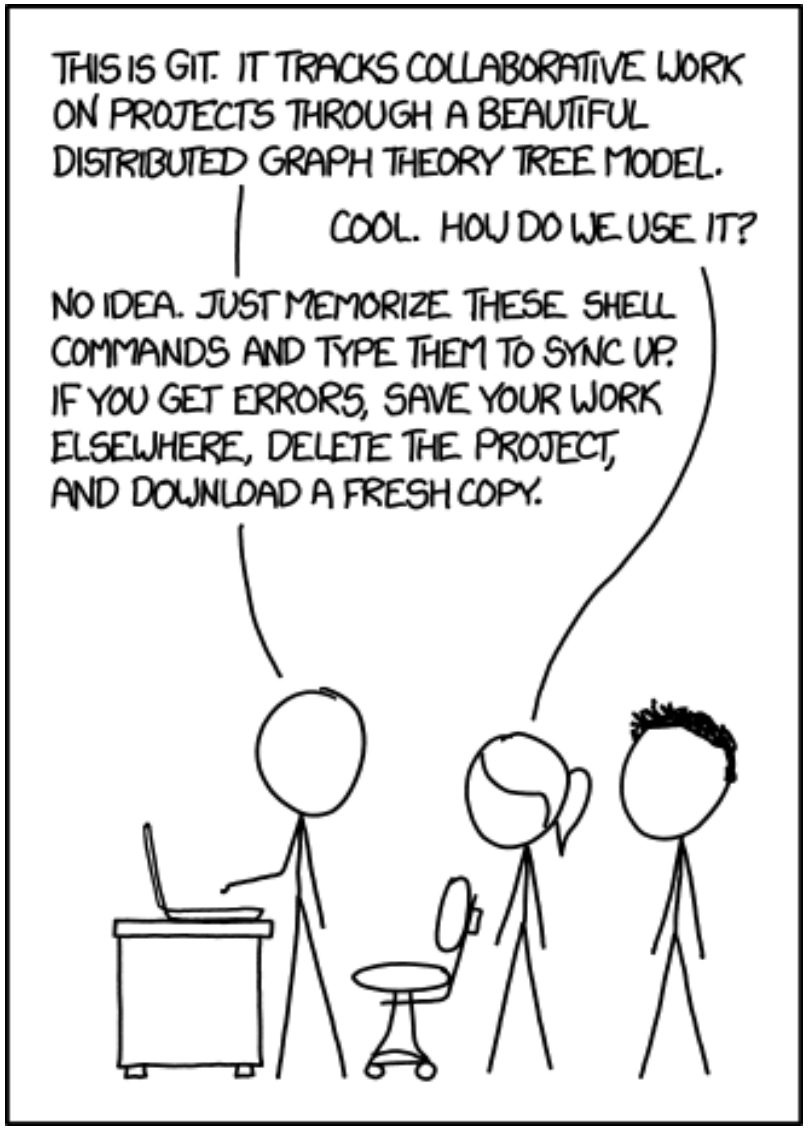
Tipos adicionales no son obligatorios en la especificación de Commits Convencionales, y no tienen un efecto implícito en el Versionado Semántico (al menos que incluyan un BREAKING CHANGE). Un ámbito puede ser añadido al tipo de un commit, para proveer información adicional contextual y debe ser contenido entre paréntesis, ej., `feat(parser): add ability to parse arrays`.

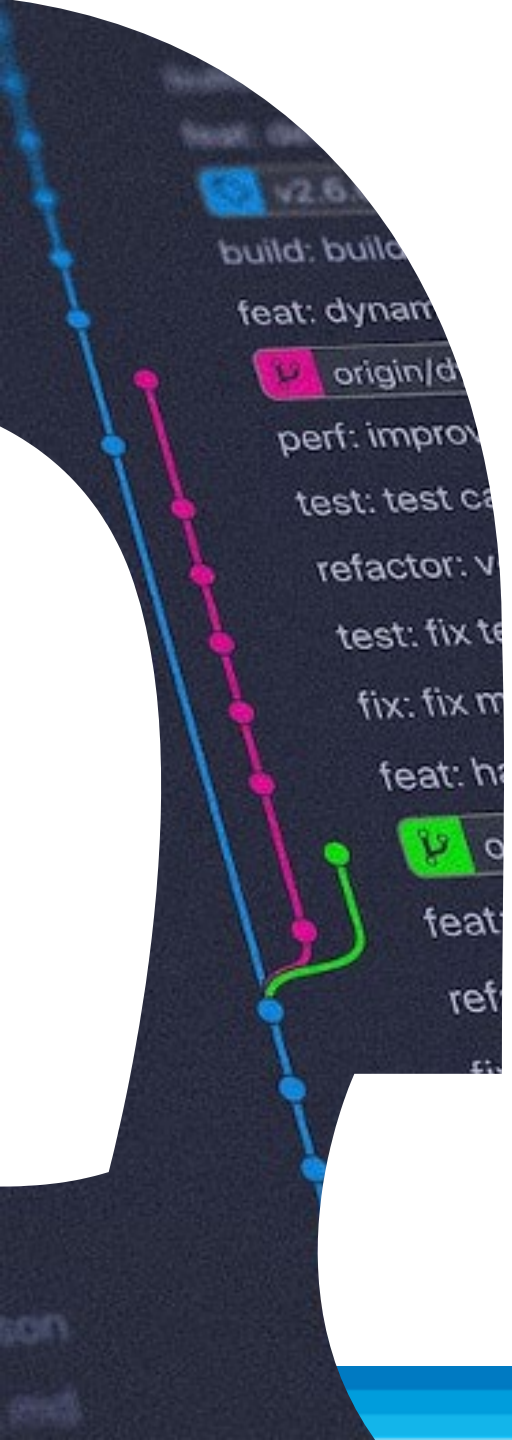
Moraleja…


Use Version Control ?
// I have no idea what this crappy
// function does and how and why it
// works.
// But it seems to be important.
// PLEASE DON'T TOUCH IT!!!!!!
GOOD COMMENTS ARE ESSENTIAL TO DELIVER HIGH QUALITY.
YEP


In case of fire
1. git commit
2. git push
3. leave building

WOW! HOW DID YOU GET LIKE THAT?
69,420,964 commits
EVERY TIME I change a line ...
69,420,964 commits
...I DO ONE PUSH
69,420,964 commits
JESUS CHRIST.


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 MEMORIZE 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.



ULPGC
Universidad de
Las Palmas de
Gran Canaria

eII
ESCUELA DE INGENIERÍA
INFORMÁTICA

Control de Versiones

Nelson Monzón
Daniel Santana Cedrés
Jonay Suárez Ramírez

Producción de Software