

Git, GitHub y el control de versiones: del caos al orden digital

El control de versiones es la "máquina del tiempo" que permite guardar, etiquetar y recuperar estados completos de un proyecto

> 94% de desarrolladores usan Git (Stack Overflow 2023)



¿Qué es un "sistema de control de versiones"?

Un VCS (Version Control System) registra quién, qué, cuándo y por qué cambió cada archivo. Permite rastrear modificaciones, comparar versiones y revertir cambios cuando sea necesario.

📁 Modelo Local

Base de datos de versiones en disco local. Ejemplos: RCS. Limitado a un solo usuario y sin capacidad de colaboración.

🏢 Modelo Centralizado (CVCS)

Servidor central que contiene todas las versiones. Clientes se conectan para obtener las últimas versiones. Ejemplos: SVN, CVS. Problema: punto único de fallo.

🔗 Modelo Distribuido (DVCS)

Cada cliente tiene una copia completa del repositorio. Permite trabajo offline y colaboración sin servidor central. Ejemplos: Git, Mercurial. Git es el más popular (>94% de desarrolladores).

"Álbum de fotos compartido donde todos tienen una copia idéntica del historial completo"

Local

1 usuario
Base de datos
local

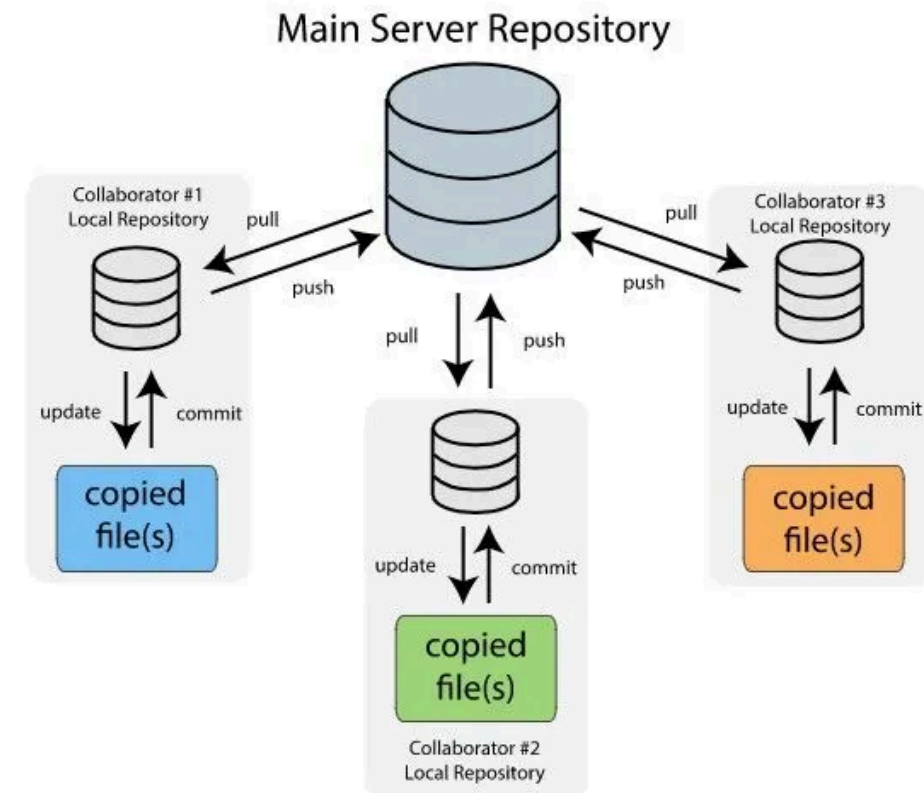
Centralizado

1 servidor central
Múltiples
clientes

Distribuido

Todos los nodos
Contienen el
historial
completo

Distributed Version Control



A Distributed Version Control System. Each collaborator has a local copy of the repository, so no Internet connection is required.

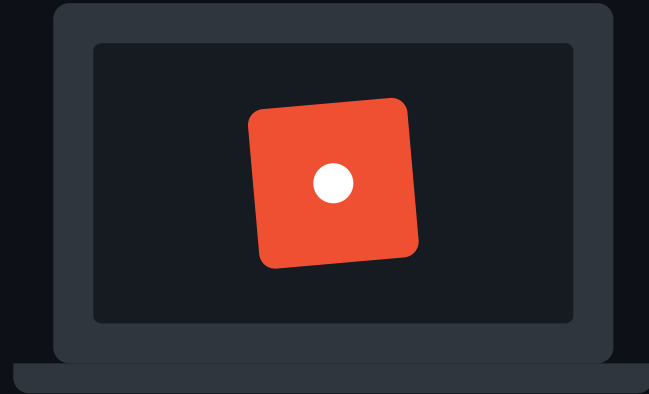
Git en 20 segundos

🕒 Origen

- ▶ Creado por Linus Torvalds en 2005 para el kernel de Linux
- ▶ Respuesta a la retirada del soporte gratuito de BitKeeper

★ Características principales

- ▶ **Gratuito y de código abierto** bajo licencia GPL v2
- ▶ **Multi-plataforma:** Windows, macOS, Linux
- ▶ **Diseñado para velocidad** : operaciones locales casi instantáneas
- ▶ **Integridad criptográfica** : SHA-1 (transición a SHA-256)



💡 Dato curioso

Un repositorio de 10 años con 30,000 commits suele ocupar menos espacio que una carpeta de fotos de un fin de semana gracias a su eficiente sistema de compresión delta.

2005

Creación de
Git por Linus
Torvalds

2008

Lanzamiento
de GitHub

2014

Microsoft
adapta Git

2020

Transición a
SHA-256

2025

>94% de
desarrolladores
usan Git

Concepto estrella: el commit

Un **commit** es una instantánea comprimida de **todo el proyecto**, no solo "diferencias". Cada commit contiene el estado completo del repositorio en ese momento.

Instantánea completa

Git guarda una copia completa de cada archivo modificado. Si un archivo no cambia, Git apunta al blob previo (eficiencia).

Identificador único

Cada commit tiene un hash SHA-1 (40 caracteres hexadecimales) que lo identifica de forma única. Transición a SHA-256 en progreso.

Integridad garantizada

El hash se calcula a partir del contenido, metadatos y el hash del commit anterior. Cualquier modificación altera el hash.



Modificación

Cambios en archivos



Staging

Preparar cambios



Commit

Guardar instantánea

"Fotografía en alta resolución que se guarda solo si algo cambió respecto a la anterior foto"

Flujo de trabajo local (3 estados)

Git utiliza tres áreas principales para gestionar tus cambios: Directorio de trabajo → Área de preparación → Repositorio local.

✍ Working Directory

Archivos que ves y modificas directamente. Git no rastrea estos cambios hasta que los preparas.

🗂 Staging Area (index)

Área intermedia donde preparas los cambios antes de confirmarlos. Permite seleccionar qué cambios incluir en el siguiente commit.

🗄 Local Repository

Base de datos Git que almacena todos los commits. Contiene el historial completo del proyecto.

📄 Comandos clave

git add

Mueve archivos del directorio de trabajo al área de preparación.

```
git add archivo.txt
```

```
git add . # Todos los archivos modificados
```

```
git add -A # Todos los cambios (incluidos eliminados)
```

git commit

Guarda los cambios preparados en el repositorio local.

```
git commit -m "Mensaje descriptivo"
```

```
git commit -am "Mensaje" # Prepara y confirma archivos ya rastreados
```

```
git commit --amend # Modifica el último commit
```



Ejemplo práctico

```
# Modificar archivo
echo "Nuevo contenido" > README.md

# Ver estado
git status

# Preparar cambios
git add README.md

# Confirmar cambios
git commit -m "Actualizar README con nueva información"
```

Instalación en Windows (capturas)

📥 Descarga

Descarga el instalador desde el sitio oficial de Git:

1

```
https://git-scm.com/download/win
```

✔ Git-2.45.x-64-bit.exe

✔ Versión portable disponible

⚙️ Configuración del asistente

Ejecuta el instalador y selecciona estas opciones recomendadas:

2

✔ Git from the command line and also from 3rd-party software

✔ Checkout Windows-style, commit Unix-style line endings

✔ Use OpenSSH

✔ Use the Windows default console window

✅ Verificación

Abre una terminal (CMD, PowerShell o Git Bash) y ejecuta:

3

```
git --version
```

Salida esperada:

```
git version 2.45.0.windows.1
```

Paso 1: Descarga

📥 Vista del sitio de descarga

Paso 2: Configuración

⚙️ Asistente de instalación

Paso 3: Verificación

🖥️ Terminal con comando git --version

Configuración mínima (una sola vez)

Antes de empezar a usar Git, necesitas configurar tu identidad. Esta configuración se guarda globalmente y se aplicará a todos tus repositorios.

👤 Configuración básica



```
git config --global user.name "Juan Pérez"
```

Establece tu nombre que aparecerá en los commits



```
git config --global user.email  
juan@example.com
```

Establece tu email para identificar tus commits



```
git config --global init.defaultBranch  
main
```

Cambia la rama por defecto de "master" a "main"

⚙️ Configuración adicional recomendada



```
git config --global core.editor "code --  
wait"
```

Usa VS Code como editor para mensajes de commit



```
git config --global pull.rebase true
```

Evita commits de merge automáticos al hacer pull



```
git config --global credential.helper  
manager
```

Guarda tus credenciales para no tener que ingresarlas cada vez



Importante

Sin nombre y email configurados, Git rechazará commits con el error "user.name not set". Esta información es obligatoria para cada commit.

Git Bash

```
user@DESKTOP:~$ git config --global user.name  
"Juan Pérez"  
user@DESKTOP:~$ git config --global user.email  
juan@example.com  
user@DESKTOP:~$ git config --global  
init.defaultBranch main  
user@DESKTOP:~$ git config --list  
user.name=Juan Pérez  
user.email=juan@example.com  
init.defaultbranch=main  
user@DESKTOP:~$ _
```

Comandos básicos (tarjeta de bolsillo)

Comandos esenciales para el día a día con Git. Domina estos y manejarás el 90% de las operaciones comunes.

📁 Repositorio

`git init`

Crea un nuevo repositorio local

`git clone URL`

Clona un repositorio remoto

`git remote -v`

Muestra repositorios remotos

`git remote add origin URL`

Añade un repositorio remoto

👁 Estado e Historia

`git status`

Muestra estado de archivos

`git status -s`

Formato corto de status

`git log --oneline`

Historial en una línea

`git log --graph --oneline`

Historial con gráfico de ramas

`git diff`

Muestra cambios no preparados

➕ Preparar y Guardar

`git add archivo`

Prepara archivo específico

`git add .`

Prepara todos los cambios

`git add -A`

Prepara todos los cambios (incluye eliminados)

`git commit -m "mensaje"`

Guarda cambios con mensaje

`git commit --amend`

Modifica último commit

🔀 Ramas

`git branch`

Lista todas las ramas

`git branch nombre`

Crea nueva rama

`git checkout rama`

Cambia a otra rama

`git checkout -b nueva-rama`

Crea y cambia a nueva rama

`git merge rama`

Fusiona rama en la actual

☁ Sincronización

`git fetch`

Descarga cambios sin fusionar

`git pull`

Descarga y fusiona cambios

`git push`

Sube cambios al repositorio remoto

`git push -u origin rama`

Sube rama nueva y configura seguimiento

↶ Deshacer cambios

`git checkout -- archivo`

Deshace cambios locales

`git reset HEAD archivo`

Quita archivo del área de preparación

`git reset --hard HEAD`

Deshace todos los cambios locales

`git revert commit`

Crea commit que deshace cambios

`git stash`

Guarda cambios temporalmente

¿Qué es GitHub?

Plataforma en la nube creada en 2008 y adquirida por Microsoft en 2018. Es el hogar de más de 100 millones de desarrolladores y empresas que colaboran en código.

★ Características principales

- | | |
|------------------|------------------------|
| <> Hospedaje Git | 🐛 Issues y seguimiento |
| ⬆ Pull Requests | ⚙ CI/CD con Actions |
| 📖 Wikis | 🌐 Páginas estáticas |
| 💻 Codespaces | 🤖 Copilot (IA) |

II. Estadísticas clave (2023)

+100M

Repositorios públicos

+4M

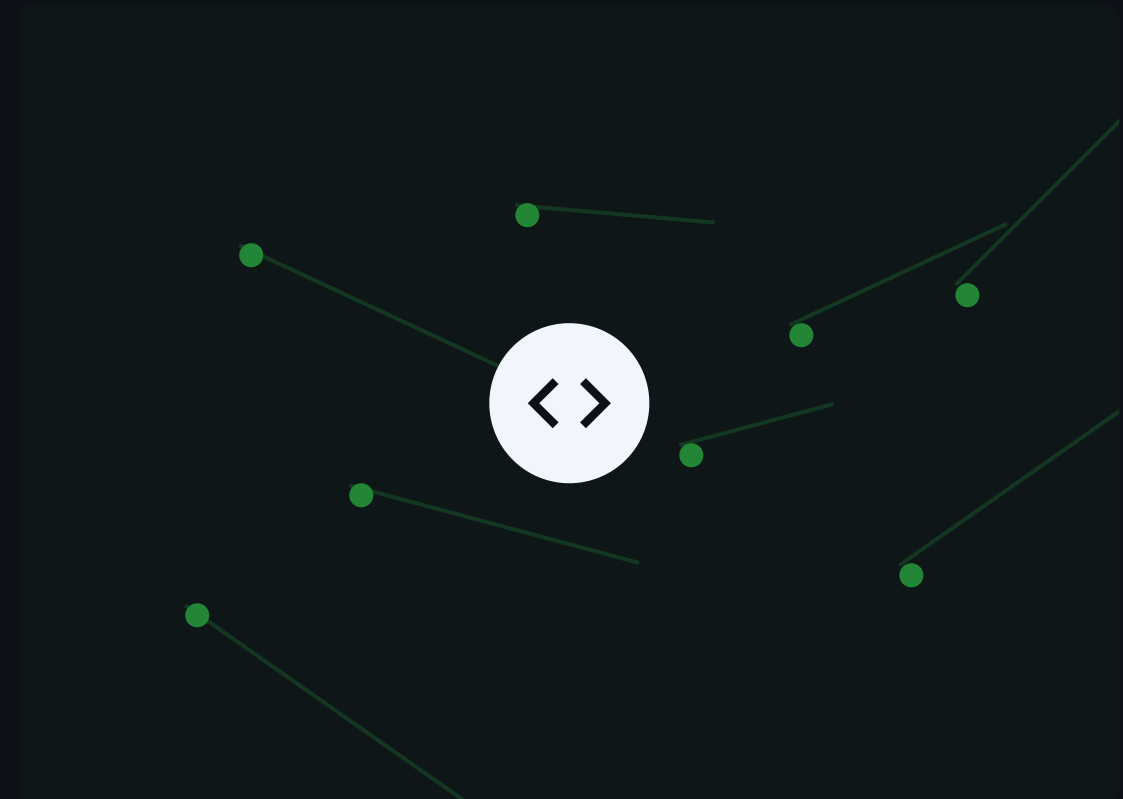
Organizaciones

+90M

Desarrolladores activos

200+

Países con usuarios



Flujo de trabajo colaborativo



Fork



Editar



Pull Request



Revisión



Merge

Despliegue



Git vs GitHub (tabla comparativa)

Git y GitHub son herramientas complementarias pero diferentes. Entender sus diferencias es clave para usarlas eficientemente.

Característica	Git	GitHub
Función principal	Sistema de control de versiones	
	Plataforma de hospedaje	
Local	✓	
	✗	
Necesita servidor	✗	
	✓	
Issues/PR	✗	
	✓	
Interfaz web	✗	
	✓	
CI/CD integrado	✗	
	✓	
Propietario	Comunidad	
	Microsoft	
Costo	Gratis	
	Gratis con opciones pagas	

💡 Cuándo usar cada uno

Git es imprescindible para cualquier proyecto de software, funciona localmente y no requiere conexión. GitHub es opcional pero muy recomendable para colaboración, backup y visibilidad. Puedes usar Git sin GitHub, pero no GitHub sin Git.

Alternativas serias a GitHub

Existen varias plataformas de hospedaje de código con características diferentes que pueden adaptarse mejor a tus necesidades específicas.



GitLab

- ✓ Plan gratuito ilimitado de repos privados
- ✓ CI/CD integrado (GitLab CI)
- ✓ Autohospedaje completo
- ✓ Issues, Merge Requests, Wiki

★ Gratis

☁ Autohospedaje



Bitbucket

- ✓ Integración con ecosistema Atlassian
- ✓ Pipelines CI/CD integrados
- ✓ Conexión nativa con Jira y Trello
- ✓ Revisión de código avanzada

★ Gratis (límites)

🏢 Empresas



Gitea / Forgejo

- ✓ Autoservicio ligero (< 100 MB)
- ✓ Código abierto (Go)
- ✓ Fácil instalación y configuración
- ✓ Interfaz similar a GitHub

★ Gratis

☁ Autohospedaje



SourceHut

- ✓ Sin JavaScript, solo HTML
- ✓ Workflow por email
- ✓ Minimalista y rápido
- ✓ Builds integrados (builds.sr.ht)

★ Gratis

📧 Minimalista



Comparación rápida

Plataforma	Repos privados gratuitos	CI/CD integrado	Autohospedaje
GitLab	Sí (ilimitados)	Sí (avanzado)	Sí
Bitbucket	Sí (5 usuarios)	Sí (Pipelines)	No
Gitea/Forgejo	Sí (ilimitados)	Parcial	Sí
SourceHut	Sí (ilimitados)	Sí (builds.sr.ht)	Parcial

Crear cuenta en GitHub (paso a paso visual)

Visitar el sitio de registro

1 Ir a github.com/signup para iniciar el proceso de registro

Elegir nombre de usuario

2 El nombre de usuario debe ser único y no se puede cambiar después. Aparecerá en tu URL: `github.com/tu-usuario`

Consejo

Usa un nombre profesional, corto y fácil de recordar. Evita números o caracteres especiales si es posible.

Email y contraseña

3 Proporciona un email válido y una contraseña segura (mínimo 8 caracteres, mayúsculas, minúsculas y números)

Verificación y términos

4 Completa la verificación humana (CAPTCHA) y acepta los términos de servicio. Las preferencias de marketing son opcionales.

Confirmar email

5 Revisa tu bandeja de entrada y haz clic en el enlace de verificación. Serás redirigido a tu dashboard personal.

Paso 1: Página de registro



Paso 2: Nombre de usuario



Paso 3: Email y contraseña



Paso 4: Verificación



Paso 5: Dashboard



Consejo de seguridad

Activa la autenticación de dos factores (2FA) desde Settings → Passwords and authentication. Usa una aplicación TOTP como Google Authenticator o Authy para mayor seguridad.

HTTPS vs SSH: cómo autenticarse

GitHub ofrece dos métodos principales para autenticarte al interactuar con repositorios remotos: HTTPS y SSH. Cada uno tiene ventajas y desventajas específicas.

Método	Ventaja	Desventaja
 HTTPS	Fácil (solo usuario + token)	Pide credenciales cada vez*
 SSH	No pide clave después	Requiere generar par de claves

Configuración HTTPS

- 1 Genera un Personal Access Token en GitHub Settings
- 2 Usa el token como contraseña
- 3 Activa el cache para no repetir credenciales:

```
git config --global credential.helper manager
```

Configuración SSH

- 1 Genera par de claves:

```
ssh-keygen -t ed25519 -C "tu_email@example.com"
```
- 2 Añade clave pública a GitHub Settings
- 3 Usa URL SSH para clonar repositorios

Recomendación

SSH es más seguro y conveniente para uso frecuente. HTTPS es más simple para principiantes o cuando trabajas detrás de firewalls restrictivos.



¿Qué significa "clonar"?

Clonar es crear una copia completa de un repositorio remoto en tu máquina local, incluyendo todo el historial de cambios.

```
git clone URL
```

📁 ¿Qué contiene un repositorio clonado?

- Carpeta **.git**: contiene toda la historia (objetos, referencias, configuración)
- Archivos del proyecto: copia de trabajo del estado actual
- Referencia al repositorio remoto (origin) para sincronización

⚙️ Opciones útiles

```
git clone --depth 1  
URL
```

Clon superficial (solo último commit)

```
git clone --branch  
rama URL
```

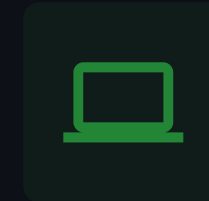
Clonar rama específica

```
git clone --recursive  
URL
```

Incluir submódulos

```
git clone URL nuevo-  
nombre
```

Cambiar nombre de carpeta



Estructura de un repositorio clonado

```
mi-proyecto/  
  .git/ ← Base de datos Git  
  objects/ ← Commits, trees, blobs  
  refs/ ← Referencias a ramas  
  HEAD ← Rama actual  
  config ← Configuración  
  README.md  
  src/  
  .gitignore
```

"Fotocopiar todo el álbum, incluidas las notas al pie de cada página"

Clonar el repo de clase (ejemplo práctico)

URL objetivo:

```
https://github.com/ricardoinstructor/CP1475.git
```

Secuencia exacta

Carpeta contenedora

Navega a una ubicación adecuada para guardar el repositorio

1

```
cd %USERPROFILE%\Desktop
```

También puedes usar: cd C:\Users\TuNombre\Documents

Clonar

Crea una copia local del repositorio remoto

2

```
git clone https://github.com/ricardoinstructor/CP1475.git
```

Git descargará todo el historial del proyecto

Explorar

Verifica el contenido y estructura del repositorio

3

```
cd CP1475
```

```
dir
```

```
type README.md
```

```
git log --oneline
```

Resultado

Se creará la carpeta CP1475/ con todo el proyecto y la historia completa de commits, ramas y etiquetas.



Windows PowerShell

```
PS C:\Users\Usuario> cd Desktop
PS C:\Users\Usuario\Desktop> git clone
https://github.com/ricardoinstructor/CP1475.git
Cloning into 'CP1475'...
remote: Enumerating objects: 25, done.
remote: Counting objects: 100% (25/25), done.
remote: Compressing objects: 100% (15/15),
done.
remote: Total 25 (delta 5), reused 20 (delta
5), pack-reused 0
Receiving objects: 100% (25/25), 5.25 KiB |
5.25 MiB/s, done.
Resolving deltas: 100% (5/5), done.
PS C:\Users\Usuario\Desktop> cd CP1475
PS C:\Users\Usuario\Desktop\CP1475> dir
Directory: C:\Users\Usuario\Desktop\CP1475
Mode LastWriteTime Length Name
----
d----- 10/20/2025 2:30 PM .git
d----- 10/20/2025 2:30 PM docs
d----- 10/20/2025 2:30 PM src
-a---- 10/20/2025 2:30 PM 512 README.md
-a---- 10/20/2025 2:30 PM 128 .gitignore
```

Estructura del repositorio clonado

```
CP1475/
  .git/ ← Historial completo
  README.md ← Información del proyecto
  src/ ← Código fuente
  docs/ ← Documentación
  .gitignore ← Archivos a ignorar
```

Explorar lo descargado: árbol de archivos

Un repositorio Git clonado contiene una estructura específica de carpetas y archivos que organizan el proyecto y su historial.

📁 Estructura típica

CP1475/

├─ README.md

├─ .git/

├─ objects/

├─ refs/

├─ HEAD

├─ src/

├─ docs/

├─ tests/

├─ .gitignore

├─ LICENSE

└─ CONTRIBUTING.md

← Portada del proyecto

← Base de datos Git (no tocar)

← Commits, trees, blobs

← Referencias a ramas

← Rama actual

← Código fuente

← Apuntes y documentación

← Pruebas unitarias

← Patrones a ignorar

← Licencia del proyecto

← Guía para contribuidores

CP1475

README.md

Información del proyecto

src/

Código fuente

docs/

Documentación

📄 Comandos útiles

tree /f

Muestra estructura de carpetas y archivos en Windows

tree

Muestra estructura en Git Bash o Linux

find . -type f

Lista todos los archivos recursivamente

git ls-files

Muestra archivos rastreados por Git

Archivos importantes

README.md

.gitignore

LICENSE

CONTRIBUTING.md

.git/

tests/

💡 Consejo

La carpeta .git contiene todo el historial del proyecto. Nunca la modifiques manualmente a menos que sepas exactamente lo que estás haciendo.

Tu primera modificación local

Ahora realizarás tu primera modificación en el repositorio clonado. Este ejercicio te ayudará a entender el flujo básico de trabajo en Git.

Ejercicio guiado

1

Abrir README.md

Usa VS Code, Bloc de notas o tu editor preferido

`code README.md`

Atajo: Ctrl+O para abrir archivo

2

Añadir tu información

Al final del archivo, añade:

`Estudiado por: [Tu nombre] - 2025-10-20`

Usa formato ISO (YYYY-MM-DD) para fechas

3

Guardar archivo

Atajo: Ctrl+S en la mayoría de editores

Comandos Git

1

Ver estado

`git status`

Verás README.md en rojo (modificado)

2

Preparar cambios

`git add README.md`

Mueve archivo al área de staging

3

Guardar cambios

`git commit -m "Añadido mi nombre al README"`

Crea un commit con mensaje descriptivo

💡

Consejos

Usa mensajes de commit claros y concisos. El formato "tipo: descripción" es recomendado. Ejemplos: "feat: añadir nueva funcionalidad", "fix: corregir error en login".

📄

README.md

1

CP1475 - Control de Versiones

2

3

Este repositorio contiene los materiales del curso de Control de Versiones.

4

5

Contenido

6

- Introducción a Git

7

- Comandos básicos

8

- Flujo de trabajo

9

10

Instalación

11

Para instalar Git, visita [git-scm.com] (https://git-scm.com)

12

13

Estudiado por: [Tu nombre] - 2025-10-20

● ● ●

Terminal

\$

git status

On branch main

Your branch is up to date with 'origin/main'.

Changes not staged for commit:

(use "git add ..." to update what will be committed)

(use "git restore ..." to discard changes in working directory)

modified: README.md

\$

git add README.md

\$

git status

On branch main

Your branch is up to date with 'origin/main'.

Changes to be committed:

(use "git restore --staged ..." to unstage)

modified: README.md

\$

git commit -m "Añadido mi nombre al README"

[main a1b2c3d] Añadido mi nombre al README

1 file changed, 1 insertion(+)

Subir (push) tu cambio a GitHub

Para subir cambios a GitHub necesitas **permiso de escritura** (write access) o haber hecho un **fork** del repositorio.

<> Comando principal

```
git push origin main
```

git push

Comando para subir cambios al repositorio remoto

origin

Nombre predeterminado del repositorio remoto

main

Nombre de la rama a subir

📄 Salida esperada

```
Enumerating objects: 5, done.
Writing objects: 100% (3/3), 283 bytes | 283.00 KiB/s, done.
To
https://github.com/ricardoinstructor/CP1475.git
1a2b3c4..5d6e7f8 main → main
```

git push -u origin main

Configura seguimiento upstream para futuros pushes

git push --all

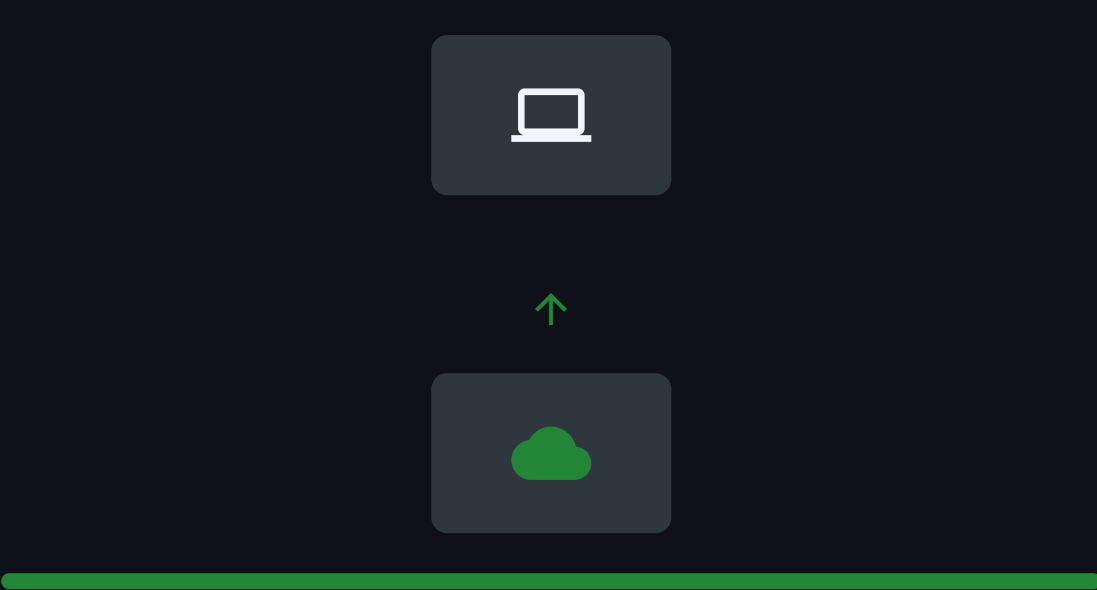
Sube todas las ramas locales al remoto

git push --tags

Sube todas las etiquetas locales al remoto

git remote -v

Muestra todos los repositorios remotos configurados



Flujo de trabajo de push

Commit local
Cambios guardados en repositorio local

Push
Subir cambios al repositorio remoto

Disponibile
Cambios visibles para otros colaboradores

Flujo colaborativo estándar: fork → pull request

El flujo **fork → pull request** permite contribuir a proyectos sin tener acceso directo al repositorio original.

🔗 Pasos del flujo colaborativo

- 1

Fork del repositorio

Presiona "Fork" en GitHub para crear una copia bajo tu usuario
- 2

Clonar tu fork

```
git clone https://github.com/TU_USUARIO/CP1475.git
```

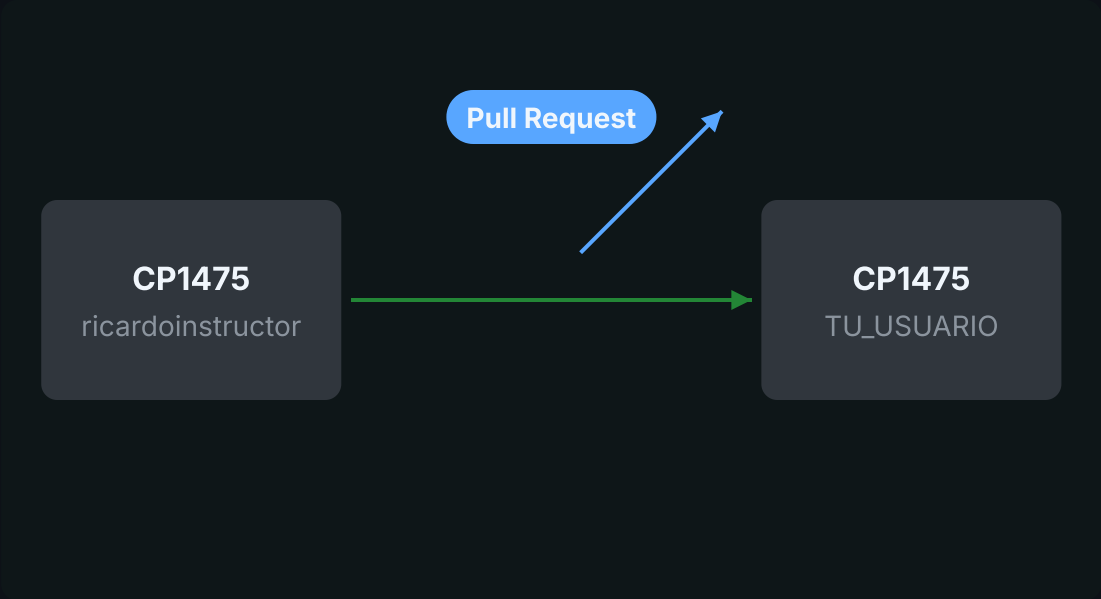
Trabajas localmente en tu copia del proyecto
- 3

Trabajar y subir cambios

Realiza modificaciones, haz commits y push a tu fork
- 4

Crear Pull Request

En GitHub: "Compare & pull request" para proponer cambios



```
git remote add upstream URL
```

Agrega repositorio original como upstream

```
git fetch upstream
```

Obtiene cambios del repositorio original

```
git merge upstream/main
```

Fusiona cambios del original a tu fork

```
git push origin main
```

Sube cambios actualizados a tu fork

★ **Beneficio principal**

El propietario original revisa y mezcla tus cambios sin dar acceso directo al repositorio principal, manteniendo el control sobre qué se integra.



Buenas prácticas en mensajes de commit

” Formato Conventional Commits

```
tipo(alcance): descripción corta

Cuerpo opcional (explica el porqué, no el qué)
```

feat	Nueva funcionalidad	fix	Corrección de error
docs	Cambios en documentación	style	Cambios de formato (no lógica)
refactor	Mejora de código sin cambiar funcionalidad	test	Añadir o modificar pruebas
chore	Tareas de mantenimiento		

<> Ejemplos prácticos

- + feat(readme): añadir instalación en Windows
- 🔧 fix(auth): corregir error de validación de token
- 📄 docs(api): actualizar documentación de endpoints
- 🔄 refactor(components): optimizar renderizado de lista

★ Beneficios

Generación automática de changelogs, semver automático, integración con herramientas de CI/CD, historial más legible y fácil de filtrar.

Plantilla de mensaje de commit

tipo:

feat, fix, docs, style, refactor, test, chore

alcance:

módulo o componente afectado

descripción:

resumen del cambio en tiempo presente

cuerpo:

explicación detallada del porqué del cambio

Flujo de trabajo con commits convencionales



Escribir

Mensaje con formato estándar



Validar

Herramientas verifican formato



Automatizar

Changelogs y versiones

Resumen visual: mapa mental de 5 nodos

"Domina estos 5 verbos y dominarás el 90% de Git"



Próximos pasos: ramas, merges y conflictos

Una vez dominados los conceptos básicos, es hora de explorar funcionalidades más avanzadas que te permitirán trabajar de forma más eficiente en equipo.

🔼 git branch

```
git branch nombre-rama
```

```
git branch -d nombre-rama
```

```
git branch -a
```

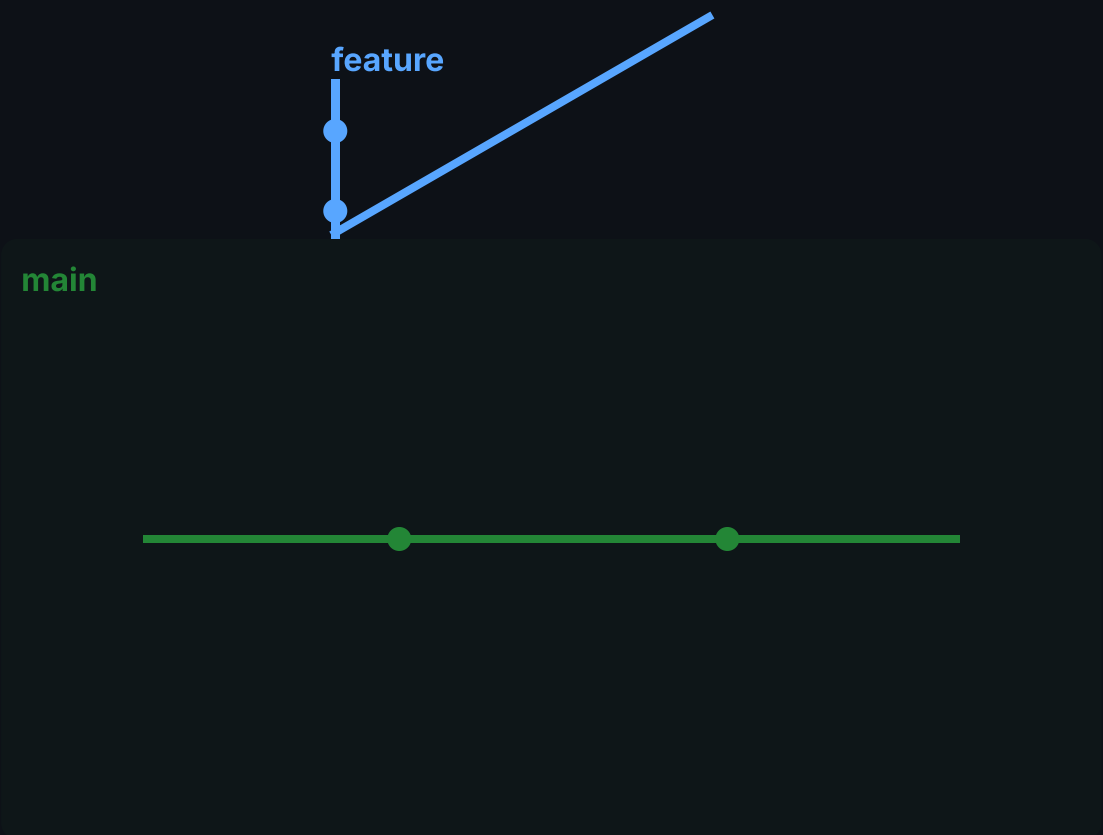
Crea, lista o elimina ramas para trabajar en paralelo sin afectar la rama principal.

⬆️ git merge

```
git merge rama
```

```
git merge --no-ff rama
```

Combina cambios de una rama en otra. Tipos: fast-forward, recursive, octopus.



↔️ git rebase

```
git rebase rama
```





```
git rebase -i HEAD~3
```

Reorganiza el historial de commits. Útil para mantener un historial limpio antes de fusionar.

📄 Pull Requests

Propón cambios para revisión antes de integrarlos. Facilita la colaboración y el control de calidad.

Comandos adicionales útiles

-  **git stash**
Guarda cambios temporalmente
-  **git cherry-pick**
Aplica commits específicos
-  **git revert**
Crea commit que deshace cambios
-  **git reset**
Mueve HEAD a un estado anterior

II. Estadísticas de uso

El **57%** de los desarrolladores usan **feature-branch workflow** (GitHub Octoverse 2023). Otros flujos populares: Gitflow (23%) y GitHub Flow (20%).

Referencias y lecturas recomendadas



Pro Git Book

<https://git-scm.com/book>

Guía completa y gratuita de Git. Capítulos esenciales para principiantes: 1-3 (básicos), 3-5 (ramas), 7-9 (herramientas).



GitHub Guides

<https://docs.github.com/en/get-started>

Tutoriales oficiales de GitHub. Guías más útiles: "Understanding the GitHub flow", "Forking projects" y "Managing branches".



Recursos adicionales



Oh Shit, Git!



GitHub Cheat Sheet



Git Immersion



Learn Git Branching



Git & GitHub Videos



Git Game



GitLab Docs

<https://docs.gitlab.com/ee/user/git.html>

Documentación de Git desde perspectiva de GitLab. Complementa la documentación oficial con ejemplos prácticos.



Atlassian Git Tutorial

<https://www.atlassian.com/git>

Tutoriales visuales sobre conceptos clave. Secciones recomendadas: "Comparing workflows" y "Merging".



Licencia de las capturas

Las capturas de pantalla utilizadas en esta presentación han sido generadas sobre software libre y/o se incluyen bajo fair-use educativo.