Gestión del Ciclo de Vida del Código Fuente con Git y GitHub

Sección I: El Papel Fundamental del Control de Versiones en la Gestión del Código Fuente

7.1.1 La Necesidad de un Repositorio de Código Fuente

En el desarrollo de software moderno, la colaboración es fundamental. Múltiples desarrolladores trabajan a menudo sobre la misma base de código para implementar nuevas funcionalidades, corregir errores y mantener el software. Sin un sistema de gestión, este proceso puede volverse caótico, llevando a la sobreescritura de trabajo, la pérdida de cambios y una incapacidad para determinar qué modificaciones causaron un fallo. Un repositorio de código fuente, gestionado por un Sistema de Control de Versiones (VCS, por sus siglas en inglés), es la solución a este problema.[1] Actúa como la única fuente de verdad (*single source of truth*) y proporciona un historial completo de cada cambio realizado en el proyecto.

7.1.2 Rol y Características de un Sistema de Control de Versiones (VCS)

El rol principal de un VCS es registrar los cambios en un archivo o conjunto de archivos a lo largo del tiempo, permitiendo a los equipos recuperar versiones específicas en cualquier momento. Sus características clave son:

1. **Trazabilidad e Historial:** Mantiene un registro detallado de quién cambió qué, cuándo y por qué (a través de los mensajes de *commit*).
2. **Concurrencia y Colaboración:** Permite que varios desarrolladores trabajen en el mismo proyecto simultáneamente sin interferir entre ellos.
3. **Aislamiento (Ramificación):** Proporciona la capacidad de crear "ramas" o líneas de desarrollo independientes (p. ej., para una nueva funcionalidad o un *bug fix*), que pueden ser trabajadas en aislamiento y luego fusionadas de nuevo en la línea principal.
4. **Seguridad y Reversibilidad:** Si se introduce un error, el VCS permite a los desarrolladores comparar versiones, identificar dónde se introdujo el error y "revertir" el código a un estado anterior y estable.

7.1.3 El Paradigma de Git: Modelo Distribuido vs. Centralizado

No todos los VCS son iguales. El diseño de Git como un Sistema de Control de Versiones Distribuido (DVCS) es fundamental para entender su poder para gestionar el trabajo concurrente.

- **Modelo Centralizado (p. ej., SVN):** En un sistema como Subversion (SVN), existe un único repositorio central. La operación principal, commit, intenta enviar los

cambios desde el cliente local del desarrollador directamente al servidor central. Si el desarrollador no tiene conexión a la red, o si hay un conflicto en el servidor, la operación falla.[2]

- **Modelo Distribuido (Git):** En Git, la operación `git commit` es un proceso puramente *local*. Captura una instantánea (*snapshot*) del proyecto y la guarda en el **repositorio local** del desarrollador.[2] Esto significa que un desarrollador puede realizar *commits* en un avión, en su casa o en cualquier lugar, sin necesidad de interactuar con otros repositorios. Cada desarrollador posee una copia completa del historial del proyecto en su máquina local.[3]

Este desacoplamiento de la operación de *commit* del servidor central es el habilitador fundamental de todo el trabajo concurrente. Permite a los desarrolladores trabajar de forma verdaderamente asíncrona y en paralelo, creando historiales de cambios completos en aislamiento antes de decidir cuándo y cómo sincronizarse con el repositorio remoto.

Además, el repositorio local actúa como un "amortiguador" (*buffer*) crucial entre las contribuciones del desarrollador y el repositorio central.[2] Esto proporciona una *seguridad psicológica* invaluable: un desarrollador puede experimentar, cometer errores y crear *commits* desordenados en su "sandbox" local sin temor a "romper" el proyecto principal. Esto conduce a una mayor *seguridad del proyecto*, ya que el desarrollador puede (y debe) limpiar y pulir su historial local (usando herramientas como `rebase` o `commit --amend`) *antes* de compartir su trabajo. Este repositorio local actúa como un primer nivel de control de calidad, asegurando que solo el código refinado llegue al repositorio remoto, un pilar fundamental del flujo de trabajo de GitHub.

Sección II: Gestión del Repositorio Local: El Ciclo de Vida del Desarrollo en GIT

7.2.1 Instalación y Configuración Inicial

Antes de poder gestionar el código fuente, Git debe estar instalado. Existen instaladores binarios para los principales sistemas operativos (Windows, macOS, Linux) disponibles en el sitio web de Git, así como métodos de instalación a través de gestores de paquetes como `apt` (para Debian/Ubuntu) o `dnf` (para Fedora).[4]

Una vez instalado, el paso *crítico* inicial es configurar la identidad del usuario. Esta información se incrusta permanentemente en la metadata de cada *commit* para fines de autoría y auditoría:

Bash

```
# Configura el nombre del autor para todos los repositorios en la máquina
$ git config --global user.name "Nombre Apellido"
# Configura el correo electrónico del autor
$ git config --global user.email "email@ejemplo.com"
```

Esta configuración es esencial y debe realizarse antes de hacer el primer *commit* en cualquier proyecto.[5]

7.2.2 El Flujo de Trabajo Fundamental: El Ciclo de Tres Árboles

El flujo de trabajo local en Git gira en torno a tres áreas conceptuales, a veces llamadas los "tres árboles" de Git.[6]

Paso 1: git init y el Directorio de Trabajo (Working Directory)

Para un proyecto nuevo, se crea un repositorio local ejecutando git init en el directorio raíz del proyecto.5 Esta es la primera área: el Directorio de Trabajo, donde los archivos se editan y modifican.

Paso 2: git status y los Estados de los Archivos

El comando git status es el "tablero de control" del desarrollador. Muestra el estado de los archivos en relación con el repositorio.4 Los estados principales son 4:

- `Untracked` (No rastreado): El archivo es nuevo; Git lo ve en el Directorio de Trabajo pero no estaba en el *snapshot* anterior.
- `Tracked` (Rastreado): Git conoce el archivo y está monitoreando sus cambios.
- `Modified` (Modificado): Un archivo rastreado ha sido cambiado en el Directorio de Trabajo.
- `Staged` (Preparado): Un archivo modificado ha sido marcado para ser incluido en el próximo *snapshot*.

Un estado de `nothing to commit, working tree clean` (nada que confirmar, árbol de trabajo limpio) indica que el Directorio de Trabajo está sincronizado con el último *commit*.[4]

Paso 3: git add y el Área de Preparación (Staging Area)

La segunda área es el Área de Preparación (o Staging Area). El comando git add <file> (o git add. para todos los cambios) toma los cambios del Directorio de Trabajo y los añade al Área de Preparación.7

El Área de Preparación no es simplemente una "lista de archivos". Es una herramienta de *artesanía* de *commits*. A diferencia de SVN, donde `add` se usa una vez por archivo, en Git `git add` funciona a nivel de *cambios*.[7] Esto permite a un desarrollador, por ejemplo, tener un archivo con modificaciones para dos *bugs* diferentes. Usando el modo interactivo (`git add -p`), puede "preparar" (stage) solo los trozos de código (*hunks*) relacionados con el primer *bug*, hacer un `commit`, y luego preparar y hacer *commit* del segundo *bug*. Esta capacidad de crear *commits atómicos* (pequeños, lógicamente autocontenidos) es fundamental para la mantenibilidad del código, facilitando las revisiones de código, la reversión de cambios (`git revert`) y la búsqueda de errores (`git bisect`).

Paso 4: git commit y el Repositorio Local (Historial)

La tercera área es el Repositorio Local (el historial de commits). El comando git commit -m "Un mensaje descriptivo" toma la snapshot precisa de los cambios preparados en el Staging Area y la guarda permanentemente en el historial del repositorio local.2

La siguiente tabla resume el flujo de datos entre estas tres áreas:

Tabla 1: La Arquitectura de Tres Árboles de Git y el Flujo de Cambios

Área	Propósito	Comando para Mover Cambios Hacia	Comando para Deshacer Cambios
Directorio de Trabajo	Contiene los archivos actuales que se están editando.	(N/A - Punto de partida)	`git restore <file>` (Descarta cambios)
Área de Preparación (Index)	Prepara el *snapshot* exacto para el próximo *commit*.	`git add <file>` (Prepara cambios)	`git restore --staged <file>` (Des-prepara)
Repositorio Local (HEAD)	Almacena el historial permanente de *commits*.	`git commit` (Guarda el *snapshot*)	`git reset HEAD~1` (Deshace el *commit*)

7.2.3 Gestión de Archivos en el Repositorio

Renombrar y Mover (git mv)

Para renombrar o mover un archivo bajo control de versiones, se utiliza git mv <origen> <destino>.8 Este comando es un atajo conveniente que realiza tres acciones:

1. Renombra el archivo en el sistema de archivos (mv).
2. Elimina el archivo antiguo del seguimiento (git rm <origen>).
3. Añade el archivo nuevo al seguimiento (git add <destino>).

Aunque Git es lo suficientemente inteligente como para detectar renombramientos *a posteriori* durante un git add. [9], usar git mv es una práctica más limpia y explícita.

Ignorar Archivos (.gitignore)

No todos los archivos de un proyecto deben ser rastreados. Los artefactos de compilación (p. ej., archivos .o o .class), dependencias de paquetes (p. ej., node_modules/), archivos de registro (.log) y, de forma crítica, archivos de configuración con secretos (p. ej., .env) deben ser ignorados.

Esto se logra creando un archivo llamado .gitignore en el directorio raíz del repositorio.[10]
Este archivo de texto plano contiene patrones que Git utilizará para ignorar archivos.

Ejemplos de patrones comunes [11]:

- *.log: Ignora cualquier archivo con la extensión.log.
- /node_modules: Ignora la carpeta node_modules en la raíz (la / inicial es importante).
- **/*.tmp: Ignora archivos .tmp en cualquier subdirectorio (** significa "coincidir en cualquier directorio").

Es crucial entender que .gitignore solo previene que archivos *untracked* (no rastreados) sean añadidos. Si un archivo ya ha sido *commiteado* al historial, añadirlo a .gitignore no hará nada. Para dejar de rastrear un archivo que ya está en el repositorio (p. ej., un archivo .env que se subió por error), se debe ejecutar: git rm --cached <file>.[11]

7.2.4 Restauración de Archivos (Deshacer Cambios)

Una de las funciones de seguridad clave de Git es la capacidad de deshacer cambios. Históricamente, esta función estaba sobrecargada en el comando git checkout, lo que causaba una gran confusión, ya que el mismo comando se usaba para cambiar de rama (una operación segura) y para descartar cambios en archivos (una operación destructiva).[13]

Desde la versión 2.23 de Git, se introdujeron comandos más claros (`git restore` y `git switch`) para separar estas responsabilidades.[14] El informe se centrará en la sintaxis moderna de `git restore`.

Caso 1: Descartar cambios en el Directorio de Trabajo (no preparados)

Si un archivo ha sido modificado pero no preparado (git status lo muestra como modified), y se desean descartar esos cambios para volver a la versión del último commit (HEAD):

- Sintaxis Moderna: `git restore <file>`[16]
- (Sintaxis Antigua): `git checkout -- <file>`[15]

Caso 2: "Des-preparar" cambios del Staging Area

Si un archivo ha sido preparado (git add <file>), pero se decide no incluirlo en el próximo commit. Esto no descarta los cambios, sino que los devuelve del Staging Area al Directorio de Trabajo (modified):

- Sintaxis Moderna: `git restore --staged <file>`[17]
- (Sintaxis Antigua): `git reset HEAD <file>`[15]

Caso 3: Restaurar un archivo a una versión de un commit anterior

Si se desea sobrescribir el archivo actual (en el Directorio de Trabajo y el Staging Area) con la versión de un commit específico del historial (identificado por su hash o un tag):

- Sintaxis Moderna: `git restore --source=<commit_hash_o_tag> <file>`[17]
- (Sintaxis Antigua): `git checkout <commit_hash> -- <file>`[20]

Sección III: Estrategias de Ramificación: Gestión del Trabajo Concurrente con GIT

7.3.1 Fundamentos de las Ramas (Branches)

Una rama (`branch`) en Git no es una copia del proyecto; es simplemente un "puntero" ligero y móvil que apunta a un *commit* específico.[21] Cuando se crea una rama, se está creando un nuevo puntero que permite que una línea de desarrollo diverja de otra. Las

ramas son la abstracción central de Git para el aislamiento, permitiendo que el flujo `edit/stage/commit` ocurra en paralelo sin afectar otras líneas de desarrollo.[21]

7.3.2 Comandos de Gestión de Ramas

- **Listar Ramas:**
  - `git branch`: Lista todas las ramas locales.[5]
  - `git branch -a`: Lista todas las ramas, incluyendo las remotas (`origin/...`).
- **Crear una Rama:**
  - `git branch <nombre-rama>`: Crea una nueva rama apuntando al *commit* actual, pero *no* cambia a ella.[5]
- **Cambiar de Rama:**
  - `git switch <nombre-rama>`: Comando moderno y preferido para cambiar a una rama existente.[16]
  - `git checkout <nombre-rama>`: Comando tradicional para cambiar de rama.[5]
- **Crear y Cambiar en un solo paso:**
  - `git switch -c <nueva-rama>`: (Moderno) Crea y cambia a la nueva rama.[16]
  - `git checkout -b <nueva-rama>`: (Tradicional) Crea y cambia a la nueva rama.[5]
- **Eliminar una Rama:**
  - `git branch -d <nombre-rama>`: Elimina la rama local. Es una operación "segura" que fallará si la rama contiene trabajo que no ha sido fusionado.[5]

7.3.3 El "Feature Branch Workflow"

La capacidad *técnica* de Git para crear ramas baratas y rápidas[21] permite una *estrategia organizacional* poderosa para gestionar el trabajo concurrente: el **Feature Branch Workflow** (Flujo de Trabajo de Ramas por Funcionalidad).[24]

La idea central de este flujo de trabajo es que la rama `main` (o `master`) *nunca* debe contener código roto o incompleto. Es la fuente de verdad estable del proyecto. Todo el desarrollo de nuevas funcionalidades o correcciones de errores debe tener lugar en una rama dedicada y descriptiva (p. ej., `feature/autenticacion-oauth` o `fix/issue-1061`).[24]

Esta *encapsulación* [24] crea una "zona de cuarentena" para el nuevo código. Permite a múltiples desarrolladores trabajar en paralelo sin perturbar la base de código principal. Más importante aún, este aislamiento es lo que habilita la revisión de código (a través de *Pull Requests*, ver Sección VI) *antes* de que el nuevo código impacte la rama `main`. Esto es fundamental para mantener la calidad y la estabilidad en entornos de Integración Continua (CI).[24]

7.3.4 Unión de Historias: `git merge`

Una vez que el trabajo en una rama de característica está completo y probado, debe ser integrado de nuevo en la rama principal. El comando para esto es `git merge`.[25]

La operación se realiza *desde* la rama receptora. Por ejemplo, para fusionar `feature/login` en `main`:

```
Bash

$ git switch main

$ git merge feature/login
```

Existen dos tipos principales de fusión que Git puede realizar [25]:

1. **Fast-Forward (Avance Rápido):** Ocurre si la rama `main` no ha tenido nuevos *commits* desde que se creó la rama `feature/login`. La historia es lineal. Git simplemente "avanza rápido" el puntero de `main` para que apunte al mismo *commit* que `feature/login`. No se crea un nuevo *commit* de fusión.
2. **3-Way Merge (Fusión de 3 vías):** Ocurre si *ambas* ramas (`main` y `feature/login`) han avanzado y tienen *commits* únicos desde su ancestro común. La historia ha divergido. Git debe crear un nuevo "*commit* de fusión" (*merge commit*) para unir las dos historias. Este *commit* especial tiene *dos* padres: la punta de `main` y la punta de `feature/login`.[25]

7.3.5 Resolución de Conflictos de Fusión

Los conflictos de fusión son una consecuencia natural e inevitable del trabajo concurrente.[25] Ocurren *exclusivamente* durante una fusión de 3 vías [25], y solo si dos desarrolladores modificaron la *misma parte del mismo archivo* en ambas ramas.[25]

Identificación del Conflicto:

Cuando esto sucede, Git detiene el proceso de fusión.25

1. `git status` mostrará los archivos problemáticos bajo el encabezado `Unmerged paths` (rutas no fusionadas).[25]
2. Git edita los archivos en conflicto e inserta marcadores de conflicto para mostrar ambas versiones del código [25]:

Aquí está el código sin conflicto.

<<<<<<< HEAD

Este es el cambio que existe en 'main' (tu rama actual o HEAD).

=======

Este es el cambio que proviene de 'feature/login' (la rama que fusionas).

>>>>>>> feature/login

Aquí hay más código sin conflicto.

3.
4.

Proceso de Resolución:

El proceso de resolución de conflictos de Git reutiliza elegantemente el flujo de trabajo fundamental (edit/stage/commit), lo que reduce la carga cognitiva en un momento estresante.25

1. **Editar:** El desarrollador abre el archivo(s) en conflicto en un editor de texto. Debe eliminar manualmente todos los marcadores (`<<<<<<<`, `=======`, `>>>>>>>`) y editar el código para que contenga la versión final deseada (que puede ser una, la otra, o una combinación de ambas).
2. **Preparar (Stage):** Una vez que el archivo está corregido, el desarrollador ejecuta `git add <archivo-resuelto>`. Esto *no* añade un archivo nuevo; le dice a Git: "El conflicto en este archivo ha sido resuelto manualmente, y esta es la versión que debe usarse".
3. **Confirmar (Commit):** Después de que *todos* los archivos en conflicto han sido añadidos (stage), el desarrollador ejecuta `git commit`. Git detectará que la fusión está en progreso y creará el *commit* de fusión para finalizar la operación.

Si un conflicto de fusión es demasiado grande o confuso, la fusión se puede cancelar y revertir al estado anterior de forma segura usando `git merge --abort`.[30]

Sección IV: Sincronización de Repositorios Locales y Remotos (GIT y GitHub)

7.4.1 Configuración de Repositorios Remotos

Un repositorio remoto (como los alojados en GitHub) es una versión de un proyecto alojada en Internet o en una red. Sirve como el punto central de sincronización para el equipo.

Caso 1: Clonación (git clone)

La forma más fácil de conectarse a un remoto es clonándolo. El comando git clone <URL> realiza varias acciones 5:

1. Descarga el historial completo del proyecto desde la URL.
2. Crea un nuevo directorio con el nombre del proyecto.
3. Inicializa un repositorio local de Git dentro de él.
4. Configura automáticamente un "remoto" llamado `origin` que apunta a la URL de origen.[31]

Caso 2: Conexión de un Repositorio Local Existente (git remote add)

Si un proyecto se inició localmente (con git init), se debe conectar manualmente a un repositorio remoto (p. ej., un repositorio vacío creado en GitHub):

```
Bash

# Añade una nueva conexión remota llamada 'origin' que apunta a la URL de GitHub

$ git remote add origin https://github.com/usuario/proyecto.git

3
```

Se puede verificar la conexión con `git remote -v`.[5]

7.4.2 Envío de Cambios al Remoto (`git push`)

Para transferir *commits* desde el repositorio local al repositorio remoto, se usa el comando `git push`.[33]

```
Bash

# Envía los commits de la rama local 'main' a la rama 'main' del remoto 'origin'

$ git push origin main
```

[34]

El Error "Non-Fast-Forward"

A veces, git push fallará con un error: non-fast-forward updates were rejected.[34] Este no es un fallo, sino el mecanismo de seguridad central de Git para la colaboración.

Significa que, mientras el desarrollador trabajaba localmente, otro desarrollador hizo `push` de sus propios *commits* al repositorio remoto. La historia del remoto ha divergido. Git rechaza el `push` porque aceptarlo sobrescribiría (y borraría) los *commits* del otro desarrollador.

El error *fuerza* al desarrollador a:

1. Obtener primero los cambios remotos (`git fetch`).
2. Integrarlos en su trabajo local (con `git merge` o `git rebase`), resolviendo cualquier conflicto en su propia máquina.
3. Intentar `git push` de nuevo, esta vez con la historia unificada.

7.4.3 Obtención de Cambios Remotos: `fetch` vs. `pull`

Esta es una de las distinciones más críticas en la sincronización remota.

- **`git fetch <remote-name>`:**
  - **Acción:** *Solo descarga* los nuevos datos (commits, ramas, tags) del repositorio remoto.[31]
  - **Efecto Local:** Actualiza las *ramas de seguimiento remoto* (p. ej., `origin/main`). **NO** toca el Directorio de Trabajo ni las ramas locales (p. ej., `main`). Los cambios descargados están "en cuarentena" en `origin/main`, listos para ser inspeccionados.
- **`git pull <remote-name> <branch-name>`:**
  - **Acción:** Es un comando compuesto que ejecuta `git fetch` e *inmediatamente después* ejecuta `git merge`.[35]

- **Efecto Local:** Descarga los datos *y* trata de fusionarlos automáticamente en la rama local actual, modificando el Directorio de Trabajo.[31]

El comando `git pull` puede ser arriesgado porque *oculta* información. El desarrollador no ve qué cambios están llegando antes de que se fusionen, lo que puede resultar en conflictos de fusión inmediatos e inesperados.[36]

Un flujo de trabajo más seguro y profesional (el "Ciclo `fetch-inspect-merge`") es evitar `git pull` en favor de:

1. `git fetch origin`: Descarga los cambios de forma segura a `origin/main`.[36]
2. `git log main..origin/main`: *Inspecciona* los cambios que han llegado (revisa el historial entre tu `main` local y el `main` remoto).
3. `git merge origin/main`: Integra *explícitamente* los cambios en la rama local, con pleno conocimiento de lo que se está fusionando.

Tabla 2: Comparativa de Sincronización: `git fetch` vs. `git pull`

Comando	Acción Principal	Modifica Ramas de Seguimiento (ej. origin/main)?	Modifica Rama Local (ej. main)?	Potencial de Conflicto Inmediato?
`git fetch`	Solo descarga	**Sí**	**No**	**No**
`git pull`	Descarga + Fusiona (`fetch` + `merge`)	**Sí**	**Sí**	**Sí**

Sección V: Herramientas Avanzadas de GIT para un Historial Limpio y Flexible

7.5.1 `git stash`: Aislamiento Temporal de Cambios

A menudo, un desarrollador está a mitad de un trabajo (con un "directorio de trabajo sucio", es decir, cambios *staged* o *unstaged*) cuando surge una tarea urgente, como arreglar un *bug* crítico en la rama `main`.[18] El desarrollador no puede hacer `commit` de un trabajo a medias, pero tampoco puede cambiar de rama (`git switch main`), ya que Git intentará llevarse los cambios o fallará.[38]

La solución es `git stash`.

- `git stash` (o `git stash push`) toma todos los cambios en archivos *tracked* (staged y unstaged) y los guarda en una "pila" (stack) local, revirtiendo el Directorio de Trabajo a un estado limpio (idéntico al *commit* `HEAD`).[18]
- (Nota: Por defecto, `git stash` *no* guarda archivos `untracked`[39]).

El desarrollador ahora puede cambiar de rama, arreglar el *bug*, y luego volver a su rama original y recuperar su trabajo:

- `git stash apply`: Reaplica los cambios guardados, pero los *mantiene* en la pila (para reutilizarlos).[39]
- `git stash pop`: Reaplica los cambios y los *elimina* de la pila.[39]

7.5.2 `git rebase`: Reescritura del Historial

`git rebase` es una alternativa a `git merge` para integrar cambios.[27] En lugar de crear un *commit* de fusión de 3 vías que une dos historias divergentes, `git rebase` toma los *commits* de la rama de característica y los *vuelve a aplicar*, uno por uno, sobre la punta de la rama `main`.[27]

El resultado es un historial de proyecto *perfectamente lineal* y limpio, como si todo el trabajo se hubiera hecho en serie, eliminando los *commits* de fusión "ruidosos".[27]

Esta elección entre `merge` y `rebase` no es solo técnica, sino filosófica, y representa una compensación entre la verdad histórica y la claridad narrativa:

Tabla 3: Comparativa Conceptual: `git merge` vs. `git rebase`

Criterio	git merge (Fusión)	git rebase (Reorganización)

Filosofía	**Verdad Histórica**[27]	**Claridad Narrativa**[27]
Mecanismo	Une historias divergentes con un *commit* de fusión de 2 padres.[27]	*Re-escribe* la historia aplicando *commits* antiguos sobre una nueva base.[27]
Historial Resultante	No lineal, gráfico complejo. Registra *exactamente* cuándo ocurrieron las integraciones.[27]	Lineal, limpio y fácil de leer. La historia parece secuencial.[27]
Naturaleza	**No destructivo**. Los *commits* existentes nunca cambian.[27]	**Destructivo** (en el sentido de que re-escribe). Crea *commits* nuevos con *hashes* diferentes.[27]
Ventaja	Preserva el contexto histórico completo; es auditable.	El historial es limpio; fácil para `git log` y `git bisect`.
"Regla de Oro"	Seguro de usar en cualquier rama.	**NUNCA usar en una rama pública/compartida** (ej. `main`).[27]

La "Regla de Oro" del Rebase es la directriz más importante: `git rebase` solo debe usarse para limpiar ramas de características *locales* y *privadas antes* de compartirlas o fusionarlas, para mantener un historial limpio.

7.5.3 `git tag`: Marcado de Versiones

A medida que un proyecto madura, es vital marcar puntos específicos en el historial como lanzamientos de versiones (p. ej., `v1.0.1`, `v2.0-beta`). Para esto se usan los *tags* (etiquetas).[40]

Un *tag* es un punto de referencia que *no cambia* y que apunta a un *commit* específico. A diferencia de una rama (que es un puntero móvil), un *tag*, una vez creado, se queda fijo en ese *commit*.[40]

Se prefieren los *tags anotados* (que almacenan metadata como autor, fecha y un mensaje) sobre los *tags ligeros* (que son solo un puntero).

- **Crear un Tag Anotado:** `git tag -a v1.4 -m "Lanzamiento de la versión 1.4"`.[40]

Un error común es asumir que `git push` envía los *tags* al remoto. Esto no es así.[41] Los *tags* deben enviarse explícitamente:

- **Enviar un solo tag:** `git push origin v1.4`[34]
- **Enviar todos los tags locales:** `git push origin --tags`[41]

Sección VI: Gestión del Trabajo Colectivo en GitHub

Mientras que Git es la herramienta de línea de comandos para el control de versiones, GitHub es la plataforma web que proporciona una interfaz gráfica y herramientas de colaboración *sobre* Git.

7.6.1 Documentación del Proyecto con Markdown (`README.md`)

El archivo `README.md` es la "portada" del repositorio en GitHub.[42] Es el primer archivo que un visitante ve y es fundamental para comunicar el propósito del proyecto. Un buen `README.md` incluye[42]:

- Qué hace el proyecto.
- Por qué es útil.
- Cómo pueden los usuarios empezar a usarlo.

Estos archivos se escriben en **Markdown** (específicamente, *GitHub Flavored Markdown*), un lenguaje de marcado ligero.[43] La sintaxis básica incluye:

- **Encabezados:** `# Encabezado 1`, `## Encabezado 2`, `### Encabezado 3`
- **Estilo de Texto:** `**Negrita**`, `_Itálica_` (o `*Itálica*`), `~~Tachado~~`
- **Listas:** `* Elemento 1`, `- Elemento 2`
- **Bloques de Código:** Usando tres comillas invertidas:
- JavaScript

```
function hello() { console.log("Hola Mundo"); }
```

-
-

7.6.2 El Flujo de GitHub: Un Modelo para la Colaboración Segura

El **GitHub Flow** es un flujo de trabajo ligero que une las capacidades de Git (ramas) con la herramienta de colaboración de GitHub (Pull Requests).[44]

El flujo consta de seis pasos:

1. **Crear una Rama (Branch):** Todo el trabajo comienza creando una rama descriptiva (ej. `fix/typo-in-docs`) a partir de `main`.[44]
2. **Realizar Cambios (Commits):** El desarrollador trabaja localmente, usando el ciclo `edit`/`add`/`commit` (Sección 7.2.2). Se hace `push` de esta rama al remoto (`origin`).
3. **Crear una *Pull Request* (PR):** Cuando el trabajo está listo para revisión (o incluso si es solo para pedir ayuda), el desarrollador abre un *Pull Request* (PR) en la interfaz de GitHub.[44]
4. **Revisión y Discusión (Review):** Este es el núcleo de la colaboración. Los compañeros revisan los cambios, comentan en líneas de código específicas, solicitan cambios o aprueban.[44]
5. **Unión (Merge):** Una vez que el PR es aprobado (y pasa cualquier prueba de CI), un mantenedor fusiona la rama de característica en la rama `main`.[44]
6. **Limpieza (Delete Branch):** La rama de característica, ahora fusionada, se elimina.[44]

El paso 6, la limpieza, es un paso de higiene crítico a menudo omitido. La rama de característica ha cumplido su propósito como *vehículo de entrega*. El código ya está en `main` y la discusión está preservada en el PR (ahora cerrado).[46] Mantener ramas "muertas" (stale branches) crea *ruido técnico* (contaminando la salida de `git branch -a`) y *ambigüedad* (¿es trabajo abandonado o activo?). Eliminarla es una señal clara de que el trabajo está 100% *hecho e integrado*.

7.6.3 Administración de *Pull Requests* (PRs)

Es fundamental entender que la herramienta `git` (la línea de comandos) no tiene concepto de "Pull Request". Un PR es una *construcción de plataforma* inventada por GitHub (y adoptada por otros) que actúa como una *capa social* superpuesta a las primitivas técnicas de Git (ramas y fusión).

Un PR es la formalización de la revisión de código. Es el mecanismo que toma una unidad técnica de trabajo (una rama Git) y la envuelve en una interfaz de usuario diseñada para la *discusión asíncrona* del equipo.[47] Es el pegamento que une el desarrollo distribuido de Git con la colaboración centralizada del equipo en GitHub.

Gestión del Flujo del PR:

- **Borradores (Draft PRs):** Si un desarrollador sube una rama pero el trabajo está en progreso (WIP), puede crear un *Draft Pull Request*. Esto señaliza al equipo que el PR no está listo para una revisión formal, previene que se fusione accidentalmente y no notifica automáticamente a los revisores.[47]
- **El Proceso de Revisión:** Los colaboradores pueden aprobar, solicitar cambios o simplemente comentar.[46] Si se solicitan cambios, el autor del PR *no* crea un nuevo PR. Simplemente añade nuevos *commits* a la *misma rama* local y hace `push` de nuevo. El PR en GitHub se actualizará automáticamente con los nuevos *commits*.[47]
- **Resolución de Conflictos en PRs:** Si la rama `main` ha cambiado mientras el PR estaba abierto, pueden surgir conflictos. Estos pueden resolverse localmente (el método preferido: `git switch mi-rama`, `git merge origin/main`, resolver conflictos localmente, y hacer `push`) o, si son simples, directamente en la interfaz de GitHub.[46]

Sección VII: Conclusiones

La gestión del código fuente con Git y GitHub es un ecosistema de dos partes: una herramienta técnica y una plataforma social.

Git es más que una simple herramienta de "guardado"; es un modelo para el desarrollo distribuido y concurrente. Su arquitectura, que separa el trabajo local del remoto y el Directorio de Trabajo del Área de Preparación, proporciona una robusta red de seguridad. Esta arquitectura permite a los desarrolladores *crear* historiales limpios y atómicos en un entorno de "sandbox" local, garantizando la seguridad psicológica para la experimentación y la seguridad del proyecto para la rama principal.

Los comandos de Git, como `fetch` (en lugar de `pull`) y el error `non-fast-forward`, están diseñados explícitamente como mecanismos de seguridad para forzar una integración local controlada antes de impactar el trabajo del equipo.

GitHub se construye sobre esta base técnica y añade la capa social esencial. El *Pull Request* es el nexo de la colaboración moderna: un foro de discusión y revisión de código construido alrededor de una unidad de trabajo de Git (una rama). El *GitHub Flow* formaliza

este proceso, asegurando que el aislamiento proporcionado por las ramas se utilice para garantizar la calidad y la estabilidad antes de la integración.

El dominio de este ciclo de vida completo —desde la "artesanía" de un *commit* atómico local, pasando por la sincronización segura con fetch y push, hasta la revisión colaborativa a través de un *Pull Request*— es fundamental para el desarrollo de software concurrente, seguro y mantenible.