

Buenas prácticas y agilidad

# Control de versiones y Git

---

Josu Gorostegui

---

## ¿Quiénes impartimos este módulo?

- **Josu Gorostegui** Software Architect. ARK Team
- **Juan Ignacio Forcén** Team leader. Verificación Documental

Módulo Agile, buenas prácticas de desarrollo y conocimientos transversales.

## ¿Por qué?

Para conocer **prácticas transversales** del desarrollo software

Se profundizará en:

- Cuestiones transversales, git, control de versiones, ...
- Generación de Código con Inteligencia Artificial
- Frameworks de Trabajo
- Cómo trabajar en equipo, qué me motiva...



**Josu Gorostegui**



<https://linkedin.com/in/josugorostegui>



<https://jgorostegui.github.io/>

## Ingeniero de software y ML

Formado en Ingeniería de Telecomunicaciones, inició su trayectoria en Donostia dedicándose a la investigación en el ámbito de imagen y vídeo.

Motivado por el avance de la inteligencia artificial, se trasladó a Pamplona para especializarse en biometría y reconocimiento facial. Posteriormente, asumió roles de Product Manager en biometría facial y de voz.

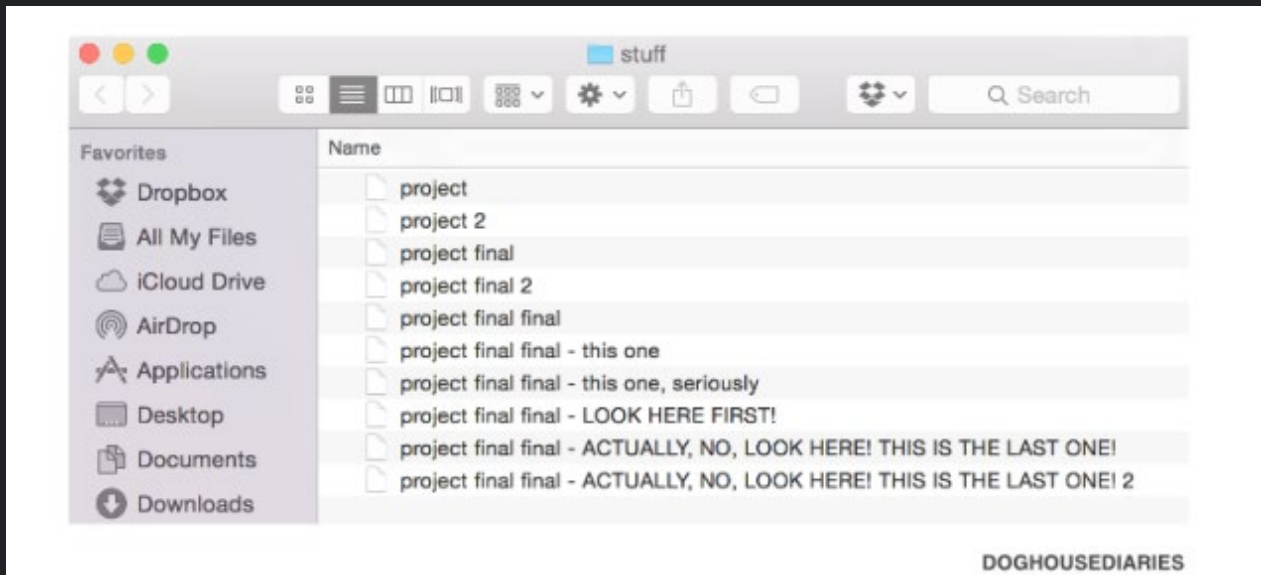
Actualmente, desempeña funciones como Arquitecto de Software en Tecnología.

- ▶ Introducción al Control de Versiones
- ▶ Qué es Git
- ▶ Fundamentos de Git
- ▶ Flujo de trabajo básico con Git
- ▶ Trabajar con Ramas
- ▶ Trabajar con repositorios remotos
- ▶ Buenas prácticas en Git

## ¿Qué es el control de versiones?

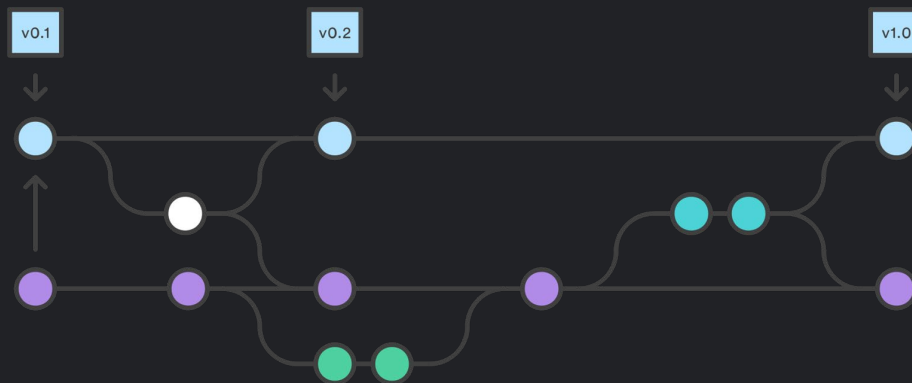
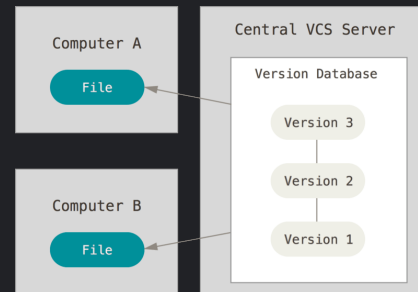
- El control de versiones es como una máquina del tiempo para nuestro código.
- A menudo es denominado "control de código fuente", es una metodología que rastrea y administra los cambios en el código de software a lo largo del tiempo.
- Es necesario que nuestro código mantenga un histórico de cambios para poder trabajar en un entorno **colaborativo**.
- Un sistema de control de versiones le permite realizar un **seguimiento** de los cambios en sus documentos.
- Permite ir a una **versión anterior** del código anterior.

## Trabajando sin control de versiones



## ¿Por qué es necesario?

1. Protección del Código Fuente
2. Trabajo Colaborativo
3. Historial Completo
4. Creación de Ramas y Fusiones
5. Trazabilidad
6. Reducción de Errores y Problemas

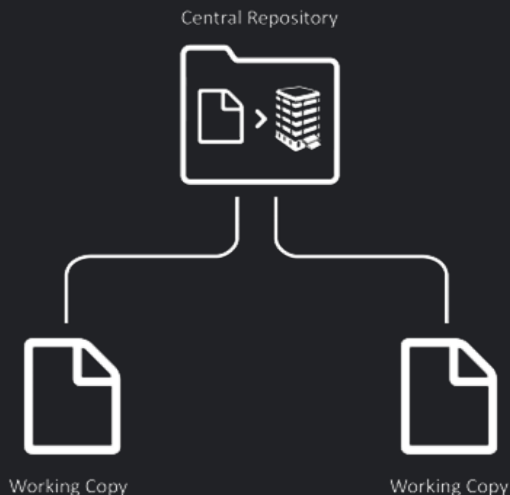




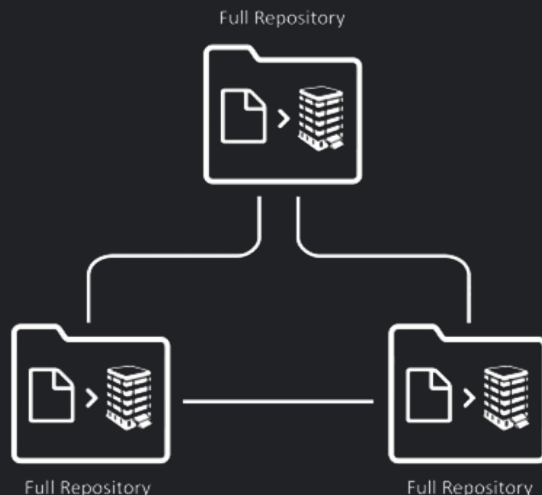
## Central vs distribuido

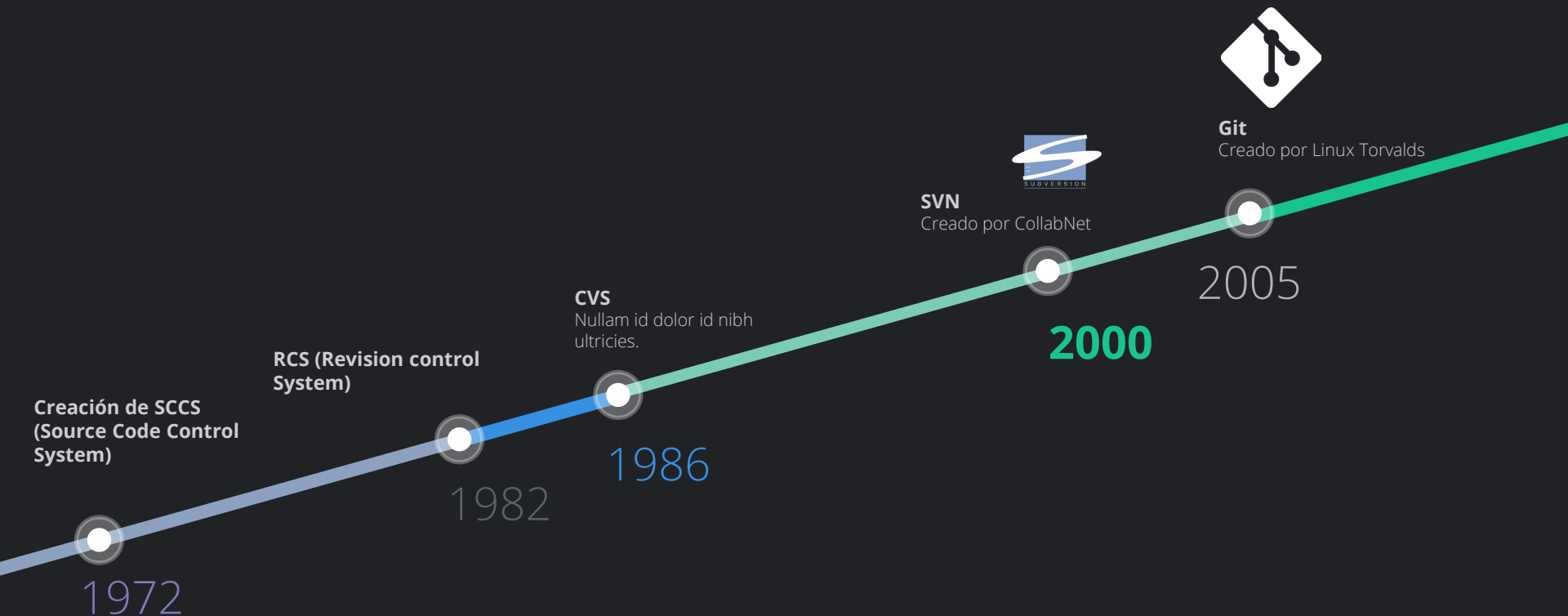
- **Central:** Se tiene una copia del proyecto en un servidor central y los integrantes del equipo realizan una actualización de estos archivos de acuerdo a los cambios que realizan.
- **Distribuido:** Los desarrolladores trabajan en su repositorio local y los cambios se actualizan entre repositorios.

### CENTRALIZED



### DISTRIBUTED





### RCS (Revision Control System)

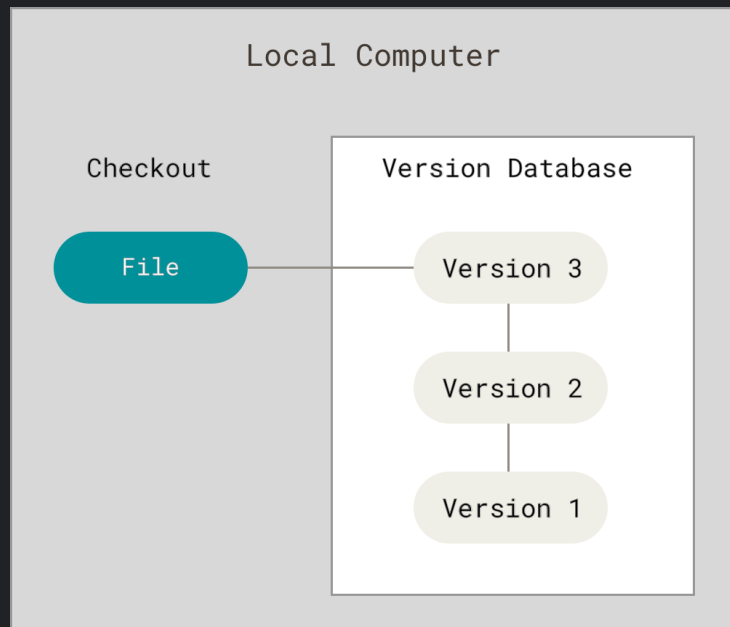
Fue uno de los primeros sistemas de control de versiones. Guardaba conjuntos de cambios en archivos especiales.

#### Ventajas

- Mejor que copiar carpetas manualmente.
- Gran salto con respecto a `\_v2\_final\_final.doc`. Por primera vez, había una forma estructurada de registrar el historial de un archivo.

#### Desventajas

- Solo funcionaba localmente.
- Si dos desarrolladores intentaban hacer cambios al mismo tiempo, surgían complicaciones.



Los sistemas centralizados como **CVS** y posteriormente **Subversion (SVN)**

### Características

- Todos los cambios se registran en un servidor central.
- Los desarrolladores obtienen la última versión del servidor y envían sus cambios.

### Ventajas

- Facilita la colaboración.
- Control centralizado.

### Desventajas

- Dependencia de la red: Si el servidor falla y no hay copias de seguridad, se pierde todo.
- Todos los desarrolladores dependen de la accesibilidad del servidor central.

Dentro de sistemas distribuidos se encuentran Git y Mercurial

### Características

- Cada desarrollador tiene una copia local completa del historial del proyecto.
- Los cambios se pueden compartir entre diferentes repositorios.

### Ventajas

- Funciona sin conexión.
- Los repositorios pueden ser respaldados por múltiples personas, reduciendo el riesgo de pérdida.
- Permite colaboraciones más flexibles.

### Desventajas

- Curva de aprendizaje: El modelo mental es un poco más complejo al principio. Hay que entender la diferencia entre tu repositorio local y el remoto (push vs. pull).

# ¿Qué es Git?

Git es un sistema de control de versiones (VCS) y gestor de código fuente (SCM).

Creador: Linus Torvalds (creador del Kernel de Linux), 2005.

Principales características:

- Distribuido: Cada clon es un repositorio completo (como vimos).
- Rendimiento: Diseñado para ser extremadamente rápido.
- Seguridad: La integridad del contenido se garantiza mediante criptografía (hashing SHA-1).
- Flexibilidad: Permite flujos de trabajo muy complejos y no lineales (ramificaciones).
- Gratuito y de Código Abierto: Accesible para todos y mantenido por una enorme comunidad.



**El Problema (1991-2005):** Gestionar el proyecto de código abierto más grande del mundo con herramientas inadecuadas. El equipo del Kernel de Linux necesitaba un sistema que fuera:

- Distribuido: Para sus miles de colaboradores globales.
- Rápido: Para manejar un historial masivo.
- Seguro: Para proteger la integridad del código.

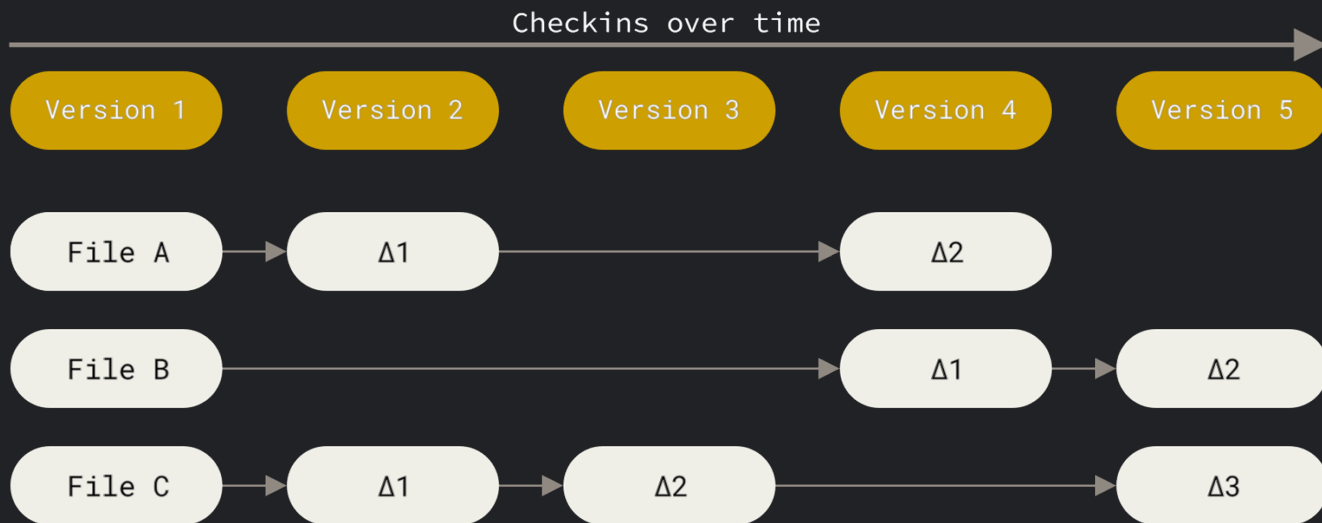
**La Crisis (Abril de 2005):** La herramienta que usaban (BitKeeper) deja de ser gratuita. El proyecto se queda sin sistema de control de versiones.

**La Solución:** Linus Torvalds, frustrado con las alternativas, crea su propia herramienta desde cero, basándola en las lecciones aprendidas durante más de una década.



## Copias instantáneas, no diferencias

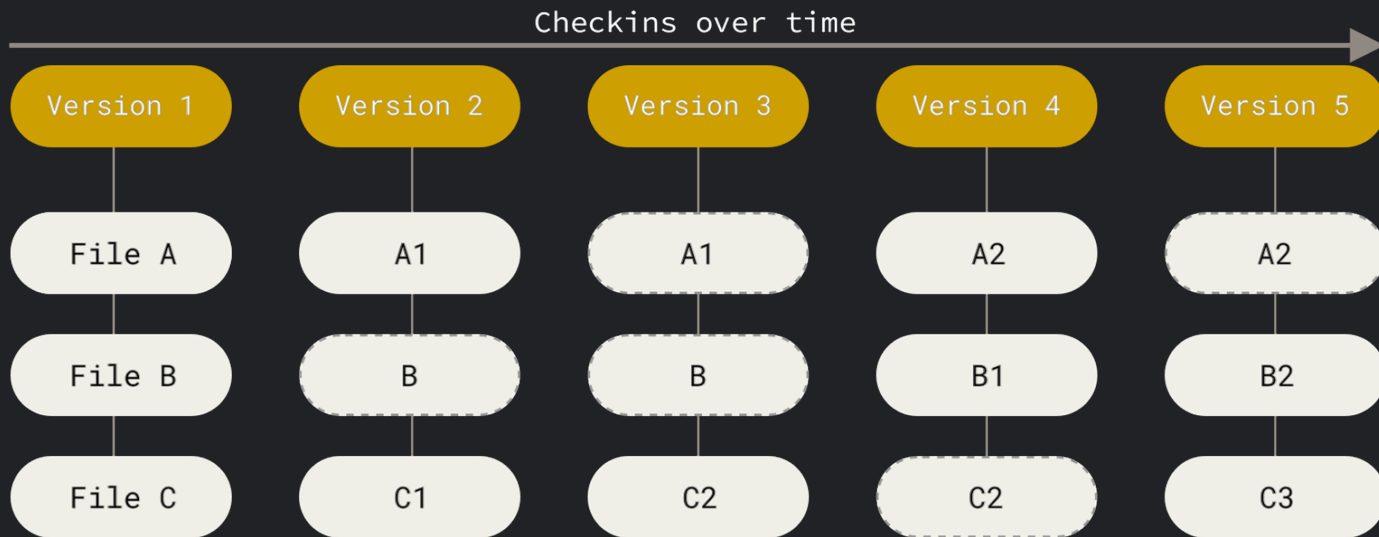
- Otros sistemas almacenan la información como un conjunto de archivos y modificaciones de ellos a través del tiempo.





## Copias instantáneas, no diferencias

- Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura.



## Git Internals

Git, es un sistema de archivos direccionable por contenido con una interfaz de usuario de sistema de control de versiones (VCS) construida sobre él.

Esto significa que Git, es un almacén de datos clave-valor que permite insertar cualquier tipo de contenido y recuperar ese contenido más tarde utilizando una clave única proporcionada por Git.

- Comandos Plumbing: Bajo nivel, utilizados para operaciones internas y scripts.
  - Ejemplo: `git hash-object`, `git cat-file`
- Comandos Porcelain: Interfaz de usuario amigable para las operaciones diarias.
  - Ejemplo: `git add`, `git commit`, `git branch`

### El directorio `.git`

Esta carpeta se crea nada más hacer `git init`, donde se encuentra todo lo almacenado/manipulado por git.

```
$ git init
```

```
$ ls -F1 .git/
```

```
config  description  HEAD  hooks/  info/  objects/  refs/
```

# Git Internals

## El directorio `.git`

Esta carpeta se crea nada más hacer `git init`, donde se encuentra todo lo almacenado/manipulado por git.

```
$ git init
```

```
$ ls -F1 .git/
```

```
config  description  HEAD  hooks/  info/  objects/  refs/
```

## git y GitHub



Herramienta de control  
de versiones



Servicio de hosting para  
proyectos de Git

# git y GitHub

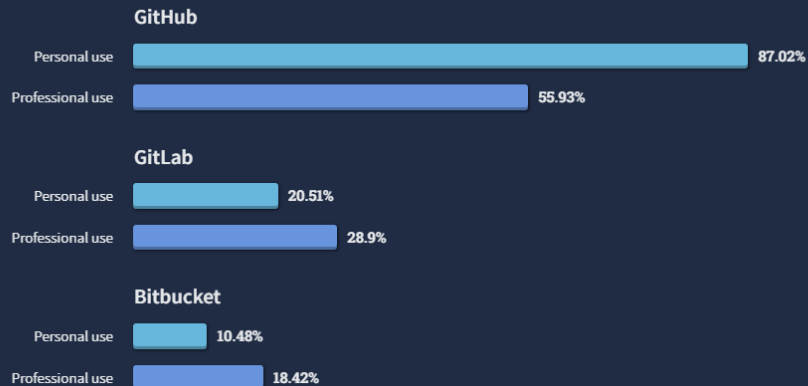


## git y GitHub

### Version control platforms

GitHub is the most popular Version Control for both personal and professional use. GitLab, Bitbucket, and Azure Repos are more likely used for professional purposes instead of personal.

67,035 responses



Diferentes plataformas y herramientas, algunas con GUI stand-alone o integradas en el editor.

### Plataformas

- GitHub permite trabajar con las herramientas de CI/CD de su elección, pero deberá integrarlas usted mismo. Los usuarios de GitHub suelen trabajar con un programa de CI de terceros
- GitLab tiene integrados flujos de trabajo de integración continua/entrega continua (CI/CD) y DevOps.



GitHub



GitLab

En Linux/Mac OS:

- Ya viene instalado por defecto, aunque no última versión
  - Ubuntu: `apt-get install git`
  - Mac OS: `brew install git`
- Última versión disponible en: <https://git-scm.com/downloads>

En Windows:

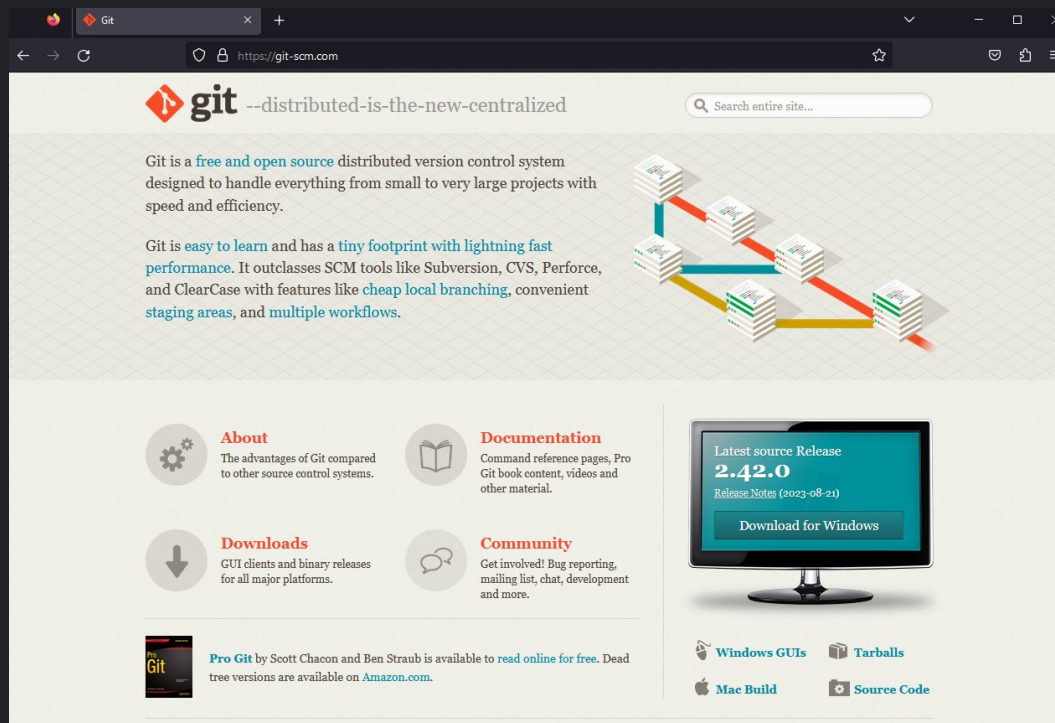
- Se descarga desde la web oficial: <https://git-scm.com/download/win>

Funciona exactamente igual en todos los sistemas operativos.



## Git

- El libro es gratuito y está traducido a todos los idiomas
- En la web también hay documentación y otros recursos



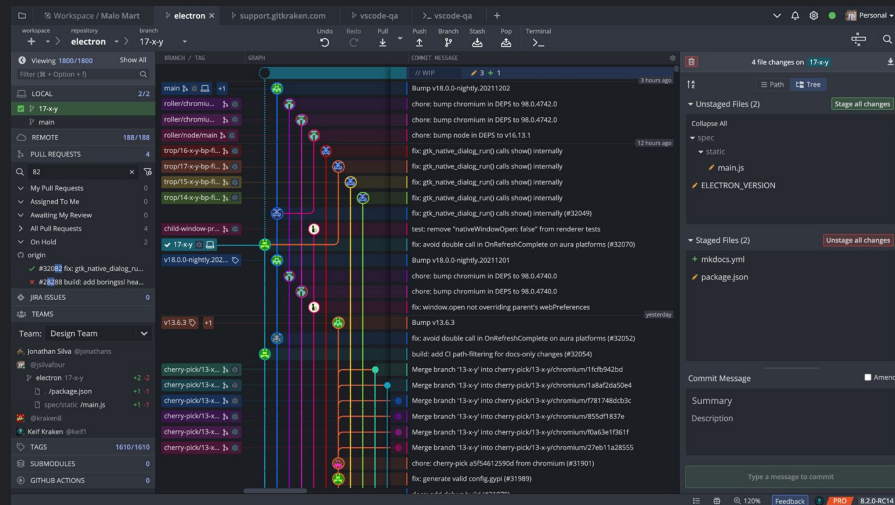
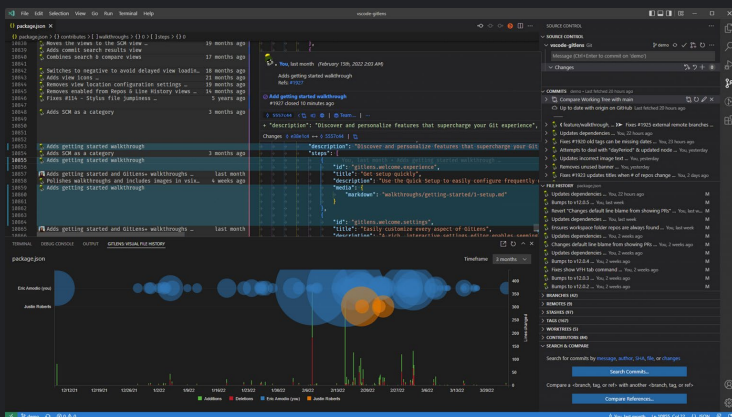
## Clientes GUI

- Fuente: <https://git-scm.com/downloads/guis>
- Más populares:
  - Clientes desde VS Code
  - GitHub desktop
  - GitKraken
  - SourceTree

The screenshot shows the 'GUI Clients' page on the official Git website (https://git-scm.com/download/gui/linux). The page features the Git logo and navigation links for 'About', 'Documentation', 'Downloads', 'GUI Clients', and 'Logos'. A sidebar on the left mentions the 'Pro Git book' by Scott Chacon and Ben Straub. The main content area is titled 'GUI Clients' and explains that Git includes built-in GUI tools like 'git-gui' and 'gitk', but also lists several third-party tools. It provides a search bar and a list of GUI clients categorized by platform: All, Windows, Mac, Linux, Android, and iOS. A note states '25 Linux GUIs are shown below'. Four specific GUI clients are highlighted with their respective logos and details:

- GitKraken**: Platforms: Linux, Mac, Windows; Price: Free / \$59+/user annually; License: Proprietary.
- Magit**: Platforms: Linux, Mac, Windows; Price: Free; License: GNU GPL.
- SmartGit**: Platforms: Linux, Mac, Windows; Price: Free for non-commercial use / \$59/user annually; License: Proprietary.
- MeGit (based on EGit)**: Platforms: Linux, Mac, Windows; Price: Free; License: EPL2.0.

## Herramientas con GUI para trabajar con Git



## Referencias

- <https://courses.cs.washington.edu/courses/cse390a/14wi/bash.html>

### Bash Shell Reference (manual)

category	command	description
basic shell	clear	clear all previous commands' output text from the terminal
	exit (or logout)	quits the shell
	alias, unalias	give a pseudonym to another command (you may need to enclose the command in quotes if it contains spaces or operators)
	history	show a list of all past commands you have typed into this shell
directories	ls	list files in a directory
	pwd	displays the shell's current working directory
	cd	changes the shell's working directory to the given directory; can be a relative or absolute path
	mkdir	creates a new directory with the given name
	rmdir	removes the directory with the given name (the directory must be empty)
file operations	cp	copies a file/directory
	mv	moves (or renames) a file/directory
	rm	deletes a file
	touch	update the last-modified time of a file (or create an empty file)
file examination	cat	output the contents of a file
	more (or less)	output the contents of a file, one page at a time
	head, tail	output the beginning or ending of a file
	wc	output a count of the number of characters, lines, words, etc. in a file
	du	report disk space used by a file/directory
	diff	output differences between two files

# Terminología general

## Repositorio

- El directorio que contiene el proyecto git, así como sus archivos.
- Si es un repositorio, tendrá una carpeta oculta con algunos archivos llamada .git

## Commit

- Como sustantivo, un punto en la historia de Git
- Como verbo: La acción de crear un nuevo snapshot en un proyecto Git, y el hecho de añadirlo a la historia.

## SHA

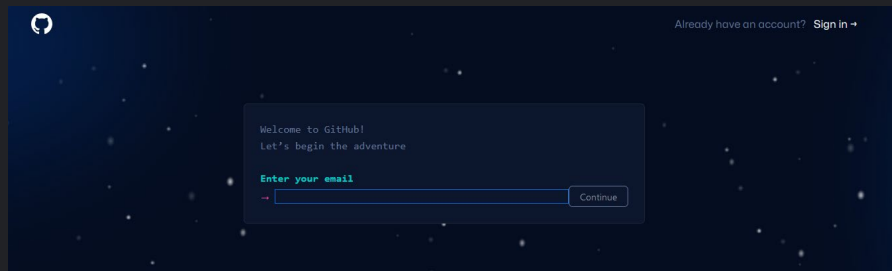
- Un SHA es básicamente un número de identificación para cada confirmación.

## SSH

- Protocolo de seguridad que permite a conectar y autenticarse con servicios y servidores remotos.

Para configurar git es necesario realizar los siguiente pasos:

1. **Crearse una cuenta en [GitHub](#).**
2. Generar claves SSH, siguiendo el [tutorial de github](#).
3. Añadir/vincular las llaves SSH a la cuenta de GitHub
4. Configurar git



## Comandos

```
$ ssh-keygen -t ed25519 -C "tunombre@gmail.com"
```

```
$ cat ~/.ssh/id_ed25519.pub
```

Para configurar git es necesario realizar los siguiente pasos:

1. Crearse una cuenta en [GitHub](#).
- 2. Generar claves SSH, siguiendo el [tutorial de github](#).**
3. Añadir/vincular las llaves SSH a la cuenta de GitHub
4. Configurar git

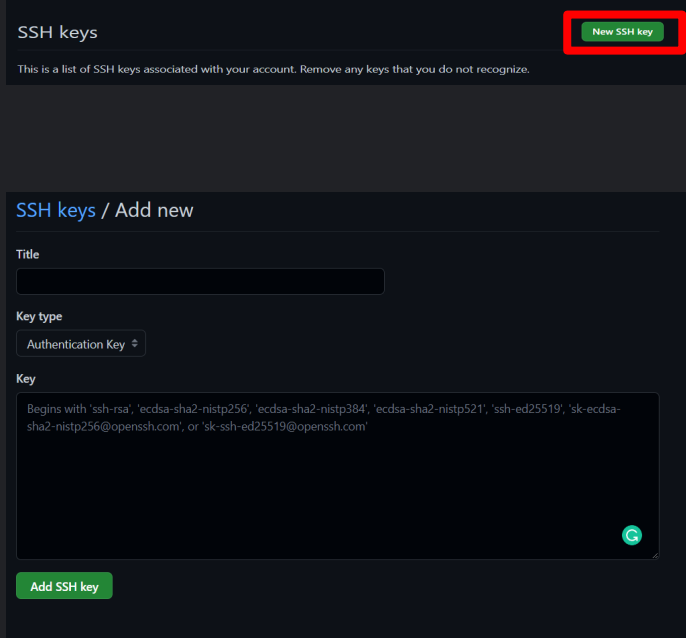
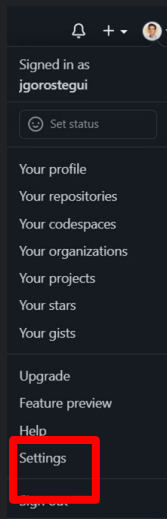
### Comandos

```
$ ssh-keygen -t ed25519 -C  
"tunombre@gmail.com"  
$ cat ~/.ssh/id_ed25519.pub
```



Para configurar git es necesario realizar los siguiente pasos:

1. Crearse una cuenta en [GitHub](#).
2. Generar claves SSH, siguiendo el [tutorial de github](#).
- 3. Añadir/vincular las llaves SSH a la cuenta de GitHub**
4. Configurar git



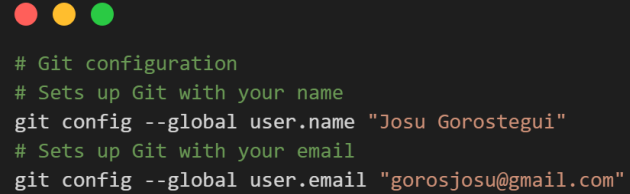


## Configurando git

Para configurar git es necesario realizar los siguiente pasos:

1. Crearse una cuenta en [GitHub](#).
2. Generar claves SSH, siguiendo el [tutorial de github](#).
3. Añadir/vincular las llaves SSH a la cuenta de GitHub

### 4. Configurar git



```
# Git configuration
# Sets up Git with your name
git config --global user.name "Josu Gorostegui"
# Sets up Git with your email
git config --global user.email "gorosjosu@gmail.com"
```

```
$ cat .gitconfig

[user]
    email = gorosjosu@gmail.com
    name = Josu Gorostegui
```

### git init

Inicializar repositorio (utilizando `git init`)

**\$** `git init`

- Inicializa un repositorio Git vacío.
- Crea una nueva carpeta `.git` en el directorio actual.
- Sólo se necesita hacer una vez por repositorio.

### git add

Añadiendo los cambios con `git add`

```
$ git add README.md
```

- Puedes agregar archivos específicos o todos los cambios.
- `git add .` agrega todos los cambios del directorio actual.
- Prepara los cambios para el próximo "commit".

### git status

Ver el estado del repo (utilizando `git status`)

#### \$ git status

- Muestra archivos modificados, agregados o eliminados.
- Indica si los archivos están listos para ser "commit".
- Esencial para el flujo de trabajo y se utiliza frecuentemente.

### git commit

Guardando Cambios con `git commit`

```
$ git commit -m "Añadiendo el archivo README.md"
```

- Crea una instantánea del proyecto.
- El parámetro `-m` permite añadir un mensaje descriptivo.
- Es fundamental para mantener un registro del progreso.

### git commit --amend

Permite modificar el último commit realizado.

- ¿Como se usa?

```
$ git commit --amend -m "nuevo mensaje de commit" # Cambia solo el mensaje
```

```
$ git commit --amend # cambios en el último commit
```

- Corregir un error tipográfico en el mensaje del commit.
- Añadir archivos olvidados al commit anterior.
- Realizar pequeños cambios adicionales antes de compartir el commit.



### ¡CUIDADO!

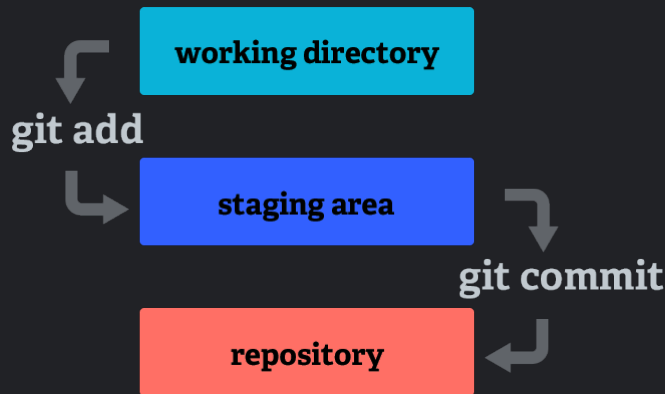
- **¡No modifiques commits que ya hayas enviado a un repositorio remoto!** Esto puede causar problemas de sincronización con otros colaboradores.
- Úsalo con cuidado y solo en commits locales.

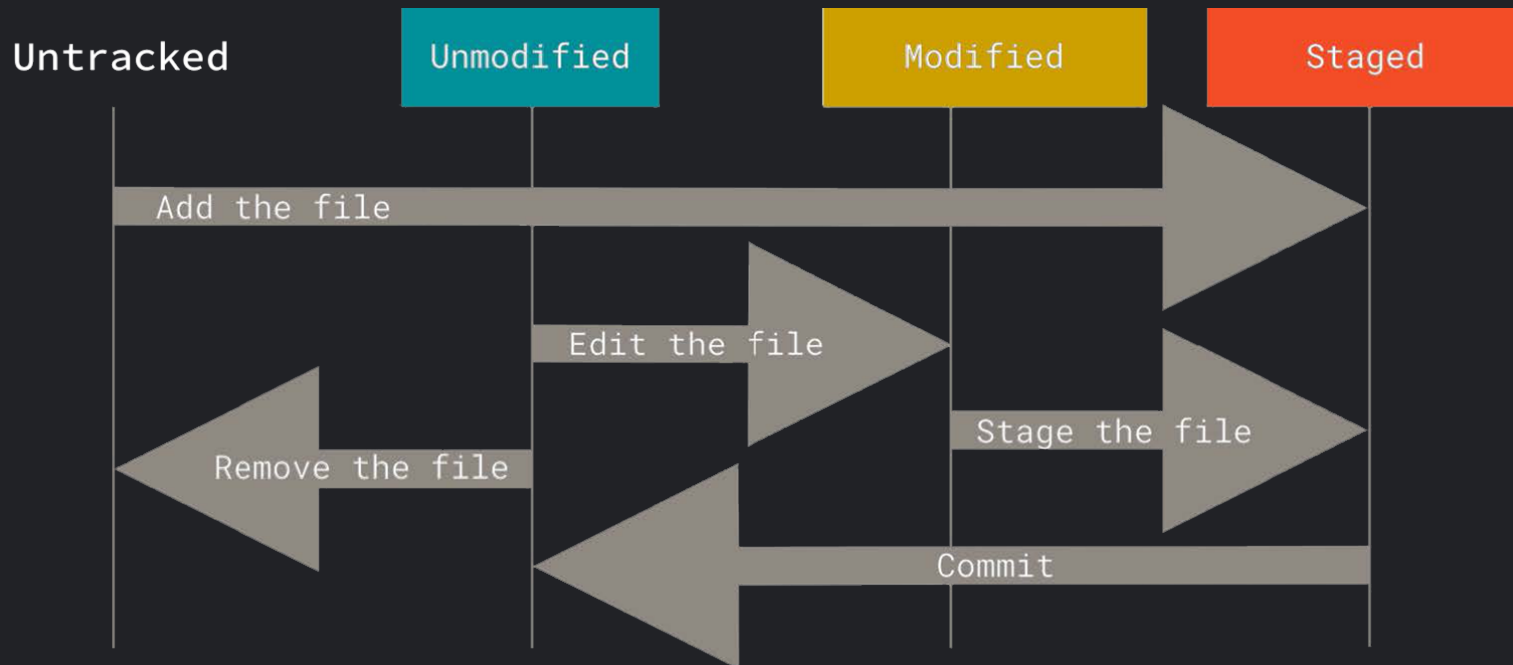
### Añadiendo archivos

Cuando se añaden (**git add**) archivos estos pasan a estar en un área llamada *staging area* o *índice*.

Los cambios no se registran hasta que no se ejecuta **git commit**

El commit representa la fotografía del proyecto en ese momento.







### git log

Visualizando el historia con `git log`

```
$ git log
```

- Muestra los "commits" en orden cronológico inverso.
- Incluye autor, fecha y mensaje de cada "commit".
- Útil para rastrear cambios y la evolución del proyecto.

Para limitar la salida del historial

```
$ git log --since=2.weeks
```

## git log (2)

`git log` ofrece opciones para personalizar la visualización del historial de commits.

Formato:

`--oneline`: Muestra cada commit en una sola línea, ideal para una vista rápida.

`--graph`: Visualiza el historial como un gráfico de ramas y fusiones.

`--pretty=format:"%h - %an, %ar : %s"`: Personaliza la información mostrada para cada commit (hash, autor, fecha, mensaje).

### Ejemplos

`git log --oneline --graph`: Vista compacta y gráfica del historial.

`git log --author="Juan Pérez" --since="2023-01-01"`:

Muestra los commits de Juan Pérez desde el 1 de enero de 2023.

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 Ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of https://github.com/dustin
|\
| * 420eac9 Add method for getting the current branch
* | 30e367c Timeout code and tests
* | 5a09431 Add timeout protection to grit
* | e1193f8 Support for heads with slashes in them
|/
* d6016bc Require time for xmlschema
* 11d191e Merge branch 'defunkt' into local
```

Un **diff** te dice **QUÉ** cambió. Un buen **commit** te dice **POR QUÉ**.

#### ¿Por qué es tan importantes

- **Para tu equipo:** Facilita la revisión de código (Pull Requests). Ahorra horas de trabajo a tus compañeros y demuestra profesionalismo.
- **Para tu "yo" del futuro:** Dentro de 6 meses, no recordarás por qué hiciste un cambio. Tu historial de commits será tu mejor mapa para entender tu propio código.
- **Para cazar errores (Bugs):** Un historial claro permite usar herramientas como git bisect para encontrar el commit exacto que introdujo un error en segundos.

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

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

Un commit profesional sigue una estructura simple y universal.

## El Asunto (Subject) - ¡El 95% de tus commits solo tendrán esto!

### 1. El Asunto (Subject) - ¡El 95% de tus commits solo tendrán esto!

- **Regla de los 50-72 caracteres:** Suficientemente corto para `git log --oneline`, suficientemente largo para ser claro.
- **Modo Imperativo:** `feat: Agrega login con Google.` (Recuerda: "Si se aplica, este commit...")
- **Incluye Referencia a tareas (si aplica):** Es una práctica muy común.
  - `fix: Corrige error de paginación en la lista de usuarios (#123)`
  - `feat(api): Añade endpoint para obtener perfil de usuario (PROJ-456)`

Un commit profesional sigue una estructura simple y universal.

## El Asunto (Subject)

**2. El Cuerpo (Body)** - Solo cuando el "porqué" no es obvio

- **¿Cuándo usarlo?** Cuando el asunto no es suficiente para explicar el contexto, las alternativas descartadas o las implicaciones del cambio.
- **Recuerda:** Siempre separado del asunto por una línea en blanco.

**Nota:** La mayoría de los commits de un proyecto son simples y no necesitan cuerpo. La clave es saber cuándo sí es necesario.

Un buen historial no es una lista de cambios, es **la historia de cómo se construyó un producto.**

### **Beneficios de un historial así:**

- Fácil de navegar: Entiendes el flujo del proyecto de un vistazo.
- Profesional: Demuestra un equipo maduro y disciplinado.
- Eficiente: Reduce el tiempo de depuración y de incorporación de nuevos miembros al equipo.

### **Convención opcional (para mantener el historial limpio)**

- *Conventional Commits*: <https://www.conventionalcommits.org/en/v1.0.0/>

### Crear el primer repositorio

Se va a crear repositorio y se utilizarán los comandos

1. Crear un directorio (utilizando `mkdir`)
2. Inicializar repositorio (utilizando `git init`)
3. Ver el estado del repo (utilizando `git status`)

Se puede ver como existen ficheros ocultos tras la inicialización.



### git diff

Para visualizar los cambios que no añadidos se utiliza: **git diff**

**\$ git diff**

**git diff** muestra las diferencias entre lo que tienes en tu directorio de trabajo y lo que está guardado en tu área de preparación (staging area) o entre dos commits, ramas, etc.





### Primer commit

Dentro del repo (creado en la diapositiva anterior), se va a añadir un archivo y realizaremos el primer commit.

Para esto es necesario recordar:

- Añadir un archivo con:
  - `git add <file>`
- Para añadir todos los archivos presentes:
  - `git add .`
- Realizar el commit con:
  - `git commit -m <message>`



### git tag

Marcando puntos importantes de nuestro historial con `git tag`

`$ git tag v1.0`

- Ideal para marcar versiones.
- Las etiquetas pueden ser ligeras o anotadas.
- Las etiquetas anotadas incluyen información adicional y se recomiendan para lanzamientos.

`$ git tag`

v0.1

v1.3

`$ git tag -l 'v1.8.5*'`

v1.8.5

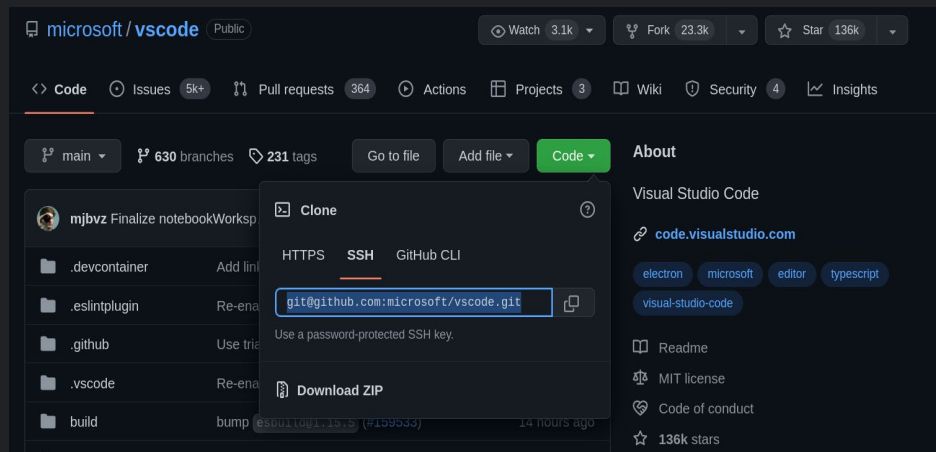
v1.8.5-rc0

v1.8.5-rc1

## Clonar y Fork

En GitHub existen más de 100M de repositorios, es la forma colaborativa de trabajar en el mundo del software.

- **Clonar:** Crea una copia del repositorio en la máquina local. Simplemente es el proceso de descargar un proyecto de GitHub desde un repositorio en línea a su máquina local usando Git.
- **Fork:** Hace una copia de un repositorio de GitHub para ser usado como base para un nuevo proyecto, pudiendo enviar los cambios al repositorio original y/o realizar unos cambios de forma independiente. Crea una copia del repositorio original (repositorio ascendente), pero el repositorio se copia en su cuenta de GitHub, no en su máquina local.



## Branch y Tag

**Branch o rama** es una *línea de desarrollo*.

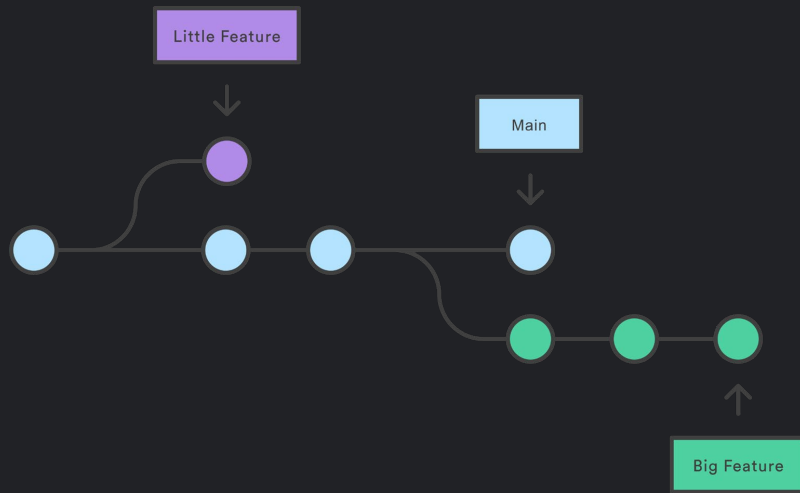
Esta línea nueva permite continuar sin alterar la línea principal.

Un repositorio git tiene las ramas main o master por defecto cuando se crea el repositorio.

En un entorno colaborativo, es habitual que varios desarrolladores compartan y trabajen sobre el mismo código fuente.

Algunos desarrolladores corregirán errores, otros implementarán nuevas features, etc.

**Tag o etiqueta** son referencias que apuntan a commits concretos en el historial de Git.



### git branch

Gestión de Ramas con **git branch**

#### \$ **git branch nueva\_rama**

- **git branch** solo listará todas las ramas.
- Usar **git branch -d nombre\_rama** para eliminar una rama.
- Las ramas son cruciales para el desarrollo paralelo y la organización de características o fixes.

### git checkout

Cambiar de Estado o Rama con **git checkout**.

El comando git checkout permite cambiar entre diferentes "commits", ramas o incluso restaurar archivos.

**\$ git checkout nombre\_rama**

- Usado tradicionalmente para cambiar entre ramas o "commits".
- **git checkout -- nombre\_archivo** puede descartar cambios en ese archivo.
- En versiones más recientes de Git, **git switch** y **git restore** se recomiendan para algunas de estas tareas.

### git switch

- Introducido en **Git 2.23**
- Enfocado en operaciones de ramas y más intuitivo para principiantes
- Tiene salvaguardas incorporadas

### Comparación con `git checkout`

- Más claro y específico
- Sólo para ramas
- Más seguro con cambios sin confirmar

Sintaxis:

```
$ git switch <nombre-rama>
```

### Ramas en git

Para crear rama y mergear los comandos son los siguientes:

- `git branch`
- `git checkout <featureX>`
- `git branch -d <featureX>`
- `git merge branch <featureX>`
- `git tag`
- `git tag <tagname>`

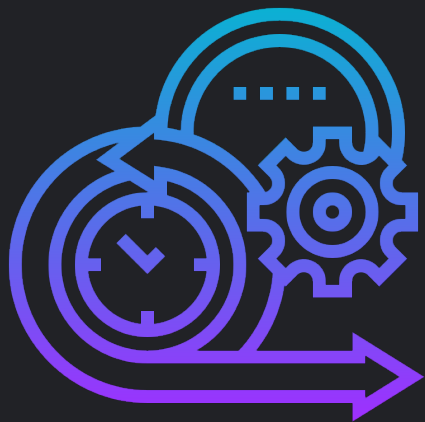




Las tareas a realizar constan de dos partes, para lo cual hay que descargar el .zip desde la [plataforma moddle](#).

1. Operaciones con archivos. Se tiene un .zip comprimido de partida, se descomprime y se realizan ciertas operaciones.
2. La carpeta del ejercicio anterior es el punto de partida del ejercicio 2 y siguientes, que trata realizar ciertas operaciones con Git.

Una vez realizadas las tareas, será necesario subir el resultado comprimido en .zip a la [plataforma moddle](#).



Buenas prácticas y agilidad

## Control de versiones y Git

---

Josu Gorostegui

---