Git

Git

Sistema de control de versiones distribuido

- Manejar versiones de un proyecto
  - En secuencia
  - En paralelo

- Distribuido
  - Ningún cliente está privilegiado (tecnológicamente) sobre otros

Centralización

Git es un sistema de control de versiones **descentralizado**

Pero: En uso común hay un repositorio **central** distinguido

- Servidor propio
- Servicio de hosting git (GitHub, GitLab, BitBucket, AWS CodeCommit, Azure DevOps, SourceForge, etc)

Usaremos: **GitLab**

Centralización

Los servicios de hosting suelen agregar features secundarios propios

- Issue Trackers
- Pull Requests
- CI/CD
- Docker Registry
- Métricas
- etc

Git se puede transferir fácilmente, estos adicionales no

Visualizacion

Es recomendable tener alguna forma de visualizar el estado del repositorio. Algunas herramientas de visualización:

- git log --all --graph --oneline
- gitg
- gitk
- magit
- GitKraken
- SourceTree
- Tortoise Git
- Hosting UI (GitHub, GitLab, BitBucket, etc)
- IDEs
- etc

https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools

Commits

Commit

- Algún estado del sistema

- Queremos referirnos a él
  - Conservarlo para el futuro
  - Compartirlo con el resto del equipo

Commit

Cada commit contiene:

- Ancestros
- Autor/Fecha de Autoría/Commit
- Mensaje de commit
- Archivos

Commits Atomicos

Un commit que aplica **un** cambio **completo**.

- Una sola razón para todo lo que cambió
- No depende de cambios posteriores

Facilita manejar commits:

- Revertir
- Reordenar
- Búsquedas en la historia

Mensajes de Commit

```
Contexto: Resumen del cambio en este commit
(línea siempre en blanco)
Este texto explica el contexto del cambio, incluyendo:
- Razones que no resultan obvias del resumen
- Efectos de los cambios que no resultan obvios
  al ver el código

Metadata que usan otros sistemas:
- Co-autores
- Issues resueltos
- Firmas digitales
```

Mensajes de Commit

Buen ejemplo:

ALSA: usb-audio: Drop bogus dB range in too low level

Browse files

```
Some USB audio firmware seem to report broken dB values for the
volume
controls, and this screws up applications like PulseAudio who blindly
trusts the given data.  For example, Edifier G2000 reports a PCM
volume from -128dB to -127dB, and this results in barely inaudible
sound.

This patch adds a sort of sanity check at parsing the dB values in
USB-audio driver and disables the dB reporting if the range looks
bogus.  Here, we assume -96dB as the bottom line of the max dB.

Note that, if one can figure out that proper dB range later, it can be
patched in the mixer maps.

BugLink: https://bugzilla.kernel.org/show_bug.cgi?id=211929
Cc: <stable@vger.kernel.org>
Link: https://lore.kernel.org/r/20210227105737.3656-1-tiwai@suse.de
Signed-off-by: Takashi Iwai <tiwai@suse.de>
```

master

v6.0-rc6 ··· v5.12-rc2

tiwai committed on 27 Feb 2021

1 parent dcf269b commit 21cba9c5359dd9d1bffe355336cfec0b66d1ee52

Mensajes de Commit

Otro buen ejemplo:

```
Fix typo
```

Mal ejemplo:

```
Fix issue #123
```

Diferencia: ¿Dónde hay que ir para entender por qué cambió el código?

Cosas que no deberían pasar

https://xkcd.com/1296/

Algunas herramientas útiles

Algunos comandos útiles

Para inspeccionar el repositorio:

- **`git log <archivo>`**
  Historial de cambios de un archivo o carpeta

- **`git blame`**
  ¿Cuándo fue la última modificación de cada línea?

- **`git bisect`**
  Busqueda binaria en la historia

Algunos comandos útiles

Para manejar interrupciones:

- **`git stash push -m <comment>`**
  `git stash pop`
  Guardar estado de archivos temporalmente

- **`git add -p`**
  Commits de partes de archivos

- **`git reset`**
  Deshacer commits

Algunos comandos útiles

Para agregar commits:

- **`git cherry-pick <commit>`**
  Copia un commit al branch actual

- **`git revert <commit>`**
  Crea un commit que hace lo opuesto a <commit>

Algunos comandos útiles

git hooks: Antes o después de ciertos comandos, se ejecutan programas dentro de .git/hooks (Si existen)

Hooks comunes:

- pre-commit
- pre-push

Usos comunes:

- Ejecutar tests/linters/autoformatters/etc

Para compartir con el resto del equipo, se puede cambiar el directorio:

```
git config --local --add core.hooksPath git-hooks
```

Algunos comandos útiles

Para manipular la historia local:

- **`git commit --amend`**
  Agregar cambios al commit actual

- **`git rebase --onto <new_root> <old_root>`**
  Trasplanta una serie de commits de un lugar a otro

- **`git rebase -i <root>`**
  Reordenar y combinar una serie de commits

EVITAR USAR CON HISTORIA PÚBLICA

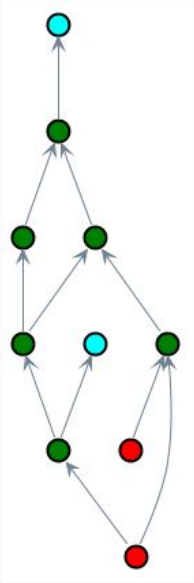
Flujos de trabajo

Historia

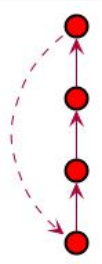
Cada commit tiene ancestros. Su contenido está relacionado al contenido de sus ancestros.

Único requisito: Sin ciclos

Historia Lineal



Historia Compleja



Historia Imposible

Historia

Git no impone una organización de la historia

El equipo debe decidir cómo organizarse

A pesar de esta libertad, existen:

- Ideas base y flujos pre-armados
- Herramientas preexistentes
  - Muchas son distribuidas con git

Trunk-based

Todo cambio se agrega a un mismo branch

Ventajas:

- Simple
- Todo el código está en el último commit

Desventajas:

- Todo el código está en el último commit
- Cuesta separar de líneas de trabajo

Trunk-based

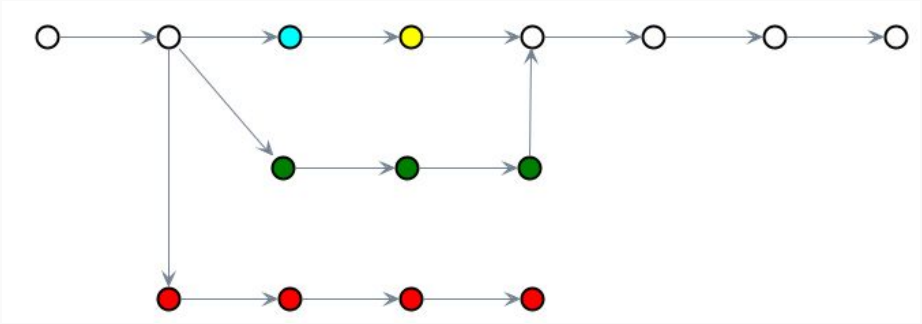
Problema principal:

El control de versiones es naturalmente distribuido
(⇒ también es concurrente)

Feature Branches

Un branch por funcionalidad **de código** independiente

(Usualmente, 1 funcionalidad para el usuario ~ N features)



Feature Branches

Ventajas:

- Solo un feature incompleto por branch
- Menos gente trabajando en cada branch
- El resto del equipo ve todos los cambios juntos

Desventajas:

- Administración de branches
- Features necesitan ser independientes entre sí

Conflicto

Al hacer merge, hay **conflicto** cuando los cambios de cada rama interactúan de forma negativa.

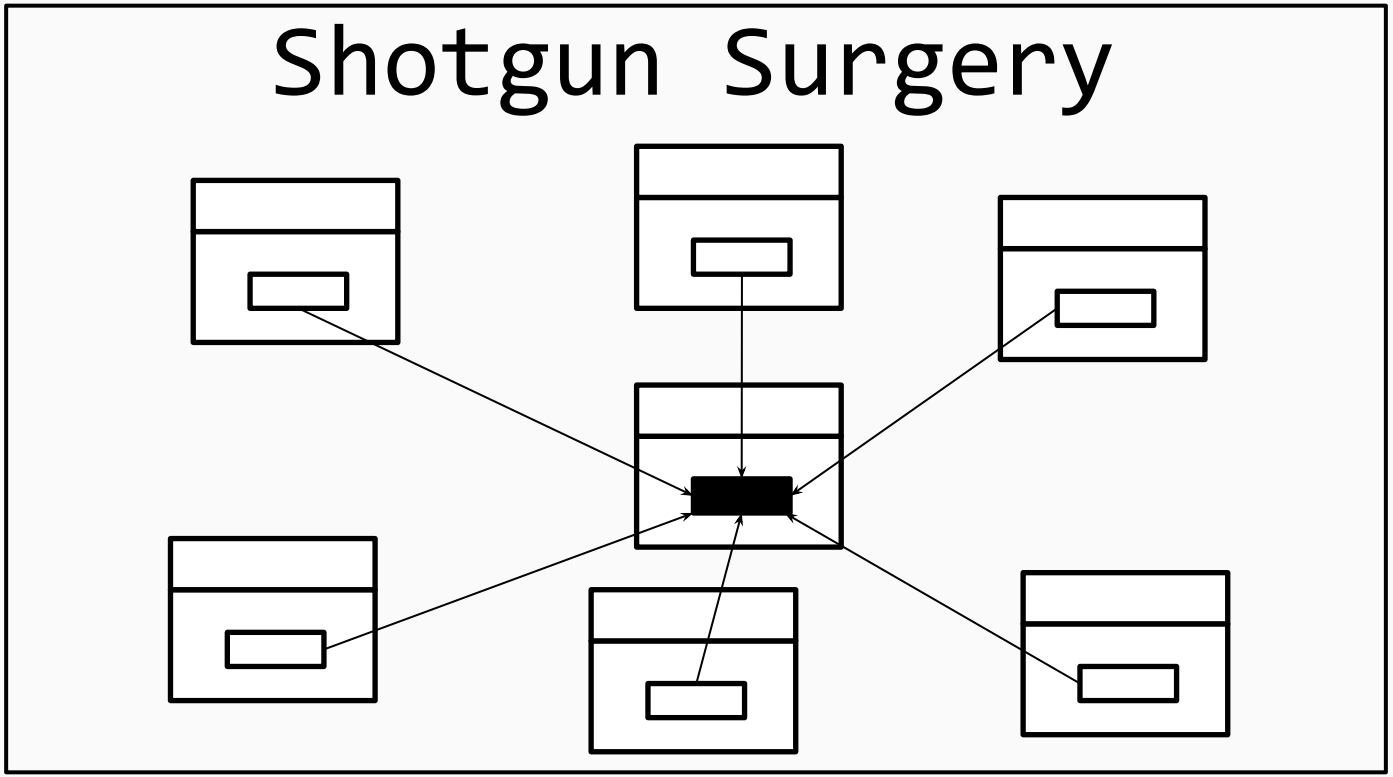
- Conflicto textual
  Mismas líneas ⇒ git puede detectarlo

- Conflicto semántico:
  Misma lógica ⇒ requiere analizar la interacción

Deseamos que los conflictos sean textuales

Conflicto

Relación con el buen diseño:



Pull Requests

Tambien llamadas **Merge Requests**

Antes de hacer un merge, hay oportunidad de verificar:

- Calidad del código
- Calidad de la aplicación
- Cumplimiento de requerimientos
- Si el cambio conviene

Para iniciar estas verificaciones, existe el proceso de **Pull Request**

Pull Requests

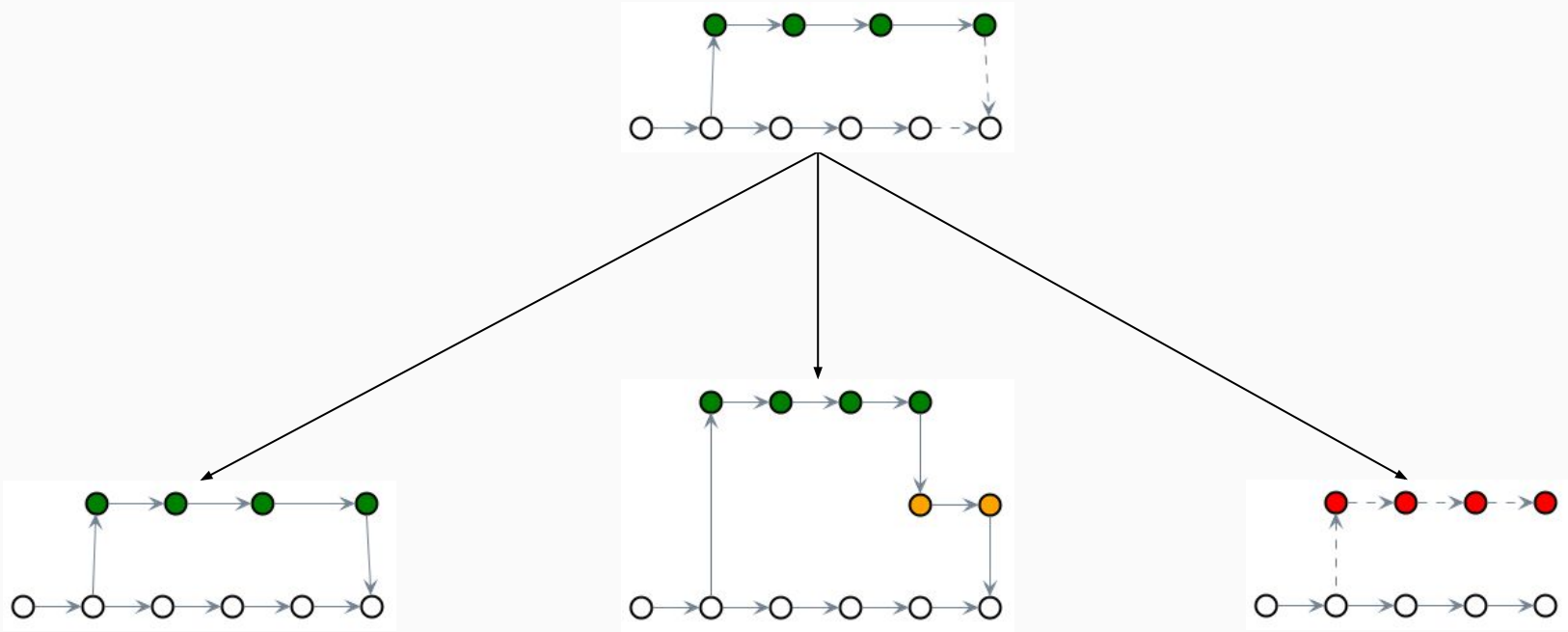
Un buen PR:

- Es atomico:
  - Una sola razón para todo lo que cambió
  - No depende de cambios externos

- Explica:
  - Razones que no resultan obvias del resumen
  - Efectos de los cambios que no resultan obvios al ver el código

Es lo mismo que queremos en commits, pero a mayor escala (horas - días)

Pull Requests



PR Aceptado

PR Aceptado con cambios

PR Rechazado

Code Review

Un pull request da la oportunidad de hacer revisión de código.

Para que sea exitosa, es recomendado que sea:

- Cooperativo
- Sin ego
- Prioridad
- Repartido

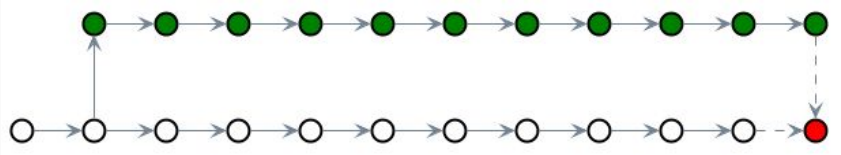
Obtenemos:

- Mejor calidad de código
- Difusión de conocimiento
  - Tecnico
  - Del proyecto (Aumentar bus factor)

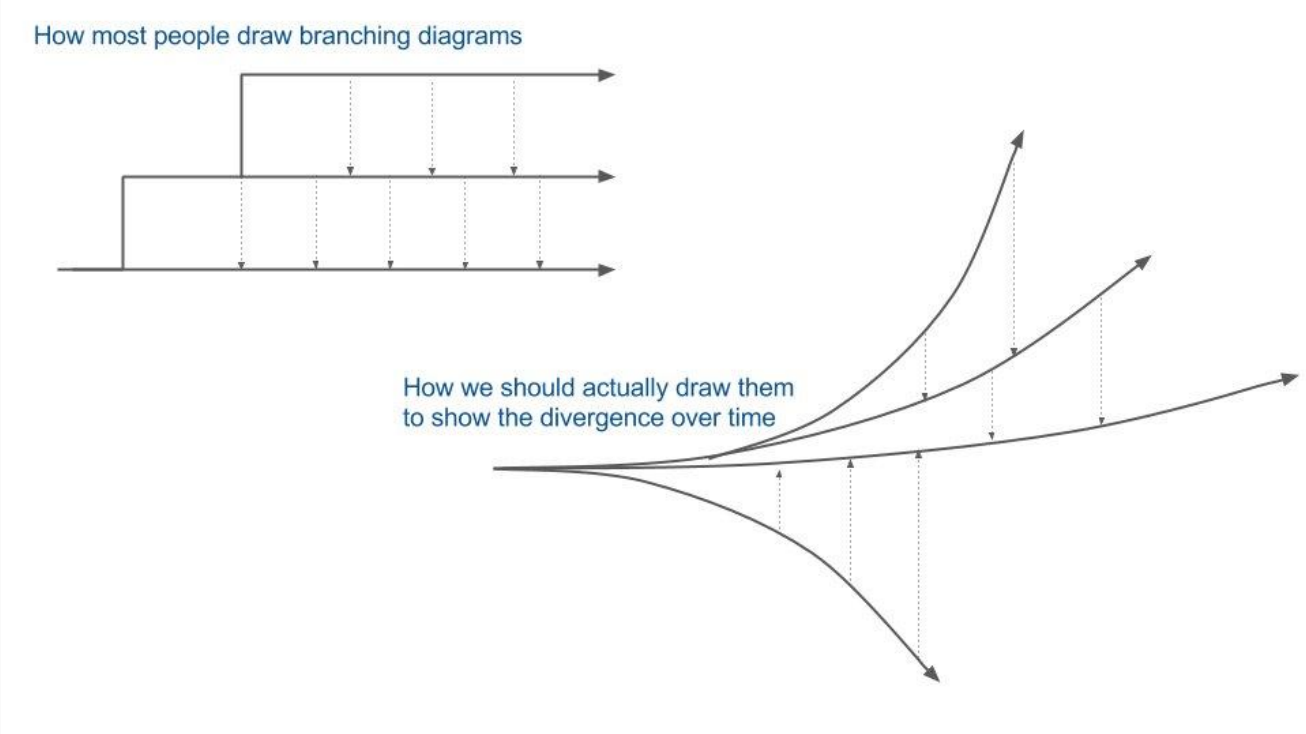
Branches a Largo Plazo

Si mantenemos un branch separando durante demasiado tiempo:

- Bugs hallados y arreglados en ambos branches
- Hay que reconciliar los conflictos



Branches a Largo Plazo



https://twitter.com/jahnnie/status/937917022247120898

Branches a Largo Plazo

Mejor solución: No hacer branches a largo plazo

- Modularizar mejor: Separar variantes por ubicación en lugar de tiempo
- Feature flags: Los branches comparten su código, la distinción se hace por configuración

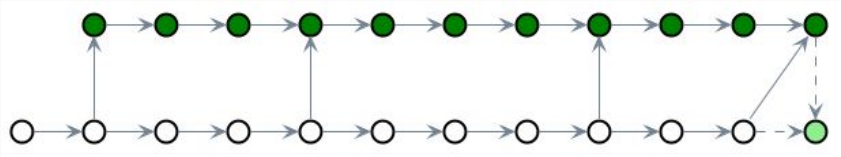
Pero: A veces representa el significado que queremos

- Manejar cuidadosamente la divergencia

Branches a Largo Plazo

Para reducir los problemas, hacer merge intermedios:

- Menos cambios en cada merge
- Esos cambios son más recientes
- Evitamos duplicar trabajo



Branches de Entorno

Branch a largo plazo que:

- Representa un entorno de despliegue
- Usa CI/CD para desplegar cambios

Algunos nombres comunes:

- production, main, master
- uat
- qa
- development, dev

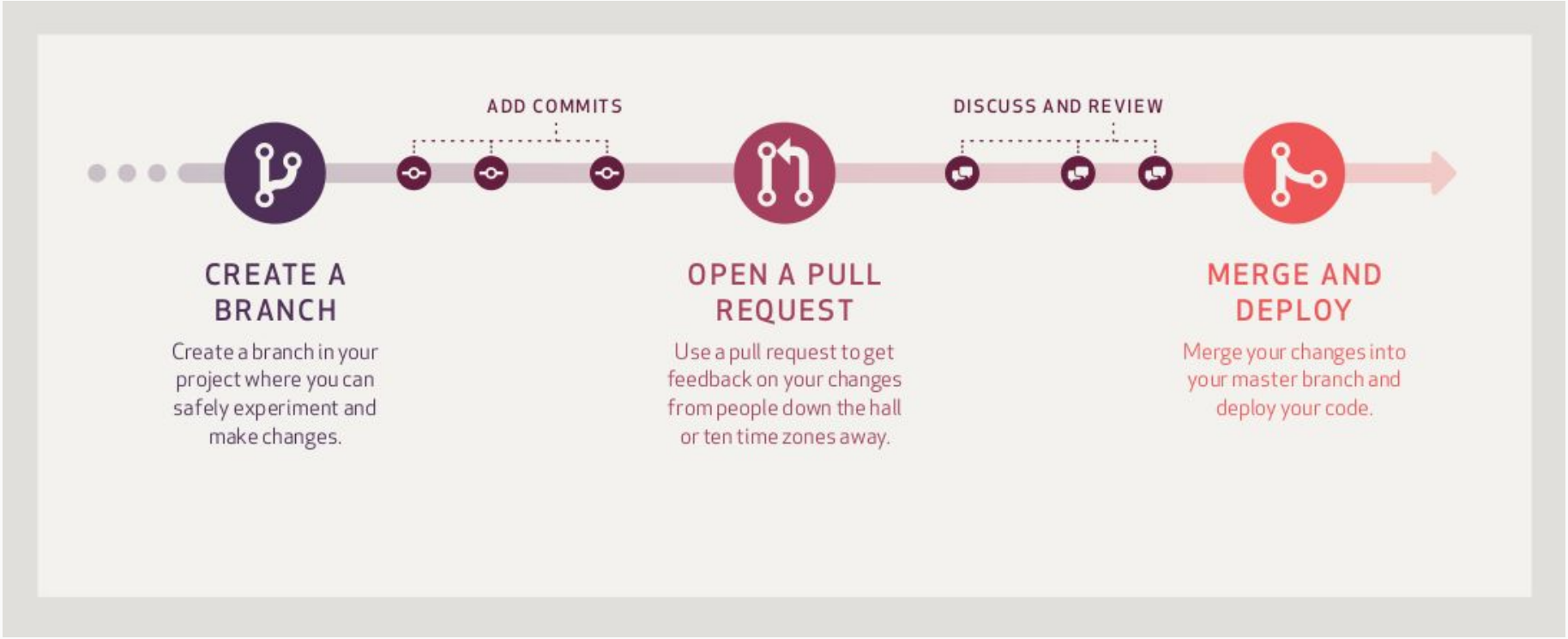
Branches de Entorno

production	uat	qa	development

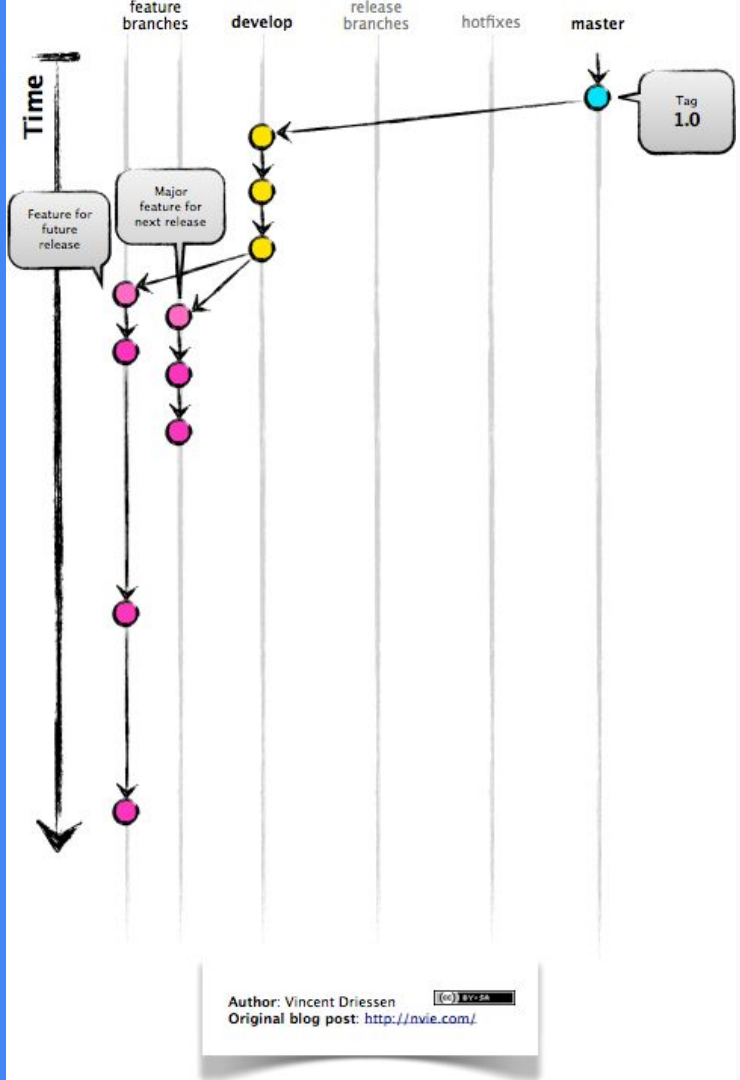
- Definimos un orden de más a menos estables

- Desarrollo
  - En entorno menos estable
  - Propagan hacia entornos más estables
- Hotfixes
  - Arreglos de problemas urgentes
  - En entorno más estable
  - Aplicados a entornos menos estables

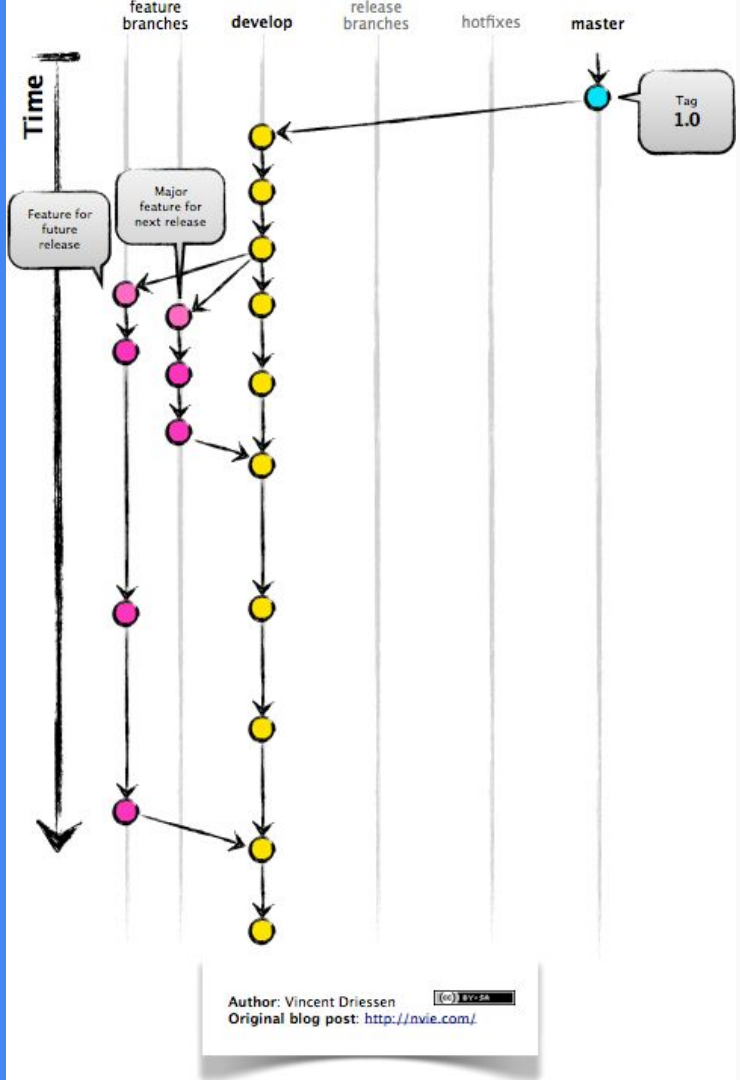
Github Flow


ADD COMMITS
DISCUSS AND REVIEW
CREATE A BRANCH
Create a branch in your project where you can safely experiment and make changes.
OPEN A PULL REQUEST
Use a pull request to get feedback on your changes from people down the hall or ten time zones away.
MERGE AND DEPLOY
Merge your changes into your master branch and deploy your code.

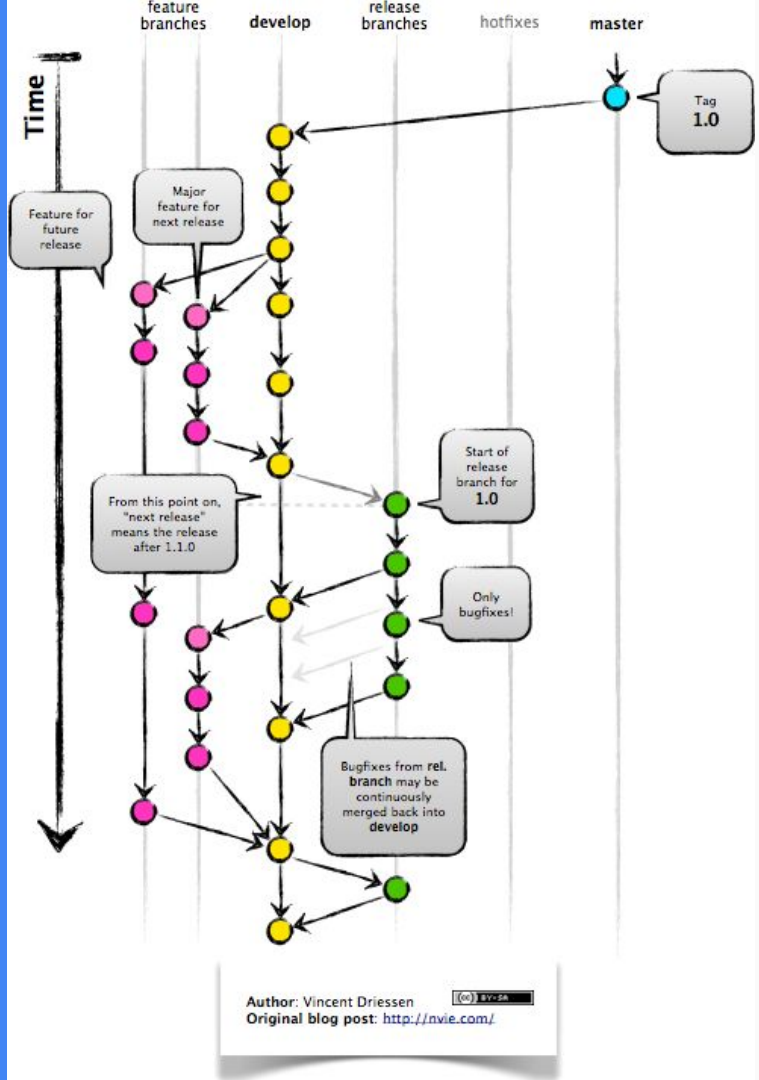
Git Flow


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Author: Vincent Driessen
Original blog post: http://nvie.com/

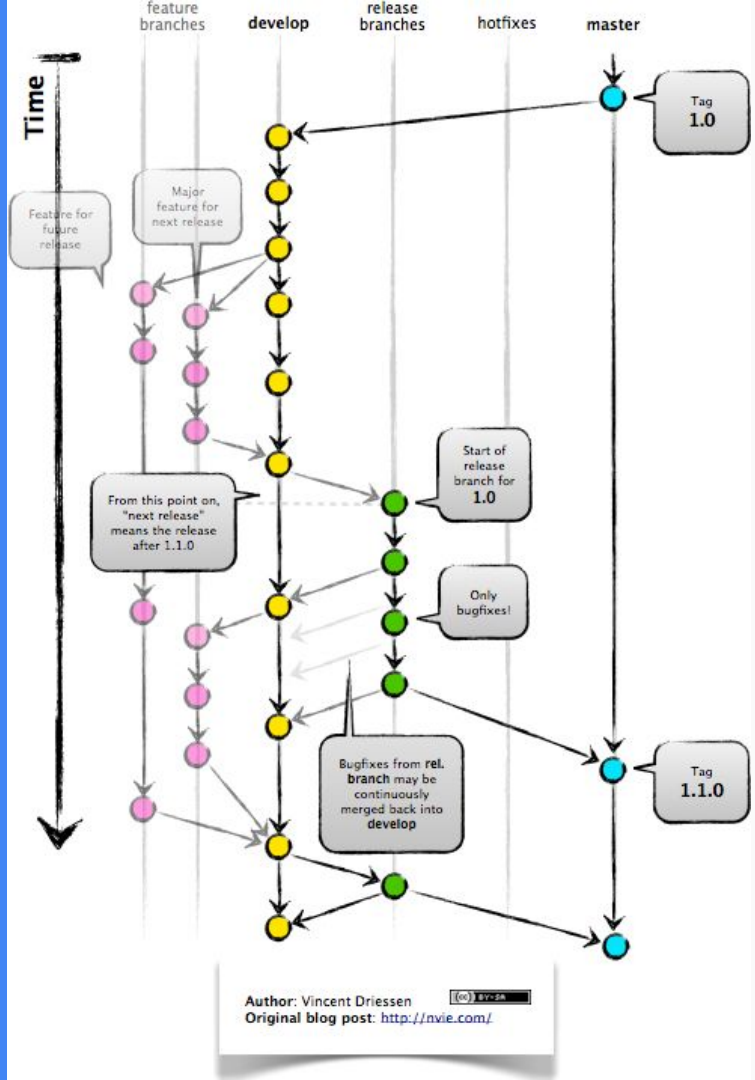
Git Flow


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Author: Vincent Driessen
Original blog post: http://nvie.com/

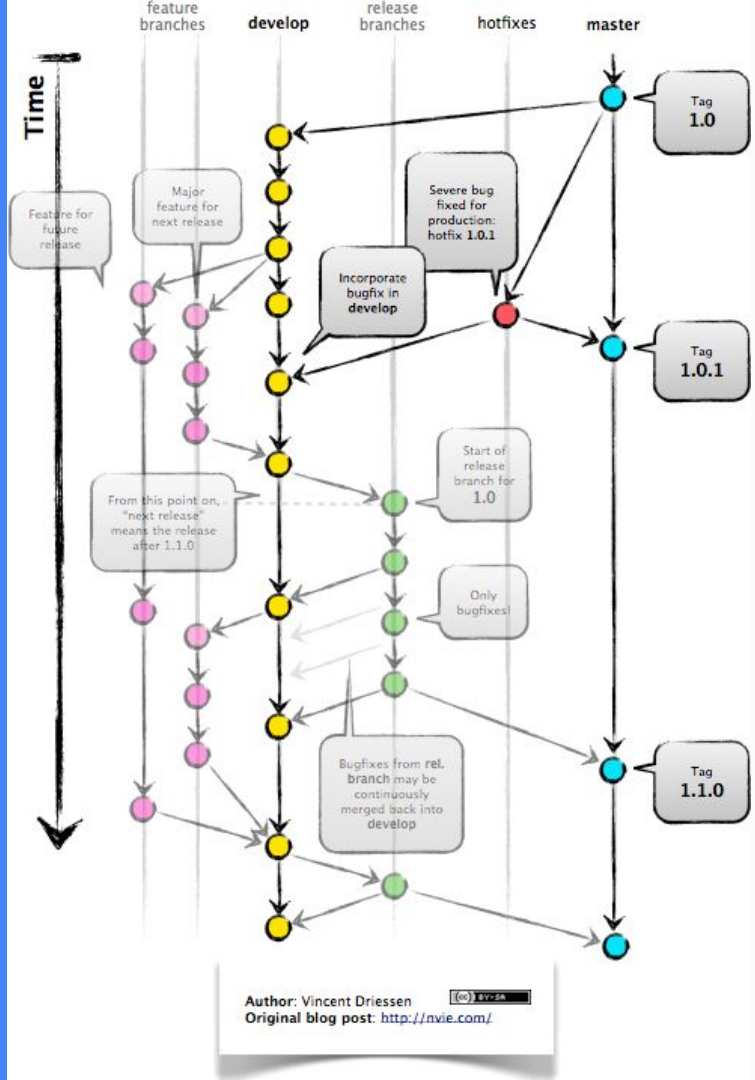
Git Flow


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Author: Vincent Driessen
Original blog post: http://nvie.com/

Git Flow


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
From this point on, "next release" means the release after 1.1.0
Start of release branch for 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/

Git Flow


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 1.0.1
Incorporate bugfix in develop
Tag 1.0.1
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/

Git Flow vs Github Flow

Github Flow:

- Verificación dentro de cada PR
- Software con despliegues simples-rápidos-baratos
  - SaaS
  - Aplicaciones web

Git Flow:

- Verificación al preparar cada release
- Software con despliegues complejos-lentos-caros
  - Aplicaciones mobile o de escritorio
  - Instalaciones on-premise
  - Múltiples versiones simultáneas

Monorepo vs Polirepo

Monorepo

Un repositorio para todo el proyecto

Caracteristicas:

- Simple
- Facilita hacer cambios coordinados a N componentes
- Fácil de filtrar para crear polyrepo con historia equivalente (git filter-repo)

Polyrepo

Un repositorio por componente del proyecto

Caracteristicas:

- Control de acceso limitando autorización a ciertos repositorios
- Facilita modularizar componentes (Evita dependencias implícitas)
- No hay forma fácil de unificar la historia para crear un monorepo equivalente

Monorepo vs Polirepo

Conway's Law: Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

Melvin Conway

Elegir una estructura de repositorios que refleje:

- Estructura de comunicación entre participantes
- Conexiones entre componentes de software
- Responsabilidades compartidas entre componentes

Integración Continua

CI/CD

Continuous Integration

- Integrar frecuentemente el trabajo realizado
- Descubrir errores rápidamente
- De forma automatica

Continuous Deployment

- Asegurar que podemos producir un entregable de forma rápida y certera
- Desplegar a entornos de prueba
- De forma automatica

CI/CD

Gitlab

```
test:
    stage: test
    script:
        - npm test
```

Github

```
name: GitHub Actions Demo
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - run: npm test
```

CI/CD

Builds

```
build:
    stage: build
    script:
        - gcc main.c -o main
    artifacts:
        paths:
            - main
```

Artefacto: Algún archivo que deseamos usar fuera del build

CI/CD

Otras etapas pueden reusar artefactos:

```
build:
    [...]
    artifacts:
        paths:
            - main

test:
    stage: test
    dependencies:
        - build
    script:
        - ./main
```

CI/CD

Runners

```
test:
    tags:
        - node   # Solo se ejecuta en un servidor asignado al tag "node"
```

La mayoría de los servicios de git dan dos opciones para ejecutar pipelines:

- En algún servidor que provee el hosting
  - Más fácil de configurar
  - Efímero
- En algún servidor que configuramos
  - Posiblemente más barato
  - Persiste en el tiempo

Portabilidad

Indicamos que el mismo trabajo se ejecute en múltiples entornos:

```
test:
    parallel:
        matrix:
            - OS: windows
              STACK: [bare, docker]
              APP: [app_web, app_windows]
            - OS: ubuntu
              STACK: [bare, docker, lxc]
              APP: [app_web, backend]
    tags:
        - ${OS}-${STACK}
    script: ["bin/test.sh ${STACK} ${APP}"]
```

GitOps

Combinando:

- Branches de entorno
- CI/CD
- Infraestructura declarativa (Próxima clase!)

➔ Cada commit representa la infraestructura en un momento

- ◆ Rastro de auditoria
- ◆ Revertir a estados anteriores
- ◆ Entornos en paralelo

Preguntas?

Recursos

- Try Git

- Branching Patterns (Martin Fowler)
- How to Write a Git Commit Message (Chris Beams)

- Gitlab CI Documentation
- GitHub Actions Documentation

- Git Flow
- GitHub Flow
- GitLab Flow