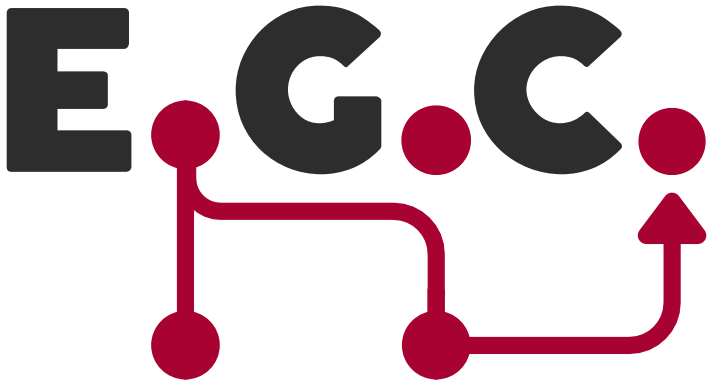


Grado en Ingeniería Informática – Ingeniería del Software

Evolución y Gestión de la Configuración

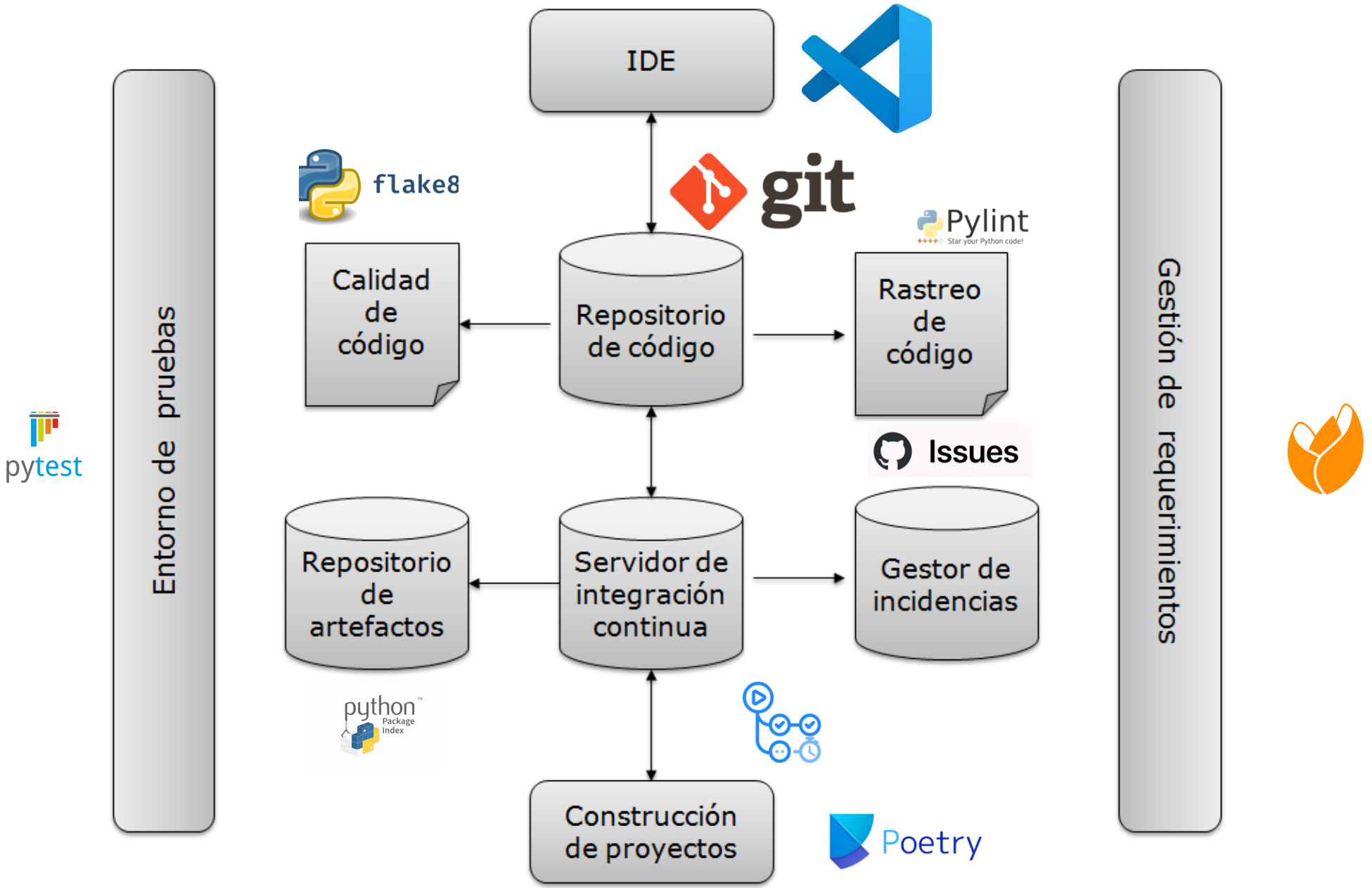
Escuela Técnica Superior de Ingeniería Informática

Tema 2: Gestión del Código Fuente

Objetivo principal: saber organizar el código fuente y usar un repositorio de código de manera eficiente pensando en la automatización

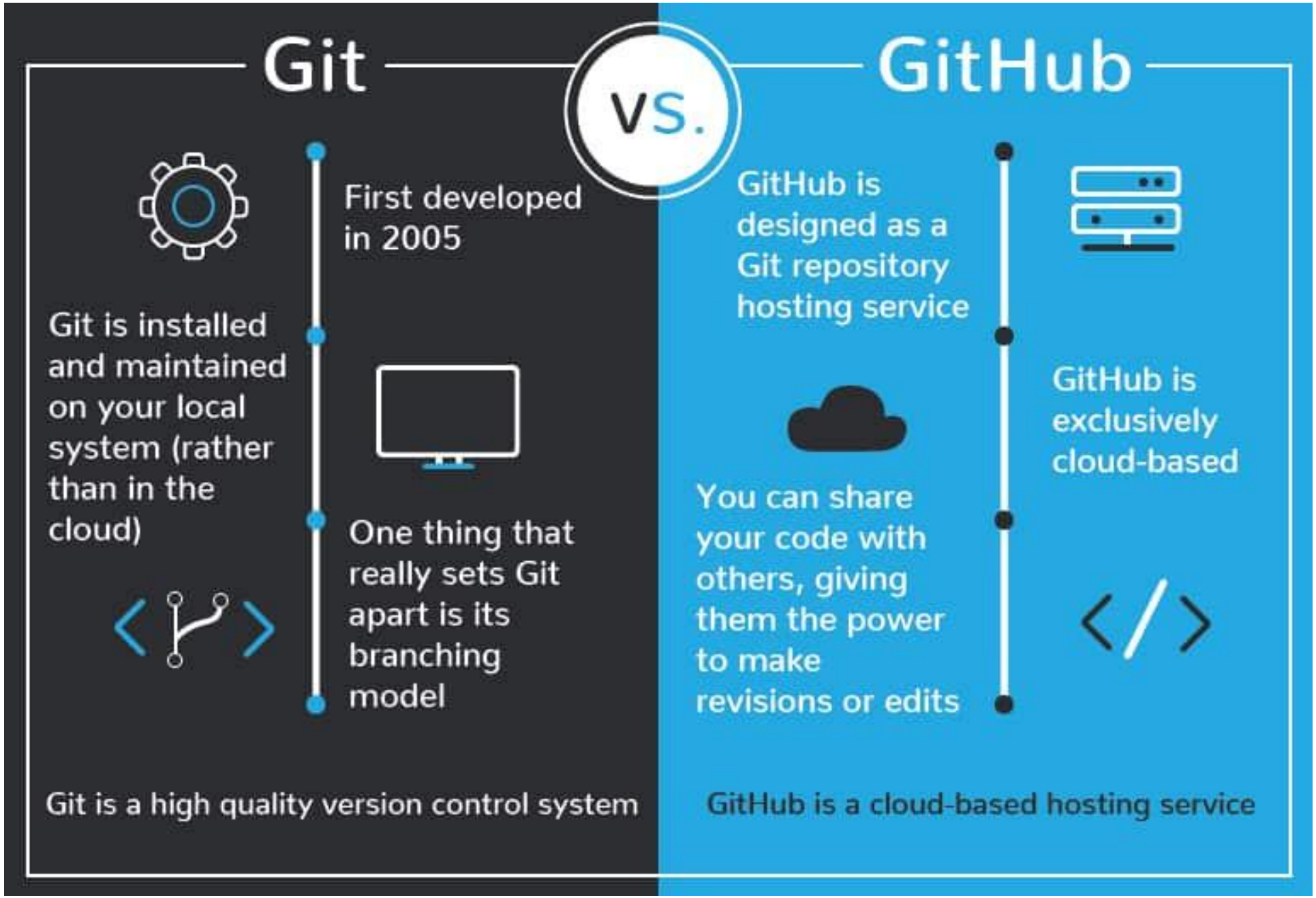
1. **Conceptos básicos**
2. Operaciones
3. Uso de repositorios
4. Resumen
5. Bibliografía

1. Conceptos básicos | ecosistemas de desarrollo



1. Conceptos básicos | git y servicios de git

¿Cuál es la diferencia entre una tecnología de repositorios de código (e.g. git) y un servicio de alojamiento de repositorios (e.g. gitlab)?


Git
vs.
GitHub
First developed in 2005
Git is installed and maintained on your local system (rather than in the cloud)
One thing that really sets Git apart is its branching model
GitHub is designed as a Git repository hosting service
GitHub is exclusively cloud-based
You can share your code with others, giving them the power to make revisions or edits
Git is a high quality version control system
GitHub is a cloud-based hosting service

- Software
- Control de versiones
- Mantenido por Linux
- Open Source
- Sin gestión de usuarios
- Instalación local
- Configuración mínima de herramientas externas
- Muy pocos competidores

- Servicio
- Alojamiento (hosting) de repositorios git
- Membresía gratuita o de pago
- Gestión de usuarios integrada
- Alojado en la web
- Mercado activo para la integración de herramientas
- Muchos competidores

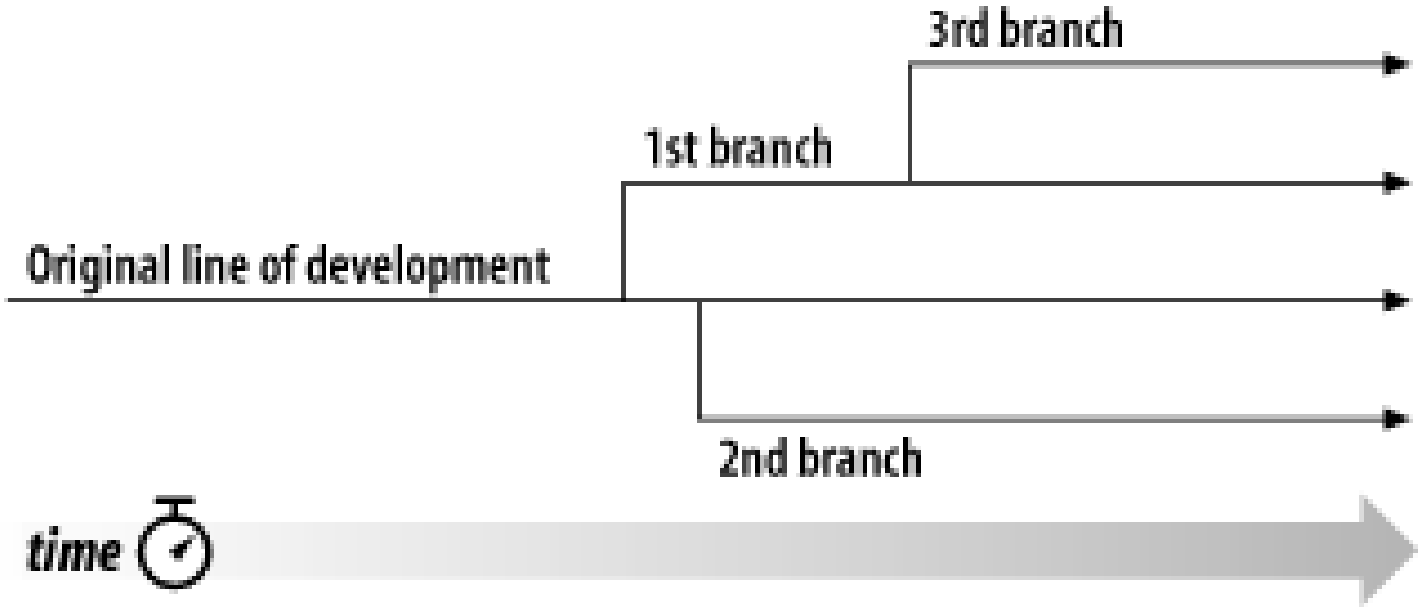
1. Conceptos básicos | repositorios distribuidos

- ❌ No hay repositorio central (aunque por convención suele haber uno “principal”)
- 💻 Cada usuario tiene su repositorio en local
- ✅ Los *commits* son locales
- 🔄 Nuevas operaciones:
  - **Push** → enviar cambios al remoto
  - **Pull** ← traer cambios del remoto
- **Rebasing:** permite cambiar la “máquina del tiempo” del repositorio local para empaquetar los cambios en un solo commit con objeto de enviarlo al repositorio remoto. ⚠️ Peligroso si no se tiene mucha destreza ⚠️

1. Conceptos básicos | branch

Branch 🌿 : una línea de desarrollo **independiente** que puede compartir parte de historia común con otras ramas:

- **Rama padre** → la que origina otra
- **Rama hija** → creada a partir de la anterior



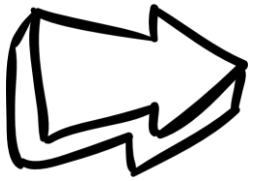
1. Conceptos básicos | contenido del repositorio

- Además del código fuente podemos guardar:
  - Pruebas, documentos, ficheros de configuración.
- **Excepciones**:
  - Los binarios
  - Los ficheros de configuración de entornos locales
  - Los archivos de log

Hay que hacer uso de .gitignore

1. Conceptos básicos
2. **Operaciones**
3. Uso de repositorios
4. Resumen
5. Bibliografía

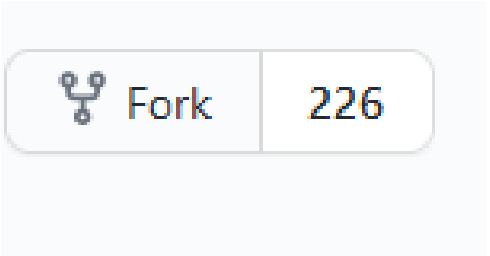
2. Operaciones | tipos



- De creación
- De escritura
- De lectura
- De ramas

2. Operaciones | operaciones de creación

Fork



$> Clone

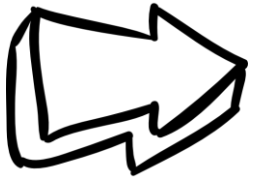
```
$> git clone https://github.com/EGCETSII/decide.git
```

$> Init

Cada miembro del equipo debe hacer commit con **un solo nombre de usuario**

```
$> git config --global user.name "Your Name"
$> git config --global user.email you@example.com
$> git init
```

2. Operaciones | tipos



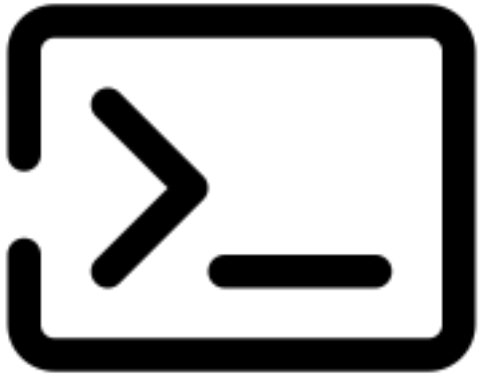
- De creación
- De escritura
- De lectura
- De ramas

2. Operaciones | operaciones de escritura

Sandbox
(sin seguimiento)

Parte de la historia
(con seguimiento)

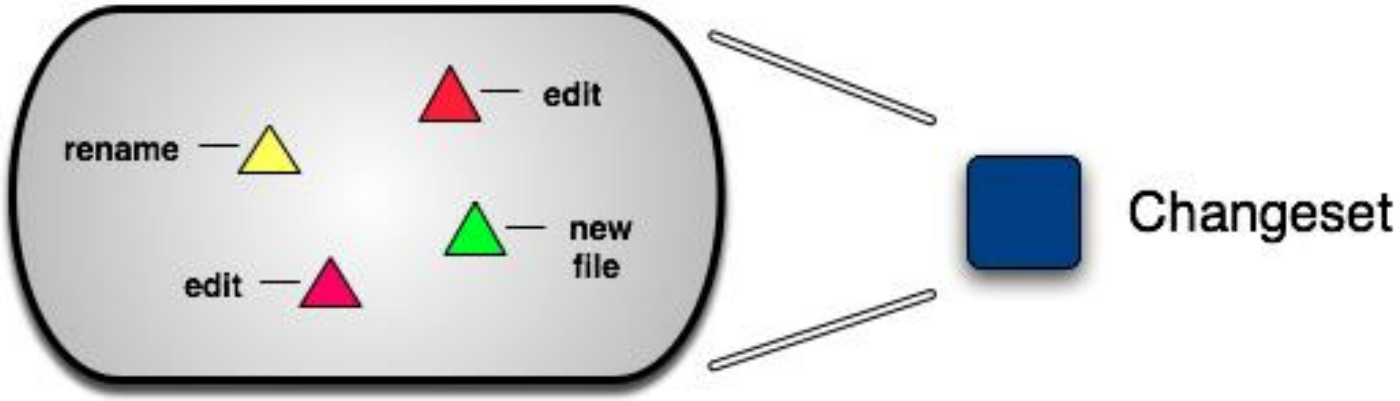
Ignorado



En EGC trabajaremos GIT con la línea de comandos

2. Operaciones | conjunto de cambios

`Changeset`: conjunto de cambios que deben ser tratados como un conjunto indivisible (en svn se conoce como "`revision`", en git como "`commit`").



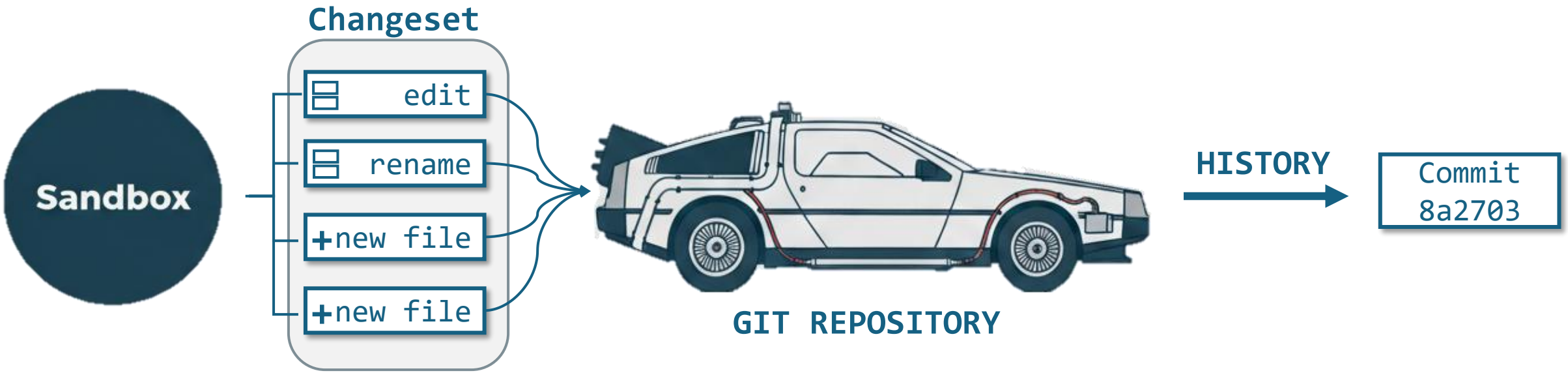
OJO: en git "commit" se usa tanto para una operación como para nombrar al conjunto de cambios

2. Operaciones | de escritura

$> add: añadir un elemento a la “máquina del tiempo” de modo que su estado y versiones sean controlados por la misma (con seguimiento).

$> commit: guardar en la “máquina del tiempo” un conjunto de changesets. Pueden ser atómicos o no.

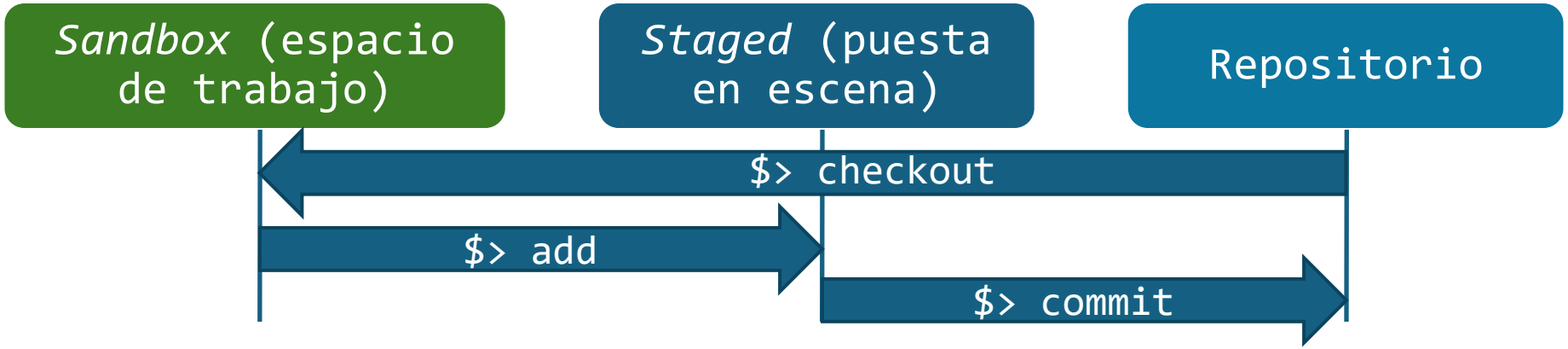
Lo ideal es tener `commits` atómicos



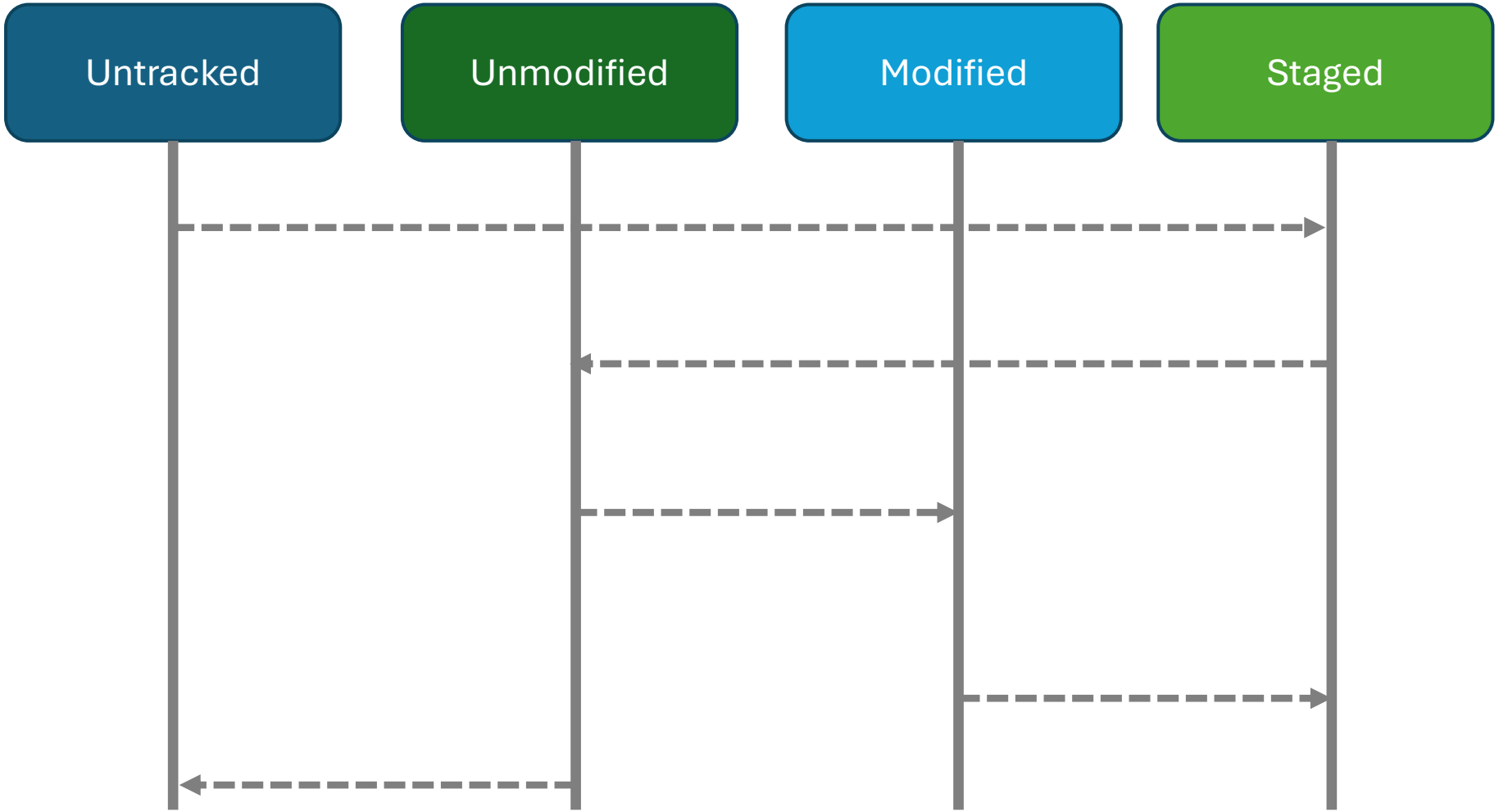
2. Operaciones | de escritura

¿Qué hay en un commit?

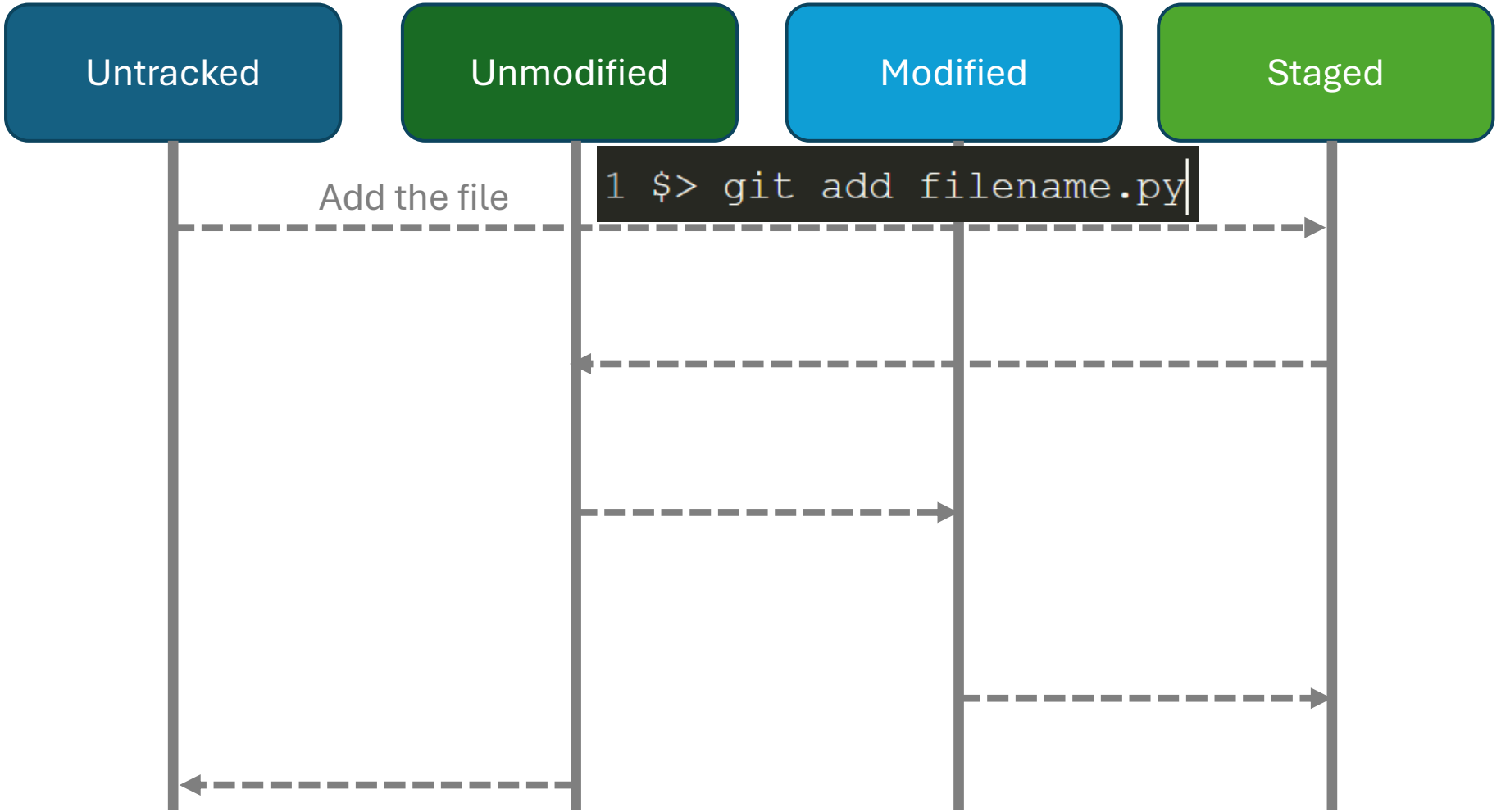
- Datos modificados + metadatos
- Un fichero *rastreado (*con seguimiento*)* puede estar en tres estados (en el caso de **`git`**):
  - Modificado (*modified*)
  - Preparado (*staged*)
  - Confirmado (*committed*)



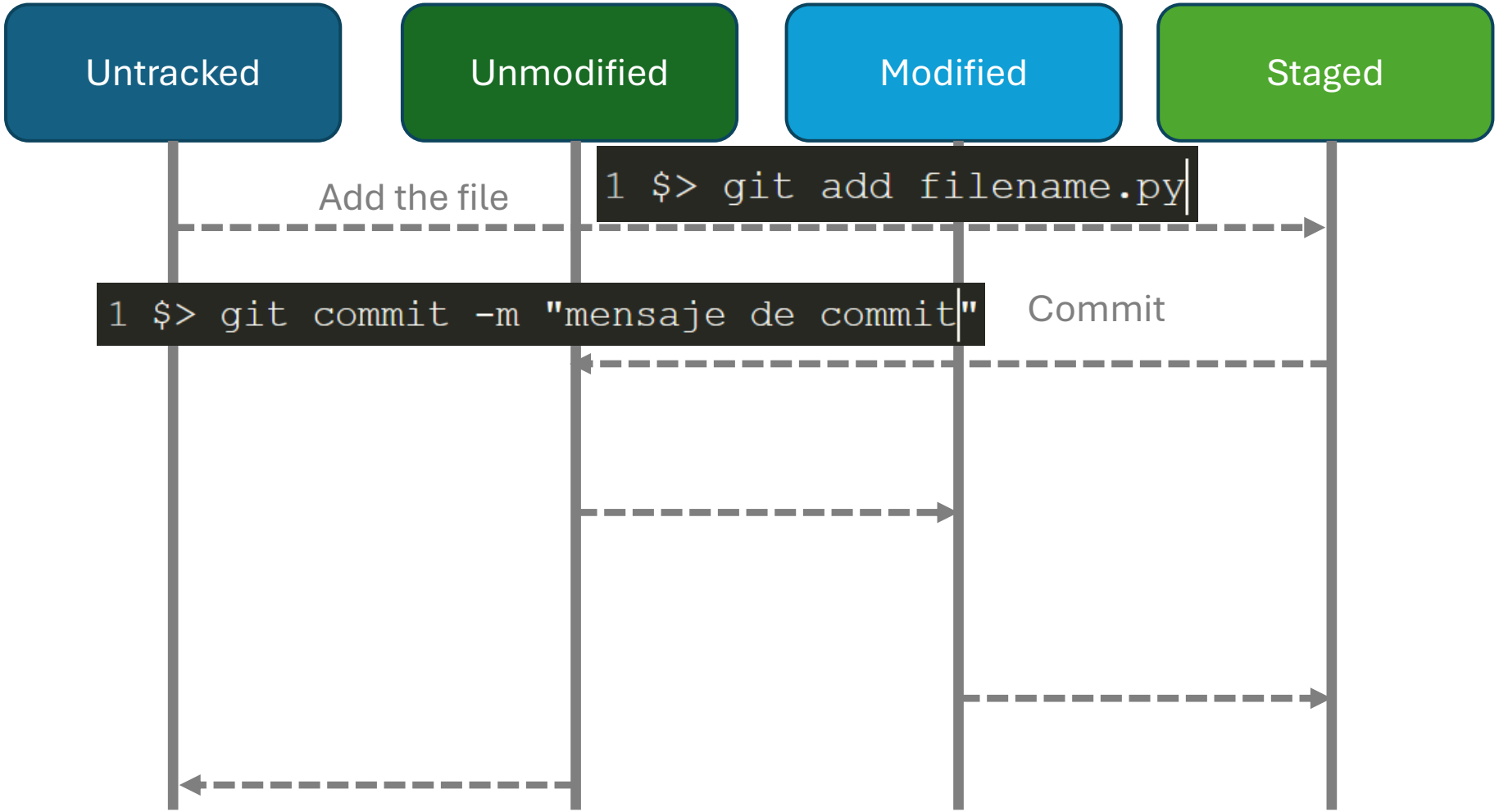
2. Operaciones | flujo frecuente



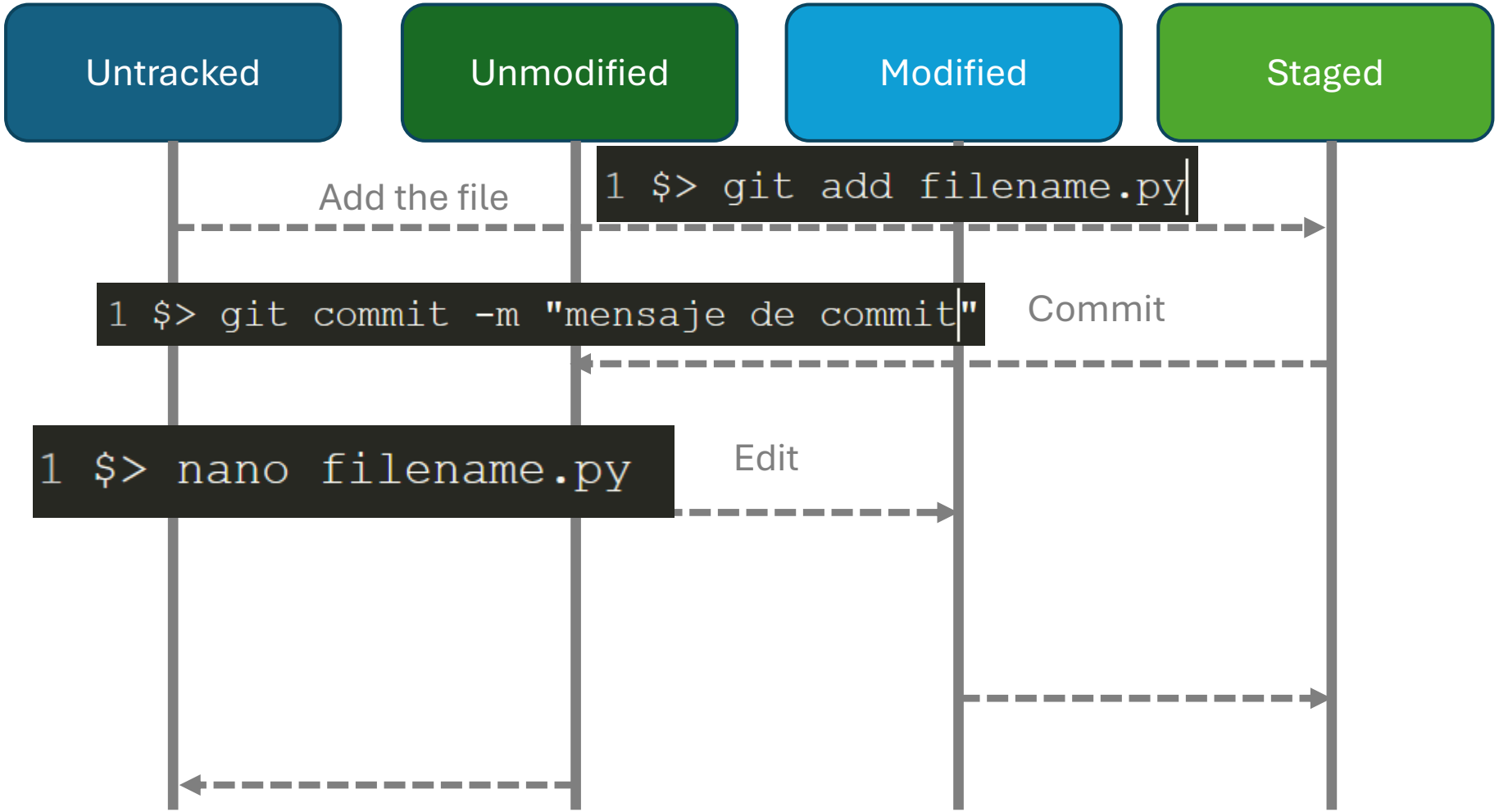
2. Operaciones | flujo frecuente



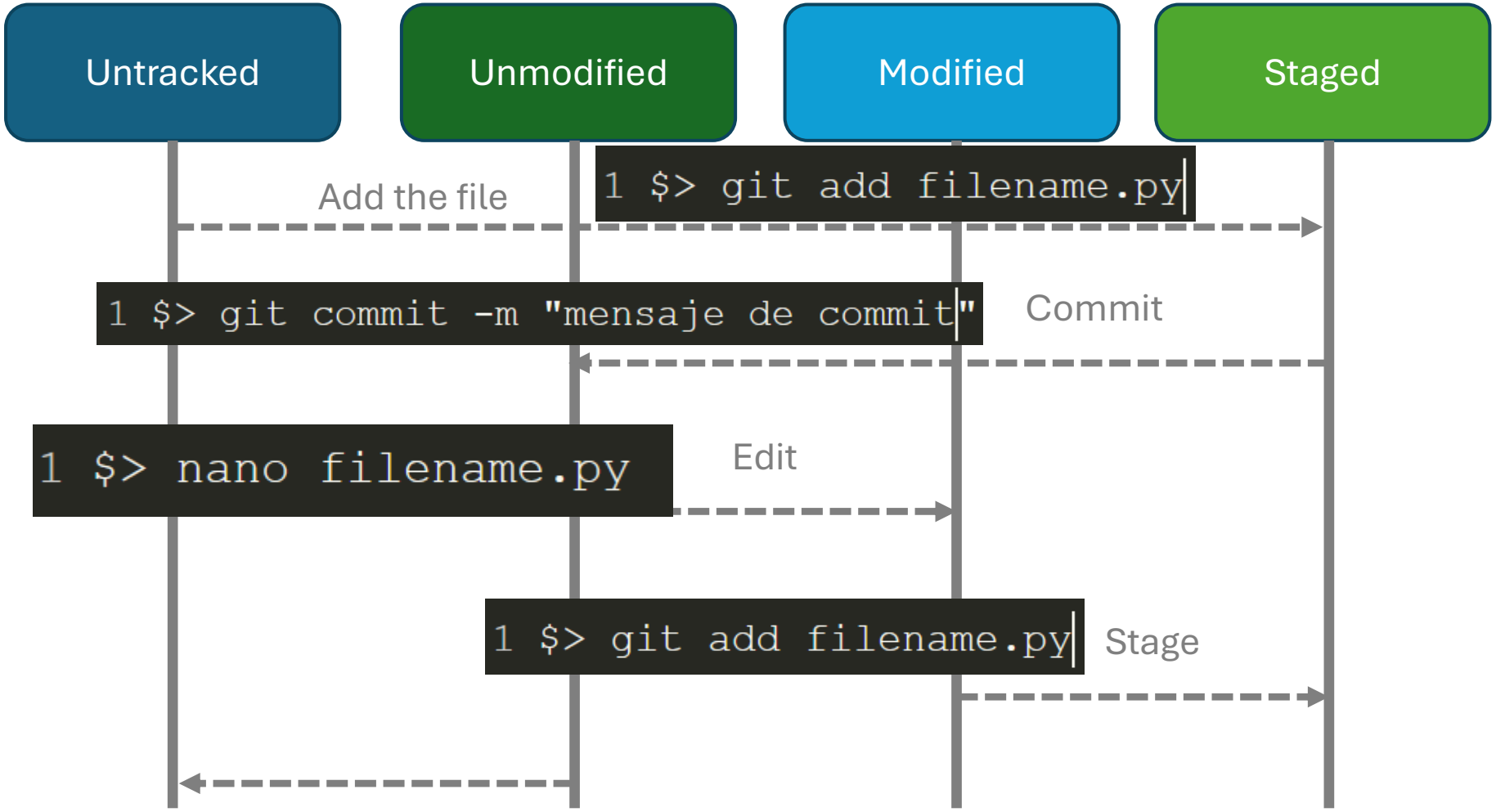
2. Operaciones | flujo frecuente



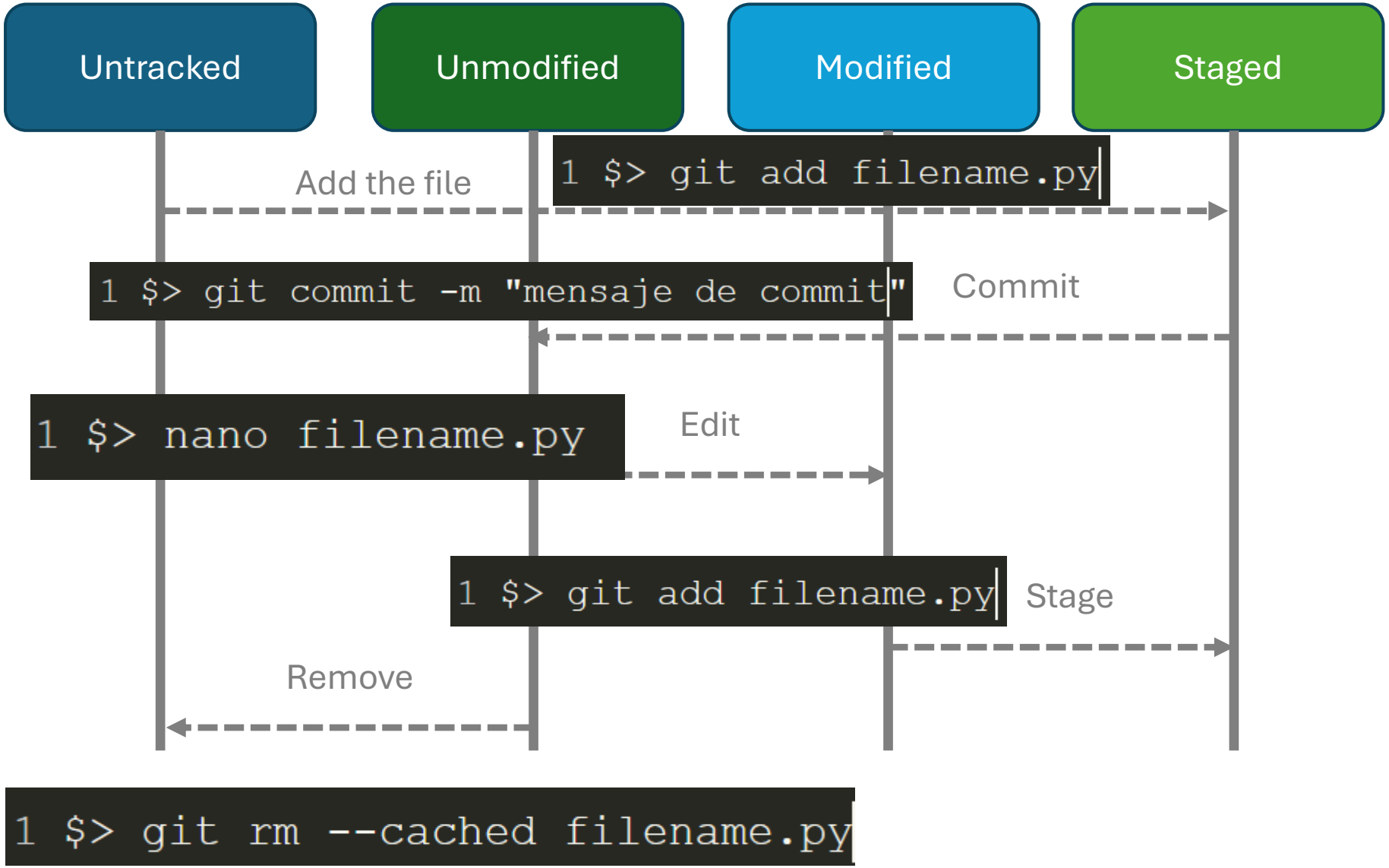
2. Operaciones | flujo frecuente



2. Operaciones | flujo frecuente



2. Operaciones | flujo frecuente

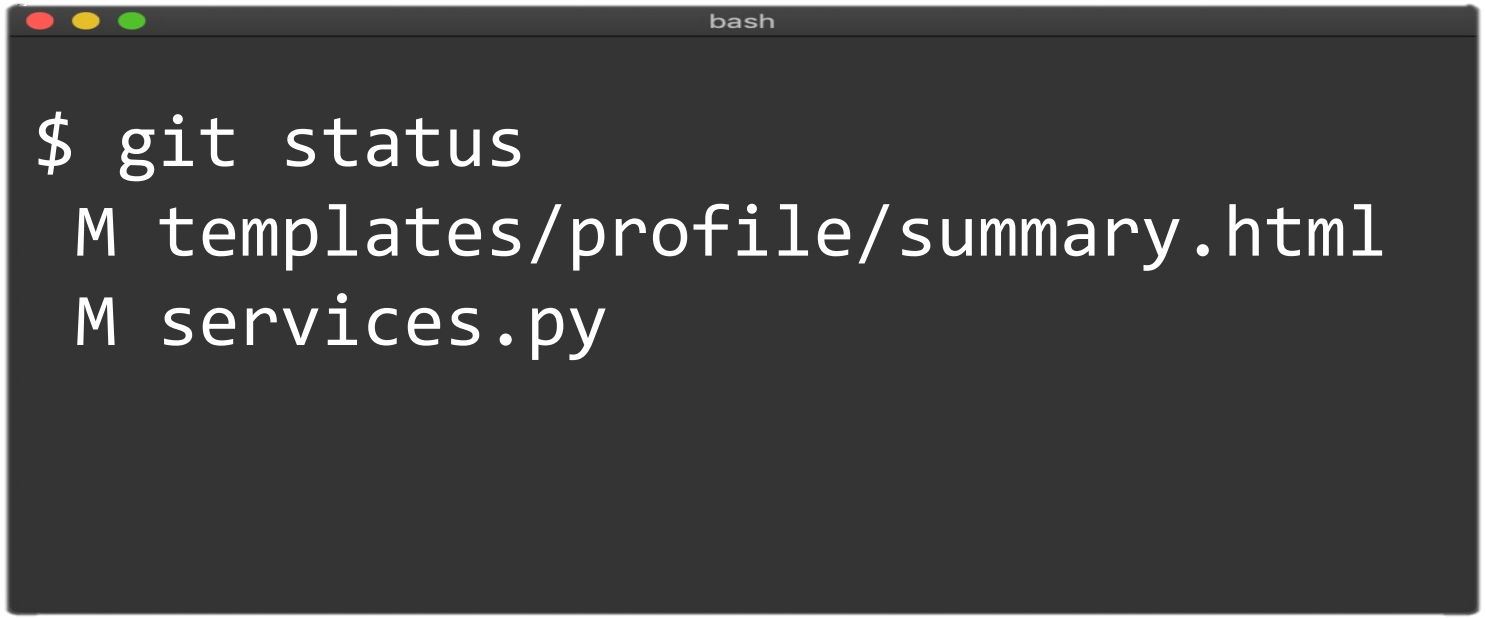


2. Operaciones | commits atómicos

¿Por qué lo ideal es tener `commits` **atómicos**?

2. Operaciones | commits atómicos

Tengo varios cambios hechos que afectan a varios ficheros. Si un cambio afecta a un conjunto de ficheros y otro cambio afecta a otro conjunto de ficheros distinto ¿cómo hago para hacer un commit atómico?



2. Operaciones | commits atómicos

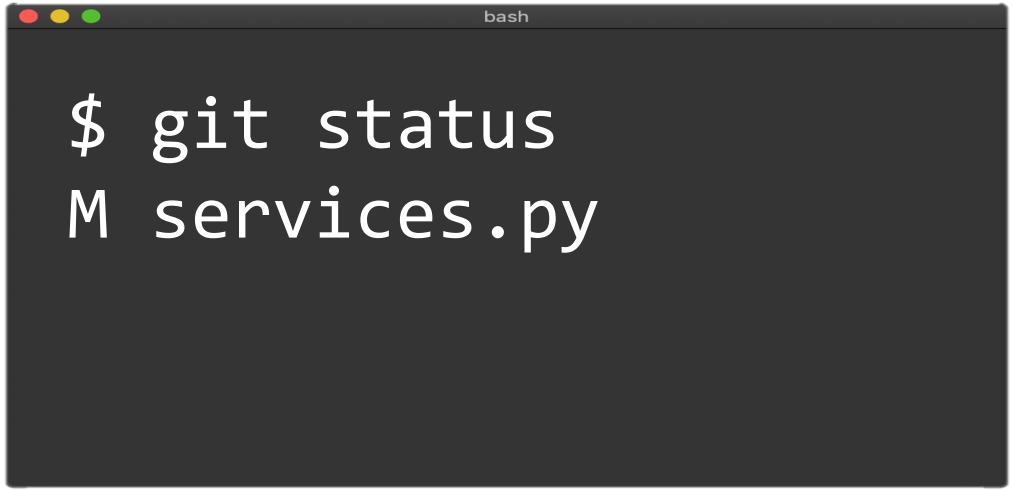
bash

```
$ git status
 M templates/profile/summary.html
 M services.py

$ git add templates/profile/summary.html

$ git status
 A templates/profile/summary.html
 M services.py

$ git commit -m "Add div HTML elements"
```

2. Operaciones | commits atómicos

Ahora tengo varios cambios hechos sobre el mismo fichero pero que afectan a partes distintas,

¿puedo hacer commits atómicos?



2. Operaciones | commits atómicos

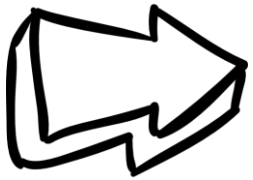
bash

```
$ git status
M services.py
$ git add -p

…

$ git diff --cached
$ git commit –m “…”
```

2. Operaciones | tipos

De creación

De escritura



De lectura

De ramas

2. Operaciones | de lectura

El concepto de **checkout** es tomar una versión de los artefactos del repositorio y llevarlo al área de trabajo (*sandbox*) para realizar modificaciones.

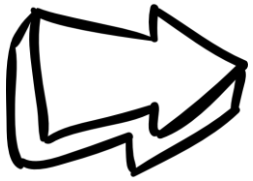
OJO: puede haber “juguetes” nuevos
OJO: en git, “checkout” es un comando histórico que puede cambiar de rama o restaurar contenido.

2. Operaciones | tipos

De creación

De escritura

De lectura



De ramas

2. Operaciones | de ramas - merge

El ciclo típico de mezclar ramas es:

I. Me pongo en la rama sobre la que quiero mezclar los cambios
II. Mezclar los cambios sobre la rama actual desde la rama origen (*source*)
III. Resolver conflictos
IV. Hacer commit del merge
V. Abortar el merge (en caso de fallo)

bash

```
$ git checkout main


$ git merge feature_x

…

$ git commit -m
"Resolve merge
conflicts"
$ git merge --abort
```

¿Se nos olvida algo?

2. Operaciones | de ramas - merge

El ciclo típico de mezclar ramas es:

I. Me pongo en la rama sobre la que quiero mezclar los cambios

II. Me aseguro de que la rama no está desactualizada

III. Mezclar los cambios sobre la rama actual desde la rama origen (*source*)

IV. Resolver conflictos

V. Hacer commit del merge

VI. Abortar el merge (en caso de fallo)

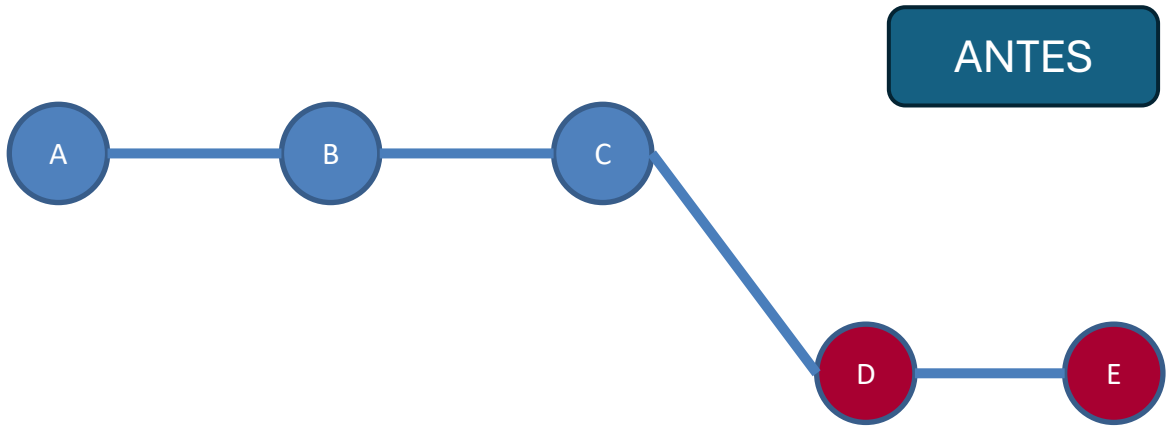
bash

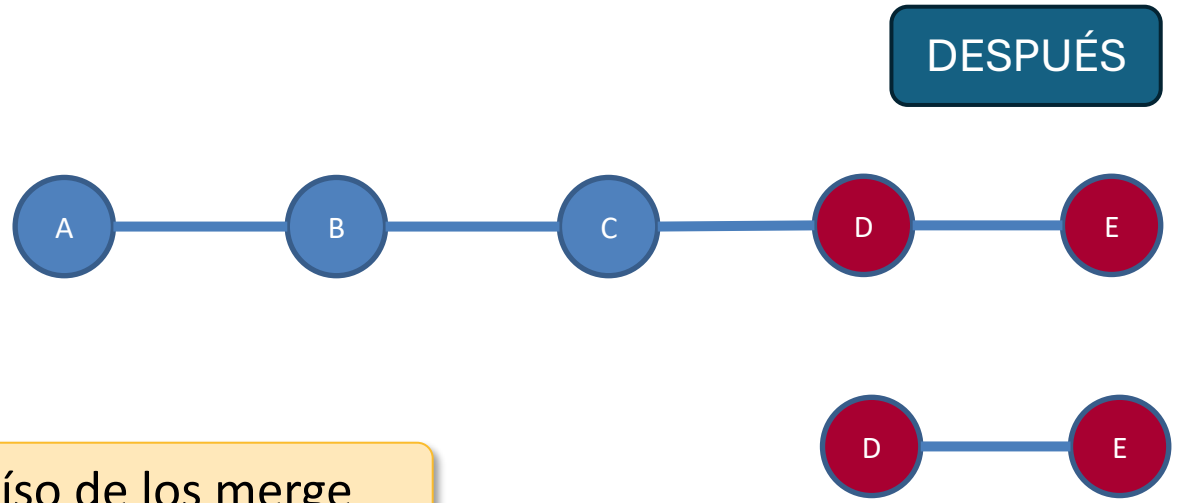
```
$ git checkout main

$ git pull origin main

$ git merge feature_x

…

$ git commit -m
"Resolve merge
conflicts"
$ git merge --abort
```

2. Operaciones | de ramas – tipos de merge

Fastforward

- Cuando la rama destino no ha tenido `commits`
- Git no hace un `commit` de `merge`
- La historia permanece lineal
- Es como si se hubiesen hecho `commits` sobre la rama destino



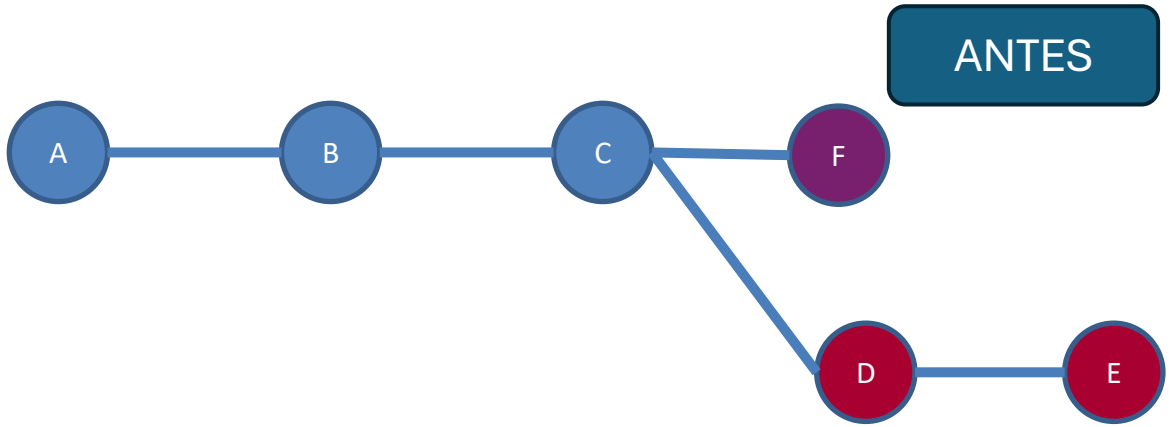


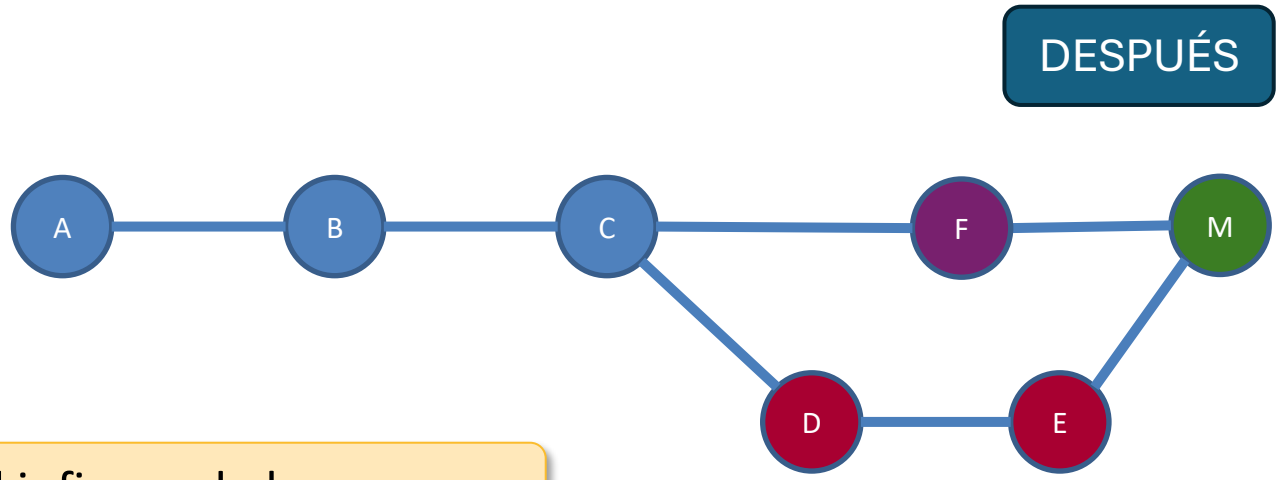
El paraíso de los merge

2. Operaciones | de ramas – tipos de merge

Recursive (3-way)

- Cuando la rama origen y destino han tenido `commits`
- Git hace un `commit` de `merge`
- La historia no es lineal





El infierno de los merge

2. Operaciones | conflictos

- **Resolución de conflictos:** cuando se hace *merge* pueden aparecer conflictos. Algunos pueden resolverse de manera automática, otros requieren intervención manual. ***Resolver conflictos es duro***.

- **Diff**. Hacer diff significa mirar la diferencia que hay entre artefactos de un repositorio.

- A partir de un *diff* se puede generar un ***patch***, es decir, un conjunto de cambios a realizar a un conjunto de artefactos. Cuando un conjunto de cambios se aplica a una serie de artefactos, se dice que se ha aplicado un patch.

* (ver http://es.wikipedia.org/wiki/Diff probar con https://www.diffchecker.com/)

1. Conceptos básicos
2. Operaciones
3. **Uso de repositorios**
4. Resumen
5. Bibliografía

3. Uso de repositorios | modelo de uso

Tenemos que definir el “***usage model***” o modo de uso de la/s herramienta/s:

¿Cómo se gestionan las ramas? ¿qué permisos se dan a los usuarios? ¿cómo se hacen los merge? ¿con qué frecuencia? ¿por qué motivos se crean las ramas? ¿cuándo se eliminan?, etc…

3. Uso de repositorios | commits

- Al establecer una guía de cómo y cuándo hacer `commit`:
  - Ordenamos y sistematizamos el trabajo
  - Permitimos generar `changelogs` automáticamente
  - Facilitamos la visualización y navegación de la historia del repositorio.
- Hay diversas guías en las que se puede basar el equipo para desarrollar la suya propia.

Format of the commit message

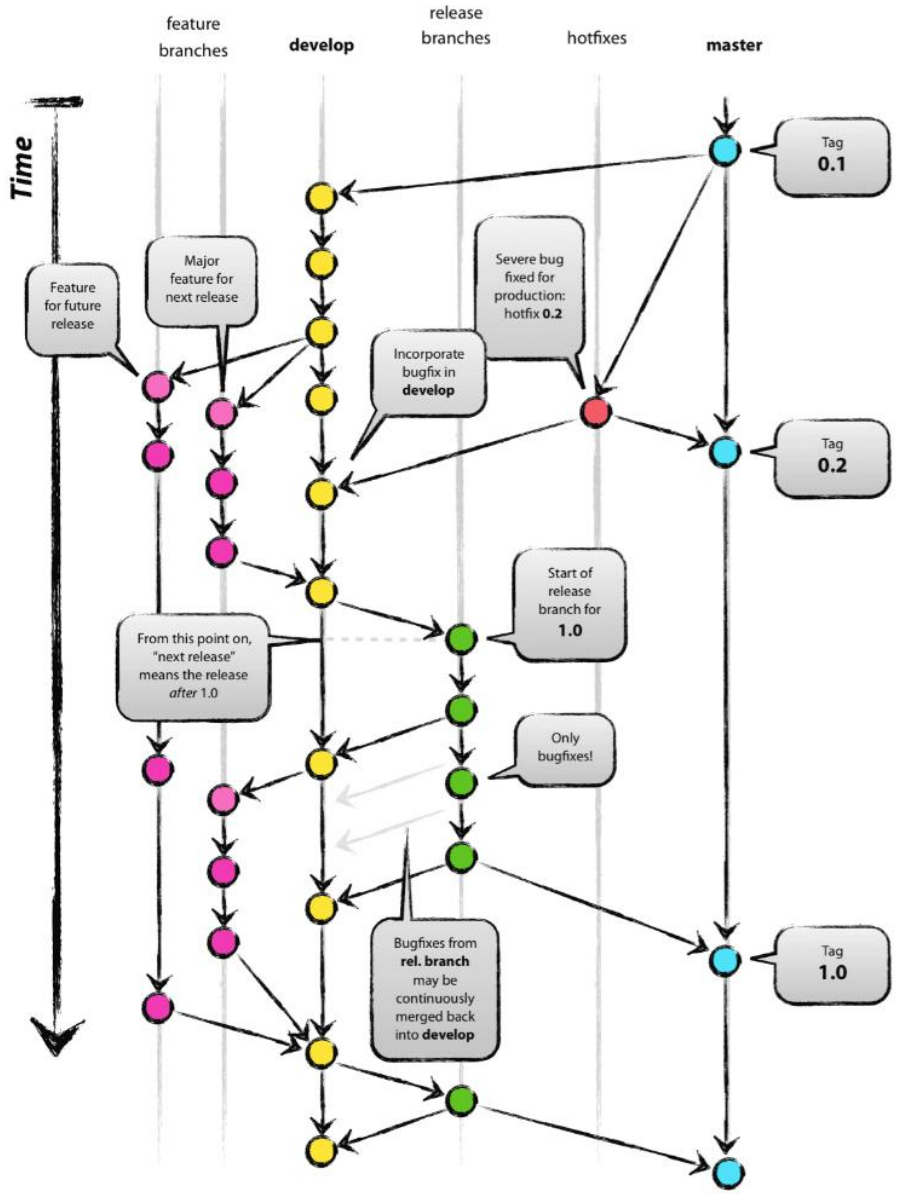
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>

http://karma-runner.github.io/0.10/dev/git-commit-msg.html

3. Uso de repositorios | principios básicos

- Todo lo que hay en `main` debe poder ser ejecutado/desplegado.
- Los `commits` deben ser gestionados correctamente siguiendo unas guías.
- Debe haber una relación entre `commits` y gestión de cambio/incidencias/`issues`.
- Se debe tener planificado y ser consciente de *cómo afecta el modelo de uso a las etapas de gestión de la construcción e integración continua*.

3. Uso de repositorios | gitflow



NO usar gitflow

http://nvie.com/posts/a-successful-git-branching-model/

3. Uso de repositorios I gitflow

Note of reflection (March 5, 2020)

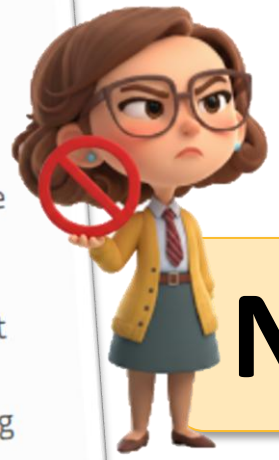
This model was conceived in 2010, now more than 10 years ago, and not very long after Git itself came into being. In those 10 years, git-flow (the branching model laid out in this article) has become hugely popular in many a software team to the point where people have started treating it like a standard of sorts — but unfortunately also as a dogma or panacea.

During those 10 years, Git itself has taken the world by a storm, and the most popular type of software that is being developed with Git is shifting more towards web apps — at least in my filter bubble. Web apps are typically continuously delivered, not rolled back, and you don't have to support multiple versions of the software running in the wild.

This is not the class of software that I had in mind when I wrote the blog post 10 years ago. If your team is doing continuous delivery of software, I would suggest to adopt a much simpler workflow (like GitHub flow) instead of trying to shoehorn git-flow into your team.

If, however, you are building software that is explicitly versioned, or if you need to support multiple versions of your software in the wild, then git-flow may still be as good of a fit to your team as it has been to people in the last 10 years. In that case, please read on.

To conclude, always remember that panaceas don't exist. Consider your own context. Don't be hating. Decide for yourself.



NO usar git flow

http://nvie.com/posts/a-successful-git-branching-model/

3. Uso de repositorios | otros modelos de uso

Modelo muy simple, GitHub lo promueve como estándar para proyectos que hacen **deploys frecuentes** (incluso varias veces al día).

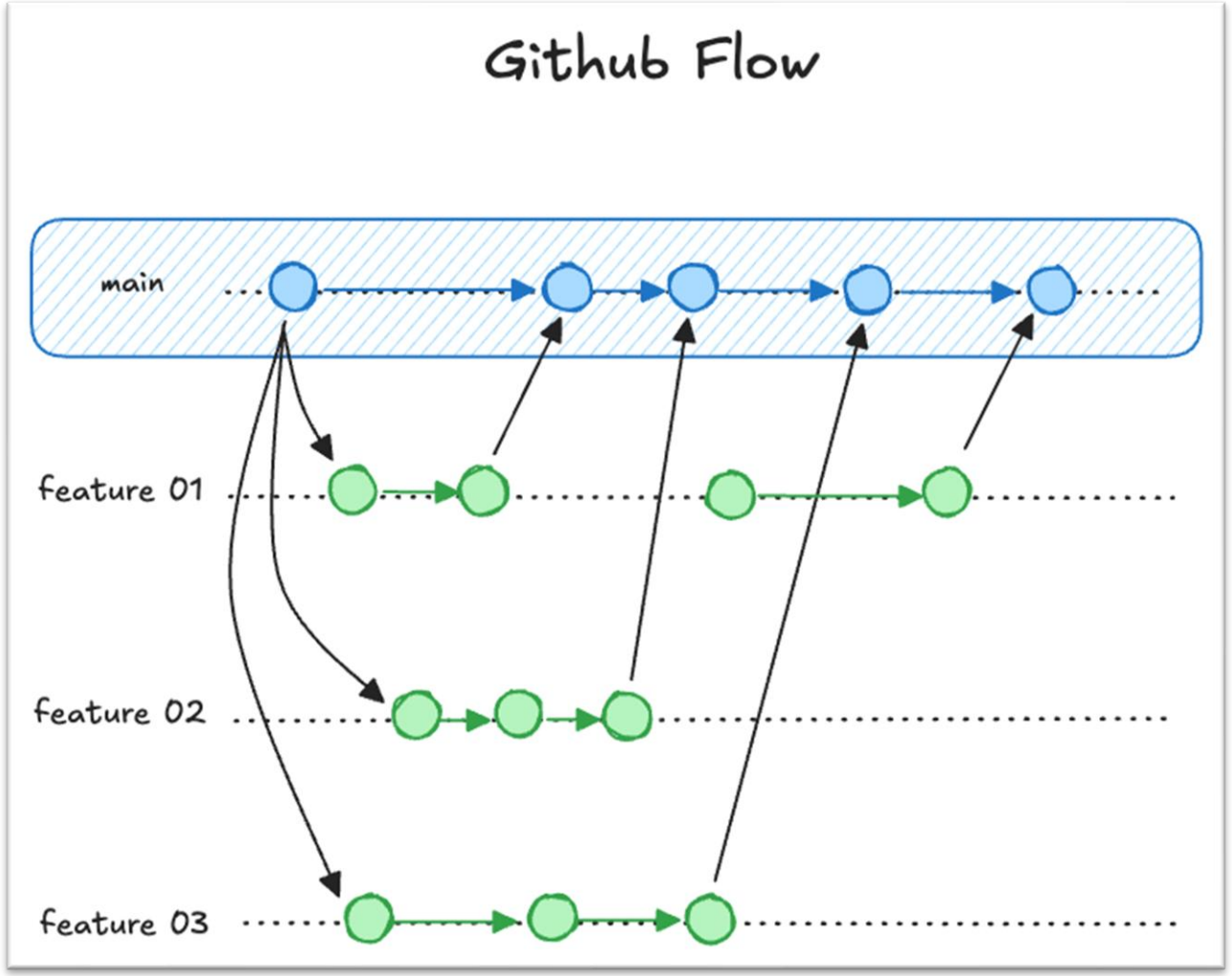
Características:

- Solo hay una rama principal: `main` (o `master`).
- Para cada `feature` o `fix`, se crea una rama corta: main → feature-xxx
- Cuando está lista, se abre un **`Pull Request (PR)`** para discutir y revisar.
- Si se aprueba, se hace `merge` a `main`.
- `main` siempre debe estar en un estado **desplegable**.

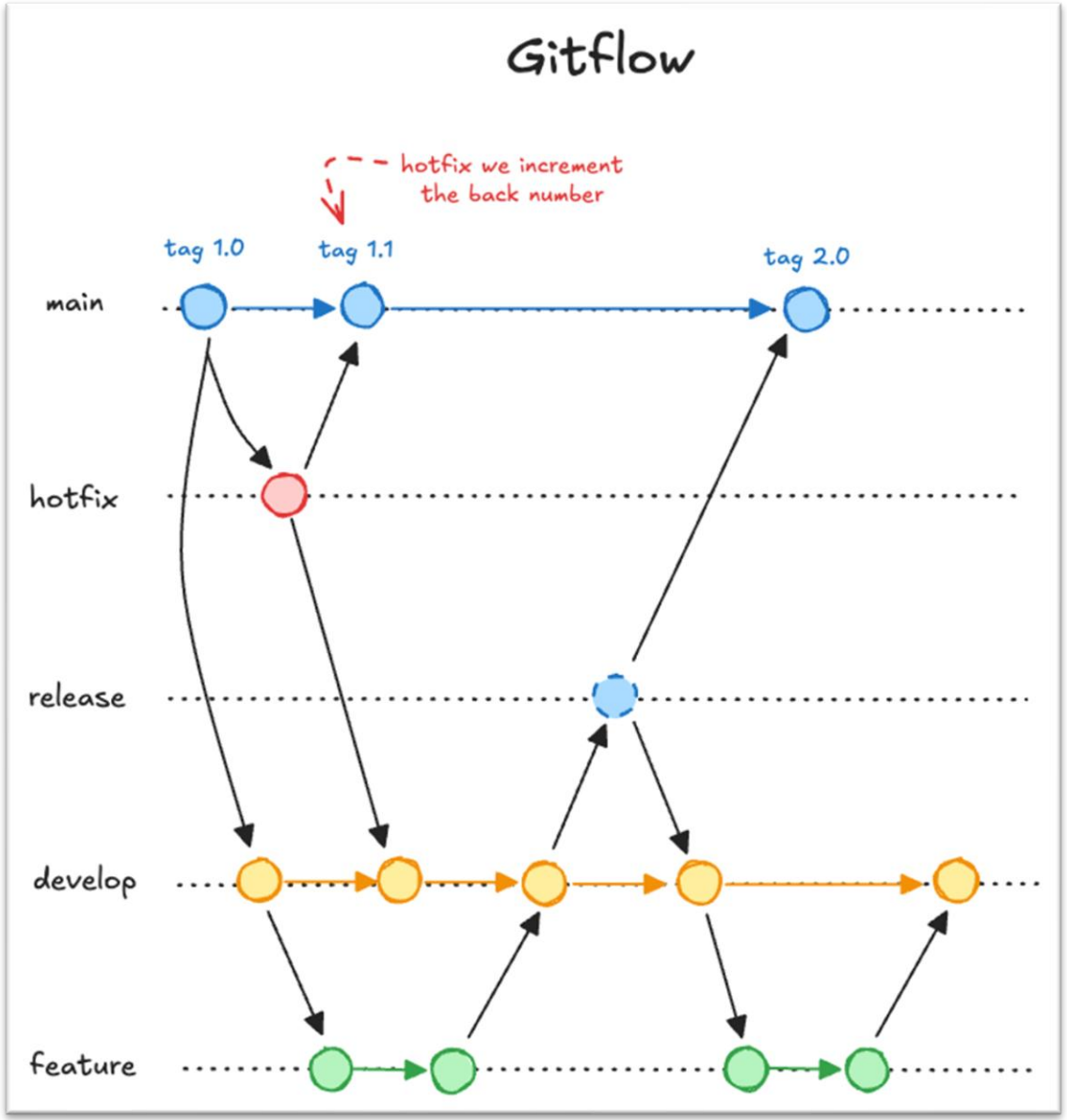
✅ **Ventajas**: simple, rápido, fomenta integración continua.

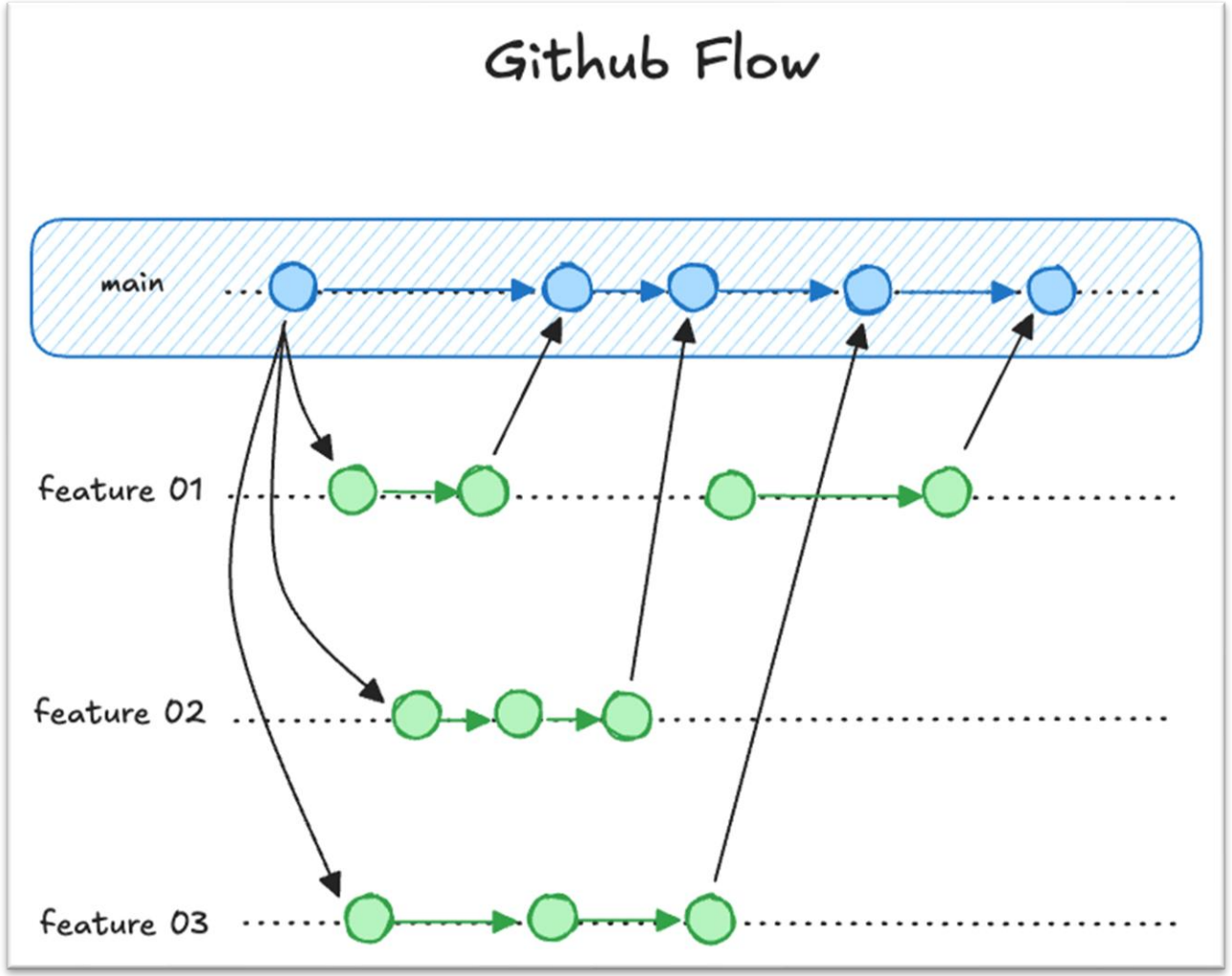
❌ **Desventajas**: no ideal para proyectos grandes con múltiples *releases* en paralelo.

❌ **Depende de PR**: PR no siempre está disponible y a veces genera cuellos de botella.



3. Uso de repositorios | otros modelos de uso



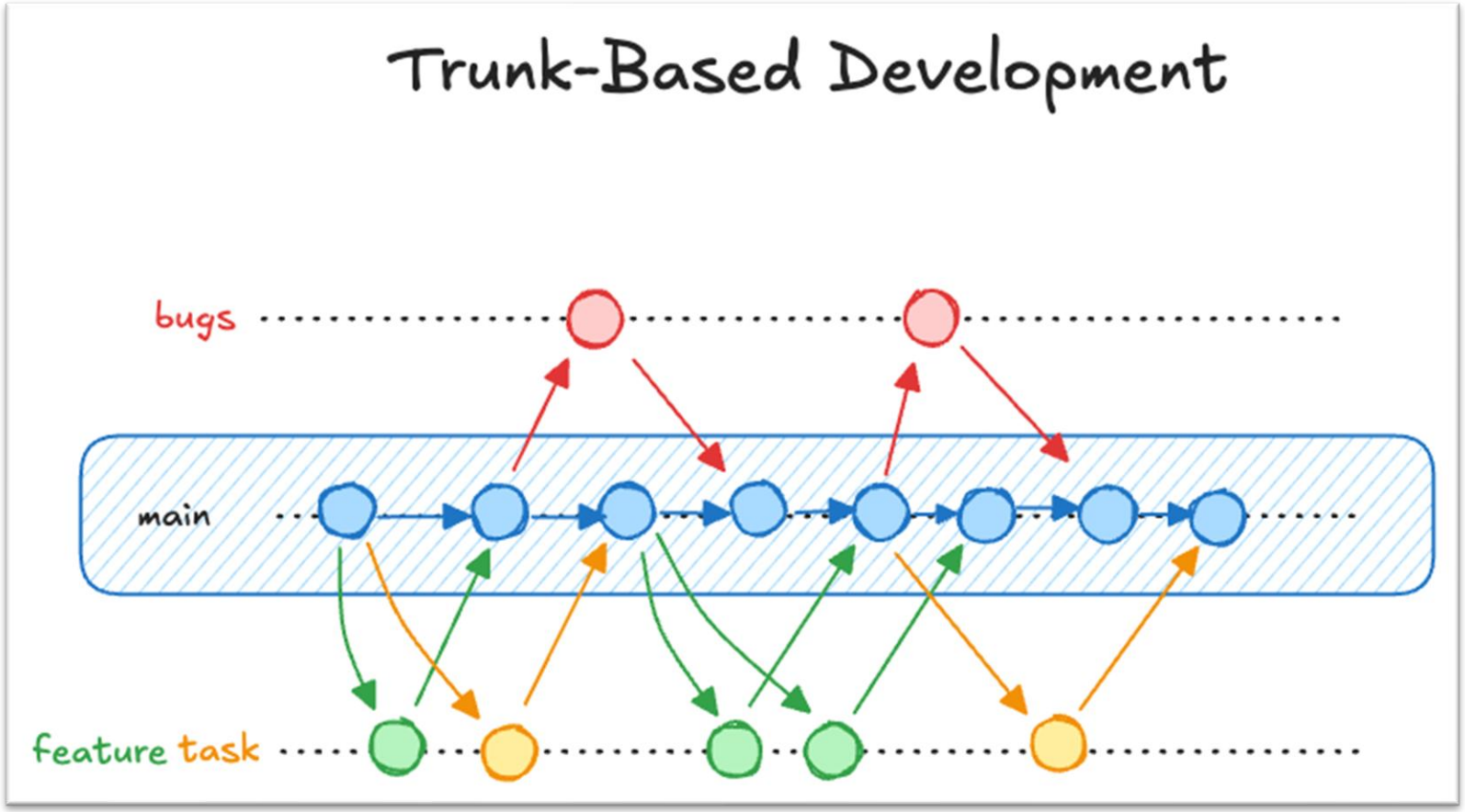


3. Uso de repositorios | otros modelos de uso

Es un modelo más moderno, usado en grandes compañías.

Características:

- Existe una única rama central: el trunk (`main`).
- Todos los devs hacen `commits` frecuentes (diarios, incluso varias veces al día) directamente en `main` o en ramas muy cortas (máx. 1-2 días).
- Se apoya en `feature flags` para esconder código incompleto sin bloquear integración.
- Se promueve la integración continua y los deploys constantes.

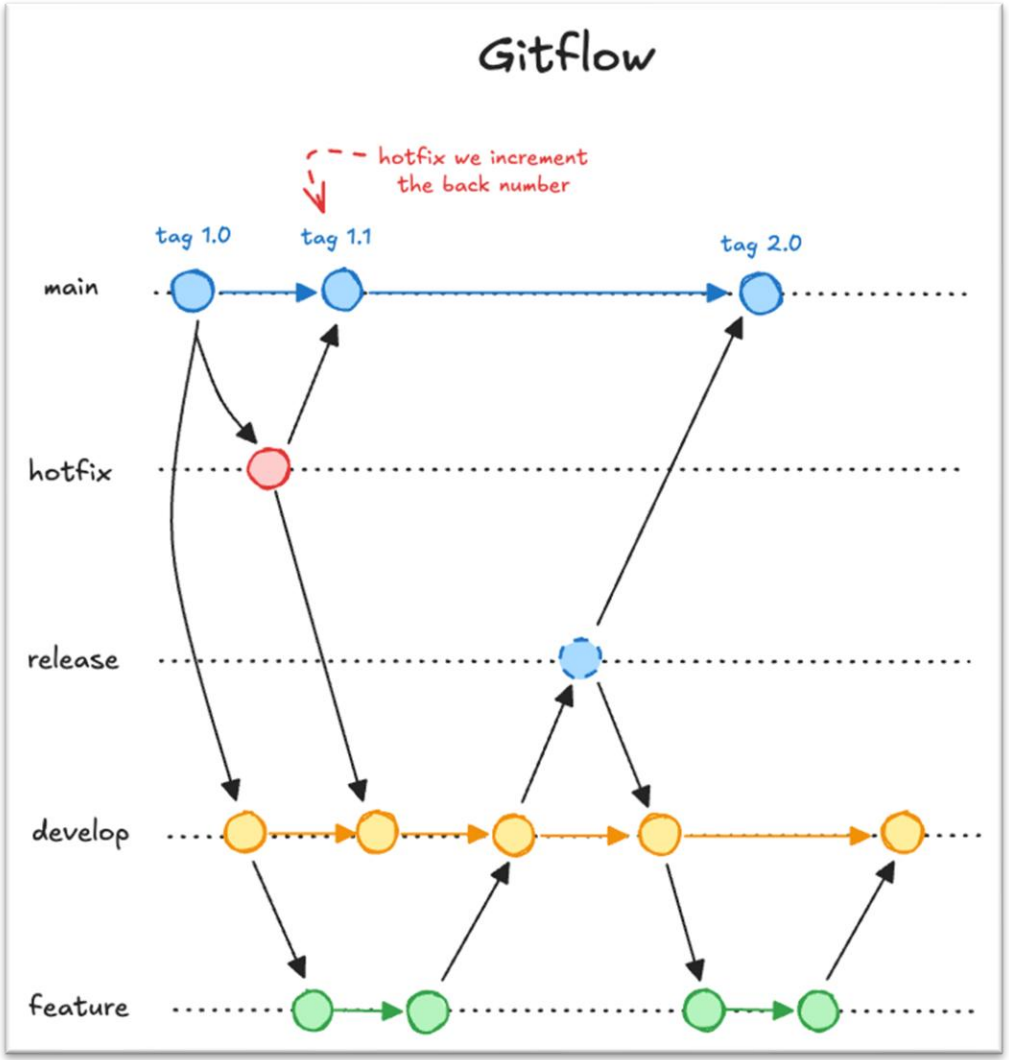


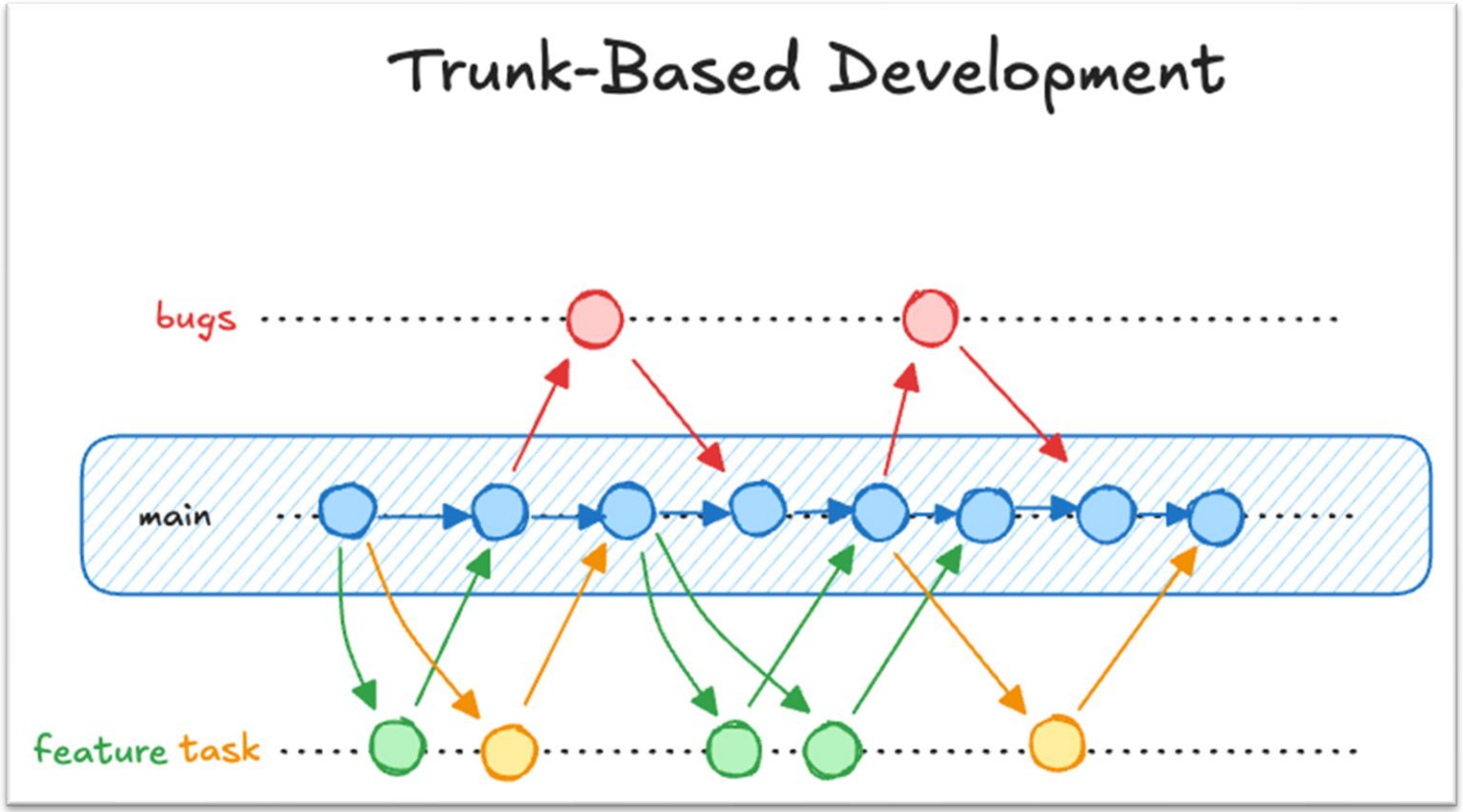
✅ **Ventajas**: minimiza conflictos de `merge` grandes, perfecto para CI/CD, código siempre fresco y cercano a producción.

❌ **Desventajas**: requiere mucha disciplina (tests automáticos, CI sólido), no es ideal si no hay CD.

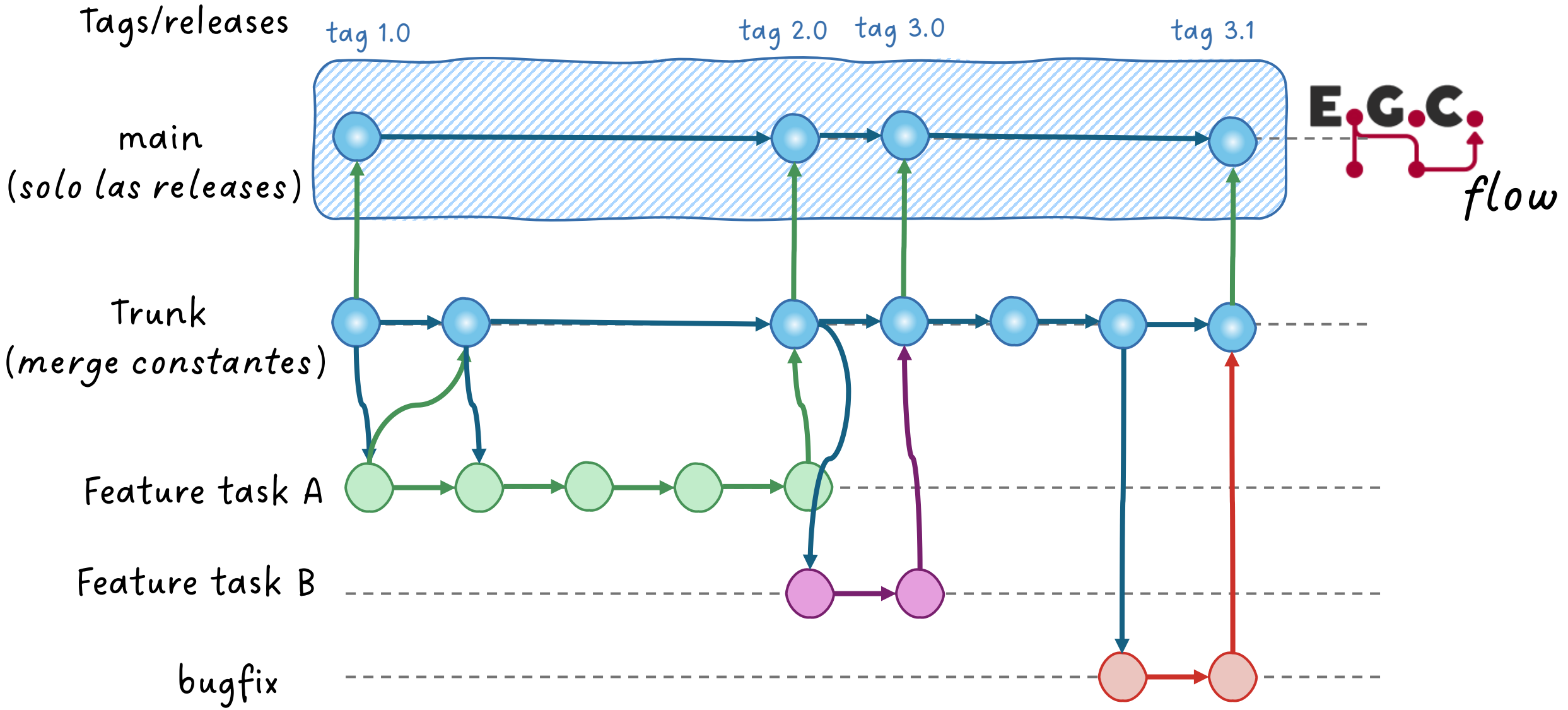
** + info:* https://trunkbaseddevelopment.com/

3. Uso de repositorios | otros modelos de uso





3. Uso de repositorios | nuestro modelo de uso



3. Uso de repositorios | nuestro modelo de uso

- Usar un modelo de ramas por *feature tasks*: al menos una rama `main` y una o varias de tareas de `features`.
- No usar un modelo de ramas por personas (a menos que esté realmente justificado).
- Las ramas que dejan de usarse, se destruyen. Puede haber ramas que no se usen que sigan, *pero debe haber una justificación para ello*.
- **No usar Pull Requests** a menos que sea para integración entre distintos equipos.
- Tener una rama `trunk` siguiendo un modelo ágil de CI en el que los `merge` sean muy frecuentes (cada vez que se acaba una tarea de una `feature`). Trunk no se desruye.
- Tener una rama `main` que hará las veces de rama `release` para etiquetar las entregas. Main no se destruye.
- Tener despliegues automáticos tanto en `trunck` como en `main`.
- Hacer **`merge` frecuentes** entre las ramas (principio de integración continua). De nada sirve tener un servidor de CI si luego no se hacen `merge` periódicos.

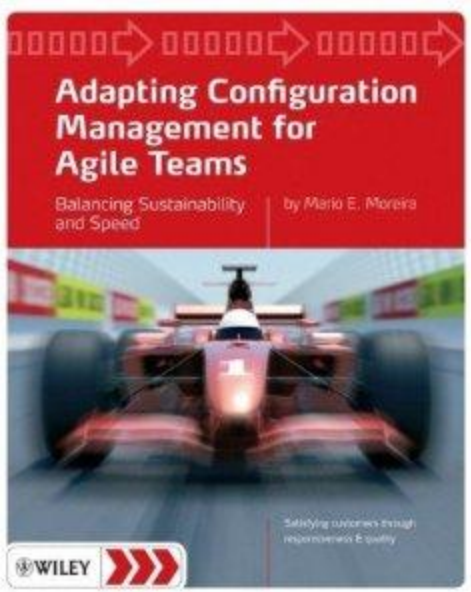
1. Conceptos básicos
2. Operaciones
3. Uso de repositorios
4. **Resumen**
5. Bibliografía

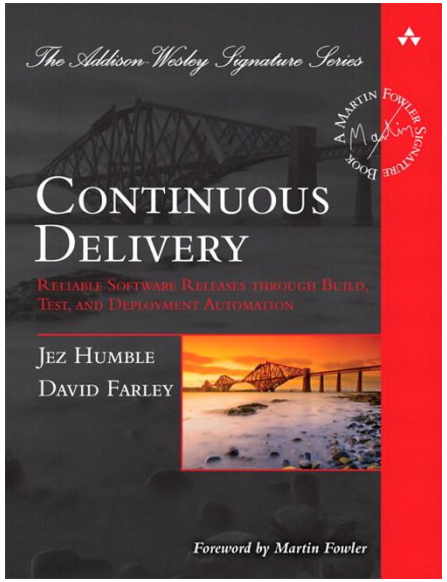
4. Resumen

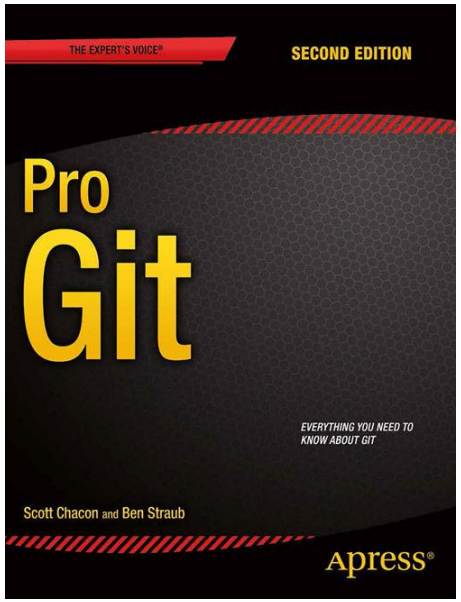
- Definimos de qué archivos vamos a hacer seguimiento y de cuáles no.
- Decidimos cómo organizar el repositorio del equipo y del proyecto y su modelo de uso (*usage model*).
- Limpiar todas las ramas que no se vayan a utilizar.
- Establecer un único nombre para cada persona que contribuya al proyecto.
- Definir una política de gestión de *commits* atómicos.
- Consensuar y usar una política de mensajes de *commit*.

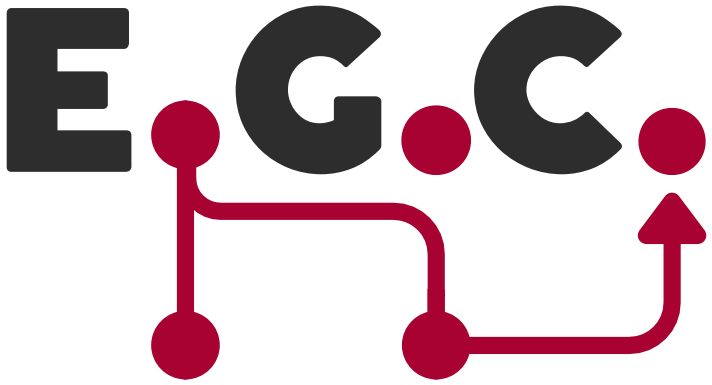
1. Conceptos básicos
2. Operaciones
3. Uso de repositorios
4. Resumen
5. **Bibliografía**

5. Bibliografía









Grado en Ingeniería Informática - Ingeniería del Software

Evolución y Gestión de la Configuración

Universidad de Sevilla 1505

Escuela Técnica Superior de Ingeniería Informática

¡Gracias!