

HOLA, ¡GIT!

TRAZABLE • CONTROLABLE • COLABORATIVO • REVERSIBLE



Paula Pereda Suárez
paaupereda@gmail.com

HOLA, ¿GIT?



¿Qué es el control de versiones?

Procesos y sistemas para **gestionar cambios** en
archivos, programas y directorios

**El control de versiones es útil para cualquier cosa
que:**

- cambie con el tiempo,
- necesite ser compartida.

¿Qué puede hacer el control de versiones?

- Rastrear archivos en diferentes estados
- Combinar diferentes versiones de archivos
- Identificar una versión en particular
- Revertir cambios

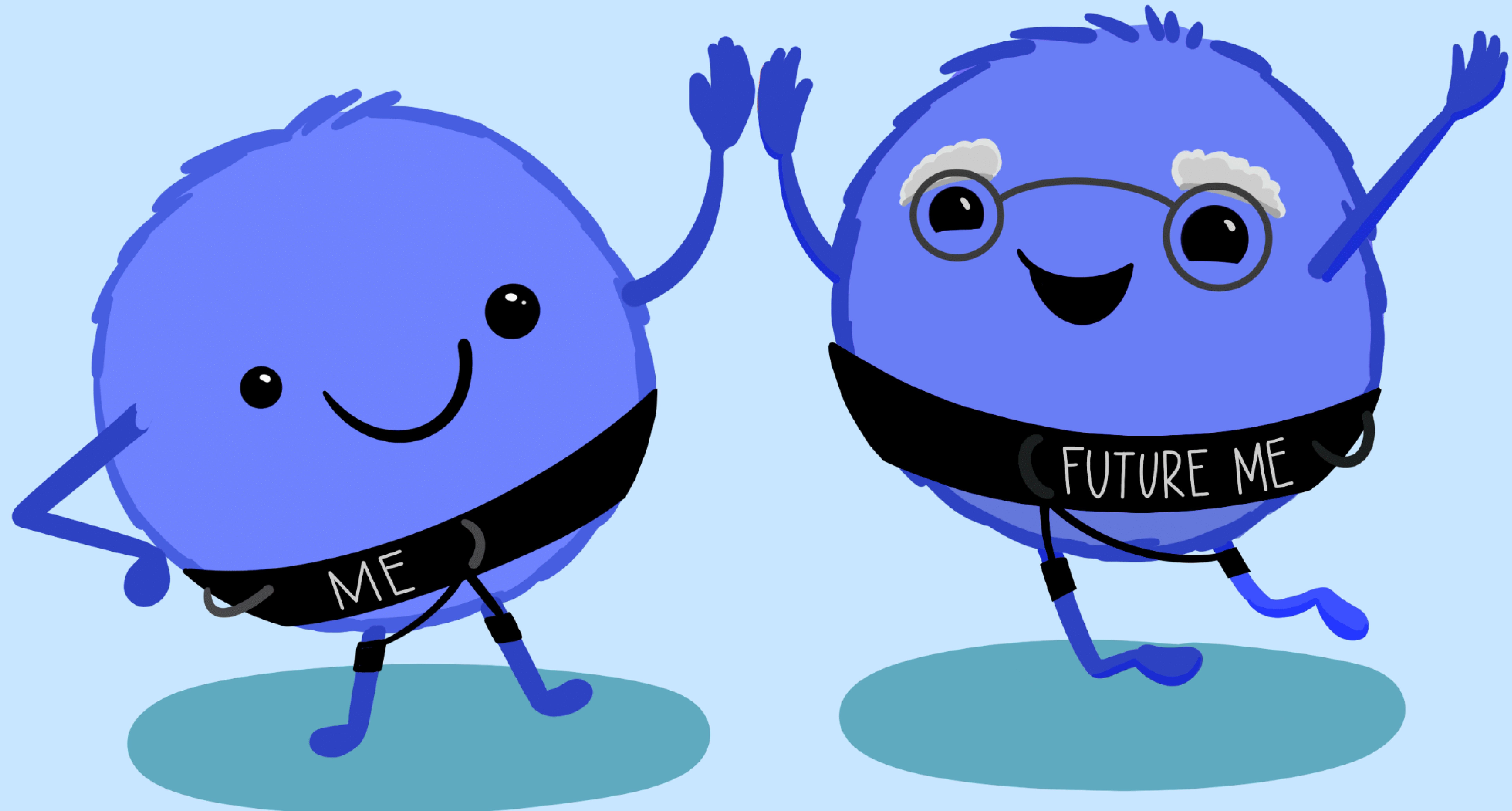
Introduciendo a Git



- Sistema de control de versiones popular para el desarrollo de software y proyectos de datos
- De código abierto
- Escalable

Beneficios de Git

- Git almacena todo, por lo que nada se pierde
- Podemos comparar archivos en distintos momentos
- Ver qué cambios se hicieron, quién los hizo y cuándo
- ¡Revertir a versiones anteriores de los archivos!

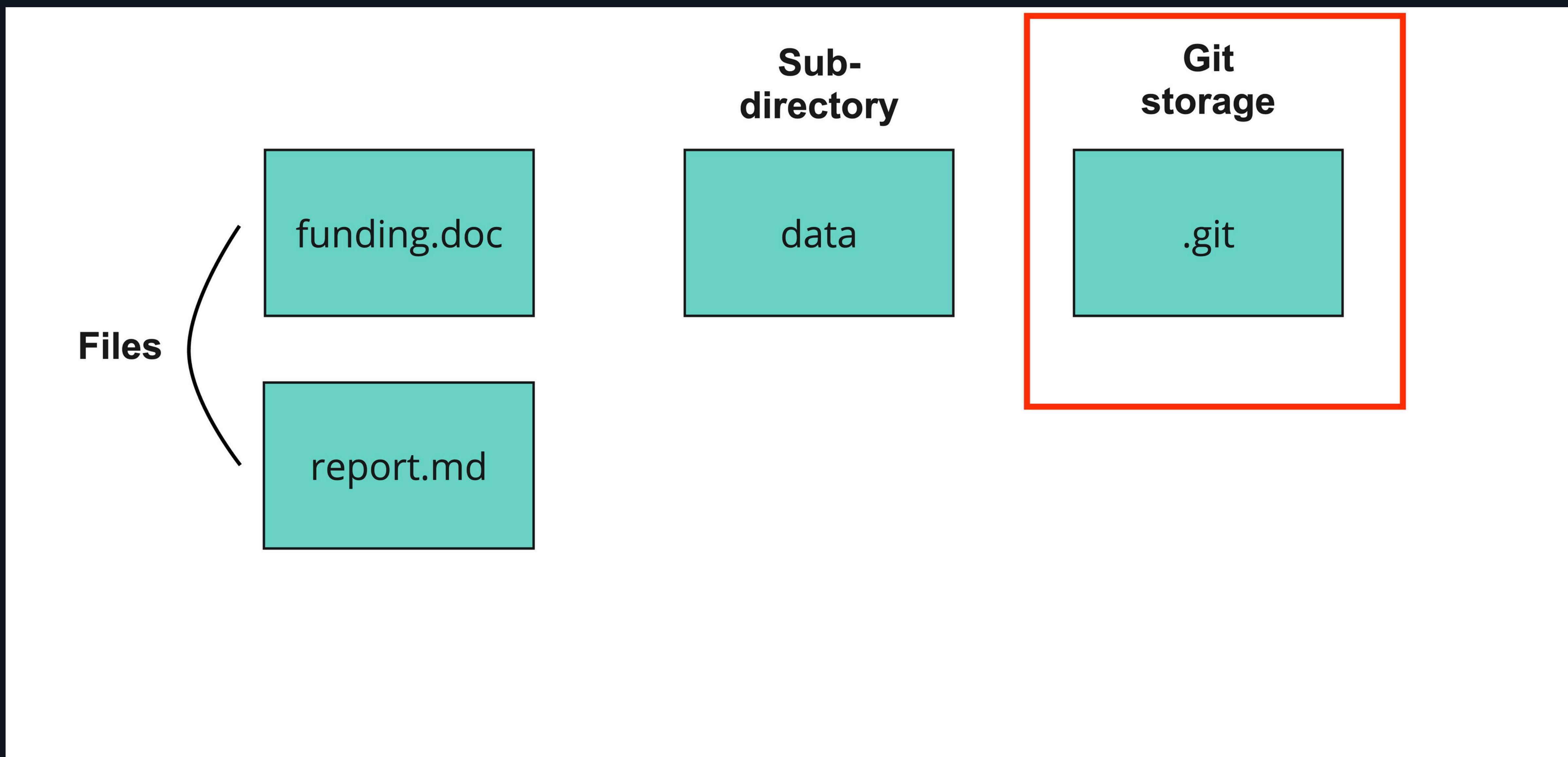


Creando repo(sitorios)

- Git almacena todo, por lo que nada se pierde
- Podemos comparar archivos en distintos momentos
- Ver qué cambios se hicieron, quién los hizo y cuándo
- ¡Revertir a versiones anteriores de los archivos!

¿Qué es un repo de Git?

Repositorio Git = directorio que contiene archivos y subdirectorios, además del almacenamiento de Git.



¿Por qué hacer repos?

Beneficios:

- Registrar versiones de forma sistemática
- Revertir a versiones anteriores
- Comparar versiones en distintos momentos en el tiempo
- Colaborar con colegas

El flujo de trabajo en Git

- Editar y guardar archivos en nuestra computadora
- Agregar los archivos al área de staging de Git

Registra qué se ha modificado

- Hacer commit de los archivos
- Git toma una instantánea de los archivos en ese momento
- Permite comparar y revertir archivos

Agregar al área de staging

- **Agregar un solo archivo**

```
git add README.md
```

- **Agregar todos los archivos modificados**

```
git add .
```

- **El . significa todos los archivos en el directorio actual y sus subdirectorios.**

Agregar al área de staging

- Agregar un solo archivo

```
git add README.md
```

- Agregar todos los archivos modificados

```
git add .
```

- El . significa todos los archivos en el directorio actual y sus subdirectorios.

Hacer un commit

```
git commit -m "Agregar README."
```

- -m: permite escribir un mensaje de registro (mensaje log) sin abrir un editor de texto.
- El mensaje de registro es útil como referencia para saber qué cambios se hicieron.
- Mejor práctica: mensajes cortos y concisos.

Staging vs. Committing en Git

Área de staging

- Zona intermedia donde se guardan los archivos preparados para el commit.
- Te permite decidir qué cambios incluir y cuáles dejar afuera.
- Podés agregar uno, varios o todos los archivos.

Hacer un commit

- Guarda de forma permanente en el historial de Git los cambios que estaban en staging.
- Cada commit es como una instantánea en el tiempo.
- Permite volver atrás, comparar versiones y entender la evolución del proyecto.

Staging area



Hacer un commit



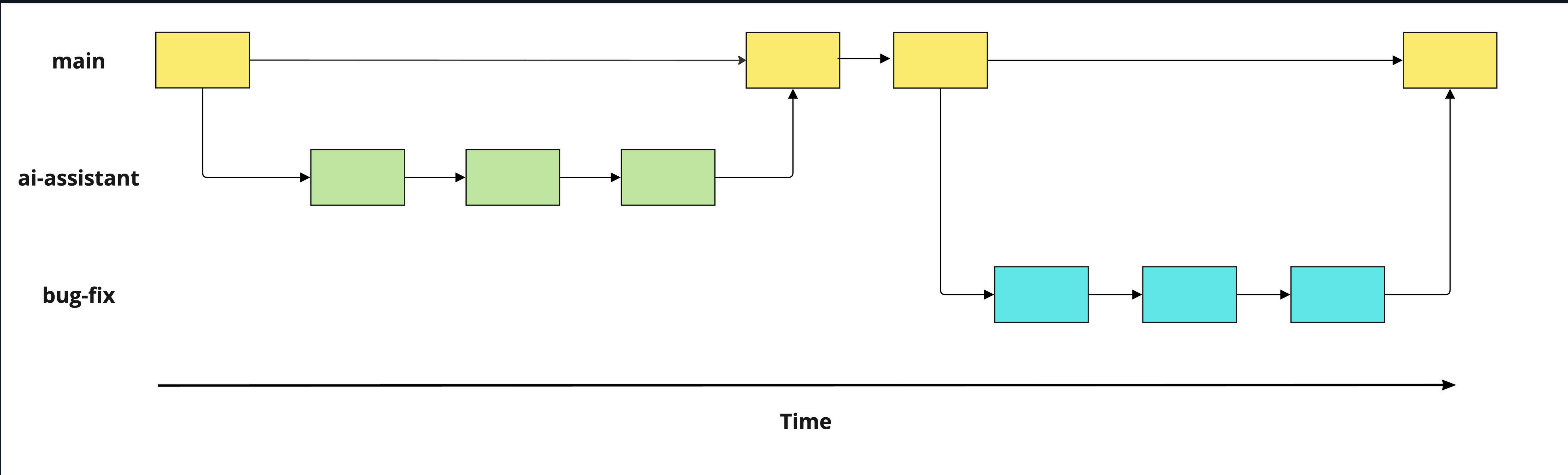
Rama (branch) en Git

- Una rama es una versión individual de un repositorio.
- Git utiliza las ramas para seguir de manera sistemática múltiples versiones de los archivos.
- En cada rama:
 - Algunos archivos pueden ser los mismos.
 - Otros pueden ser diferentes.
 - Algunos pueden no existir en absoluto.

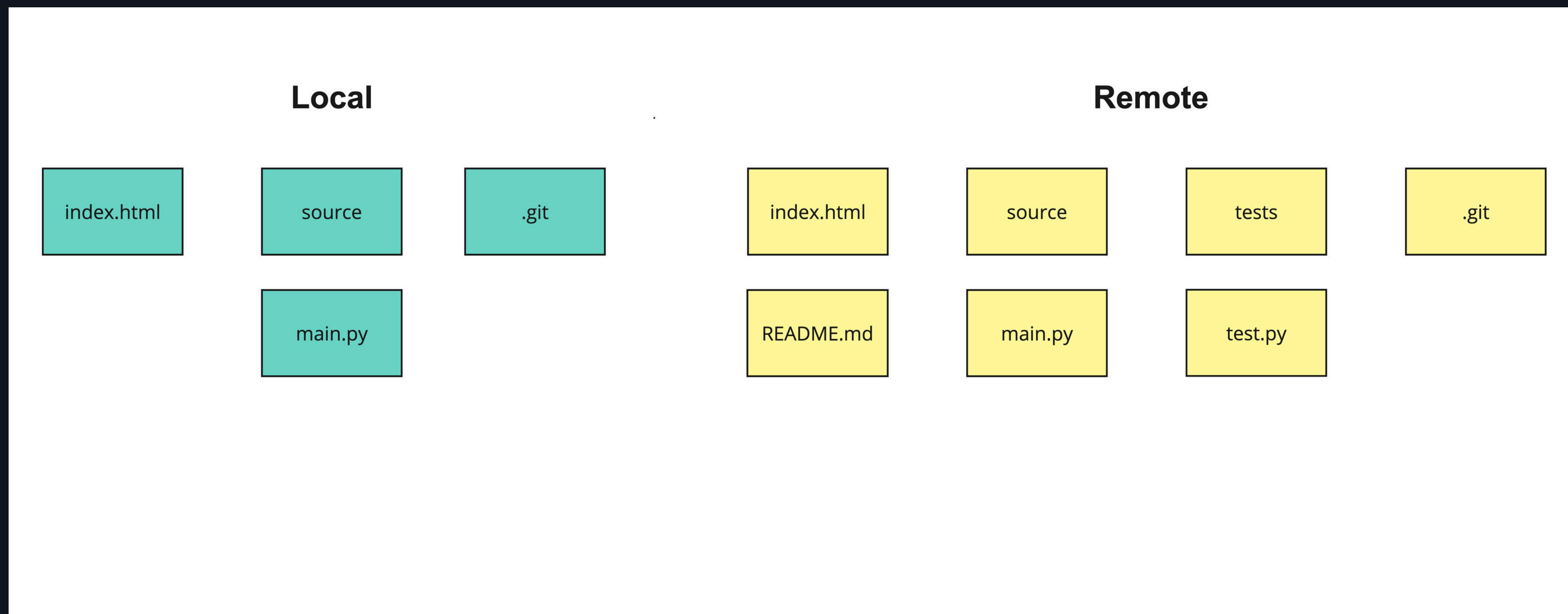
Una analogía útil: pensar en una rama como una línea de tiempo paralela. Podés trabajar en cambios sin afectar la rama principal, y luego decidir si fusionarlos (merge) o mantenerlos separados.

¿Por qué usar ramas (branches)?

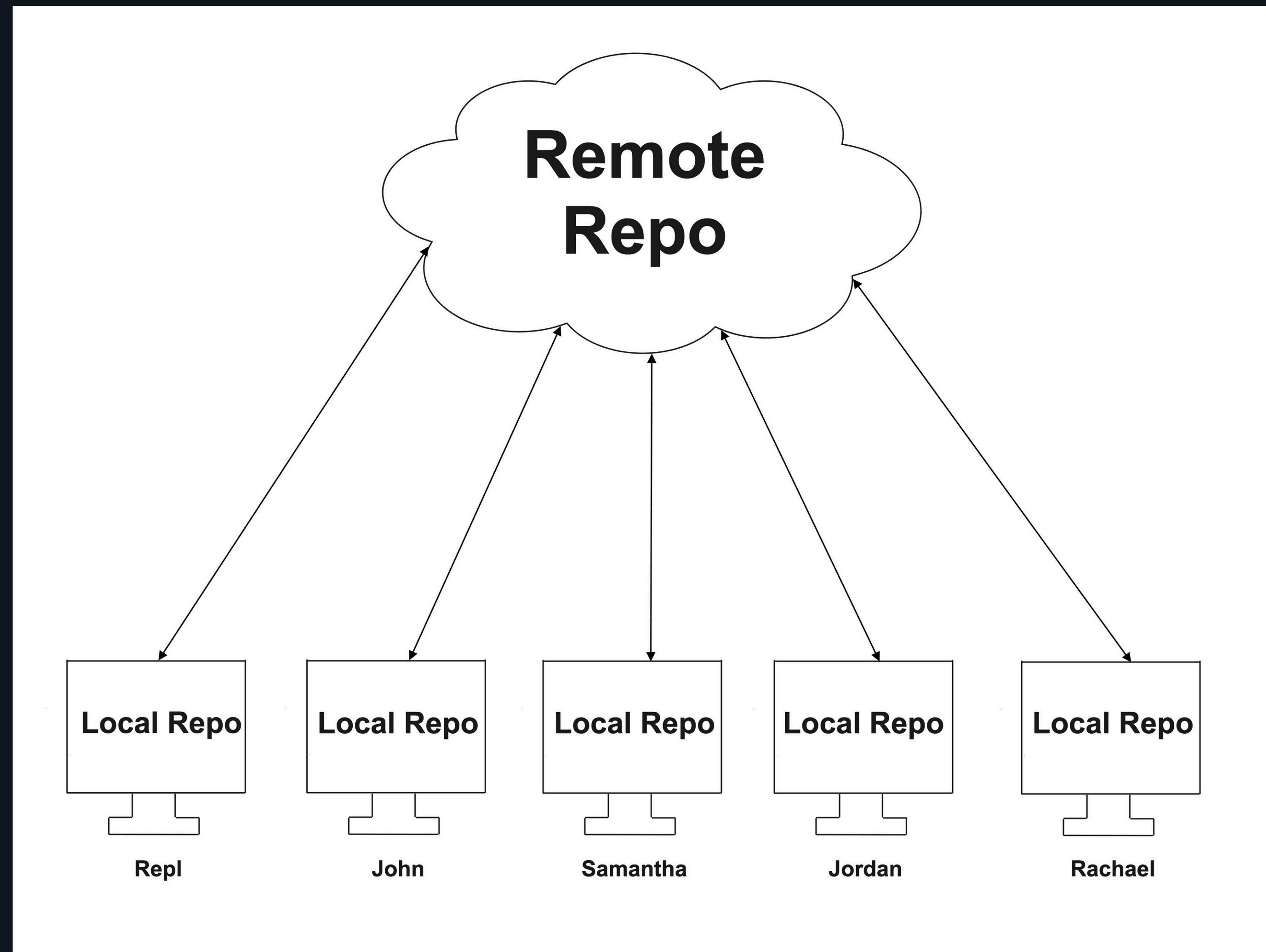
- Permiten que varios desarrolladores trabajen en un mismo proyecto al mismo tiempo sin pisarse los cambios.
- Comparar el estado de un repositorio entre ramas, identificando qué se mantiene igual y qué se modificó.
- Combinar contenidos (fusionar ramas) para llevar nuevas funciones o correcciones al sistema en producción.
- Cada rama debe tener un propósito específico, por ejemplo: desarrollar una nueva funcionalidad, corregir un error o hacer pruebas.

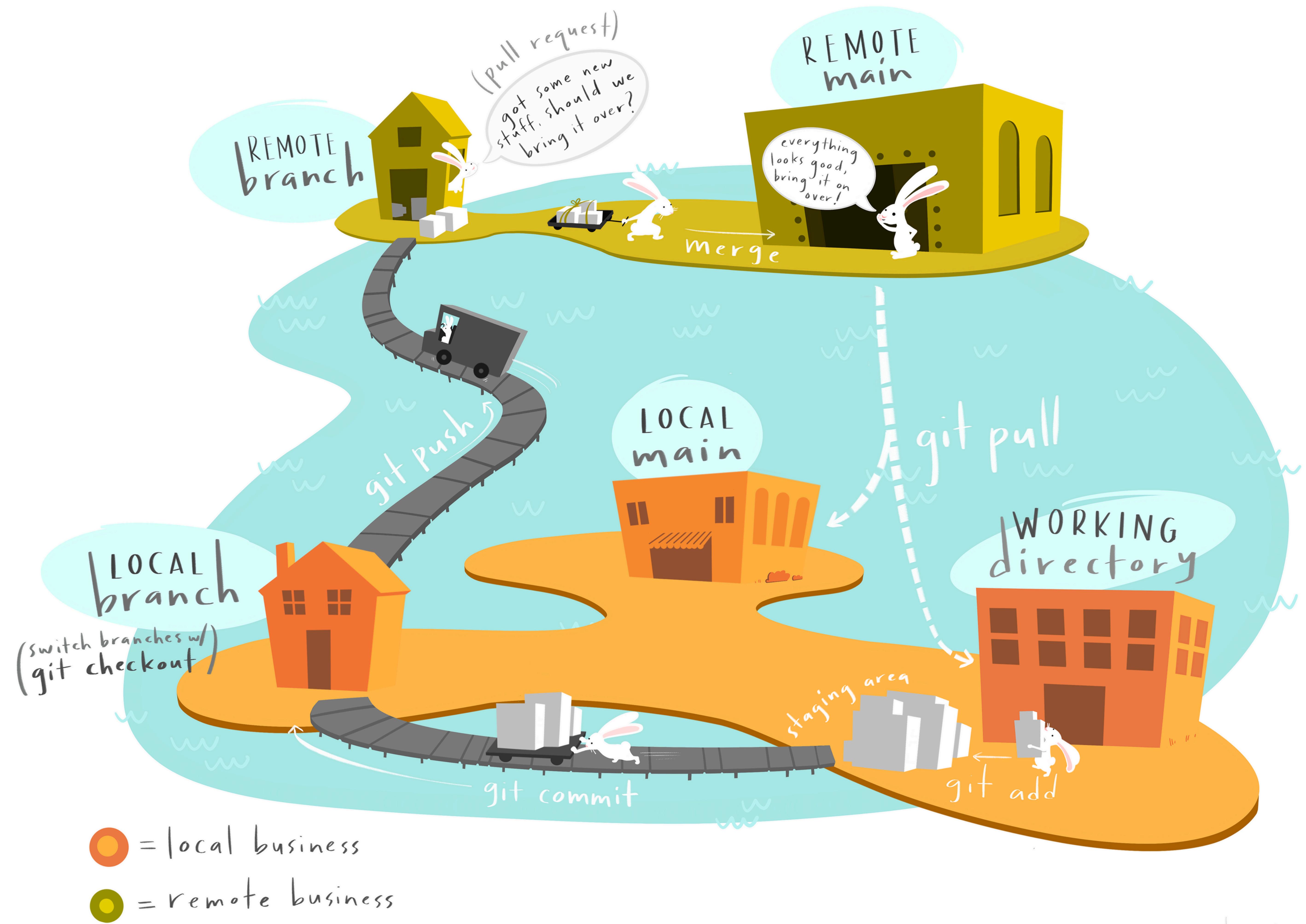


Local versus remoto



Colaborando en proyectos de Git





@allison-horst

Obtener (fetch) desde un remoto

- Obtener desde el remoto origin

git fetch origin

- Obtener todas las ramas remotas
- Puede crear nuevas ramas locales si estas solo existían en el remoto
- Fetch no fusiona el contenido del remoto con tu repositorio local.
- Solo descarga y actualiza la información de referencias (commits, ramas) desde el remoto.
- En otras palabras, git fetch actualiza lo que sabe tu repo sobre el remoto, pero no toca tus archivos locales

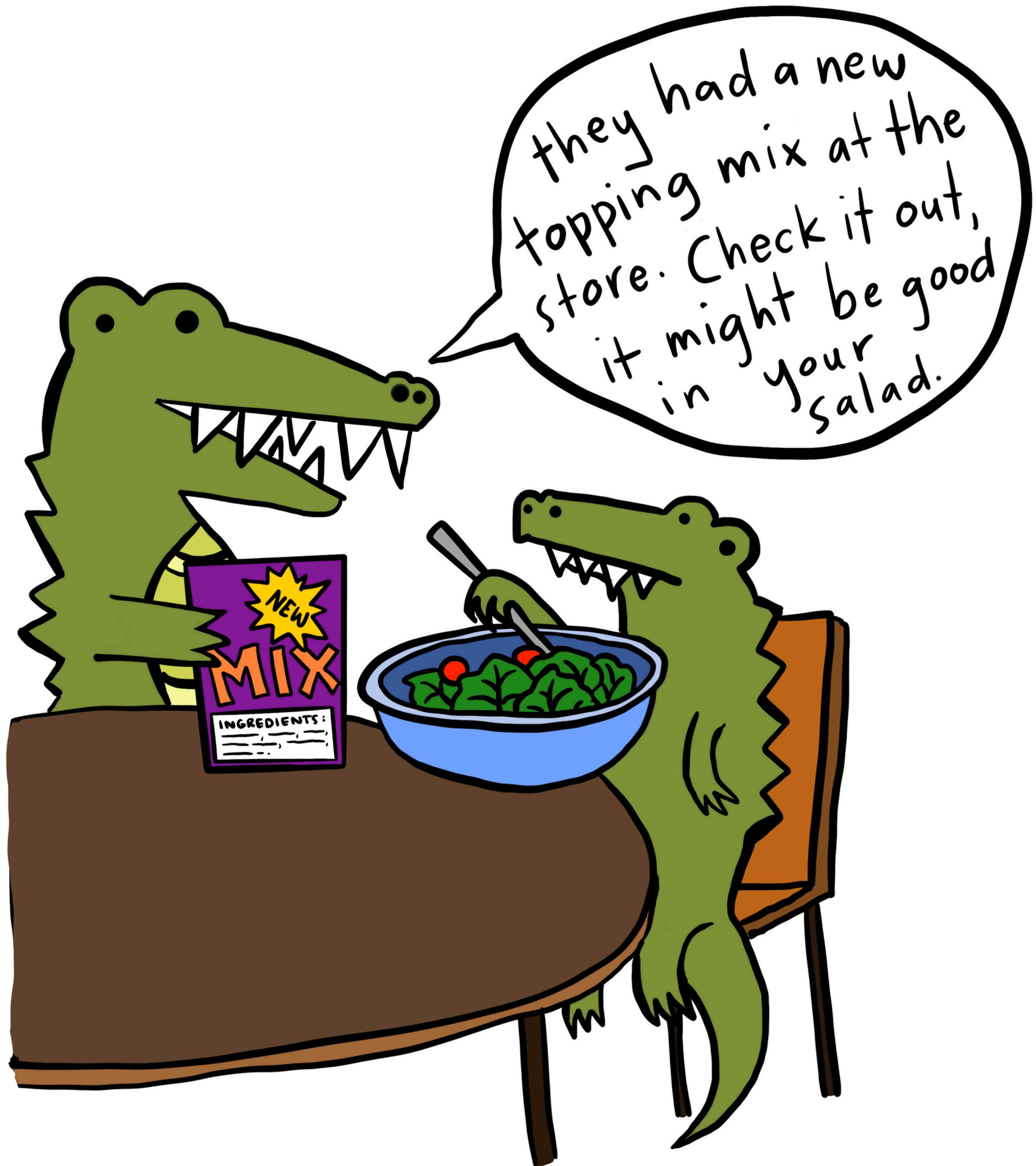
Hacer pull desde un remoto

- La sincronización entre local y remoto es un flujo de trabajo común.
- Git nos simplifica este proceso.
- Pull = Fetch + Merge desde el remoto por defecto (main) hacia la rama actual del repositorio local.

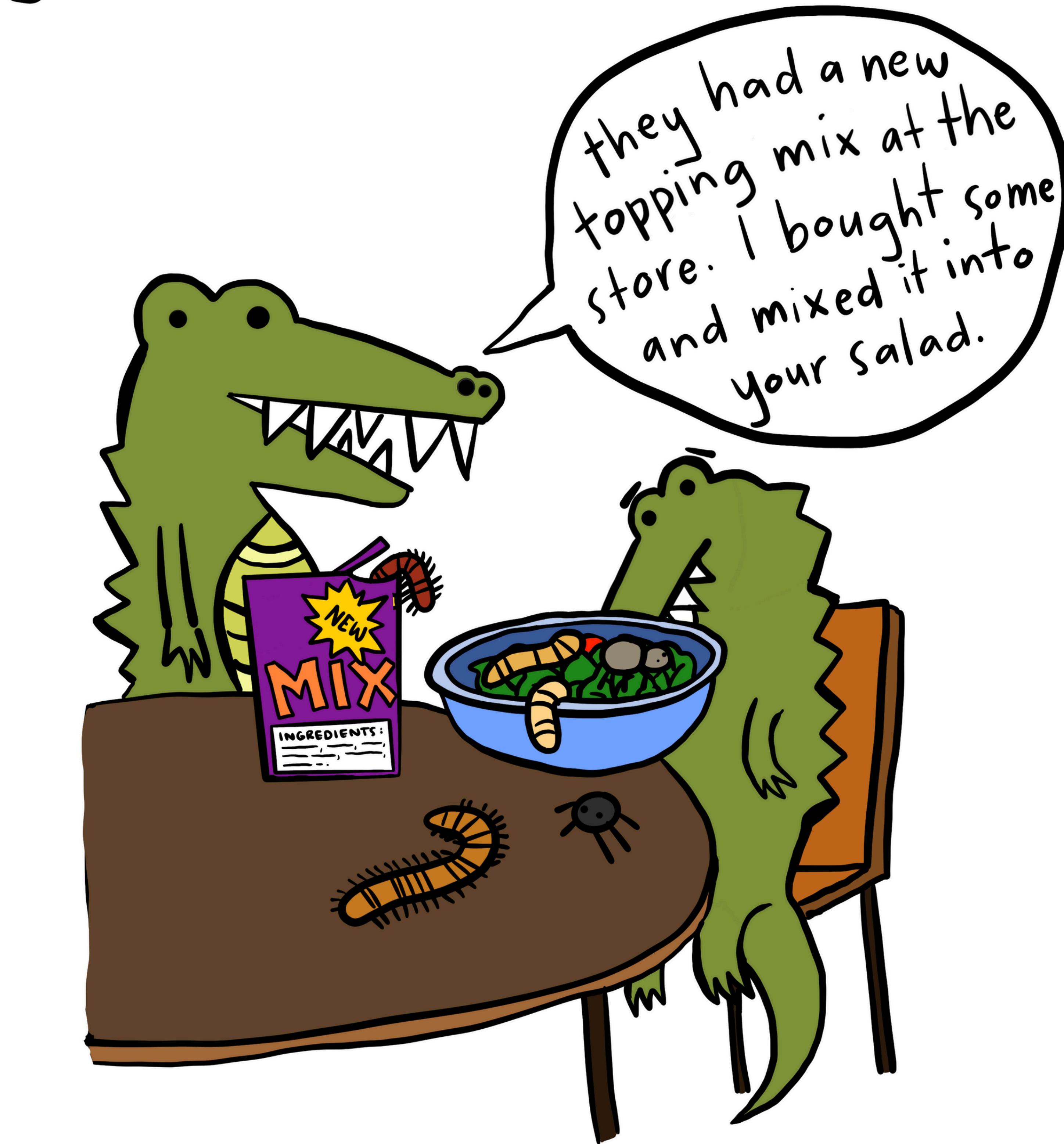
```
git pull origin
```

En otras palabras, git pull descarga los cambios del remoto y los integra directamente en tu rama local.

git fetch: bring stuff home



git pull: bring stuff home AND merge it in



check what you're getting before merging it in with git fetch

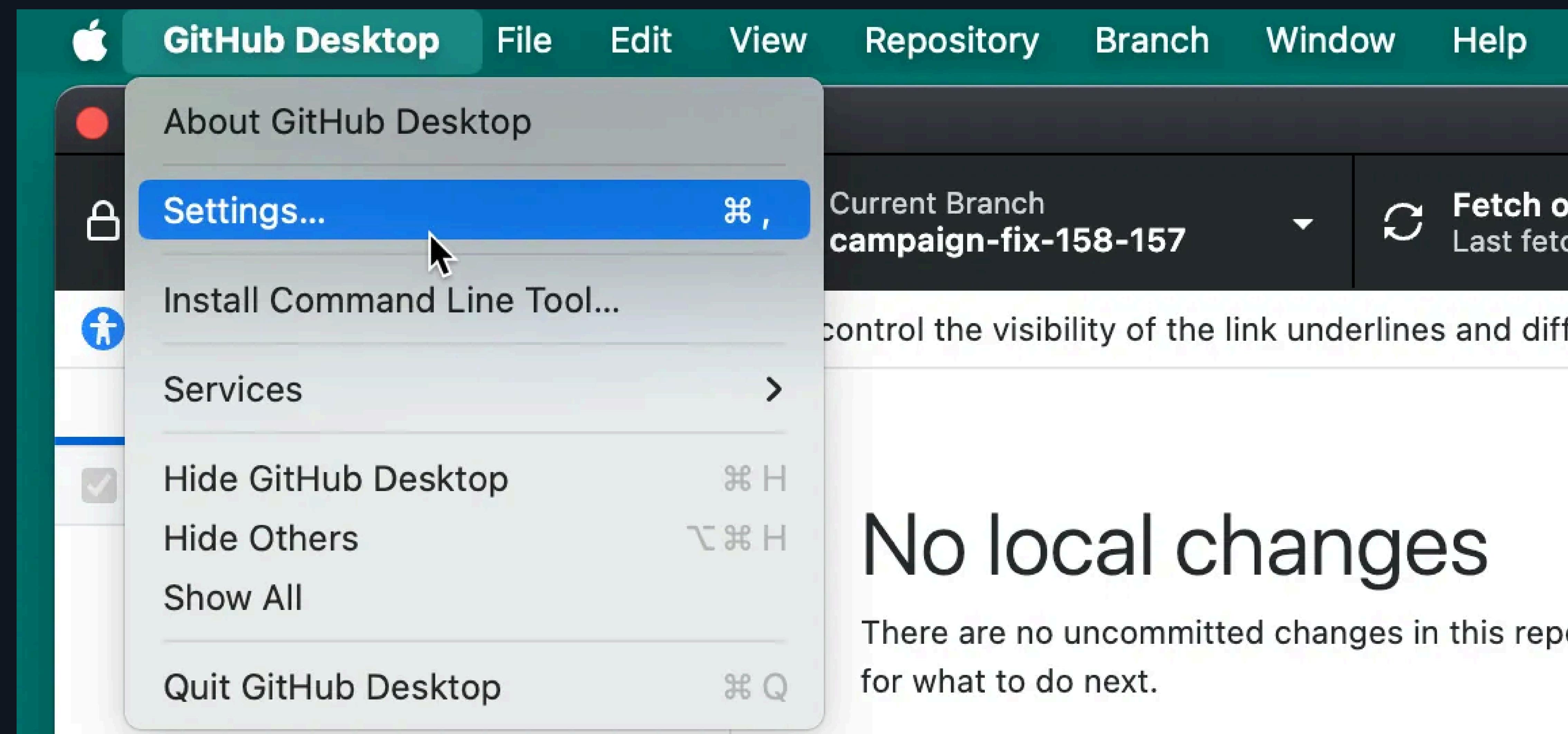


Primeros pasos con GitHub Desktop

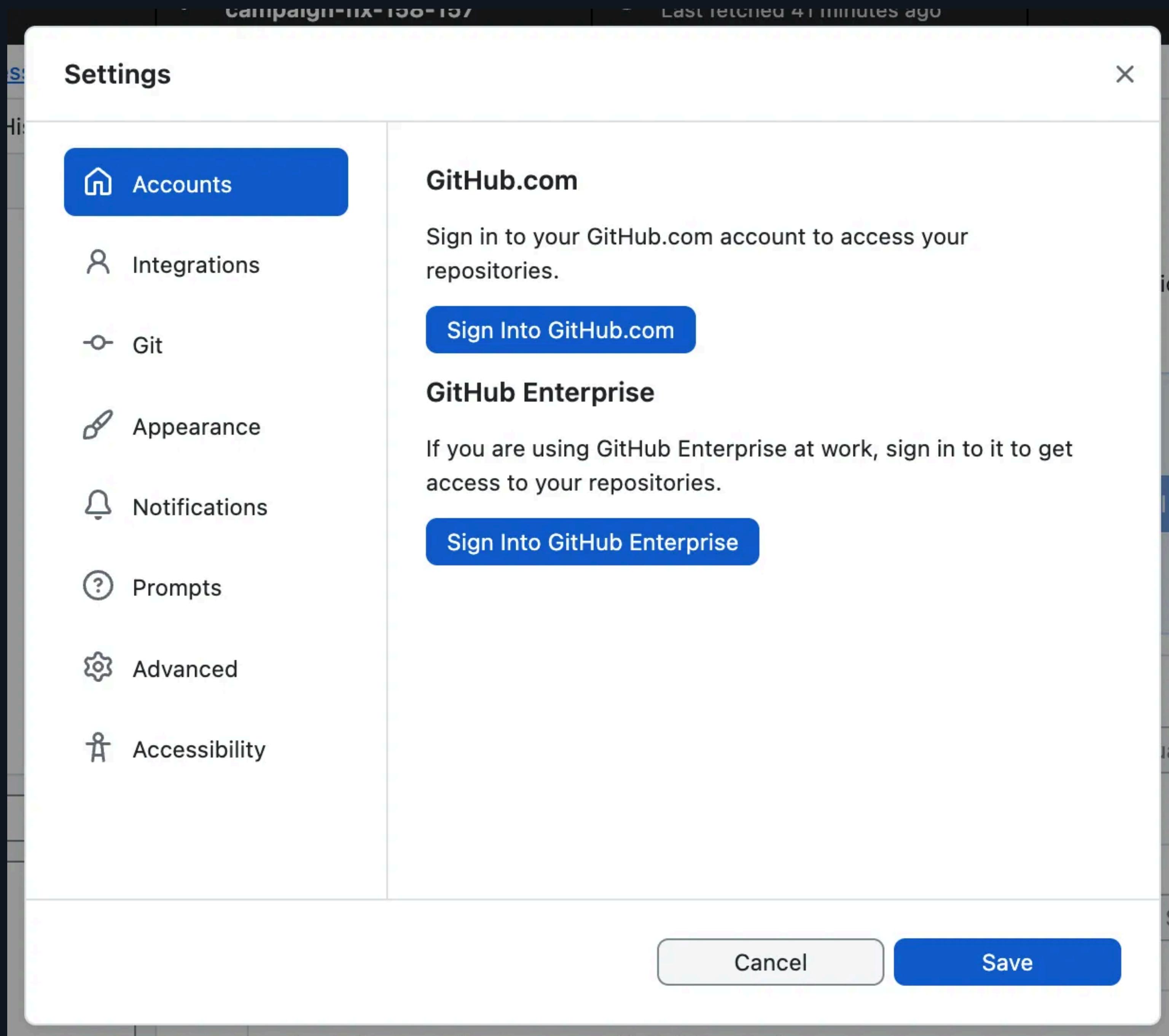
- GitHub Desktop es una aplicación gratuita y de código abierto que te ayuda a trabajar con código alojado en GitHub u otros servicios de alojamiento Git.
- Con GitHub Desktop podés ejecutar los comandos de Git — como hacer commits o enviar (push) cambios— a través de una interfaz gráfica, en lugar de usar la línea de comandos.

Parte 1: Instalación y autenticación

1. Antes de poder autenticarte en GitHub, necesitás una cuenta. Para más información, consultá [Crear una cuenta en GitHub](#).
2. En la barra de menús, seleccioná GitHub Desktop y luego hacé clic en Configuración (Settings).



3. En la ventana de Configuración, dentro del panel Cuentas (Accounts), hacé clic en el botón correspondiente de Iniciar sesión (Sign Into).



4. Seguí los pasos para iniciar sesión.

Proyectos de R

¿Qué es? Un archivo .Rproj que define la raíz del proyecto: dónde vive tu código, datos y outputs, y cómo se abre el entorno.

¿Por qué conviene?

- Rutas reproducibles (base estable para here::here()).
- Evita setwd() y “funciona en mi máquina”.
- Se integra con Git/GitHub y con renv (librería por proyecto).
- Limpieza: configurá Save workspace to .RData on exit = Never.
- Buenas prácticas
- Estructura clara: data/raw/, scripts/, R/, outputs/.
- Ignorá data/raw/ y archivos generados en .gitignore.
- Comando básico

{here}

¿Qué es? Paquete que genera rutas relativas robustas desde la raíz del proyecto (.Rproj)

¿Por qué es mejor?

- Reemplaza `setwd()` y rutas absolutas frágiles.
- Funciona igual en distintas máquinas/carpeta de usuario.
- Ideal para lectura/escritura

{pacman}

¿Qué es? Un gestor que instala si falta y carga igual tus paquetes con una sola función.

¿Por qué es mejor?

- **Simplifica listas largas de library().**
- **Acelera talleres y equipos: menos fricción de instalación.**
- **Convive perfecto con renv: lo instalado queda fijado en renv.lock**

{renv}

¿Qué es? Aísla una librería por proyecto y fija versiones en `renv.lock` para reproducibilidad

¿Por qué es mejor?

- Elimina “me rompe por versiones”.
- `snapshot()` captura el estado; `restore()` lo reconstruye en otra máquina.
- No ensucia tu librería global.
- Comandos básicos (mantra: `init` → `use` → `snapshot` → `restore`)

```
if (!requireNamespace("renv", quietly = TRUE)) install.packages("renv")

renv::init() # crea renv/ y renv.lock # ... usá tus paquetes (ideal con
pacman::p_load(...))

renv::snapshot() # fija versiones en renv.lock

# En otra compu (o luego de clonar):

renv::restore() # reconstruye el entorno
```

Versionar: sí renv.lock; no renv/library/.

Junto a Git: commiteá renv.lock cada vez que cambien
paquetes/versiones.