



GIT

Sistema de control de versiones

El control de versiones es la práctica de **rastrear y administrar los cambios en el código** del software

Los sistemas de control de versiones (SCM) **son herramientas** de software que **ayudan** a los equipos de software a **administrar los cambios** en el código fuente a lo largo del tiempo

A medida que los entornos de desarrollo se han acelerado, los sistemas de control de versiones ayudan a los equipos de software a **trabajar de forma más rápida e inteligente**

¿Qué es Git?

Git es un sistema de control de versiones **distribuido gratuito y de código abierto** diseñado para manejar desde proyectos pequeños hasta proyectos muy grandes, con rapidez y eficiencia.

Git es fácil de aprender y ocupa poco espacio con un rendimiento ultrarrápido. Supera a las herramientas de SCM como Subversion, CVS, o ClearCase con características como ramas locales, áreas de preparación y múltiples flujos de trabajo.



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

¿Qué es GitHub?

GitHub es el mayor proveedor de **alojamiento de repositorios Git**, y es el punto de encuentro para que millones de desarrolladores colaboren en el desarrollo de sus proyectos.

Un gran porcentaje de los repositorios Git se almacenan en GitHub, y muchos proyectos de código abierto lo utilizan para **hospedar su Git**, realizar su seguimiento de fallos, hacer revisiones de código y otras cosas.

Por tanto, aunque no sea parte directa del proyecto de código abierto de Git, es muy probable que durante tu uso profesional de Git necesites interactuar con GitHub en algún momento.

Alternativas: GitLab, Bitbucket, etc

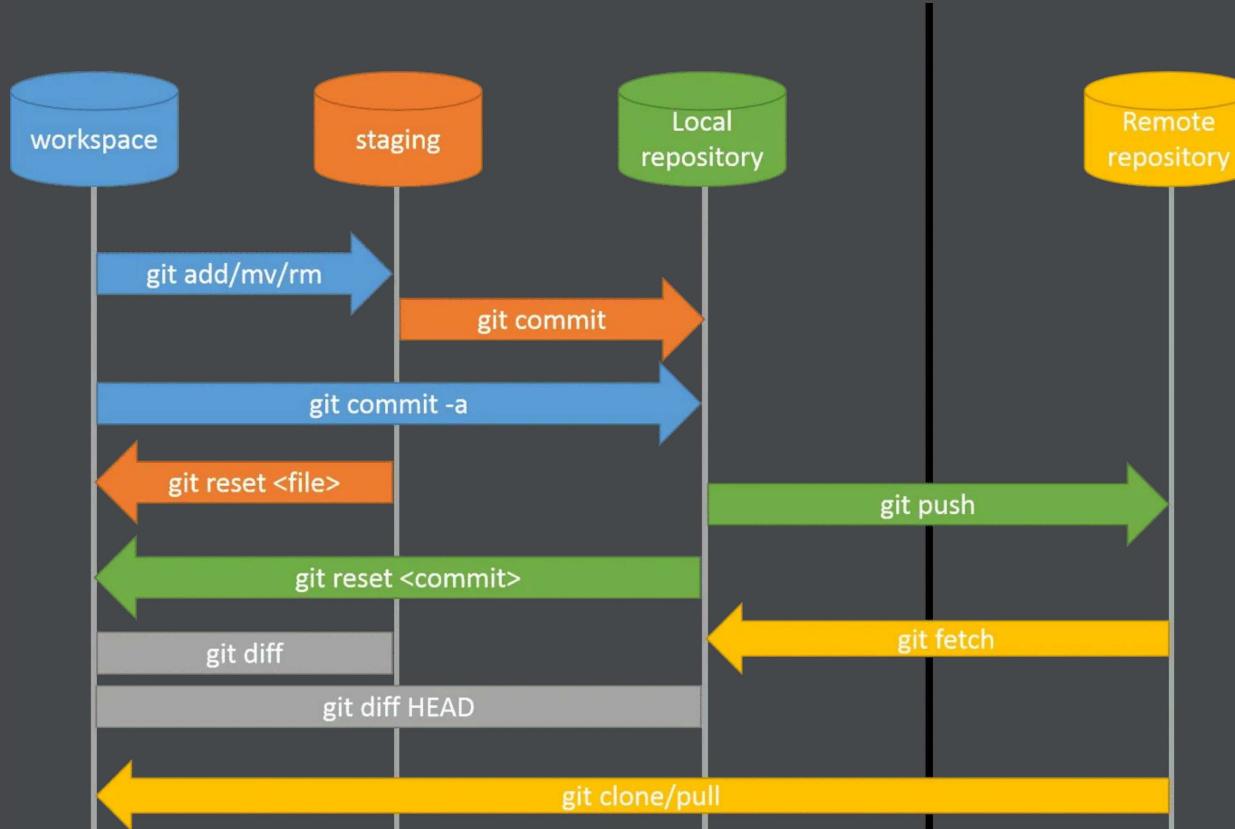


<https://github.com/>

¿GitHub vs Git?

	VS	
1 GitHub is a service		1 Git is a software
2 GitHub is a graphical user interface		2 Git is a command-line tool
3 GitHub is hosted on the web		3 Git is installed locally on the system
4 GitHub is maintained by Microsoft		4 Git is maintained by linux
5 GitHub is focused on centralized source code hosting		5 Git is focused on version control and code sharing
6 GitHub is a hosting service for Git repositories		6 Git is a version control system to manage source code history

¿Dónde está mi código?



Comandos (clone)

Git clone es un comando para descargar el código fuente existente desde un repositorio remoto (como Github, por ejemplo). En otras palabras, Git clone básicamente hace una copia idéntica de la última versión de un proyecto en un repositorio y la guarda en su computadora.

Hay un par de formas de descargar el código fuente, pero sobre todo prefiero él clone con https:

```
git clone https://name-of-the-repository-link
```

Por ejemplo, si queremos descargar un proyecto de Github, todo lo que tenemos que hacer es hacer clic en el botón verde (clonar o descargar), copiar la URL en el cuadro y pegarlo después del comando git clone que he mostrado en la esquina superior derecha.

Esto hará una copia del proyecto en tu espacio de trabajo local para que puedas comenzar a trabajar con él.

Comandos (branch)

Las ramas son muy importantes en el mundo de git. Dicho de manera rápida y básica, una rama no es más que un nombre que se da a un commit, a partir del cual se empieza a trabajar de manera independiente y con el que se van a enlazar nuevos commits de esa misma rama. Las ramas pueden mezclarse de modo que todo el trabajo hecho en una de ellas pase a formar parte de otra.

Mediante el uso de ramas, varios desarrolladores pueden trabajar en paralelo en el mismo proyecto simultáneamente. Podemos usar el comando git branch para crear, listar y eliminar ramas. Creando una nueva rama:

```
git branch <branch-name>
```

Este comando creará una rama **localmente**. Para insertar la nueva rama en el repositorio remoto, debes usar el siguiente comando:

```
git push -u <remote> <branch-name>
```

Para ver las ramas:

```
git branch or git branch --list
```

Para borrar las ramas:

```
git branch -d <branch-name>
```

Comandos (checkout)

Este es también uno de los comandos Git más utilizados. Para trabajar en una rama, primero debe cambiarse a ella. Usamos git checkout principalmente para cambiar de una rama a otra. También podemos usarlo para verificar archivos y confirmaciones.

`git checkout <name-of-your-branch>`

Hay algunos pasos que debes seguir para cambiar con éxito entre ramas:

- Los cambios en tu rama actual deben confirmarse o guardarse antes de cambiar.
- La rama que deseas verificar debe existir en tu local.

También hay un comando de acceso directo que te permite crear y cambiar a una rama al mismo tiempo:

`git checkout -b <name-of-your-branch>`

Este comando crea una nueva rama en su local (-b significa rama) y marca la rama como nueva justo después de que se haya creado, si observas bien, el cambio se encuentra en -b).

Comandos (status)

El comando de estado de Git nos brinda toda la información necesaria sobre la rama actual

`git status`

Podemos recopilar información acerca de:

- Si la rama actual está actualizada
- Si hay algo que necesita un commit, un add, o borrarse
- Si hay archivos preparados, sin preparar o sin seguimiento
- Si hay archivos creados, modificados o eliminados

Comandos (add)

Cuando creamos, modificamos o eliminamos un archivo, estos cambios ocurrirán en nuestro local y no se incluirán en la próxima confirmación (a menos que cambiemos las configuraciones).

Necesitamos usar el comando git add para incluir los cambios de un archivo(s) en nuestro próximo commit.

Para agregar un solo archivo:

`git add <file>`

Para añadir todo de una vez:

`git add -A`

Cuando vimos la captura de pantalla anterior en la cuarta sección, vimos que hay nombres de **archivos que se encontraban en rojo, lo que significa que son archivos sin preparar**. Los archivos no preparados no se incluirán en sus confirmaciones. Para incluirlos, necesitamos usar git add.

Importante: el comando git add no cambia el repositorio y los cambios no se guardan hasta que usamos git commit.

Comandos (commit)

Este es quizás el comando más utilizado de Git. Una vez que llegamos a cierto punto en el desarrollo, queremos guardar nuestros cambios (tal vez después de una tarea o problema específico).

Un **commit** es un conjunto de cambios en los archivos que hemos dado por buenos y que queremos almacenar como una instantánea de cara al futuro. Los commits se relacionan unos con otros en una o varias secuencias para poder ir viendo la historia de un determinado archivo a lo largo del tiempo. Es el concepto central de todo sistema de control de código.

Git commit es como establecer un punto de control en el proceso de desarrollo al que puede volver más tarde si es necesario. **También necesitamos escribir un mensaje corto** para explicar lo que hemos desarrollado o cambiado en el código fuente.

`git commit -m "commit message"`

Importante: Git commit guarda tus cambios solo localmente

Comandos (push)

Después de confirmar los cambios (con git commit), lo siguiente que hay que hacer es enviar estos cambios al servidor remoto. Git push sube tus confirmaciones al repositorio remoto.

`git push <remote> <branch-name>`

Sin embargo, si tu rama se creó recientemente, también debes cargar la rama con el siguiente comando:

`git push --set-upstream <remote> <name-of-your-branch>`

O bien:

`git push -u origin <branch_name>`

Importante: Git push solo carga los cambios que están confirmados.

Comandos (pull)

El comando git pull se usa para obtener actualizaciones del repositorio remoto. Este comando es una **combinación de git fetch y git merge**, lo que significa que, cuando usamos git pull, obtienes las actualizaciones del repositorio remoto (git fetch) e inmediatamente aplica los últimos cambios en su local (git merge). (En simples palabras, sirve para traer el repositorio remoto a tu repositorio local).

`git pull <remote>`

Esta operación puede causar conflictos que debes resolver manualmente

Comandos (revert)

A veces necesitamos deshacer los cambios que hemos hecho. Hay varias formas de deshacer nuestros cambios de forma local o remota (depende de lo que necesitemos), pero debemos usar estos comandos con cuidado para evitar eliminaciones no deseadas.

Una forma más segura de deshacer nuestras confirmaciones es usando git revert. Para ver nuestro historial de confirmaciones, primero debemos usar

`git log --oneline`

Luego, solo necesitamos especificar el código hash junto a nuestro commit que nos gustaría deshacer

`git revert <commit-id>`

El comando Git revert deshará la confirmación dada, pero creará una nueva confirmación sin eliminar la anterior.

La ventaja de usar git revert es que no toca el historial de commits. Esto significa que aún puede ver todas las confirmaciones en su historial, incluso las revertidas. Otra medida de seguridad aquí es que todo sucede en nuestro sistema local a menos que los insertemos en el repositorio remoto. Es por eso que git revert es más seguro de usar y es la forma preferida de deshacer nuestros commits.

Comandos (merge)

Cuando hayas completado el desarrollo en tu rama y todo funcione bien, el paso final es fusionar la rama con la rama principal (dev o master branch). Esto se hace con el comando git merge.

Git merge básicamente integra su rama de características (feature branch) con todas sus confirmaciones en la rama dev (o master). Es importante recordar que primero debes estar en la rama específica que deseas fusionar con tu rama de características.

Por ejemplo, cuando deseas fusionar tu rama de características en la rama develop:

Primero debes cambiar a la rama develop:

`git checkout develop`

Antes de hacer la fusión, debes actualizar la rama de desarrollo local:

`git fetch`

Finalmente, puedes hacer la fusión:

`git merge <branch-name>`

Sugerencia: asegúrate que tu rama de desarrollo tenga la última versión antes de fusionar las ramas, de lo contrario, puede enfrentar conflictos u otros problemas no deseados

Comandos (cherry-pick)

git cherry-pick es un potente comando que permite que las confirmaciones arbitrarias de Git se elijan por referencia y se añadan al actual HEAD de trabajo.

La ejecución de cherry-pick es el acto de elegir una confirmación de una rama y aplicarla a otra. git cherry-pick puede ser útil para deshacer cambios.

`git cherry-pick <commit-id>`

Por ejemplo, supongamos que una confirmación se aplica accidentalmente en la rama equivocada. Puedes cambiar a la rama correcta y ejecutar cherry-pick en la confirmación para aplicarla a donde debería estar.

Comandos (tag)

El objeto tipo etiqueta (tag) es muy parecido al tipo commit. Su principal diferencia reside en que generalmente apunta a una confirmación de cambios (commit) en lugar de a un árbol (tree). Es como una referencia a una rama, pero permaneciendo siempre inmóvil, apuntando siempre a la misma confirmación de cambios, dando un nombre más amigable a esta.

Crear un nuevo tag:

```
git tag <nombre-tag>
```

Listar tags:

```
git tag -l
```

¿A que se refiere origin?

Origin es un nombre abreviado para el repositorio remoto desde el que se clonó originalmente un proyecto. Más precisamente, se utiliza en lugar de la URL del repositorio original y facilita su referencia.

Por lo tanto, para enviar los cambios al repositorio remoto, puedes usar cualquiera de los siguientes comandos:

`git push origin nombre-de-la-rama`

o

`git push https://github.com/nombre-de-usuario/nombre-del-repositorio.git nombre-de-la-rama`

¿Y HEAD?

El concepto de HEAD se refiere al **commit en el que está tu repositorio posicionado en cada momento**.

Por regla general HEAD suele coincidir con el último commit de la rama en la que estés, ya que habitualmente estás trabajando en lo último. Pero si te mueves hacia cualquier otro commit anterior entonces el HEAD estará más atrás.

Stash

El área de Stash no es más que un **espacio en paralelo en el que podemos guardar los cambios que tengamos sin commitear en nuestro working tree**, es decir, cuando hagamos un stash lo que pasará es que volveremos al estado de nuestro último commit y los cambios que hubiera se habrán archivado en la 'pila' de stash

Esto resulta interesante cuando necesitamos **dejar temporalmente a un lado nuestros cambios sin necesidad de commitearlos antes de llegar a un punto estable**, o simplemente cuando queremos cerciorarnos de que cualquier error detectado se está produciendo por ese código añadido desde el último commit

Si ejecutamos el comando **git stash list** veremos que se pueden guardar múltiples estados de nuestro código en la lista de stashes. Ojo, que aunque hablamos en términos de pila de elementos y veamos comandos como **git stash pop**, sí que podremos acceder a cualquier stash intermedio

Pull request

Las pull requests son una funcionalidad que **facilita la colaboración entre desarrolladores** que usan GitHub, GitLab, Bitbucket, etc. Ofrecen una interfaz web intuitiva para debatir los cambios propuestos antes de integrarlos en el proyecto oficial.

En su forma más sencilla, las solicitudes de incorporación de cambios (pull request) **son un mecanismo para que los desarrolladores notifiquen a los miembros de su equipo que han terminado una función**. Una vez la rama de función está lista, el desarrollador realiza la solicitud de incorporación de cambios mediante su cuenta de la plataforma en cuestión. Así, todas las personas involucradas saben que pueden o deben revisar el código y fusionarlo con la rama main.

Pero la solicitud de incorporación de cambios **es mucho más que una notificación**: es un foro especializado para debatir sobre una función propuesta. Si hay algún problema con los cambios, los miembros del equipo pueden publicar feedback en las solicitudes de incorporación de cambios e incluso modificar la función al enviar confirmaciones de seguimiento. El seguimiento de toda esta actividad se realiza directamente desde la solicitud de incorporación de cambios.

GUI Clientes

<https://git-scm.com/download/gui/linux>

Workflows en Git

Una “estrategia de branching” es una **serie de reglas** que aplica un equipo de desarrollo de software cuando **necesita escribir código** para incorporar una nueva funcionalidad o hacer una corrección, fusionarlo y enviarlo al repositorio donde se encuentra alojado el resto del código del software en uso.

Una estrategia de branching define como un equipo utiliza las branches para lograr un proceso de desarrollo concurrente, a través de un conjunto de reglas y convenciones que establecen:

- ¿Cuándo un desarrollador debe crear una branch?
- ¿De qué otra branch debe derivarse la nueva branch?
- ¿Cuándo debe el desarrollador hacer merge?
- ¿Y a qué branch debería hacer el merge?
- ...

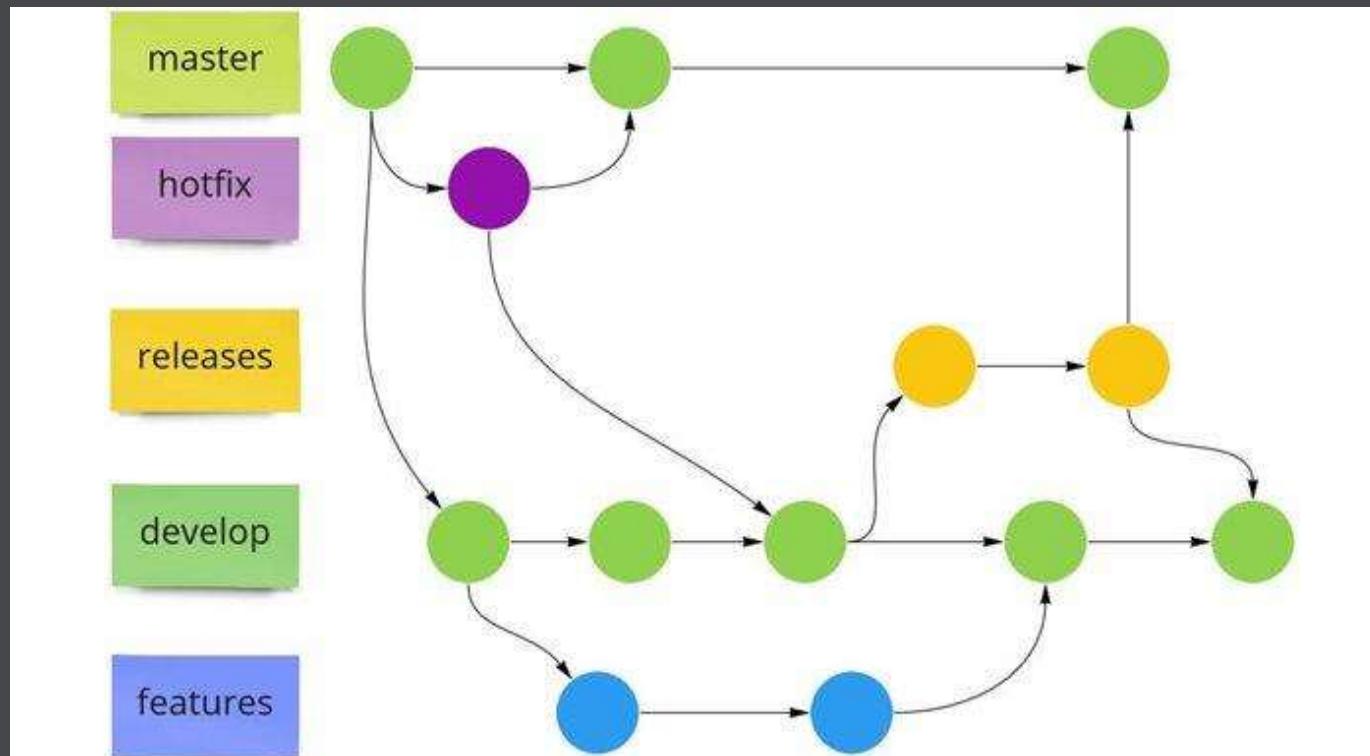
Por qué un Workflow en Git

Una estrategia de branching garantiza que **todos los miembros** del equipo sigan el **mismo proceso** para realizar cambios en código. La estrategia **correcta** mejora la colaboración, la eficiencia y la precisión en el proceso de entrega de software, mientras que la estrategia **incorrecta** (o ninguna estrategia) conduce a muchos errores y problemas de integración, que se traducen en horas de trabajo, mucho esfuerzo y sobre todo pérdida de tiempo y dinero.

Para reducir estos problemas, una estrategia de branching debe buscar:

- Permitir a los desarrolladores optimizar su productividad.
- Habilitar el desarrollo en paralelo.
- Permitir un conjunto de releases estructurados y planificados.
- Proporcionar una ruta clara para promover los cambios del software al entorno de producción.
- Evolucionar y adaptarse a los cambios que se realizan a diario en el código.
- Admitir múltiples versiones de software.
- ...

Workflows (Git Flow)



Workflows (Git Flow)

Es el flujo de trabajo más popular y extendido. Se basa en dos ramas principales con una vida infinita. Para cada tarea que se le asigna a un desarrollador se crea una rama feature en la cual se llevará a cabo la tarea. Una vez que ha finalizado, realizará un pull request (validación) contra develop para que validen el código.

Pasamos a detallar las dos ramas principales que se utilizan:

- **Master**: Contiene el **código de producción**. Todo el código de desarrollo, a través del uso de releases, se mergea (fusiona) en esta rama en algún momento.
- **Develop**: Contiene código de pre-producción. Cuando un desarrollador finaliza su feature, lo mergea contra esta rama.

Durante el ciclo de desarrollo, se usan varios tipos de ramas para dar soporte:

- **Feature**: Por cada tarea que se realiza, se crea una nueva rama para trabajar en ella. Esta rama parte de develop.
- **Hotfix**: Parte de master. Rama encargada de corregir una incidencia crítica en producción.
- **Releases**: Parte de develop. Rama encargada de generar valor al producto o proyecto. Contiene el código que se desplegará, y una vez que se han probado las features integradas en la release, se "mergeará" a la rama master.

Workflows (Git Flow)

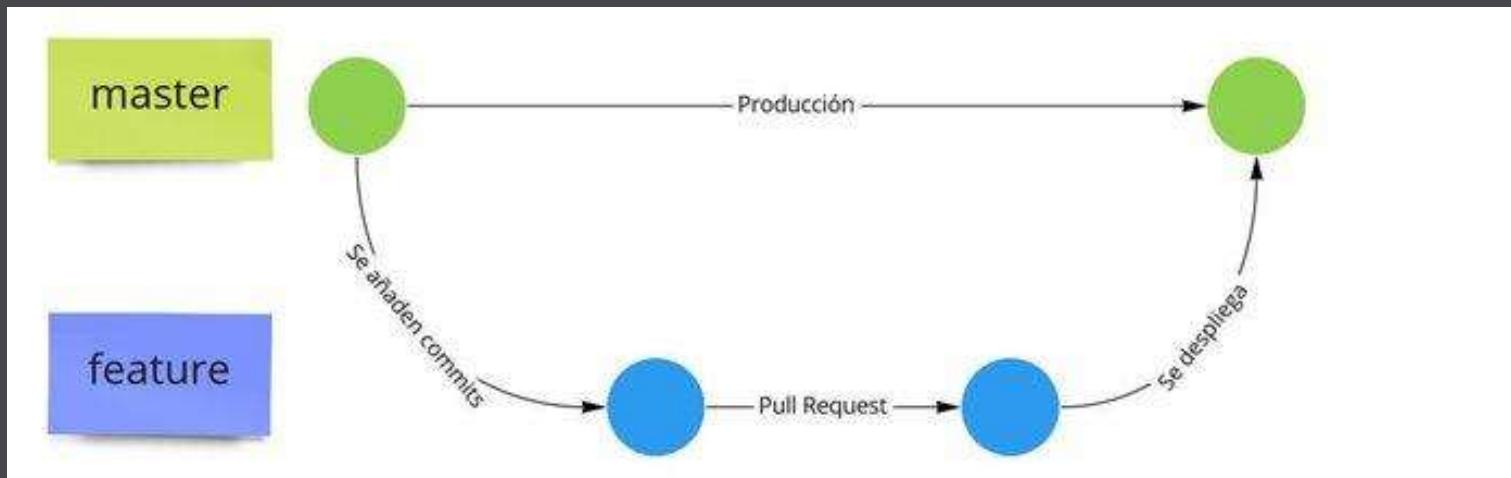
Pros

- Fácil de comprender el flujo de ramas.
- Ideal para **productos estables** que no requieren de desplegar cambios inmediatamente.
- Muy recomendable cuando el equipo tiene todo tipo de desarrolladores. El control de las features más la release hace que el código no se deteriore.
- Perfecto para productos **open-source** en los que pueden colaborar todo tipo de desarrolladores.

Contras

- No es el más indicado si tu proyecto necesita iterar muy rápido y subir a producción varias veces al día o semana.
- En caso de que el proyecto utilice varias herramientas de integración continua, la rama **develop** puede convertirse en una **rama redundante de master**.
- El uso de la rama master como rama protegida. Muchas herramientas de automatización usan la rama master por defecto.
- Gran complejidad en las ramas creadas de hotfix y releases. La integración continua elimina la necesidad de la creación de estas ramas, facilitando el despliegue.

Workflows (GitHub Flow)



Workflows (GitHub Flow)

La diferencia principal con GitFlow es la **desaparición de la rama develop**. Se basa en los siguientes principios:

- Todo lo que haya en la **rama master debe ser desplegado**.
- Para cualquier característica nueva, crearemos una rama de master, usando un nombre descriptivo.
- Debemos hacer commit en esta rama en local y hacer push con el mismo nombre en el server.
- Si necesitamos feedback, utilizaremos las herramientas de mergeo como **pull request**.
- Una vez revisado el código, podemos mergear contra master.
- Una vez mergeado contra master, debemos desplegar los cambios.

Workflows (GitHub Flow)

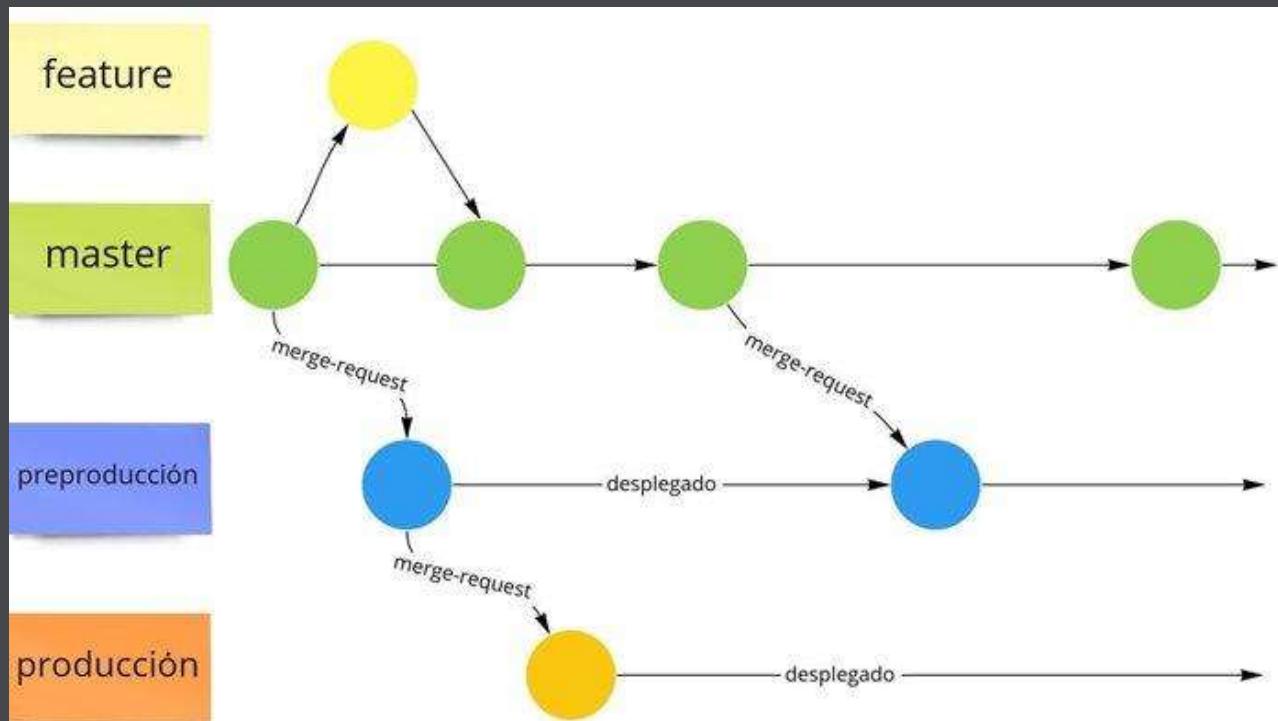
Pros

- Útil y amigable con las actuales herramientas de CI/CD.
- Recomendado para **features de duración corta** (diarias o incluso de horas).
- Flujo ligero y recomendado si el proyecto requiere de una **entrega de valor constante**.

Contras

- **Inestabilidad de master** si no se utilizan las herramientas de testing/PR correctamente.
- No recomendado para múltiples entornos productivos.
- Dependiendo el producto, podemos tener **restricciones de despliegues**, sobre todo en **aplicaciones SaaS**.

Workflows (GitLab Flow)



Workflows (GitLab Flow)

GitLab Flow es una alternativa/extensión de GitHub Flow y Git Flow, que nace debido a las carencias que adoptan estos dos flujos. Mientras que una de las consignas de GitHub es que todo lo que haya en master es desplegado, hay ciertos casos en los que no es posible cumplirlo o no se necesita. Por ejemplo: aplicaciones iOS cuando pasan a la App Store Validation o incluso tener ventanas de despliegue por la naturaleza del cliente.

El flujo propone utilizar master, ramas features y ramas de entorno. Una vez que una feature está finalizada hacemos una **merge request contra master**. Una vez que master tiene varias features, llevamos a cabo una **merge request a preproducción** con el conjunto de features anteriores, que a su vez, son candidatas de **pasar a producción haciendo otro merge request**. De esta manera conseguimos que el código subido a producción sea muy estable, ya que validamos features tanto a nivel individual como en lote.

La naturaleza de este flujo no requiere generar ramas de releases, ya que cada entorno será desplegado con cada merge request aceptada.

Workflows (GitLab Flow)

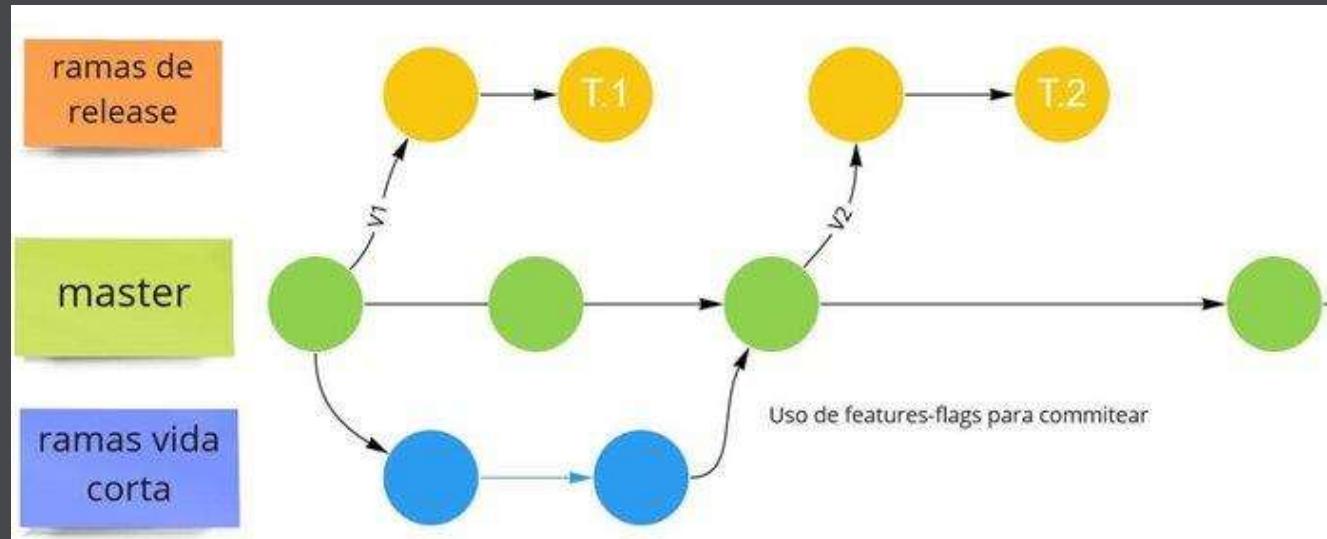
Pros

- Mucha confianza en la **versión de producción**.
- Ciclo de desarrollo muy seguro. Se revisa el código tanto a nivel individual en la feature, como a nivel global al pasar a preproducción o producción, mitigando el impacto.
- Previene el “overheap” de crear releases, taggings, merge a develop.

Contras

- Requiere de un equipo que valide las MR tanto de las features, como de los diferentes entornos.
- **Tiempo demasiado alto** para la entrega de valor. Desde que se crea y se valida la feature, hasta que llega a producción, tiene que pasar por muchas validaciones.
- **Más complejo** que GitHub Flow.

Workflows (Trunk-based Flow)



Workflows (Trunk-based Flow)

Este flujo es muy similar a GitHub Flow, con la característica nueva de las releases branch y el **cambio de filosofía** que presenta. Los principios que rigen este flujo son los siguientes:

- Los desarrolladores debemos colaborar siempre sobre el trunk (o master).
- Bajo ningún concepto debemos crear features branch utilizando documentación técnica. En caso de que la evolución sea compleja, haremos uso de **features flags** (condicionales en el código) para activar o desactivar la nueva característica.
- Preferiblemente usaremos la metodología **Pair-Programming** en vez de hacer uso de PR.

Se sacan ramas de release para poder desplegar el código en diferentes entornos, ya sea mobile, web, etc.

Workflows (Trunk-based Flow)

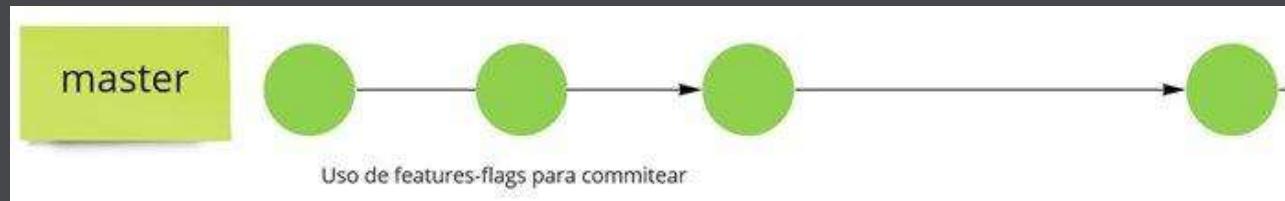
Pros

- Muy útil si nuestro proyecto necesita iterar rápido y entregar valor lo antes posible.
- Responde muy bien a proyectos agile pequeños.
- Preparado para equipos que utilizan **pair-programming**.
- Funciona muy bien con un **equipo experimentado y cerrado**.

Contras

- Debemos ser responsables de nuestro código y hacer el esfuerzo de subir **código de calidad**.
- El proyecto debe tener QA y CD muy maduro, de lo contrario se introducirán muchos **bugs** en la rama trunk.
- No es recomendable utilizarlo en proyectos **open-source**, ya que requieren de una verificación extra de código.

Workflows (Master-only Flow)



Workflows (Master-only Flow)

El flujo de trabajo usa solo una rama infinita. Usaremos master en esta descripción, ya que probablemente sea el nombre más común y ya es una convención de Git, pero también puedes usar, por ejemplo, current, default, mainline, ...

Realizaremos cada **feature o hotfix** sobre la **misma rama**, testearemos y faremos commit **en local**. Una vez que este cambio se haya aprobado, faremos push contra master en origin, desplegándose en producción inmediatamente.

Este flujo está orientado a **proyectos con desarrolladores muy experimentados** y en el cual se fomente el pair-programming.

Debemos usar **features-flags** para poder **integrar el código en la rama master** y evitar conflictos a lo largo del tiempo.

Workflows (Master-only Flow)

Pros

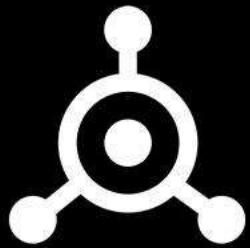
- Solo mantenemos una rama.
- Tendremos el histórico del proyecto limpio.
- Con el uso de pair-programming y desarrolladores experimentados, **la entrega de valor en producción es inmediata.**

Contras

- No soporta múltiples entornos productivos.
- Requiere de **desarrolladores experimentados** para no dejar inestable la rama principal.
- El proyecto requiere de **guidelines de código muy estrictas.**

¿Cuál es el mejor workflow?

Ten en cuenta tu proyecto, ciclos de release y equipo...
Inspírate en los workflows existentes...
...y crea tu propio modelo!



KEEP
CALM
DEVCENTER
IS
WORKING
ON IT