Laboratorio GIT

Microcredencial USECHIP

Germán Cano Quiveu

Índice

1 Introducción

La duración estimada de esta sesión es de **6 horas**. El objetivo es que el alumno comprenda no solo los comandos, sino la filosofía de diseño de Git.

Al finalizar este laboratorio, serás capaz de:
- Gestionar el ciclo de vida de los archivos en un repositorio local.
- Comprender el modelo de datos de Git (Snapshots vs Deltas).
- Utilizar GitHub como plataforma de colaboración remota.
- Resolver conflictos básicos y navegar por el historial.

> Esta guía utiliza como referencia técnica **Pro Git v2.1** y la metodología visual de **Learning Git** (O'Reilly).

1.1 Sistemas de Control de Versiones

Los sistemas de control de versiones o *Version Control Systems (VCS)* son herramientas que permiten registrar y gestionar los cambios realizados sobre un conjunto de archivos a lo largo del tiempo. Estos sistemas facilitan la colaboración entre desarrolladores, el seguimiento de modificaciones, la recuperación de versiones anteriores y la auditoría del historial de un proyecto.

Un VCS permite responder a preguntas como:

- ¿Qué cambios se realizaron y cuándo?
- ¿Quién realizó un cambio específico?
- ¿Cómo se puede volver a un estado anterior del proyecto?

Existen diferentes clasificaciones según cómo se estructure el VCS y cómo se gestione el almacenamiento de versiones.

1.1.1 VCS Centralizados

En los sistemas de control de versiones centralizados existe un único repositorio central que almacena todo el historial del proyecto. Los desarrolladores obtienen una copia de trabajo desde este repositorio, realizan modificaciones y luego envían los cambios nuevamente al servidor central. Ejemplos de VCS centralizados incluyen CVS, Subversion (SVN) y Perforce. Entre sus principales desventajas se encuentran:

- La dependencia de un único servidor central.
- La falta de redundancia: si el servidor deja de estar disponible, los desarrolladores no pueden acceder al historial ni enviar cambios.
- Limitaciones para el trabajo sin conexión.

1.1.2 VCS Distribuidos

En los sistemas de control de versiones distribuidos, cada desarrollador clona el repositorio completo, incluyendo todo el historial de cambios. Esto implica que cada copia local funciona como un repositorio completo. Ejemplos de VCS distribuidos son Git, Mercurial y Bazaar. Las principales ventajas de este enfoque son:

- Redundancia total del historial del proyecto.
- Posibilidad de trabajar sin conexión.
- Mayor flexibilidad en los flujos de trabajo.
- Operaciones locales más rápidas, al no depender constantemente de un servidor remoto.

1.2 Git

Durante los primeros años del mantenimiento del kernel de Linux, los cambios se realizaban mediante parches enviados por correo electrónico. En 2002, se comenzó a utilizar BitKeeper, un sistema de control de versiones distribuido pero propietario. En 2005, tras la ruptura de la relación con la empresa desarrolladora y debido a las restricciones de licencia, Linus Torvalds decidió desarrollar un nuevo sistema de control de versiones. Git fue diseñado con los siguientes objetivos:

- Alta velocidad en las operaciones.
- Diseño simple y eficiente.
- Funcionamiento completamente distribuido.
- Capacidad para manejar proyectos de gran tamaño y con miles de contribuyentes, como el kernel de Linux.

Con estos principios se desarrolló Git, cuyo núcleo inicial fue implementado en aproximadamente diez días. Desde entonces, Git se ha convertido en el sistema de control de versiones más utilizado a nivel mundial.

1.2.1 Snapshots

Una de las principales diferencias entre Git y otros sistemas de control de versiones radica en la forma en que almacenan los datos. Los VCS tradicionales gestionan los cambios como una serie de diferencias (*deltas*) entre archivos a lo largo del tiempo.

Git, en cambio, almacena la información como una secuencia de *snapshots*. Cada confirmación (*commit*) guarda una instantánea completa del estado del proyecto. Si un archivo no ha cambiado respecto al commit anterior, Git simplemente almacena una referencia al archivo existente, optimizando el uso del espacio. Este enfoque permite:

- Recuperar versiones completas de forma eficiente.
- Aumentar la integridad de los datos.
- Facilitar operaciones como ramificación y fusión.

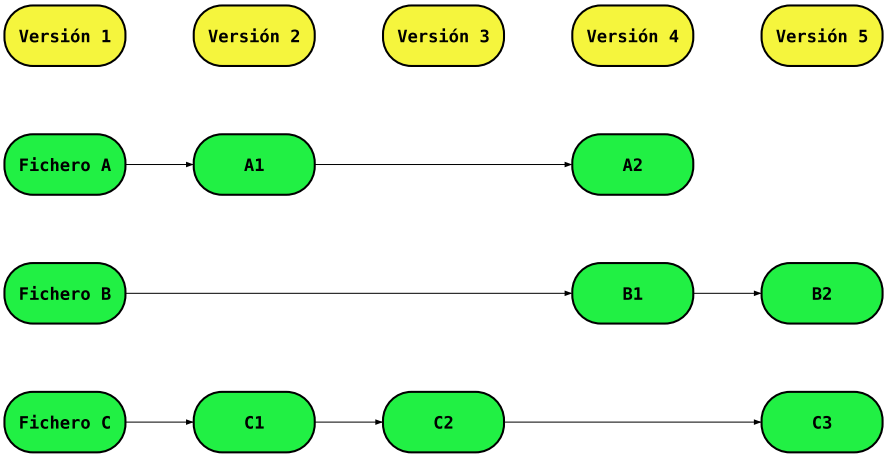

Figura 1: almacenamiento de información en VCS tradicionales

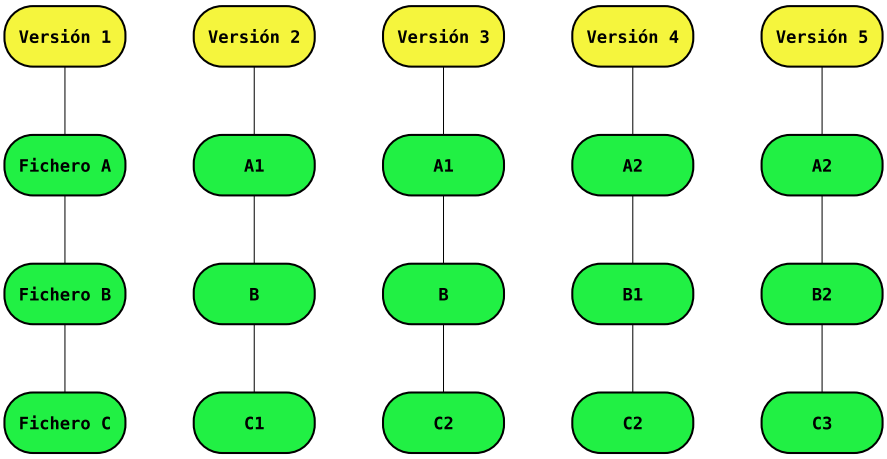

Figura 2: almacenamiento de información en Git

1.2.2 Operaciones en Local

Una característica fundamental de Git es que casi todas las operaciones se realizan de forma local. Dado que el repositorio completo (con todo el historial de commits, ramas y etiquetas) se encuentra en tu máquina, acciones como consultar el historial de hace un año, comparar dos versiones de un archivo o crear una rama nueva no requieren comunicación con un servidor remoto.

Esto proporciona:

- **Velocidad extrema:** Las consultas al historial no dependen de la red.
- **Independencia:** Puedes trabajar sin conexión, realizando commits y gestionando ramas de forma normal.
- **Seguridad:** Solo las operaciones de sincronización (`push`, `pull`, `fetch`) requieren conexión a internet.

1.2.3 Integridad de los Datos y SHA-1

Git está diseñado para que sea imposible alterar el contenido de un archivo o directorio sin que el sistema se dé cuenta. Todo objeto en Git es verificado mediante una función hash criptográfica llamada SHA-1 antes de ser almacenado.

El identificador de cada objeto (commit, archivo o árbol) es una cadena de 40 caracteres hexadecimales generada a partir de su contenido. Por ejemplo: `24b9da6552252987aa493b52f8696cd6d3b00373`.

Gracias a este mecanismo:
- El identificador es una «huella digital» del contenido.
- Cualquier corrupción de datos o modificación malintencionada en el historial sería detectada inmediatamente al cambiar el hash resultante.
- Git referencia todo por su hash en lugar de por su nombre de archivo, lo que garantiza la integridad total de la base de datos del proyecto.

1.2.4 Estados y Áreas de Trabajo en Git

Git organiza el trabajo mediante un modelo claro que combina **estados** y **áreas de trabajo**. Mientras que los estados describen la situación de un archivo dentro del flujo de versionado, las áreas representan los distintos lugares donde Git almacena y gestiona esos archivos. Comprender esta distinción es fundamental para entender cómo funciona Git.

1.2.5 Áreas de Trabajo de Git

Git utiliza tres áreas principales:

- El directorio de trabajo (*working directory*)
- El área de preparación (*staging area* o *index*)
- El repositorio Git (*repository*)

El **directorio de trabajo** contiene los archivos reales del proyecto que el desarrollador edita directamente. El **área de preparación** almacena los cambios que han sido seleccionados para el próximo commit. El **repositorio Git** contiene el historial completo del proyecto, almacenado como una secuencia de commits en la carpeta `.git`.

1.2.6 Los Tres Estados de Git

A lo largo del flujo de trabajo, los archivos pueden encontrarse en uno de los siguientes estados:
- Modificado (*Modified*)
- Preparado (*Staged*)
- Confirmado (*Committed*)

Estos estados están directamente relacionados con las áreas de trabajo de Git.

1.2.7 Modificado (Working Directory)

Un archivo se encuentra en estado *modificado* cuando ha sido alterado en el directorio de trabajo, pero sus cambios aún no han sido añadidos al área de preparación. En este estado,

Git detecta que el archivo es distinto respecto a la última versión confirmada, pero dichos cambios todavía no formarán parte del siguiente commit.

1.2.8 Preparado (Staging Area)

Un archivo pasa al estado *preparado* cuando sus cambios se añaden al área de preparación mediante comandos como `git add`. En este punto, el archivo queda marcado explícitamente para ser incluido en el próximo commit. El área de preparación permite construir commits de forma precisa, eligiendo exactamente qué cambios se registrarán, incluso cuando se han realizado múltiples modificaciones en el directorio de trabajo.

1.2.9 Confirmado (Repository)

Un archivo alcanza el estado *confirmado* cuando los cambios preparados se guardan de forma permanente en el repositorio local mediante un commit. Git almacena entonces un snapshot del estado del proyecto, que pasa a formar parte del historial y puede ser recuperado en cualquier momento.

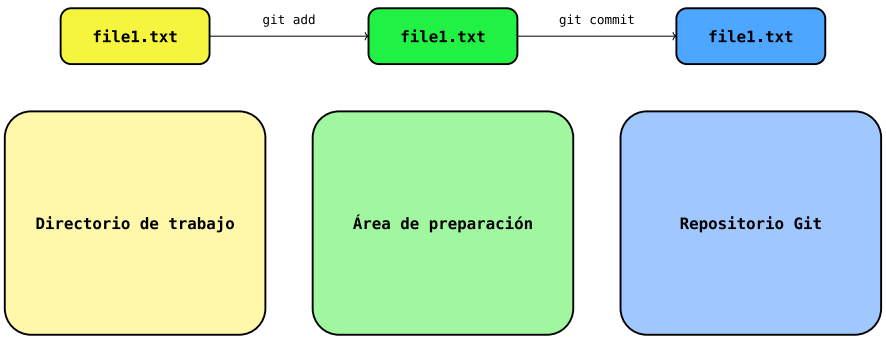

Figura 3: los tres estados de Git

1.2.10 Flujo de Trabajo Básico

El flujo de trabajo típico en Git sigue esta secuencia:

- Los archivos se modifican en el directorio de trabajo.
- Los cambios seleccionados se añaden al área de preparación.
- Los cambios preparados se confirman en el repositorio Git.

Este modelo, basado en áreas bien definidas y estados explícitos, es una de las principales diferencias de Git frente a otros sistemas de control de versiones y proporciona un alto grado de control sobre el proceso de versionado.

> **El flujo básico:** Modificas archivos -> Los añades al Staging Area con `git add` -> Los confirmas permanentemente con `git commit`.

1.3 Ramificaciones (Branching)

Una **rama** en Git no es una copia de archivos, sino un puntero móvil que apunta a un commit específico. Esto las hace increíblemente ligeras.

1.3.1 El puntero HEAD

Git utiliza el puntero especial `HEAD` para saber en qué rama te encuentras. Al cambiar de rama con `git switch` o `git checkout`, `HEAD` se desplaza y Git actualiza los archivos de tu Directorio de Trabajo para que coincidan con el estado de esa rama específica.

> Crear una rama en Git es una operación instantánea (escribir 41 bytes en un archivo), a diferencia de otros sistemas donde era necesario copiar todo el código fuente a una carpeta nueva.

2 Instalación y Configuración Inicial

Para comenzar a trabajar con Git, es necesario preparar el entorno de desarrollo. Aunque Git puede utilizarse a través de diversas interfaces gráficas (GUIs), en este curso nos centraremos en el uso de la línea de comandos.

2.1 Instalación del Software

El proceso de instalación varía según el sistema operativo, pero el resultado final será el mismo: el acceso al comando `git` en nuestra terminal.

2.1.1 En sistemas GNU/Linux

La mayoría de las distribuciones Linux incluyen Git en sus repositorios oficiales. Para sistemas basados en Debian o Ubuntu, el proceso es el siguiente:

```
sudo apt update
sudo apt install git
```

Para comprobar que la instalación se ha completado correctamente, ejecuta:

```
git --version
```

2.1.2 En Microsoft Windows

Existen dos formas principales de trabajar en Windows. La elección dependerá de si prefieres una emulación o un entorno Linux real.

1. Opción Estándar (Git Bash): Es la opción más sencilla para principiantes. Descarga el instalador desde https://git-scm.com y asegúrate de marcar la casilla de **Git Bash**. Esto instala una terminal ligera que emula el comportamiento de Unix.

2. Opción Avanzada (WSL - Windows Subsystem for Linux): Si prefieres trabajar en un entorno Linux real (como Debian o Ubuntu) sin salir de Windows, puedes usar WSL. Para instalarlo, abre PowerShell como administrador y ejecuta:

```
wsl --install
```

Tras reiniciar, tendrás una terminal de Linux (por defecto Ubuntu, aunque puedes instalar Debian desde la Microsoft Store). Dentro de esa terminal, la instalación de Git se realiza exactamente igual que en Linux:

```
sudo apt update && sudo apt install git
```

> **Importante:** Si usas WSL, Git vivirá dentro de tu instancia de Linux. Los archivos de Windows son accesibles desde `/mnt/c/`, pero se recomienda trabajar dentro del sistema de archivos de Linux para obtener el máximo rendimiento.

2.1.3 En macOS

La forma más rápida es a través de las herramientas de línea de comandos de Xcode. Simplemente abre una terminal y escribe `git`. Si no está instalado, el sistema lanzará un aviso para descargarlo. También puedes usar Homebrew:

```
brew install git
```

3 Práctica Guiada: Proyecto Rainbow

Metodología: A diferencia de un tutorial paso a paso, aquí se te indicará **qué** debes conseguir, pero no **cómo**. Deberás consultar tu **Anexo de Comandos** para encontrar la orden exacta.

3.1 Hito 1: Inicialización del Entorno

El primer paso es crear el «suelo» sobre el que trabajaremos.

1. Crea una carpeta en tu ordenador llamada `proyecto-rainbow`.
2. Abre esa carpeta en tu terminal.
3. Inicialízala como un repositorio Git.

> **Pista:** Si usas `ls -a` (Mac/Linux) o activas «Ver elementos ocultos» (Windows), deberías ver una carpeta `.git`.

> 📝 **Verificación: Verificación de Estado:** Ejecuta el comando para ver el estado del repositorio.
> - Debes ver que estás en la rama `main` (o master).
> - Debes ver que no hay commits todavía (`No commits yet`).

3.2 Hito 2: Configuración de Identidad

Git necesita saber quién eres para firmar tu trabajo.

Acción: Configura tu **Nombre** y tu **Email** a nivel local para este proyecto.

> 📝 **Verificación: Autoevaluación:** Lista la configuración de Git. Busca las líneas `user.name` y `user.email`. ¿Aparecen tus datos correctamente?

3.3 Hito 3: Ciclo de Vida (Untracked -> Staged)

1. Abre tu editor de texto favorito (Bloc de notas, VS Code, Nano, Vim…).
2. Crea un archivo llamado `colores.txt` dentro de la carpeta del proyecto.
3. Escribe la palabra: `Rojo`.
4. Guarda y cierra el archivo.

> 📝 **Verificación: Autoevaluación:** Comprueba el estado (`status`).

Reto: En este momento el archivo es **Untracked**. Ejecuta el comando necesario para pasarlo al **Staging Area**.

📝 **Verificación: Autoevaluación:** Comprueba el estado (`status`).

3.4 Hito 4: Confirmación (Commit)

El archivo está preparado, pero aún no está guardado en el historial permanentemente.

Reto: Realiza un **Commit** con el mensaje «Añadido color Rojo».

Ejercicio de Abstracción: Toma un papel y un bolígrafo. Dibuja un círculo que represente tu primer commit (C1). Dibuja una etiqueta rectangular llamada `main` y otra `HEAD` apuntando a ese círculo. **Así es como Git ve tu proyecto ahora mismo. Este grafo seria util actualizarlo a lo largo de los ejercicios**

📝 **Verificación: Auditoría del Historial:** Ejecuta el comando para ver el historial (`log`) y verifica los datos:
1. **Autor:** ¿Aparece tu nombre y correo tal como los configuraste?
2. **Fecha:** ¿Coincide la hora con la actual?
3. **Hash:** Fíjate en el código alfanumérico (ej: `a1b2c...`). Ese es el Hash único de tu commit.

3.5 Hito 5: Modificación y Diferencias

El proyecto evoluciona.

1. Abre `colores.txt` y añade una segunda línea con el texto: `Naranja`.
2. Guarda el archivo.

Reto:
1. Antes de guardar nada, ejecuta el comando para ver las **diferencias** (`diff`). Asegúrate de que Git ha detectado que has añadido una línea (`+ Naranja`).
2. Pasa los cambios al Staging Area.
3. Crea un nuevo commit con el mensaje «Añadido color Naranja».

📝 **Verificación: Autoevaluación:** Consulta el historial resumido (`--oneline`). Deberías ver dos commits:
- `(HEAD -> main) Añadido color Naranja`
- `Añadido color Rojo`

3.6 Hito 6: Viaje en el Tiempo (Checkout)

Vamos a volver al pasado para ver cómo era el proyecto al principio.

Acción:
1. Busca el **Hash** (los 7 primeros caracteres) del **primer commit** (el de «Rojo»).

2. Utiliza el comando de navegación para moverte a ese commit específico.

> **Advertencia:** Git te mostrará un aviso de **«Detached HEAD»**. Significa que estás mirando una snapshot antigua y no la versión actual.

> 📝 **Verificación: Verificación:** Abre el archivo `colores.txt` en tu editor.
> - ¿Qué contenido tiene? ¿Por qué?

Retorno al presente: Ejecuta el comando para volver a la rama `main`. Verifica que el contenido de colores.txt

3.7 Hito 7: Creación de Ramas (Universos Paralelos)

Queremos experimentar sin romper la versión principal.

Acción:
1. Crea una nueva rama llamada `frios`.
2. Cámbiate a esa rama.

> 📝 **Verificación: Autoevaluación de HEAD:** Ejecuta el log decorado (`git log --oneline --decorate`). ¿Donde se encruente el **HEAD**?

3.8 Hito 8: Divergencia

Estando en la rama `frios`:
1. Edita el archivo `colores.txt`.
2. Añade al final del fichero: `Azul`.
3. Guarda y cierra.
4. Añade el fichero al **Staging Area**
4. **Haz un commit** en esta rama con el mensaje «Configuración Azul».

3.9 Hito 9: Fusión (Merge Fast-Forward)

El experimento ha gustado. Queremos traer los cambios de `frios` a `main`.

1. **Importante:** Cámbiate a la rama `main` (comprebe el contenido de colores.txt)
2. Ejecuta el comando para **fusionar** la rama `frios` dentro de la actual.

> 📝 **Verificación: Resultado Esperado:** Git te dirá `Fast-forward`. Ahora `colores.txt` en `main` contiene «Azul».

3.10 Hito 10: Simulación de Trabajo en Paralelo

Vamos a simular que dos desarrolladores trabajan a la vez partiendo del mismo punto.

1. **Desarrollador A (Amarillo):**
   - Asegúrate de estar en `main`.
   - Crea una rama `dev-amarillo` y entra en ella.
   - Añade el texto «Amarillo».
   - Añade el fichero al **staging area**
   - Haz commit: «Cambio A: Amarillo».
2. **Desarrollador B (Rojo):**
   - **Vuelve a la base:** Cámbiate a `main`.
   - Crea una rama `dev-rojo` y entra en ella.
   - Añade el texto «Rojo Fuego».
   - Añade el fichero al **staging area**
   - Haz commit: «Cambio B: Rojo Fuego».

Situación: Tienes dos ramas que han modificado la misma línea de formas distintas.

3.11 Hito 11: El Conflicto (Merge Wars)

Es hora de integrar el trabajo.

1. Vuelve a `main`.
2. **Fusión fácil:** Fusiona la rama `dev-amarillo`. (Todo irá bien, `main` ahora es Amarillo).
3. **Fusión difícil:** Intenta fusionar la rama `dev-rojo`.

¡ERROR! Git intentará mezclar «Amarillo» con «Rojo Fuego» y fallará.

> 📝 **Verificación: Autoevaluación:** Ejecuta `git status`. Debes ver: `both modified: colores.txt`

3.12 Hito 12: Resolución Manual

1. Abre `colores.txt` con tu editor.
2. Busca los marcadores `<<<<<<<`, `=======` y `>>>>>>>`.
3. Decide cómo queda el archivo final (ej: borra los marcadores y deja solo el texto que prefieras, o combínalos).
4. Guarda el archivo limpio.
5. Añádelo al Staging Area
6. Haz el commit para confirmar la resolución.

3.13 Hito 13: Etiquetas (Tags)

Vamos a marcar este momento como nuestra versión 1.0.

Acción:
1. Crea una etiqueta llamada `v1.0.0` con el mensaje «Primera Release».
2. Lista las etiquetas para comprobar que existe.

3.14 Hito 14: La Máquina del Tiempo con Etiquetas

Acción:

1. Haz un cambio extra en el archivo (ej: añade «Violeta») y haz commit. Ahora estamos en el futuro (v1.1 hipotética).
2. **Viaja al pasado:** Muevete a la etiqueta `v1.0.0`.

> 📝 **Verificación: Verificación:** Comprueba el archivo. No debería tener «Violeta». Estás viendo la versión exacta que etiquetaste antes.

Para terminar, vuelve a la rama `main`.

3.15 Hito 15: Reescribiendo la Historia (Git Rebase) [Bonus]

Hasta ahora hemos usado `merge` para unir ramas (creando nudos). Vamos a probar una técnica avanzada para dejar el historial limpio.

Escenario: Estás desarrollando una nueva funcionalidad, pero mientras tanto, alguien ha añadido documentación en la rama principal.

1. **Preparación:** Asegúrate de estar en `main`.
2. **La Rama de Funcionalidad:**
   - Crea una rama `feature-limpia` y entra en ella.
   - Añade una línea nueva a `colores.txt`: `Verde`.
   - Añade el fichero al **Staging area**
   - Haz commit: «Añadir Verde».
3. **El Cambio en Main:**
   - Vuelve a `main`.
   - Crea un archivo NUEVO llamado `leeme.txt` y escribe dentro «Proyecto de prueba».
   - Añade el fichero al **Staging area**
   - Haz commit: «Añadir documentación».
4. **El Rebase:**
   - Vuelve a la rama `feature-limpia`.
   - Ejecuta el comando `rebase` sobre `main`.

Resultado: Git cogerá tu cambio («Añadir Verde») y lo «levantará» momentáneamente. Luego aplicará el cambio de main (crear `leeme.txt`) y finalmente volverá a pegar tu cambio al final.

> 📝 **Verificación:**
> 1. Ejecuta `git branch`. ¿Sigues viendo la rama `feature-limpia`?
> 2. Ejecuta `git log --oneline --graph --all`.
>   - Observa que la historia ahora es una línea recta.
>   - Observa que `feature-limpia` está por delante de `main`.

3.16 Hito 16: El Final Limpio (Fast-Forward y Limpieza)

Ahora mismo tu rama `feature-limpia` tiene todo el código correcto, pero `main` se ha quedado atrás.

1. Cámbiate a `main`.
2. Haz un `merge` con `feature-limpia`.
   - Al ser la historia lineal (gracias al rebase), Git no creará un «nudo» ni un commit de mezcla. Simplemente deslizará el puntero `main` hacia adelante. Esto se llama **Fast-Forward**.
3. **Limpieza:** Ahora que `main` tiene todo, borra las demas ramasa.

> 📝 **Verificación:**
> - Solo debería quedar una rama: `main`.
> - Si ejecutas `ls` o miras la carpeta, debes tener `colores.txt` (con Verde) y `leeme.txt`.

4 Práctica: Colaboración con GitHub

Hasta ahora, nuestro arcoíris solo existe en tu ordenador. En entornos profesionales, el código debe residir en un servidor para permitir el respaldo, la colaboración y el despliegue. En esta sección, aprenderemos a sincronizar nuestro trabajo con **GitHub** utilizando protocolos seguros.

4.1 Hito 1: Creación del Repositorio en la Nube

Antes de enviar archivos, necesitamos un «almacén» vacío en el servidor.

1. Inicia sesión en https://github.com.
2. Haz clic en el botón **«New»** (o el icono +) para crear un nuevo repositorio.
3. Nombre del repositorio: `proyecto-rainbow`.
4. **IMPORTANTE:** No marques ninguna casilla (README, .gitignore o licencia). Queremos un repositorio totalmente vacío.

> 📝 **Verificación: Observación:** Tras crear el repositorio, GitHub te muestra varias opciones de URL. Fíjate en el botón que dice **SSH** (junto a HTTPS). Si lo pulsas, la URL cambiará a un formato que empieza por `git@github.com`. Esa es la que usaremos.

4.2 Hito 2: Seguridad y Autenticación (Configuración SSH)

GitHub ya no recomienda usar contraseñas simples. La forma profesional de identificarse es mediante **Criptografía Asimétrica**.

Este sistema se basa en dos archivos vinculados matemáticamente:
1. **Clave Privada:** Es tu identidad secreta. Se guarda en tu PC y nunca se comparte.
2. **Clave Pública:** Es tu identidad visible. Se sube a GitHub para que pueda verificar tu firma.

4.2.1 1. Generar el par de claves

El comando es estándar, pero la terminal varía según tu sistema:

Linux / macOS: **Abre tu terminal habitual. Windows:** Abre **Git Bash** (el icono de colores).

Copia y ejecuta este comando:

```
ssh-keygen -t ed25519 -C "tu@email.com"
```

> **Instrucciones:** Pulsa **Enter** a todas las preguntas para aceptar la ruta por defecto. Si dejas la **passphrase** vacía, no te pedirá contraseña al hacer push.

4.2.2 2. Activar el Agente SSH (Vital)

Para que tu terminal pueda usar la clave, necesitamos encender el gestor de identidad (SSH Agent) y cargar tu fichero.

Ejecuta estos dos comandos en orden:

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

> **¿Qué hace esto?**
> 1. `ssh-agent -s`: Inicia el proceso del agente en segundo plano.
> 2. `ssh-add`: Carga tu clave privada en la memoria para que Git la use.
>
> **Si saltas este paso, obtendrás un error de «Permission denied».**

4.2.3 3. Obtener la Clave Pública

Necesitamos copiar el contenido de la clave pública (extensión `.pub`) para dársela a GitHub.

En Linux / macOS / Git Bash:

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

> **Nota para Windows:** Si usas Git Bash, asegúrate de que busca en la ruta correcta (usualmente `/c/Users/TuUsuario/.ssh/`).

Copia **todo el texto** que aparece en pantalla, comenzando por `ssh-ed25519` y terminando por tu etiqueta (email).

4.2.4 4. Instalar la Clave Pública en GitHub

1. Ve a GitHub -> Haz clic en tu foto de perfil -> **Settings**.
2. En el menú lateral izquierdo, selecciona **SSH and GPG keys**.
3. Pulsa el botón verde **New SSH key**.
4. **Title:** Pon un nombre que identifique tu PC (ej: «PC Laboratorio»).
5. **Key:** Pega el contenido de la clave pública que acabas de copiar.
6. Pulsa **Add SSH key**.

4.2.5 5. Probar la conexión

Verifica que GitHub reconoce tu firma criptográfica:

```
ssh -T git@github.com
```

📝 **Verificación: Verificación:** La primera vez, SSH te avisará de que no conoce al servidor (`The authenticity of host...`). Escribe `yes` para confirmar. Deberías ver el mensaje: **«Hi [Usuario]! You've successfully authenticated»**.

Fase Autónoma: A partir de este punto, deberá deducir los comandos basándose en los objetivos descritos y la documentación del Anexo.

4.3 Hito 3: Vinculación del Remoto (Remote Add)

Es necesario registrar la URL del servidor en la configuración local de Git.

Objetivo:
1. Obtenga la URL SSH del repositorio creado en el Hito 1 (formato `git@github.com:...`).
2. Vincule dicha URL al repositorio local bajo el alias estándar `origin`.

📝 **Verificación: Validación:** Ejecute el comando para listar los remotos con detalle (`-v`). Debe observar las direcciones de `fetch` y `push` apuntando a GitHub.

4.4 Hito 4: Sincronización Inicial (Push Upstream)

El repositorio local contiene commits que no existen en el servidor.

Objetivo: Envíe (`push`) la rama `main` al repositorio remoto `origin`. **Nota Técnica:** Al ser el primer envío, debe establecer la rama remota como **upstream** (rastreo) para simplificar comandos futuros (use el flag `-u`).

📝 **Verificación: Resultado:** Actualice la página del repositorio en el navegador. Deberá visualizar el código fuente (`colores.txt`) y el historial de commits idéntico al local.

4.5 Hito 5: Simulación de Colaboración (Pull)

Simularemos la intervención de un segundo desarrollador modificando el código directamente en el servidor.

1. **Modificación Remota:**
   - En la interfaz web de GitHub, edite el archivo `leeme.txt` (o créelo si no existe).
   - Añada el texto: «Modificación realizada desde el servidor».
   - Confirme el cambio con el botón «Commit changes».
2. **Estado Local:**
   - Verifique en su terminal que el archivo `leeme.txt` local **no** contiene aún estos cambios.

Objetivo: Ejecute el comando necesario para descargar y fusionar los cambios remotos en su rama local actual.

📝 **Verificación: Validación:** Inspeccione el contenido local de `leeme.txt`. El texto nuevo debe aparecer.

4.6 Hito 6: Estrategia de Ramas (Gitflow Simplificado)

En este ejercicio trabajaremos sobre una rama distinta al `main` para realizar nuevos desarrollos.

Objetivo:
1. Asegúrese de estar en `main` y actualizado.
2. Cree una rama llamada `develop` y cámbiese a ella.
3. Suba la rama `develop` al servidor estableciendo el upstream.

📝 **Verificación: Validación:** Vaya a GitHub.com. Arriba a la izquierda de la lista de archivos, haga clic en el desplegable que dice **main**. Ahora debería ver y poder seleccionar **develop**.

4.7 Hito 7: Trabajo en Feature Branch (Aislamiento)

En un equipo real, trabajar directamente sobre `develop` es peligroso: si cometes un error, bloqueas el trabajo de todos tus compañeros. Por eso, cada nueva tarea se hace en su propia «burbuja» aislada.

Objetivo:
1. Estando en `develop`, cree una rama nueva `feature-experimento` para aislar su trabajo
2. Cree un archivo `experimento.txt` con contenido `TODO: mejora cromática`, añádalo al staging area y haga commit.
3. Suba esta rama específica al servidor

Concepto Clave: Ahora tienes una copia de seguridad de tu trabajo en el servidor (`feature/experimento`), pero **aún no** has tocado la rama de integración (`develop`). Esto permite que otros sigan trabajando sin verse afectados por tus cambios incompletos.

4.8 Hito 8: Integración y Despliegue (Merge Local)

Vamos a simular que el experimento ha sido un éxito. Ahora debemos llevar esos cambios hasta la rama de producción (`main`), pasando primero por integración (`develop`).

Realizaremos esta operación **manualmente en local** para entender el movimiento de datos.

Objetivo:

1. **Integrar:** Sitúese en la rama `develop` y fusione (`merge`) la rama `feature/experimento`.
2. **Desplegar:** Sitúese en la rama `main` y fusione la rama `develop`.
3. **Sincronizar:** Envíe la rama `main` actualizada al servidor (`push`).

> **Nota:** Observe que al hacer el `push` final, GitHub detectará automáticamente que `main` ha avanzado y mostrará los cambios del experimento en la página principal del repositorio.

4.9 Hito 9: Exclusión de Archivos (.gitignore)

Es buena práctica evitar que archivos sensibles o temporales contaminen el repositorio.

Objetivo:

1. Cree un archivo local llamado `credenciales.key`.
2. Configure el repositorio para que Git **ignore** automáticamente cualquier archivo con extensión `.key`.
3. Compruebe el estado del repositorio (`status`).

> 📝 **Verificación: Validación:** El archivo `credenciales.key` no debe aparecer en la lista de **Untracked files**. Git debe omitirlo completamente.

5 Conceptos Avanzados: Flujos de Trabajo Profesionales

Ahora que dominas los comandos, es vital entender el **protocolo**. En el mundo real, aunque técnicamente puedes hacer un merge a `main` (como hicimos en el Hito 8), las empresas suelen prohibirlo por seguridad.

5.1 1. ¿Qué es realmente un Pull Request (PR)?

Técnicamente, un Pull Request termina siendo un `git merge`. Sin embargo, la diferencia es el **proceso**:

- **Merge Local:** Tú eres juez y parte. Fusionas tu código sin preguntar.
- **Pull Request:** Es una **Sala de Espera**. Tu código se queda en un «limbo» donde ocurren dos cosas antes de entrar a la rama principal:
  1. **Code Review:** Tus compañeros leen tu código para detectar errores lógicos.
  2. **CI/CD (Integración Continua):** Robots automáticos ejecutan tests para asegurar que no has roto nada.

Solo cuando los humanos (Code Review) y los robots (CI/CD) dan el visto bueno, se permite la fusión.

5.2 2. Ramas Protegidas (Protected Branches)

Seguramente te has preguntado: **«¿Por qué hacer todo este lío del PR si puedo hacer merge en mi PC y subirlo?»**.

La respuesta es la **Seguridad**. En las empresas, la rama `main` (y a veces `develop`) está **bloqueada**.

> **Simulación:** Si intentaras hacer `git push origin main` en una empresa real, recibirías este error:
> `remote: error: GH006: Protected branch 'main'. Push rejected.`

El Pull Request es la única «llave» administrativa que permite saltarse ese bloqueo de forma controlada.

5.3 3. Arquitecturas de Colaboración

Existen dos formas principales de organizar el trabajo en equipo con Git:

5.3.1 A. Modelo de Repositorio Compartido (Shared Repository)

El que hemos usado en esta práctica. Típico de empresas y equipos internos. Todo el equipo tiene acceso de escritura al mismo repositorio central.

5.3.2 B. Modelo Open Source (Fork & Pull)

El que se usa para colaborar en proyectos públicos (Linux, React, etc.).

Como no tienes permiso para escribir en el repositorio de otro (ej. Microsoft), el flujo es un «triángulo»:

1. **Fork:** Creas una copia exacta del repositorio ajeno en **tu** cuenta de GitHub.
2. **Push:** Subes los cambios a **tu** copia.
3. **Pull Request:** Solicitas desde GitHub que el dueño del repositorio original acepte los cambios que tienes en tu copia.

Conclusión: Git es la herramienta (el martillo), pero flujos como **Gitflow** o **GitHub Flow** son la técnica de construcción. Un buen desarrollador domina ambos.

6 Práctica: Optimización del Flujo con VS Code

Hasta ahora has operado en la terminal, lo cual te otorga el máximo control y entendimiento sobre Git. Muchos profesionales prefieren la terminal por su velocidad y precisión.

Sin embargo, existen tareas donde una interfaz gráfica (GUI) aporta valor añadido, especialmente en la **visualización de diferencias** y la **resolución de conflictos**.

Objetivo: Transicionar del uso de comandos manuales a un flujo de trabajo visual para la gestión de ramas, visualización de diferencias (`diff`) y resolución de conflictos.

6.1 Hito 1: Verificación del Entorno Integrado

VS Code no requiere plugins externos para operar con Git, ya que utiliza la instalación del sistema subyacente.

1. Abra VS Code.
2. Abra la terminal integrada (`Terminal > New Terminal`).
3. Verifique que VS Code detecta su instalación de Git:

```
git --version
```

Configuración Recomendada: Para evitar que Git abra editores externos (como Vim o Nano) al solicitar mensajes de commit o fusiones, configuraremos VS Code como el editor predeterminado del sistema:

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

6.2 Hito 2: La Interfaz de Control de Código Fuente

En la barra lateral izquierda, localice el icono de **Source Control** (tres nodos conectados, o `Ctrl + Shift + G`).

Esta vista reemplaza al comando `git status`. Se divide en:

- **Changes:** Archivos modificados en el Working Directory (Rojos en terminal).
- **Staged Changes:** Archivos preparados para commit (Verdes en terminal).

Actividad:

1. Abra el archivo `colores.txt` y añada un nuevo color: `Magenta`.
2. Observe cómo aparece inmediatamente bajo la sección «Changes» con una `M` (Modified).

6.3 Hito 3: Análisis de Diferencias (Visual Diff)

Una de las mayores ventajas de la GUI es la inspección de cambios antes de confirmar.

Actividad: Haga clic **una vez** sobre `colores.txt` en el panel de Source Control.

- Se abrirá una vista dividida (**Diff View**).
- A la izquierda (Rojo): La versión original.
- A la derecha (Verde): Su modificación actual.

> **Ventaja:** Esto permite auditar visualmente el código para evitar subir `console.log`, comentarios basura o errores evidentes antes de siquiera hacer el staging.

6.4 Hito 4: Staging Selectivo e Interactivo

En lugar de escribir `git add .` (que añade todo ciegamente), VS Code permite precisión quirúrgica.

Actividad:
1. Pase el ratón sobre el archivo `colores.txt` en el panel lateral.
2. Pulse el icono `+` (Stage Changes).
3. Observe cómo el archivo se mueve de la lista **Changes** a **Staged Changes**.

Nota: Si quisiera sacar el archivo del área de preparación (unstage), simplemente pulsaría el icono `-`.

6.5 Hito 5: Ciclo de Commit y Sincronización (Configuración SSH)

La interfaz simplifica el ciclo `commit` -> `push`, pero requiere una configuración extra para funcionar con nuestras llaves SSH.

6.5.1 1. Realizar el Commit

1. En la caja de texto superior del panel lateral, escriba el mensaje: «Añadir Magenta vía GUI».
2. Pulse el botón azul **Commit** (o `Ctrl + Enter`).

6.5.2 2. El intento de Sincronización (Error Esperado)

Localice el botón azul **Sync Changes** (flechas rotatorias) en el panel lateral o en la barra de estado inferior. **Púlselo**.

¿Qué ocurre? Es muy probable que VS Code le muestre un error de **«Permission denied»** o le pida su contraseña repetidamente.

La Causa: Cuando activó el agente SSH en la lección anterior (`eval $(ssh-agent)...`), lo hizo **solo para esa ventana de terminal**. VS Code es un programa independiente y no «ve» su llave cargada en memoria.

6.5.3 3. La Solución: Archivo Config

Para no tener que activar el agente manualmente cada vez que abrimos VS Code, crearemos un archivo de configuración permanente.

1. En la terminal integrada de VS Code, vaya a su carpeta `.ssh`:

```
cd ~/.ssh
```

2. Cree (o abra) el archivo `config` usando el propio VS Code:

```
code config
```

3. Pegue el siguiente contenido en el archivo (ajustando `id_ed25519` si usó otro nombre):

```
Host github.com
  HostName github.com
  User git
  IdentityFile ~/.ssh/id_ed25519
```

4. Guarde (`Ctrl+S`) y cierre el archivo.

Actividad Final: Vuelva a pulsar el botón **Sync Changes**. Ahora VS Code leerá ese archivo, encontrará la llave automáticamente y enviará sus cambios a GitHub sin errores.

6.6 Hito 6: Gestión Visual de Ramas

VS Code facilita la navegación entre ramas sin necesidad de memorizar comandos.

6.6.1 1. Crear una Rama

1. Mire la **Barra de Estado** (la franja azul o gris en la parte inferior de la ventana).
2. A la izquierda, verá el nombre de su rama actual (ej. `main` o `master`). **Haga clic en ese nombre**.
3. Se abrirá un menú superior. Seleccione: `+ Create new branch...`.
4. Escriba el nombre: `gui-test` y pulse Enter.
5. Observe abajo a la izquierda que VS Code ha cambiado automáticamente a la nueva rama.

6.6.2 2. Modificar y Cambiar de Rama

1. En la rama `gui-test`, modifique `leeme.txt` añadiendo una línea al final. Haga Commit.
2. Para volver a `main`: Haga clic en `gui-test` (abajo a la izquierda) y seleccione `main` en la lista.

6.6.3 3. Fusionar (Merge) desde la GUI

Ahora estamos en `main` y queremos traer los cambios de `gui-test`.

1. Abra la paleta de comandos (`F1` o `Ctrl+Shift+P`).
2. Escriba `Git: Merge Branch...` y selecciónelo.
3. Elija la rama que quiere absorber: `gui-test`.
4. VS Code realizará la fusión. Si no hay conflictos, será instantáneo.

6.7 Hito 7: Resolución Gráfica de Conflictos

La resolución de conflictos mediante marcadores de texto (`<<<< HEAD`) es propensa a errores humanos. VS Code ofrece un editor de tres vías («3-way merge») o lentes de código («CodeLens»).

Simulación:

1. Genere un conflicto: Modifique la misma línea de un archivo tanto en `main` como en `gui-test` con textos diferentes y haga commit en ambas.
2. Intente hacer el Merge como en el paso anterior.
3. VS Code le avisará de un conflicto.

Acción: Al abrir el archivo conflictivo, VS Code no mostrará solo texto plano. Resaltará los bloques y ofrecerá opciones clicables:

- `Accept Current Change`: Mantiene lo que tienes en tu rama actual (`main`).
- `Accept Incoming Change`: Sobrescribe con lo que viene de la otra rama (`gui-test`).
- `Accept Both Changes`: Conserva ambas líneas.

Seleccione la opción deseada haciendo clic. VS Code limpia los marcadores automáticamente. Guarde y realice el commit para finalizar.

> 📝 **Verificación: Conclusión:** El uso de la GUI no reemplaza el conocimiento de la terminal, sino que lo complementa.
> - Use la **Terminal** para configuraciones, arreglos complejos o automatización.
> - Use **VS Code** para el ciclo diario (escribir, ramas rápidas, resolver conflictos), ya que reduce drásticamente los errores de descuido.

7 Práctica Final: Git aplicado a KiCad

En el desarrollo de hardware, el mayor desafío es saber **qué ha cambiado** en la placa. A diferencia del código, leer un diff de texto con coordenadas (`X: 10.5 -> 10.6`) es imposible para un humano.

En esta práctica final, configuraremos el repositorio para ignorar archivos temporales y utilizaremos una herramienta especializada para visualizar cambios en el PCB.

7.1 Hito 1: Preparación del Entorno (.gitignore)

Antes de nada, debemos asegurar que Git solo vigila los archivos de diseño reales. KiCad genera multitud de archivos temporales que ensuciarían el repositorio.

1. **Diagnóstico:** Abre tu proyecto de KiCad en VS Code, inicia Git (`git init`) y haz `git status`. Verás archivos `.kicad_prl`, `fp-info-cache`, backups, etc.
2. **Creación del Filtro:** Crea un archivo `.gitignore` y pega esta configuración estándar para hardware:

```
# Ignorar configuraciones locales (vista, capas visibles)
*.kicad_prl
*.kicad_dru
fp-info-cache

# Ignorar copias de seguridad automáticas
*.kicad_pcb-bak
*.sch-bak
*-bak
*.lck

# Ignorar salidas de fabricación (se deben regenerar, no guardar)
/gerbers/
/plot/
*.zip
```

3. **Primer Commit (Línea Base):** Ahora que `git status` sale limpio (solo muestra `.kicad_sch`, `.kicad_pcb` y `.kicad_pro`), guarda la versión inicial:

```
git add .
git commit -m "Estructura inicial del proyecto limpio"
```

7.2 Hito 2: Instalación de KiCad-Diff (Entorno Virtual)

Usaremos la herramienta `KiCad-Diff`. Al ser un programa complejo, necesita su propio entorno aislado.

1. **Clonar la herramienta:** Sal de tu carpeta de proyecto y descarga el código fuente «al lado»:

```
cd ..
git clone https://github.com/Gasman2014/KiCad-Diff.git
cd KiCad-Diff
```

2. **Crear el Entorno Virtual (VENV):**

> **Usuarios de Linux (Debian/Ubuntu):** Antes de nada, instala las librerías gráficas del sistema: `sudo apt install build-essential libgtk-3-dev python3-dev libgl1-mesa-dev python3-wxgtk4.0`

Crea y activa el entorno:

```
python3 -m venv kicad-venv
```

Activar el entorno:

- Linux/Mac: `source kicad-venv/bin/activate`
- Windows: `.\kicad-venv\Scripts\activate`

3. **Instalar dependencias:** Instala las librerías y da permisos de ejecución:

```
pip install -r requirements.txt
chmod +x plot_kicad_pcb kidiff
```

7.3 Hito 2.1: Parcheado de Compatibilidad (Debian/KiCad 8)

El código original de la herramienta es antiguo y usa funciones que KiCad 7 y 8 han cambiado o eliminado. Si intentas ejecutarlo tal cual, fallará. Aplicaremos tres «parches» rápidos modificando el código fuente automáticamente:

```
sed -i 's/popt.SetPlotInvisibleText(False)/# popt.SetPlotInvisibleText(False)/g'
kidiff/plot_kicad_pcb.py
sed -i 's/settings.plot_prog/settings.pcb_plot_prog/g' kidiff/kidiff.py
sed -i 's/popt.SetSvgPrecision(aPrecision=5, aUseInch=False)/
popt.SetSvgPrecision(5)/g' kidiff/plot_kicad_pcb.py
```

> **¿Qué estamos arreglando?**
> 1. Comentamos `SetPlotInvisibleText` (obsoleto).
> 2. Corregimos un nombre de variable mal escrito en el gestor de errores (`plot_prog`).
> 3. Quitamos el argumento `aUseInch` en `SetSvgPrecision` que ya no existe en la nueva API de KiCad.

7.4 Hito 3: Simulación de Cambios

1. Vuelve a tu carpeta de proyecto: `cd ../mi-proyecto-kicad`
2. Abre KiCad, mueve algún componente o pista en el PCB, guarda y cierra.

7.5 Hito 4: Ejecución del Comparador Visual

El script `kidiff` necesita encontrar dos cosas: su ayudante `plot_kicad_pcb` (en el PATH) y la librería `pcbnew` de KiCad (en el PYTHONPATH).

Ejecución: Asegúrate de ver `(kicad-venv)` en la terminal.

1. Configura las rutas (Copia y pega este bloque entero):

```
export PYTHONPATH=$PYTHONPATH:/usr/lib/kicad/lib/python3/dist-packages
export PATH=$PATH:$<ruta_completa>/KiCad-Diff/bin
```

2. Ejecuta la herramienta:

```
kidiff mi_placa.kicad_pcb
```

(Nota: Si usas Windows, simplemente ejecuta `python ../KiCad-Diff/kidiff mi_placa.kicad_pcb`).

Resultado Esperado:
1. Se abrirá una ventana pidiendo confirmar versiones.
2. Se generará un servidor web y se abrirá tu navegador.
3. Verás una galería interactiva con el «Antes» y el «Después».

7.6 Hito 5: El Ciclo de Trabajo en Hardware

Ahora que has verificado visualmente que no has roto nada crítico (como mover un conector por error), puedes proceder a guardar tu trabajo.

1. Cierra el servidor web (Ctrl+C en la terminal).
2. Confirma los cambios en Git:

```
git add --all
git commit -m "Modificación de ruteo verificada visualmente"
```

7.7 Hito 6: Recuperación de Desastres

Un error común es borrar una pista crítica o estropear el ruteo sin querer y guardar el archivo.

Acción:
1. En el PCB, borre todo lo que ha hecho o mueva componentes al azar (simulando un error fatal). Guarde.
2. Utilice VS Code o la terminal para descartar cambios:

```
git restore .
```

3. Vuelva a abrir KiCad. Su trabajo habrá sido restaurado perfectamente al estado del último commit.

8 Ejercicio de Evaluación: Gestión del Ciclo de Vida de Software

El objetivo de este ejercicio es validar la capacidad del alumno para traducir requisitos funcionales y de negocio a operaciones de control de versiones con Git.

Metodología: A continuación se presentan una serie de necesidades del proyecto. Usted tiene total autonomía para decidir qué comandos ejecutar para satisfacerlas.

8.1 Contexto del Proyecto

Usted es el responsable de configuración de un proyecto de control de motores (`main.py`). El software debe evolucionar gestionando versiones estables, experimentales y parches de seguridad.

8.2 Hito 1: Infraestructura Base

El proyecto requiere un repositorio local limpio llamado `desafio-python`. El código inicial (`main.py`) debe cumplir la siguiente especificación:

```python
def iniciar_motor():
    # Configuración por defecto
    velocidad = 10
    print(f"Sistema arrancado a {velocidad} RPM")

if __name__ == "__main__":
    iniciar_motor()
```

Requisito: Asegure este estado inicial en la línea de historia principal.

8.3 Hito 2: Desarrollo Paralelo (Feature Isolation)

El equipo de I+D solicita probar una configuración de **100 RPM**. **Restricción:** Este cambio es experimental y **no debe afectar** a la versión estable actual del proyecto mientras se desarrolla.

Acción: Implemente el mecanismo necesario para aislar este cambio (variable `velocidad = 100`) en una línea de trabajo paralela llamada `prueba-turbo`.

8.4 Hito 3: Hotfix en Producción

Mientras I+D trabaja en el turbo, se detecta un fallo de seguridad en la versión estable (la línea principal).

Acción: Sin incluir los cambios del turbo (que aún no están listos), modifique la versión estable bajando la `velocidad` a **5**. Guarde este cambio crítico en el historial principal.

8.5 Hito 4: Integración de Características

El experimento «turbo» ha sido aprobado, pero con modificaciones. Se requiere integrar el desarrollo paralelo dentro de la línea principal del producto.

Requisito:
1. Unifique ambas líneas de trabajo.
2. Es previsible que exista incompatibilidad entre el hotfix (5 RPM) y el experimento (100 RPM).
3. **Decisión de Ingeniería:** El valor final tras la unificación debe ser un compromiso de **50**. Resuelva la incoherencia manualmente y finalice la integración.

8.6 Hito 5: Gestión de Releases (Versionado)

El producto ha alcanzado la estabilidad.

1. **Release 1.0:** Marque el estado actual de forma inmutable como la versión `v1.0` («Versión Estable 50 RPM»).
2. **Evolución:** El desarrollo continúa en la línea principal. Suba la velocidad a **200** para la nueva generación de motores.
3. **Release 2.0:** Marque este nuevo estado como la versión `v2.0`.

8.7 Hito 6: Escenario de Mantenimiento LTS (Long Term Support)

El Reto Final: Un cliente crítico que opera con la versión `v1.0` necesita un ajuste urgente a **55 RPM**.

Restricciones Bloqueantes:
- El cliente **NO puede recibir** el código de la versión `v2.0` (200 RPM).
- El cambio debe quedar registrado oficialmente como una versión `v1.1`.

Tarea: Genere una nueva versión `v1.1` que nazca estrictamente de la `v1.0`, contenga el cambio a 55 RPM, y no tenga rastro de la evolución hacia la v2.0.

8.8 Criterios de Validación

El ejercicio se considerará superado si el repositorio cumple las siguientes condiciones verificables:

1. **Historial Gráfico:** Al visualizar el grafo (`git log --graph --all`), se observa:
   - Una bifurcación y unión (diamante) correspondiente a la integración del Hito 4.
   - Una divergencia final donde `v1.1` y `v2.0` siguen caminos separados desde `v1.0`.
2. **Integridad de Archivos:**
   - `git checkout v2.0` -> `main.py` tiene 200 RPM.
   - `git checkout v1.1` -> `main.py` tiene 55 RPM (y NO 200).
3. **Limpieza:** No existen marcadores de conflicto (`<<<<<<<`) en el código final de ninguna versión.

Fin del Ejercicio

9 Anexo: Guía de Referencia de Comandos Git

Esta sección compila los comandos esenciales utilizados durante el curso. Úsala como referencia rápida cuando necesites recordar la sintaxis exacta.

9.1 1. Configuración e Inicio

Comando	Descripción
`git version`	Muestra la versión de Git instalada en el sistema.
`git config --list`	Lista todas las opciones configuradas activas actualmente.
`git config --global user.name <nombre>`	Define el nombre de usuario por defecto para **todos** los proyectos del PC.
`git config --global user.email <email>`	Define el email por defecto para **todos** los proyectos del PC.
`git config user.name <nombre>`	Define el nombre de usuario **solo para el proyecto actual** (sobrescribe la configuración global).
`git config user.email <email>`	Define el email **solo para el proyecto actual**. Útil para separar identidad personal de la profesional/académica.
`git init`	Inicializa un repositorio Git nuevo en el directorio actual.
`git init -b <branch>`	Inicializa el repositorio creando inmediatamente la rama principal con el nombre indicado (ej. `main`).

9.2 2. Flujo de Trabajo Diario (Local)

Comando	Descripción
`git status`	**El más importante.** Muestra qué archivos han cambiado, qué está preparado para commit y la situación de la rama.

Comando	Descripción
`git add <fichero>`	Añade un archivo específico al área de preparación (**staging area**).
`git add .`	Añade cambios del directorio actual (nuevos, modificados y borrados) al área de preparación.
`git add --all`	Añade **absolutamente todo** el contenido del repositorio (equivalente a `-A`), independientemente de en qué subcarpeta te encuentres.
`git commit -m "<mensaje>"`	Guarda los cambios preparados en el repositorio local de forma permanente.
`git restore <fichero>`	Deshace los cambios en un archivo (vuelve al estado del último commit).
`git restore .`	Deshace **todos** los cambios en el directorio de trabajo (vuelve al estado limpio del último commit).
`git log`	Muestra el historial cronológico de commits.
`git log --oneline --graph --all`	Muestra el historial resumido y dibuja gráficamente las ramas y uniones.
`git log --all`	Muestra el historial de **todas** las ramas (incluidas las remotas como `origin/main`). Vital para ver commits de compañeros que aún no has fusionado.

9.3 3. Inspección Detallada de Cambios (Diff)

Comandos para revisar diferencias exactas. Es la herramienta de diagnóstico principal.

Comando	Descripción
`git diff`	Muestra diferencias de archivos modificados **no preparados** (Working Directory vs Staging Area).
`git diff --staged`	Muestra cambios que **ya has preparado** (con `git add`) comparados con el último commit. Revisar esto siempre antes de hacer commit.
`git diff -b`	**Ignore Space Change.** Muestra los cambios ignorando la cantidad de espacios en blanco. **Uso:** Vital cuando solo has cambiado la

Comando	Descripción
	indentación del código y quieres ver si has tocado la lógica real.
`git diff -w`	**Ignore All Space.** Ignora **totalmente** los espacios en blanco (incluso saltos de línea). Más agresivo que `-b`.
`git diff <hash>`	Compara tu **directorio de trabajo actual** con un commit antiguo específico. **Pregunta:** «¿Cuánto ha cambiado mi proyecto actual respecto a la versión de hace un mes?»
`git diff <rama>`	Compara tu **directorio de trabajo actual** contra la última versión de otra rama (ej. `main`). **Uso:** Para ver qué te falta o qué tienes de más respecto a la rama principal.
`git diff <hash1> <hash2>`	Compara dos commits específicos entre sí. El orden importa (muestra cómo convertir hash1 en hash2).
`git diff <rama1>..<rama2>`	Muestra las diferencias entre la punta de dos ramas distintas.
`git diff <hash1> <hash2> <fichero>`	Muestra la evolución de **un solo archivo** entre dos versiones, ignorando el resto del proyecto.
`git diff --stat`	Muestra un resumen numérico (+líneas, -líneas) sin el código. Útil para visión general.

9.4 4. Gestión de Ramas (Branching)

Comando	Descripción
`git branch`	Lista todas las ramas locales existentes.
`git branch --all`	Lista **todas** las ramas: tanto las locales como las remotas (conocidas como `remotes/origin/...`).
`git branch -vv`	**Modo detallado.** Muestra las ramas locales junto a su rama remota de rastreo (**upstream**). **Uso:** Vital para saber si tu rama está vinculada correctamente a `origin` y si vas por delante/detrás.

Comando	Descripción
`git branch -r`	Lista **únicamente** las ramas remotas. Útil para ver qué hay en GitHub sin mezclarlo con lo local.
`git branch <nombre>`	Crea una nueva rama, pero **no** cambia a ella automáticamente.
`git branch -d <nombre>`	Elimina una rama local (solo si ya ha sido fusionada).
`git switch <nombre>`	Cambia el entorno de trabajo a la rama especificada (Alternativa moderna y recomendada a `checkout`).
`git switch -c <nombre>`	Crea una rama nueva y cambia a ella en un solo paso.
`git checkout <nombre>`	Comando clásico para cambiar de rama.
`git checkout -b <nombre>`	Comando clásico equivalente a crear rama y cambiar a ella.
`git merge <rama>`	Fusiona la historia de la `<rama>` indicada dentro de la rama en la que estás situado actualmente.
`git merge --abort`	Cancela un proceso de fusión en caso de conflicto y restaura el estado anterior.

9.5 5. Trabajo con Repositorios Remotos

Comando	Descripción
`git clone <url> <carpeta>`	Descarga un repositorio remoto completo a la máquina local.
`git remote -v`	Lista los repositorios remotos vinculados y sus URLs.
`git remote add <nombre> <url>`	Vincula un nuevo repositorio remoto (comúnmente llamado `origin`) al repositorio local.
`git fetch <remoto>`	Descarga los datos del remoto pero **no** los fusiona con tu trabajo actual (es una operación segura).
`git pull <remoto> <rama>`	Descarga los cambios del remoto y trata de fusionarlos automáticamente (`fetch` + `merge`).

Comando	Descripción
`git push <remoto> <rama>`	Sube tus commits locales a la rama especificada del servidor remoto.
`git push -u origin <rama>`	**El estándar.** Sube la rama y establece el vínculo (upstream) a la vez. La `-u` es abreviatura de `--set-upstream`.
`git branch -u <remoto>/<rama>`	Establece la rama remota «upstream». Permite usar `git pull` y `git push` sin argumentos en el futuro.
`git checkout --track origin/<rama>`	Crea una rama local con el mismo nombre que la remota y las vincula automáticamente.
`git push <remoto> -d <rama>`	Elimina una rama definitivamente en el servidor remoto.
`git remote prune origin`	Limpia tu lista local de ramas remotas, borrando las referencias a ramas que ya no existen en GitHub.

9.6 6. Manipulación del Historial y Reparación

Comandos avanzados para corregir errores, mover punteros y viajar en el tiempo.

Comando	Descripción
`git reset --hard <hash>`	**Destructivo.** Mueve la rama actual al `<hash>` (o `<rama>`) indicado. **Efecto:** Borra todos los commits posteriores y **actualiza tus archivos** para que sean idénticos a ese momento. Ideal para eliminar errores recientes.
`git reset --soft <hash>`	Mueve la rama al `<hash>` (o `<rama>`) indicado pero **mantiene** los cambios en el área de preparación. **Uso:** Permite «deshacer» el commit pero no el trabajo, para volver a empaquetarlo de otra forma.
`git branch -f <rama> <hash>`	**«Teletransporte».** Fuerza a una etiqueta de rama (que no sea la actual) a apuntar a un commit específico. Útil para recolocar ramas que se quedaron atrás o se desviaron.
`git checkout <hash>`	**«Detached HEAD».** Mueve tu vista a un commit antiguo para inspeccionarlo. No mueve

Comando	Descripción
	ninguna rama. Si haces commits aquí, quedarán huérfanos al salir (a menos que crees una rama nueva).

9.7 7. Etiquetas (Tags)

Marcadores fijos para versiones de software (v1.0, v2.0).

Comando	Descripción
`git tag`	Lista solo los nombres de las etiquetas existentes.
`git tag -n`	Lista las etiquetas mostrando también la primera línea de su mensaje.
`git show <tag>`	Muestra toda la información de la etiqueta: mensaje completo, autor, fecha y el commit al que apunta.
`git tag <nombre>`	Crea una etiqueta ligera (ej: `v1.0`) en el commit actual (sin mensaje).
`git tag -a <nombre> -m "msg"`	Crea una etiqueta anotada (con fecha, autor y mensaje). Recomendada para releases públicas.
`git push origin <tag>`	Sube una etiqueta específica al servidor remoto.
`git push origin --tags`	Sube **todas** las etiquetas locales que faltan en el remoto.
`git tag -d <nombre>`	Elimina una etiqueta localmente.

> **Consejo:** Si en algún momento te sientes perdido, ejecuta `git status`. El 90% de las veces, Git te sugerirá en la propia terminal qué comando tiene sentido ejecutar a continuación.

9.8 8. Guardado Temporal (Stashing)

Permite guardar el trabajo a medias en una «pila» temporal para limpiar el directorio de trabajo sin hacer commits incompletos.

Comando	Descripción
`git stash`	Guarda los cambios de los archivos modificados (trackeados) en una pila temporal y revierte el directorio de trabajo al último commit limpio.
`git stash -u`	Igual que el anterior, pero incluye también los archivos nuevos (**untracked**) que aún no habías añadido con `git add`. Por defecto `stash` los ignora.
`git stash pop`	Saca los últimos cambios guardados de la pila, los aplica a tu rama actual y, si no hay conflictos, los borra de la pila.
`git stash apply`	Aplica los cambios guardados pero **los mantiene** en la pila. Útil si quieres aplicar el mismo «parche» en varias ramas distintas.
`git stash list`	Muestra una lista de todos los estados guardados temporalmente (ej. `stash@{0}`, `stash@{1}`).
`git stash drop <id>`	Elimina una entrada específica de la lista (ej. `stash@{0}`).
`git stash clear`	Borra definitivamente **todas** las entradas guardadas en el stash. **Cuidado:** No se puede deshacer.