

PROMETEO

Unidad 2: Control de versiones con Git y Github

Git es mucho más que una herramienta técnica; es la base del trabajo colaborativo en el desarrollo moderno. Antes de su aparición, los equipos dependían de sistemas centralizados como CVS o Subversion (SVN), donde todo el historial del proyecto se almacenaba en un único servidor.

Sesión 5: Conceptos básicos de Git: repositorio, commit, historial

Cómo Git revolucionó la gestión del código

Git es mucho más que una herramienta técnica; es la base del trabajo colaborativo en el desarrollo moderno. Antes de su aparición, los equipos dependían de sistemas centralizados como CVS o Subversion (SVN), donde todo el historial del proyecto se almacenaba en un único servidor. Si ese servidor fallaba, se perdía todo el progreso. Git, en cambio, introdujo un modelo distribuido: cada programador tiene una copia completa del proyecto, incluyendo todos los cambios históricos. Esto garantiza seguridad, autonomía y flexibilidad.

El objetivo de Git es permitir que los desarrolladores trabajen simultáneamente sin pisarse los cambios. Cada uno puede crear su propia versión del proyecto, experimentar con nuevas funcionalidades y, cuando todo esté listo, integrar su trabajo con el del resto. De este modo, se combina la libertad individual con el control colectivo.

El repositorio: el corazón del proyecto

Un repositorio (o repo) es el espacio donde "vive" un proyecto. Contiene tanto los archivos actuales como todo el historial de versiones. Puede ser local, almacenado en el ordenador del desarrollador, o remoto, alojado en servicios como GitHub, GitLab o Bitbucket. El repositorio local permite trabajar sin conexión y guardar cambios de forma segura, mientras que el remoto actúa como punto de sincronización con el resto del equipo.

Los commits: fotografías del progreso

Cada vez que realizas un cambio significativo —añadir una nueva función, corregir un error o actualizar documentación— puedes confirmarlo mediante un commit. Este commit captura el estado exacto del código en ese momento, junto con un mensaje descriptivo. Podríamos compararlo con el "guardar partida" de un videojuego: si algo falla después, puedes volver a un punto anterior del proyecto.

Un buen mensaje de commit es breve y claro, e indica la acción realizada y su propósito, por ejemplo:

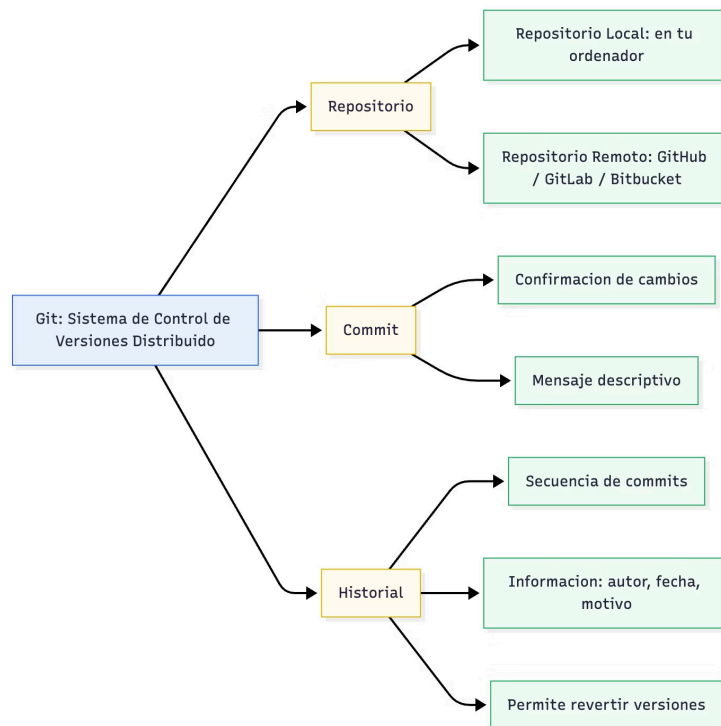
- "Corrige validación de formulario de registro"
- "Añade test unitario para función de login"

El historial: la memoria colectiva del código

El historial de commits (o log) es la cronología completa de todas las modificaciones hechas en el proyecto. Contiene quién hizo el cambio, cuándo y por qué. Esta trazabilidad es esencial en entornos profesionales porque permite auditar el código, revisar decisiones y resolver conflictos.

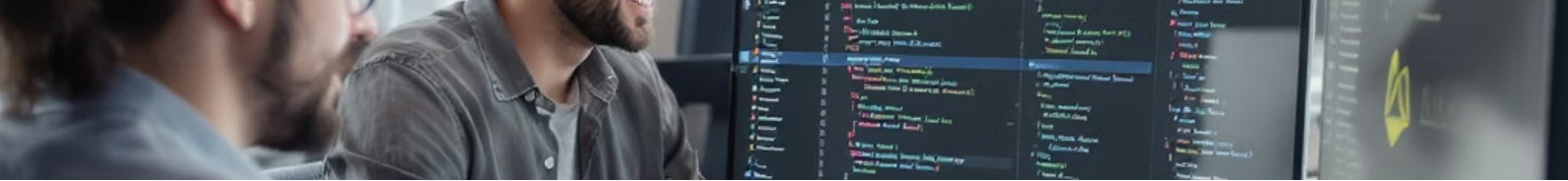
Por ejemplo, si una actualización provoca un error en producción, se puede usar el historial para identificar exactamente qué commit lo introdujo y revertirlo. Git, por tanto, no solo registra líneas de código: registra decisiones técnicas. Por eso se ha convertido en una herramienta imprescindible no solo para programadores, sino también para diseñadores, analistas de datos y gestores de proyectos que necesitan controlar versiones de documentos o scripts.

Esquema Visual: Estructura conceptual de Git



Explicación del esquema: El nodo principal representa a Git como un sistema de control de versiones distribuido. Desde él salen tres pilares esenciales: Repositorio, Commit e Historial. El repositorio se divide en local (trabajo personal en el ordenador) y remoto (colaboración online). El commit es la acción central de confirmación que guarda los cambios con una descripción. El historial agrupa todos los commits en orden temporal, ofreciendo trazabilidad y control total sobre el desarrollo.

Este esquema resume cómo Git conecta el trabajo individual con la colaboración global: cada commit forma parte del historial, y cada historial se sincroniza entre los repositorios locales y remotos.



Caso de Estudio: Visual Studio Code y la colaboración a escala global

Contexto

Visual Studio Code (VS Code), desarrollado por Microsoft, es uno de los editores de código más populares del mundo, con millones de usuarios activos. Lo interesante es que su desarrollo es abierto: cualquiera puede contribuir al proyecto a través de su repositorio público en GitHub. Coordinar miles de contribuciones semanales de programadores distribuidos por todo el mundo sería imposible sin Git.

Estrategia

El equipo principal de VS Code gestiona el proyecto utilizando Git y GitHub de manera ejemplar:

- **Repositorio principal:** todos los archivos del código fuente y su historial están alojados en un repositorio público.
- **Commits controlados:** cada cambio, desde una pequeña corrección hasta una nueva función, se envía mediante un commit acompañado de un mensaje claro y estructurado.
- **Pull Requests:** los desarrolladores externos proponen sus modificaciones creando un "pull request" (solicitud de fusión). Antes de aprobarlo, los revisores analizan los commits, los comentarios y los resultados de las pruebas automáticas.
- **Historial transparente:** cualquier persona puede consultar el historial del proyecto, ver quién introdujo una función o cuándo se corrigió un error.

Resultado

Gracias a este flujo de trabajo basado en Git, VS Code mantiene:

- **Coherencia:** cada contribución está documentada y validada.
- **Escalabilidad:** se pueden integrar miles de aportaciones sin perder control.
- **Transparencia:** el historial abierto refuerza la confianza de la comunidad.

Este modelo demuestra cómo Git permite sostener proyectos de software a escala planetaria sin un único servidor central. La trazabilidad de los commits es, literalmente, el hilo conductor que mantiene unido el trabajo de miles de desarrolladores anónimos.

Herramientas y Consejos Profesionales

Mensajes de commit claros y consistentes

Utiliza el formato "verbo + objetivo" (por ejemplo, "Añade validación de campos en formulario"). Algunos equipos siguen convenciones como Conventional Commits, que facilitan el mantenimiento automático del changelog y la integración continua.

Comando git status como brújula diaria

Antes de hacer un commit o un push, ejecuta git status para comprobar qué archivos han cambiado y cuál es su estado (modificados, nuevos o eliminados). Es la forma más sencilla de evitar errores.

Archivo .gitignore para mantener tu repositorio limpio

Este archivo indica a Git qué elementos debe ignorar: archivos temporales, logs, dependencias o configuraciones locales. En proyectos de Node.js, por ejemplo, se suele incluir node_modules/ para evitar subir miles de archivos innecesarios.

1

Visualiza sin miedo

Si prefieres evitar la línea de comandos, usa GitHub Desktop, Sourcetree o el panel de control integrado de Visual Studio Code. Estas herramientas permiten realizar commits, comparar versiones y sincronizar con un clic.

2

Usa ramas incluso para pequeñas pruebas

Aunque esta sesión se centra en repositorios e historial, ya puedes adoptar el hábito de trabajar en ramas separadas. Así mantienes tu historial principal (main o master) siempre estable.

3

Aplica revisiones de código con GitHub o GitLab

Cada commit puede asociarse a un pull request, donde tus compañeros revisan tu código antes de fusionarlo. Esta práctica eleva la calidad y reduce errores en entornos profesionales.

4

Automatiza con hooks

Git permite ejecutar scripts automáticos (llamados hooks) antes o después de determinadas acciones. Por ejemplo, puedes configurar un pre-commit hook que ejecute pruebas automáticas antes de aceptar un cambio.

Mitos y Realidades del Uso de Git

✗ **Mito:** "Git solo es útil para proyectos grandes."

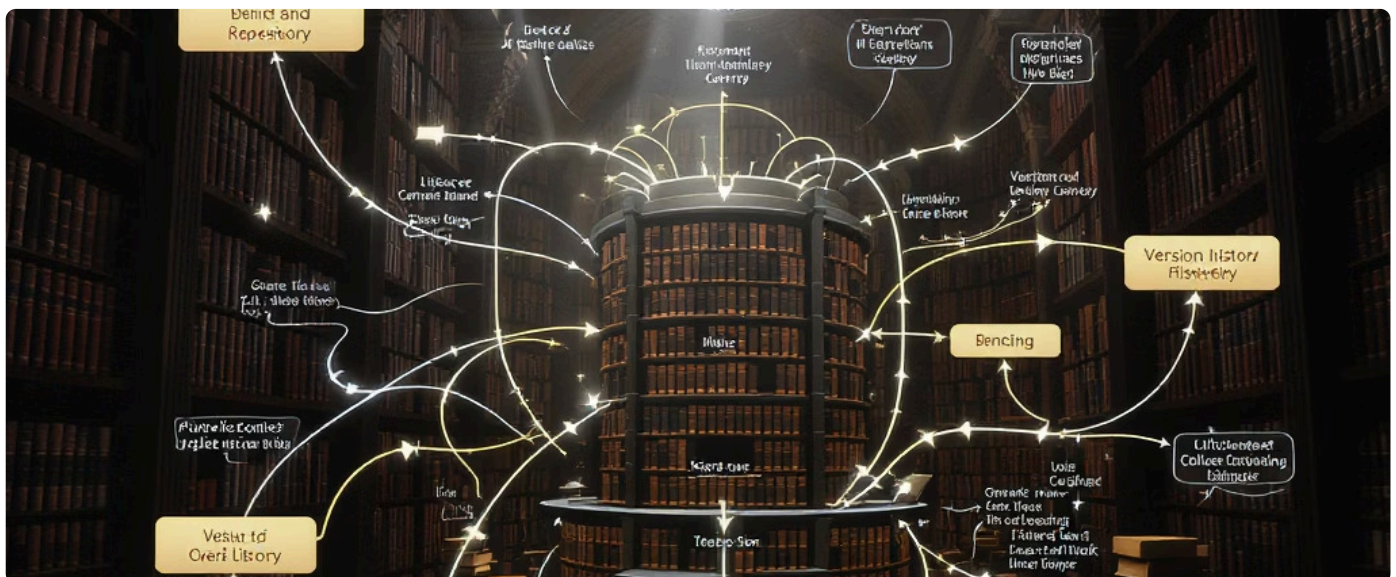
→ **FALSO.** Git aporta valor incluso en proyectos personales o educativos. Permite hacer retrocesos, comparar versiones y mantener un control riguroso del progreso. Usar Git desde el principio evita perder trabajo y fomenta hábitos profesionales.

✗ **Mito:** "Si uso GitHub, ya estoy usando Git automáticamente."

→ **FALSO.** GitHub es una plataforma que utiliza Git, pero no es Git en sí. Puedes usar Git completamente sin conexión, sin subir nada a GitHub. GitHub solo actúa como repositorio remoto y entorno de colaboración visual.

📄 Resumen Final (para examen)

- **Git** = sistema de control de versiones distribuido.
- **Repositorio** = espacio donde se almacena el proyecto y su historial.
- **Commit** = confirmación de cambios con mensaje descriptivo.
- **Historial** = registro cronológico de todos los commits.
- **GitHub / GitLab** = plataformas para almacenar y colaborar de forma remota.
- **Ventaja clave:** trazabilidad total y trabajo colaborativo sin depender de un servidor central.



Sesión 6: Comandos esenciales: git init, git add, git commit, git push, git pull

Fundamentos: El flujo de trabajo básico con Git

Para dominar Git no basta con entender qué es un repositorio o qué representa un commit: el verdadero dominio empieza cuando sabes cómo se conectan los comandos en el flujo de trabajo diario. Git se basa en una secuencia lógica de pasos que reflejan el proceso natural de desarrollo: crear un proyecto, registrar los cambios, confirmarlos y compartirlos con los demás. Esta secuencia, aunque sencilla, es el eje sobre el que gira toda la colaboración moderna en programación.

El flujo esencial de Git

Inicializar el repositorio (git init)

Este comando crea el entorno de control de versiones dentro de una carpeta. Es como "activar" Git en un proyecto. Una vez ejecutado, Git empezará a rastrear los cambios en los archivos de esa carpeta.

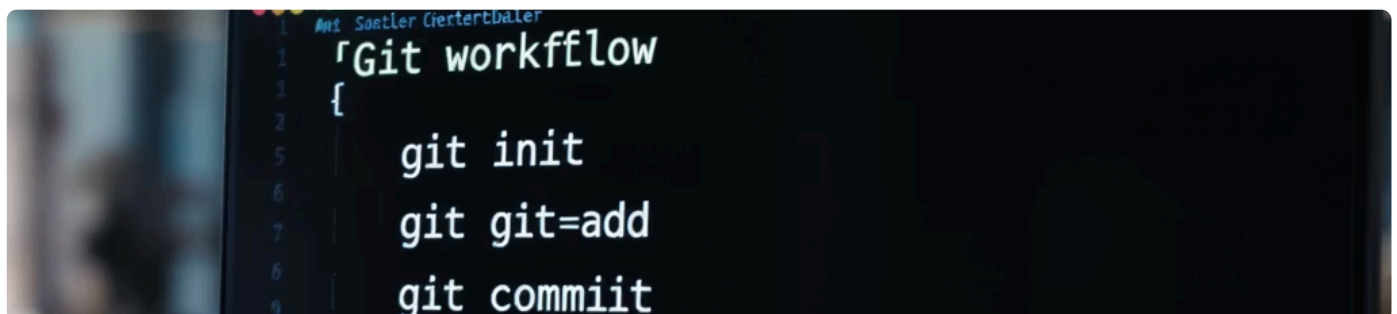
- Si el proyecto aún no tiene control de versiones, git init crea la carpeta .git, que contiene toda la información del historial y configuración.
- Si el proyecto proviene de un repositorio remoto, este paso se sustituye por git clone.

Preparar los archivos (git add)

Después de modificar archivos, Git no los guarda automáticamente. Antes hay que añadirlos al área de preparación (staging area). Este paso permite elegir qué cambios se incluirán en el siguiente commit.

- git add archivo.txt añade un archivo individual.
- git add . añade todos los archivos modificados en el proyecto.

Esta etapa evita incluir cambios accidentales y da control sobre lo que se registrará en el historial.



1

Confirmar los cambios (git commit)

Cuando estás satisfecho con las modificaciones, confirmas el estado actual del código mediante un commit.

- Cada commit es una "fotografía" exacta del proyecto en ese momento.
- Se acompaña de un mensaje que explica qué se cambió y por qué.

Ejemplo:

```
git commit -m "Corrige bug en formulario de registro"
```

Este paso es la base del versionado, ya que genera un punto al que se puede volver si algo sale mal.

2

Subir los cambios al repositorio remoto (git push)

Si trabajas en equipo o usas GitHub/GitLab, tus commits locales deben sincronizarse con el repositorio remoto.

- `git push origin main` envía los commits de la rama principal al servidor remoto llamado origin.
- Este comando actualiza la versión compartida del proyecto y la hace visible para los demás.

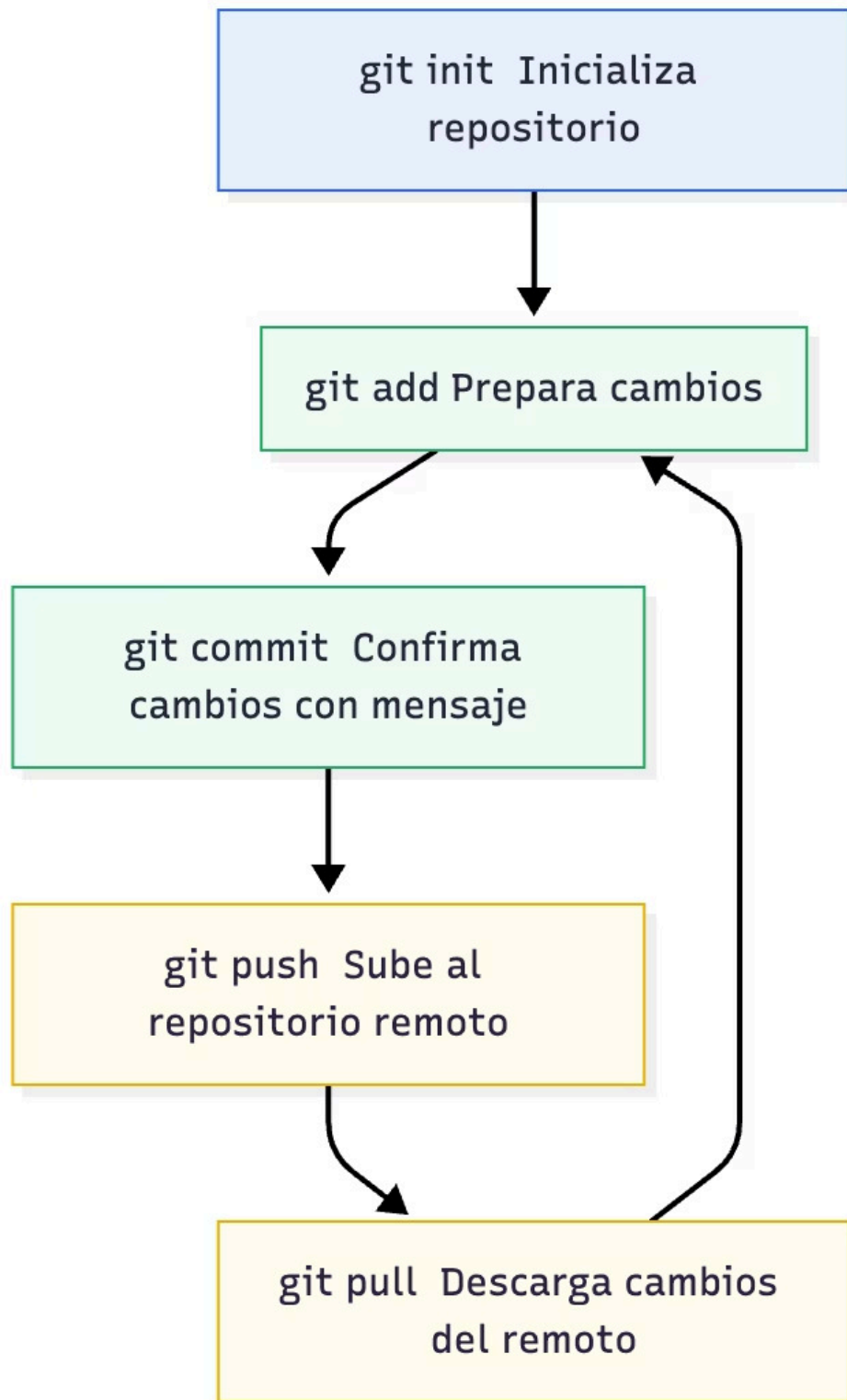
3

Traer actualizaciones (git pull)

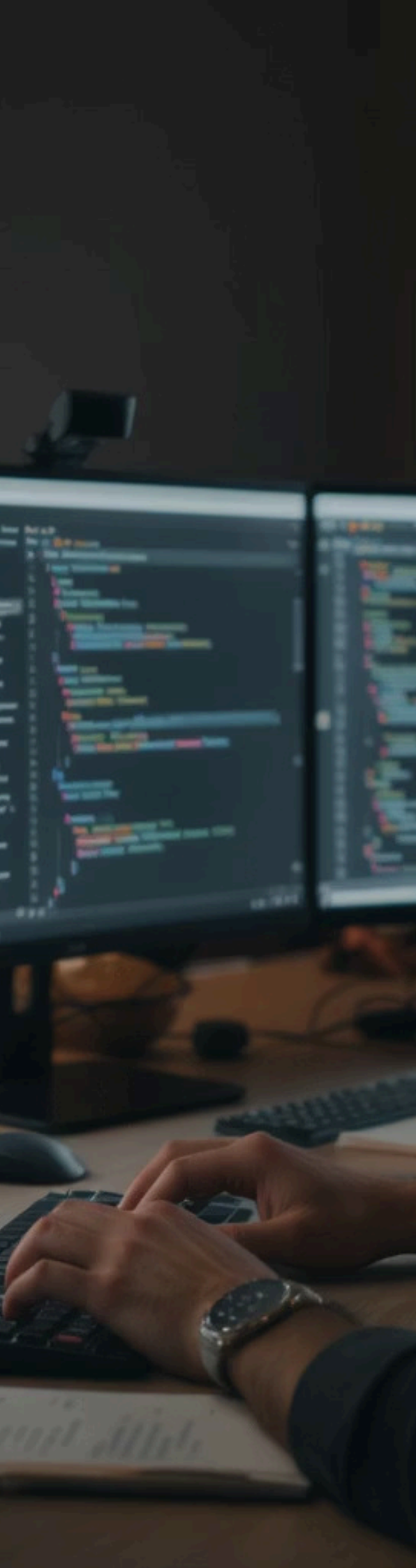
Del mismo modo, para mantener tu repositorio local al día, necesitas traer los cambios que otros compañeros hayan subido.

- `git pull origin main` descarga y fusiona las actualizaciones del remoto.
- Si hay conflictos (dos personas modificaron la misma línea de código), Git pedirá resolverlos manualmente.
- Este ciclo —init → add → commit → push → pull— es la base de todo flujo de trabajo en Git, tanto en proyectos personales como en grandes equipos distribuidos.

Esquema Visual: Flujo de trabajo fundamental de Git



Explicación del esquema: El diagrama representa el ciclo continuo de trabajo con Git. Todo empieza con `git init`, que crea la estructura del repositorio. A partir de ahí, los archivos pasan al área de preparación con `git add`. Con `git commit` los cambios se registran en el historial. Luego, `git push` sincroniza el trabajo con el servidor remoto (por ejemplo, GitHub). Finalmente, `git pull` actualiza el repositorio local con los cambios de otros. El proceso vuelve a empezar cada vez que se modifican archivos, creando una dinámica iterativa de mejora continua.



Caso de Estudio: Sincronización colaborativa en una empresa web

Contexto

Imagina una empresa llamada PixelForge, especializada en desarrollo web. Su equipo de seis programadores trabaja en distintas ciudades. Todos colaboran en un mismo proyecto alojado en GitHub: la web de un nuevo cliente corporativo. Antes de adoptar Git, el equipo intercambiaba archivos comprimidos por correo electrónico, lo que provocaba caos: se sobrescribían versiones, se perdían cambios y nadie sabía cuál era la versión más reciente.

Estrategia con Git

El equipo decidió profesionalizar su flujo de trabajo adoptando Git y el modelo distribuido. Así estructuraron su proceso:

Inicialización y clonación:

El jefe de proyecto crea el repositorio remoto en GitHub. Luego cada desarrollador ejecuta:

```
git clone https://github.com/pixelforge/cliente-  
corporativo.git
```

Así todos obtienen una copia completa del proyecto y su historial.

Desarrollo local:

Cada miembro trabaja en su parte del código (por ejemplo, el módulo de contacto). Después de modificar los archivos, ejecuta:

```
git add .  
git commit -m "Crea formulario de contacto con validación"
```

Sincronización:

Para compartir su trabajo con el equipo:

```
git push origin main
```

Los demás actualizan su copia local ejecutando:

```
git pull origin main
```

Esto garantiza que todos trabajan siempre con la versión más reciente.

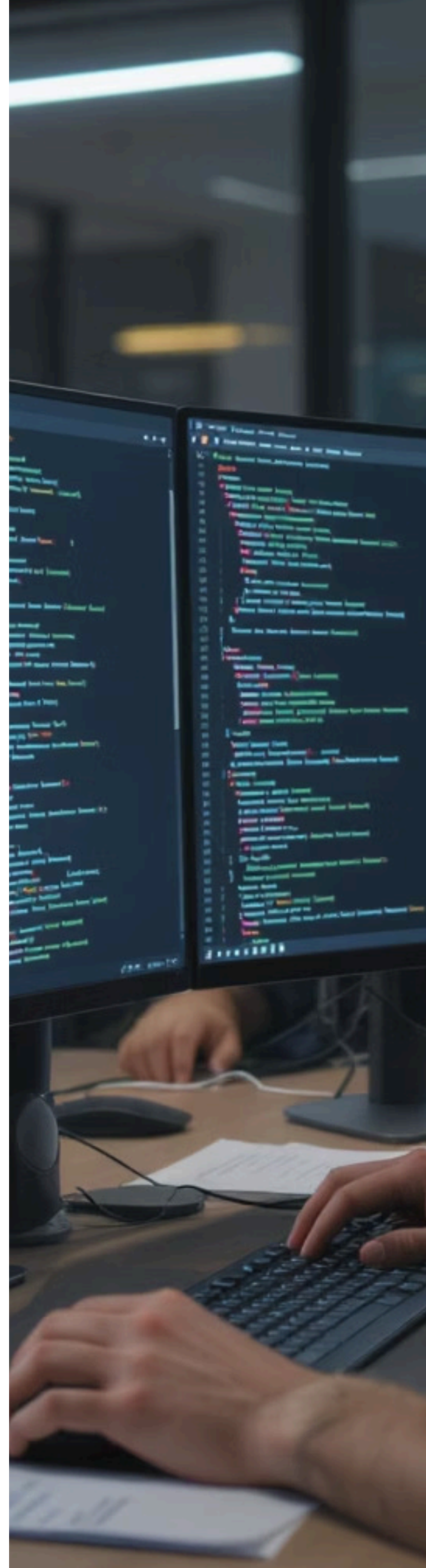
Gestión de conflictos:

Si dos personas editan la misma línea, Git detecta el conflicto y pide resolverlo manualmente. La política de PixelForge exige que el desarrollador afectado revise el conflicto y confirme el resultado con un nuevo commit.

Resultado

- **Productividad duplicada:** se eliminan los tiempos muertos por intercambios manuales.
- **Trazabilidad total:** cada cambio queda documentado con autor, fecha y motivo.
- **Seguridad:** el repositorio remoto actúa como copia de respaldo del trabajo del equipo.

Git se convierte así en el eje central del flujo de trabajo de PixelForge, permitiendo desarrollar en paralelo sin perder coherencia ni control.



Herramientas y Consejos Profesionales

1. Usa git status como punto de control constante

Antes de hacer nada, ejecuta git status. Te muestra los archivos modificados, los pendientes de agregar y el estado actual de tu rama. Es el mejor aliado para evitar cometer errores.

2. Crea commits pequeños y descriptivos

Un commit no debería incluir decenas de cambios distintos. Divide las tareas en pasos lógicos: un commit por función, corrección o mejora.

Ejemplo:

✓ "Agrega validación de campos vacíos"

✗ "Actualiza varios archivos"

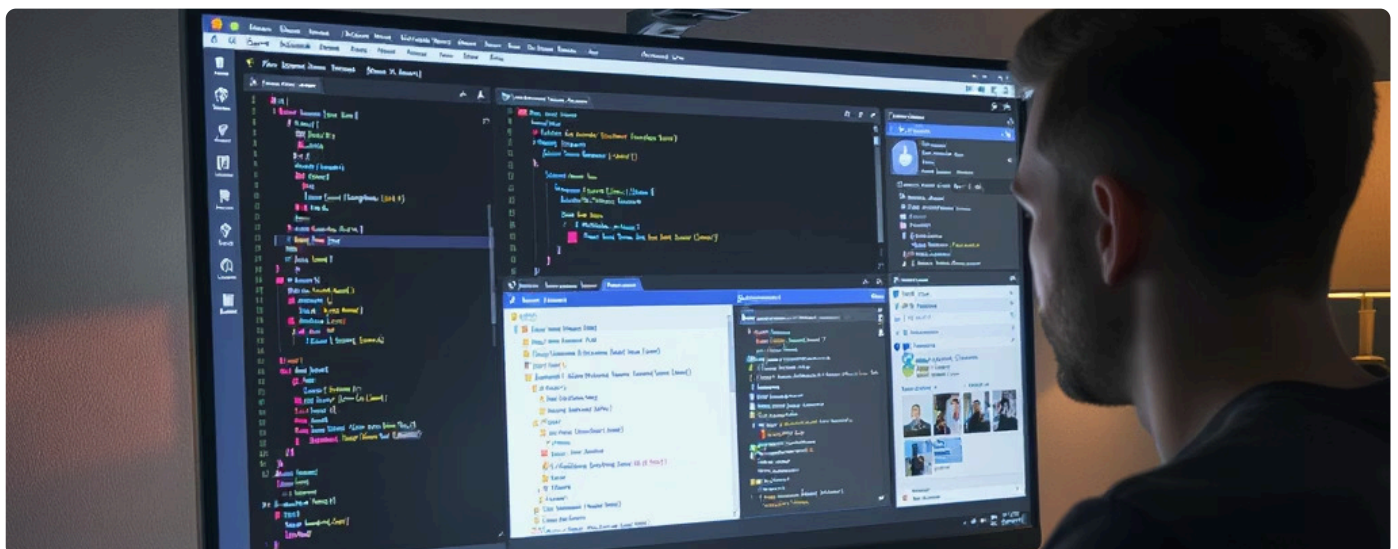
Esto facilita entender el historial y revertir cambios concretos.

3. Usa git log --oneline para revisar la historia

Este comando resume los commits mostrando solo el identificador abreviado y el mensaje. Ideal para inspeccionar rápidamente la evolución del proyecto.

4. Sincroniza frecuentemente

Haz git pull antes de empezar a trabajar cada día para tener el código más reciente, y git push al finalizar la jornada para compartir tu progreso. Así evitas conflictos innecesarios.



5. Mantén tu repositorio limpio

Incluye un archivo .gitignore para evitar subir archivos temporales, cachés o dependencias. Esto mantiene el repositorio liviano y profesional.

6. Usa herramientas visuales si estás empezando

Si prefieres no usar la terminal, prueba:

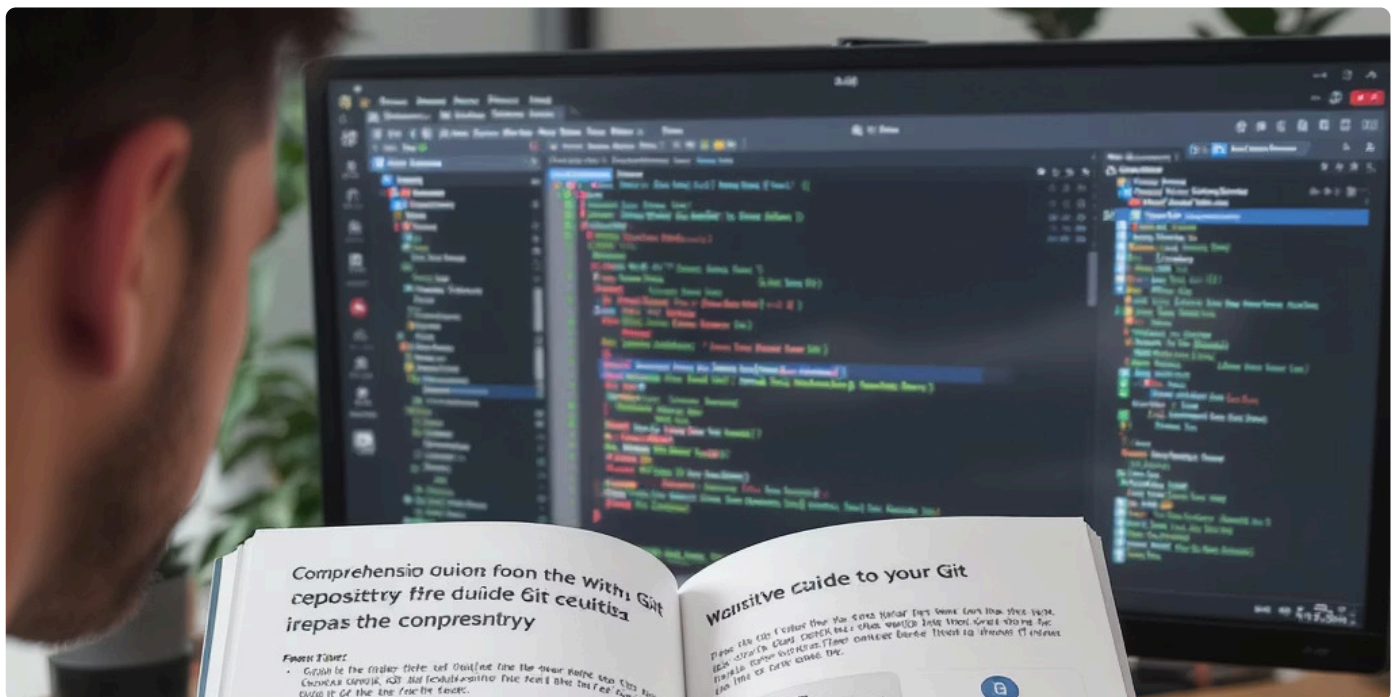
- **GitHub Desktop:** interfaz sencilla para gestionar commits y sincronizar.
- **Sourcetree:** herramienta profesional gratuita con soporte para Git y Mercurial.
- **VS Code Panel de Control:** integrado en el editor, ideal para flujos rápidos.

7. Aprende comandos intermedios pronto

Cuando te sientas cómodo con los básicos, explora comandos como:

- `git diff` → compara diferencias entre versiones.
- `git checkout` → navega entre commits o ramas.
- `git rebase` → reordena y limpia el historial.

Comprenderlos te permitirá optimizar proyectos grandes con múltiples colaboradores.



Mitos y Realidades

✗ **Mito:** "Git necesita conexión a internet para funcionar."

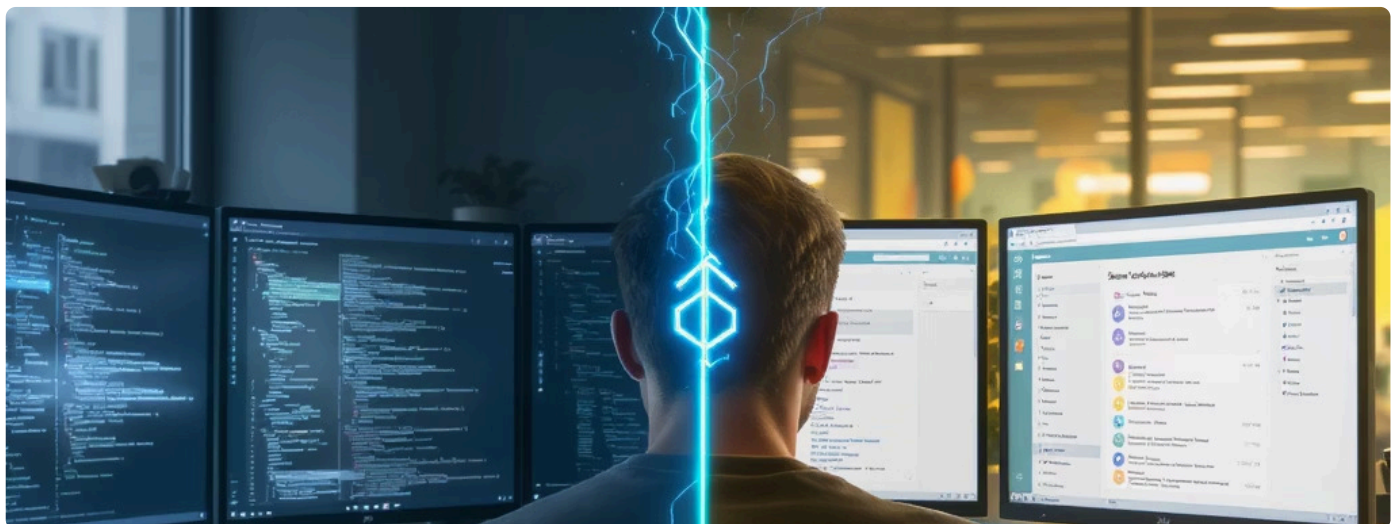
→ **FALSO.** Git es un sistema distribuido: todo el historial y las operaciones básicas (add, commit, diff, revert) funcionan localmente. Solo necesitas conexión al usar comandos que interactúan con el remoto (push, pull, fetch). Esto lo convierte en una herramienta robusta para trabajar incluso sin conexión.

✗ **Mito:** "Solo los programadores profesionales necesitan aprender Git."

→ **FALSO.** Git es útil para cualquier persona que trabaje con versiones de documentos: diseñadores que gestionan archivos de Illustrator, escritores que versionan manuscritos, analistas que mantienen scripts o incluso equipos de marketing que editan material digital. Git es control de cambios, no solo código.

📄 Resumen Final (para examen)

- **git init** → Crea un nuevo repositorio local.
- **git add** → Añade archivos al área de preparación.
- **git commit** → Guarda los cambios con un mensaje.
- **git push** → Sube los commits al repositorio remoto.
- **git pull** → Descarga los cambios más recientes del remoto.
- **Consejo clave:** sincroniza y documenta frecuentemente para evitar conflictos.
- Git no necesita conexión salvo al interactuar con el servidor remoto.



Sesión 7: Ramas, merges y resolución de conflictos (Git avanzado).

Fundamentos: El poder de las ramas en Git

En los proyectos modernos, especialmente los que involucran equipos distribuidos, resulta imprescindible trabajar en paralelo sin romper el código principal. Git resuelve este desafío mediante el uso de ramas (branches), una de sus características más poderosas y distintivas. Una rama es una línea independiente de desarrollo dentro del mismo repositorio. Imagina el proyecto como un árbol: el tronco es la rama principal (main o master), y cada nueva funcionalidad, prueba o corrección se desarrolla en una rama lateral que luego se une al tronco mediante una fusión o merge.

¿Por qué usar ramas?

Las ramas permiten:

- Desarrollar sin interferir con el código estable.
- Probar ideas nuevas sin riesgo.
- Colaborar simultáneamente con otros desarrolladores.
- Organizar el ciclo de desarrollo siguiendo estrategias profesionales como GitHub Flow o Git Flow.

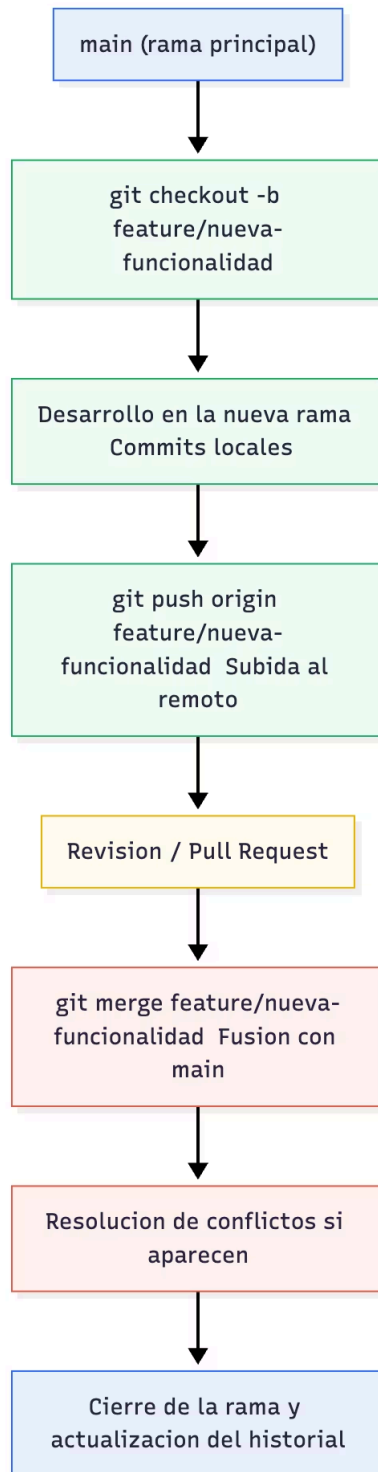
Cada rama puede contener varios commits y ser fusionada (mergeada) con la rama principal cuando el trabajo esté listo. Si varias personas modifican el mismo archivo o incluso la misma línea, Git genera un conflicto, que debe resolverse manualmente antes de continuar.

El ciclo de trabajo avanzado con ramas

1. Crear una nueva rama con `git checkout -b nombre-rama`.
2. Realizar los cambios y confirmarlos con commits.
3. Sincronizar con el remoto (`git push origin nombre-rama`).
4. Fusionar la rama con main mediante `git merge nombre-rama`.
5. Resolver conflictos, si los hubiera, y cerrar la rama una vez integrada.

Este flujo permite mantener un desarrollo limpio y trazable. Las ramas no son un lujo de proyectos grandes: son la forma más profesional de mantener la estabilidad del código en cualquier entorno.

Esquema Visual: Flujo de trabajo con ramas y merges



Explicación del esquema: El flujo visualiza cómo las ramas permiten crear caminos de desarrollo paralelos. La rama principal (main) actúa como base estable. Desde ella se genera una nueva rama con `git checkout -b`, donde se realizan cambios. Una vez probada la funcionalidad, se sube al remoto para revisión (pull request). Finalmente, se fusiona con la rama principal mediante `git merge`. Si surgen conflictos, se resuelven antes de confirmar la fusión. El resultado es un proyecto estructurado, donde cada contribución tiene su propio espacio de trabajo aislado pero perfectamente integrable.



Caso de Estudio:

GitHub Flow en una empresa tecnológica

Contexto

La empresa CodeWave, dedicada al desarrollo de aplicaciones SaaS, adoptó Git como pilar central de su metodología ágil. El equipo de desarrollo, compuesto por más de 20 programadores, necesitaba un sistema que les permitiera trabajar de forma paralela sin interrumpir las actualizaciones semanales de producción.

Estrategia

Para ello implementaron el modelo GitHub Flow, una estrategia ligera y moderna basada en ramas cortas y revisiones continuas:

- **main** → siempre estable y lista para desplegar.
- **feature/** → cada nueva funcionalidad se desarrolla en una rama independiente.
- **hotfix/** → ramas rápidas para solucionar errores urgentes.

Ejemplo: Un desarrollador crea una rama para una nueva función de registro:

```
git checkout -b feature/registro-usuarios
```

1. Realiza commits frecuentes documentando los avances.
2. Cuando termina, envía los cambios al remoto y crea un pull request en GitHub.
3. Otros compañeros revisan el código, comentan y aprueban.
4. Finalmente, se fusiona con la rama main:

```
git merge feature/registro-usuarios
```

Resolución de conflictos

En una ocasión, dos programadores modificaron el mismo archivo de configuración. Git detectó el conflicto durante la fusión y marcó las líneas en conflicto así:

```
<<<<<<< HEAD
config.timeout = 5000
=====
config.timeout = 3000
>>>>>>> feature/nueva-config
```

El equipo revisó ambas versiones, decidió cuál mantener (o combinarlas) y guardó los cambios. Tras esto, realizó un nuevo commit confirmando la resolución.

Resultado

- **Cero interrupciones en producción:** las pruebas y errores se gestionan en ramas aisladas.
- **Mayor transparencia:** cada cambio queda documentado en su propio pull request.
- **Crecimiento escalable:** el número de contribuciones semanales aumentó un 60% sin afectar la estabilidad del código.

GitHub Flow demostró que, con ramas y merges bien gestionados, el trabajo colaborativo se vuelve predecible, seguro y ágil.

```
1  @tab-width: 14em;
2
3  atom-pane-container
4    atom-pane {
5      -webkit-flex-direction: column;
6    }
7  }
8
9  .tab-bar {
10     width: @tab-width;
11     max-width: @tab-width;
12     min-width: @tab-width;
13     height: auto;
14     padding: 0;
15     -webkit-flex-direction: column;
16
17     .tab {
18       top: 0;
19       width: 100%;
20       height: auto;
21       -webkit-flex: 0 0 100%;
22
23       &.active {
24         width: 100%;
25         -webkit-flex: 1 0 100%;
26       }
27
28       .title {
29         text-align: left;
30       }
31     }
32
33     .placeholder {
34       position: relative;
35       width: auto;
36       height: 1px;
```

Herramientas y Consejos Profesionales

1. Usa git branch y git checkout -b para gestionar ramas

- `git branch` → muestra todas las ramas existentes.
- `git checkout -b nombre-rama` → crea y cambia directamente a la nueva rama.

Nombrar correctamente las ramas es buena práctica:

- `feature/` para nuevas funcionalidades.
- `fix/` o `hotfix/` para correcciones.
- `experiment/` para pruebas o prototipos.

2. Fusiona con precaución y orden

Antes de ejecutar un `git merge`, asegúrate de que tu rama principal está actualizada:

```
git checkout main
git pull origin main
git merge feature/nueva-funcionalidad
```

Esto reduce conflictos y mantiene la coherencia entre ramas.

3. Herramientas visuales para merges y conflictos

Resolver conflictos directamente en la terminal puede ser confuso al principio. Por eso se recomiendan entornos visuales como:

- **Visual Studio Code:** resalta automáticamente las diferencias y permite elegir versiones.
- **GitKraken:** interfaz intuitiva para ramas y merges con vista gráfica del historial.
- **Sourcetree:** gratuita, ideal para gestionar repositorios y conflictos visualmente.

4. Aplica políticas de revisión de código

Usa pull requests o merge requests en GitHub o GitLab. Cada fusión debe pasar revisión de otro miembro del equipo. Esto mejora la calidad del código y evita errores en producción.

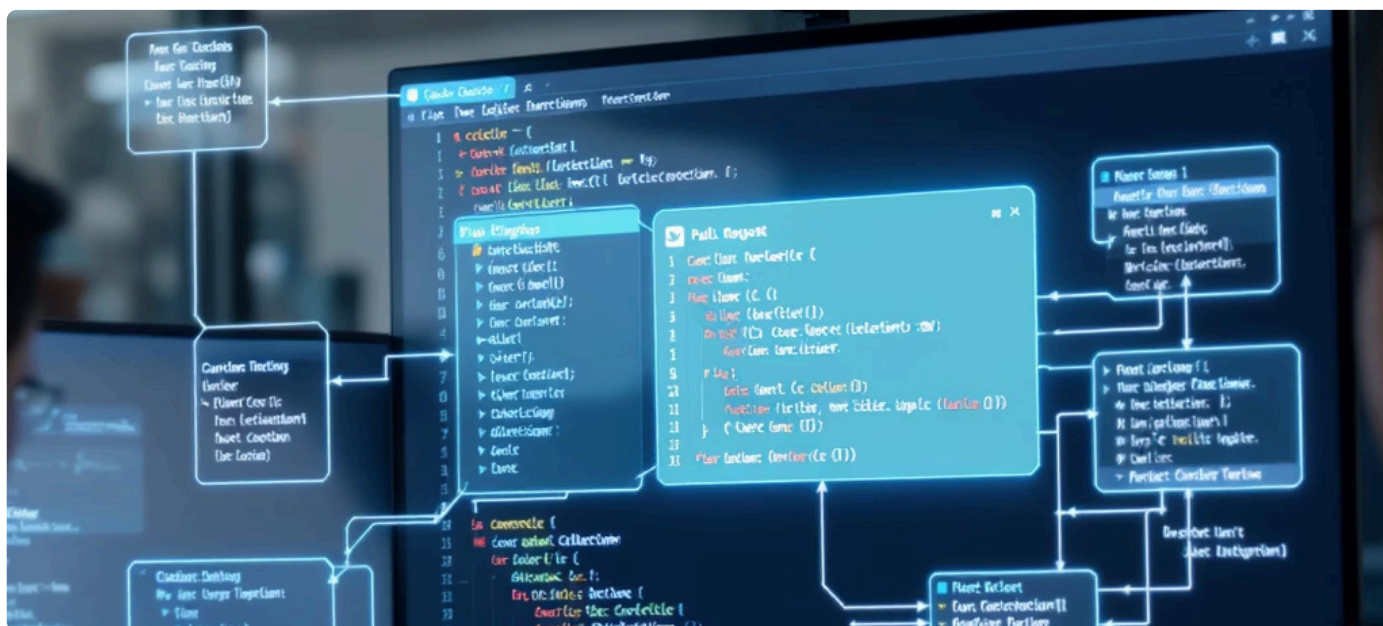
5. Mantén ramas cortas y actualizadas

Cuanto más tiempo mantengas una rama sin fusionar, más probable será que sufra conflictos. Lo ideal es mantenerlas activas solo el tiempo necesario y actualizarlas con frecuencia usando:

```
git pull origin main --rebase
```

6. Automatiza fusiones con integraciones continuas


Herramientas como GitHub Actions, GitLab CI/CD o CircleCI pueden ejecutar pruebas automáticas cada vez que se crea un pull request. Así solo se fusionan ramas que superen las validaciones.



Mitos y Realidades

 **Mito:** "Las ramas son solo para proyectos grandes."

→ **FALSO.** Incluso en proyectos personales, las ramas son esenciales para mantener el código limpio. Permiten probar nuevas ideas sin alterar la versión principal y volver atrás sin perder nada.

 **Mito:** "Los conflictos indican que Git falló."

→ **FALSO.** Los conflictos no son errores del sistema, sino una consecuencia natural del trabajo colaborativo. Indican que dos personas cambiaron las mismas líneas y que Git necesita intervención humana para decidir cuál conservar. Saber resolverlos es parte del dominio profesional de Git.

Resumen Final (para examen)

- **Rama** = línea de desarrollo independiente dentro del repositorio.
- **git branch** y **git checkout -b** → crean y gestionan ramas.
- **git merge** → fusiona una rama con otra (generalmente con main).
- **Conflicto** → ocurre cuando dos cambios afectan la misma línea de un archivo.
- **Resolución** → revisar, elegir versión correcta y confirmar con un nuevo commit.
- **Estrategias comunes:** GitHub Flow (ramas cortas y revisión continua) y hotfix (correcciones urgentes).