

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

Centralización

Las forjas suelen agregar features secundarios propios

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

Git se puede transferir fácilmente entre proveedores, estos adicionales no

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

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

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

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

<https://xkcd.com/1296/>

Visualización

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>

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

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

No hay relación entre tamaño de commit y atomicidad del mismo

Solve race condition in audio system

[50 lineas de comentario]

[1 caracter modificado]

Autoformat all code with prettier

[1000000 líneas modificadas]

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

- **git rebase**

Reordenar commits

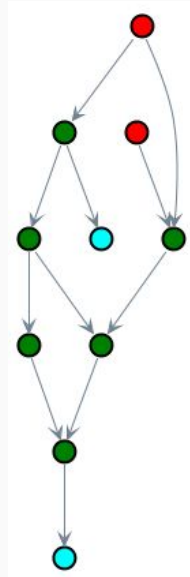
Flujos de trabajo

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

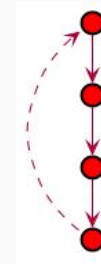
Único requisito: Sin ciclos



Historia Lineal



Historia Compleja



Historia Imposible

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
(\Rightarrow también es concurrente)

Múltiples desarrolladores pueden trabajar sobre el mismo código simultáneamente, sin posibilidad de fingir que su trabajo fue secuencial

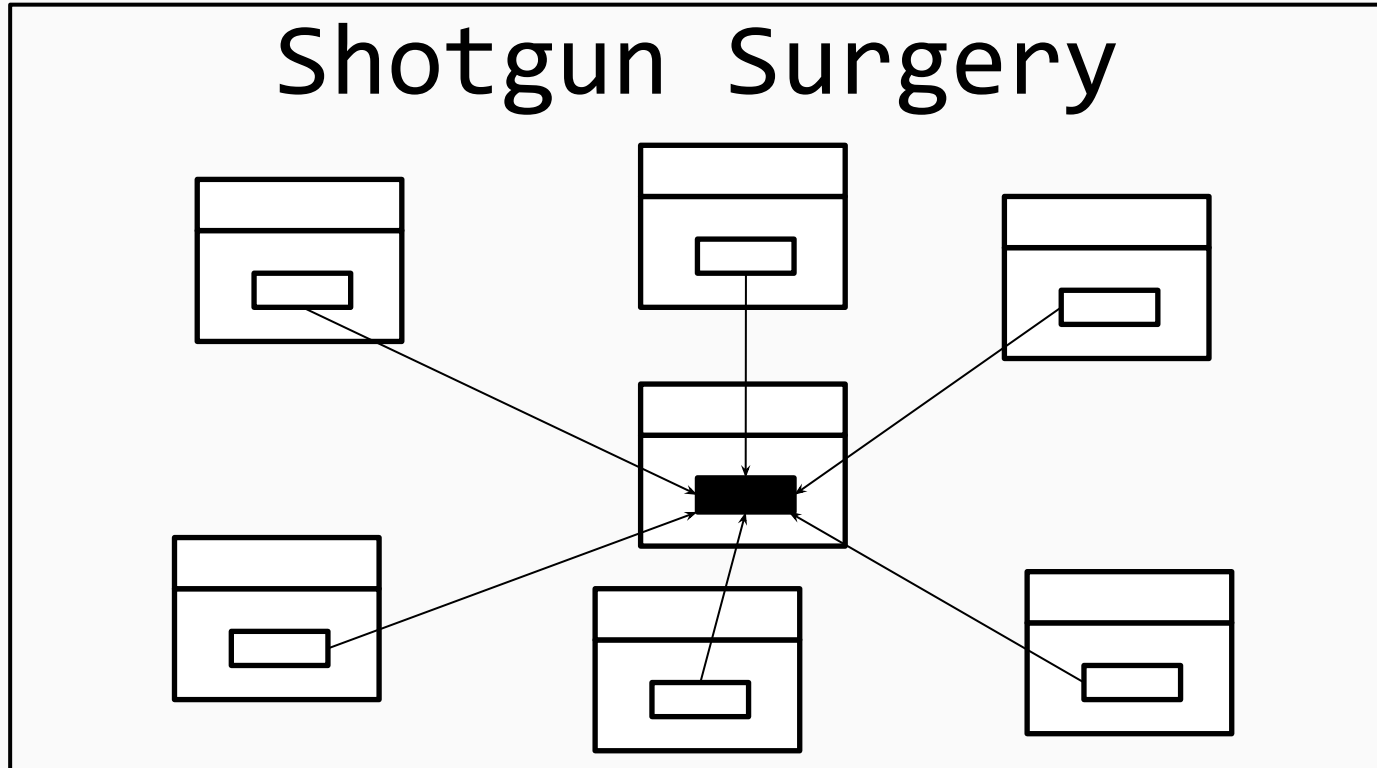
Conflicto

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

- Conflicto textual
Mismas líneas \Rightarrow git puede detectarlo
- Conflicto semántico:
Misma lógica \Rightarrow requiere analizar la interacción

Deseamos que los conflictos sean textuales

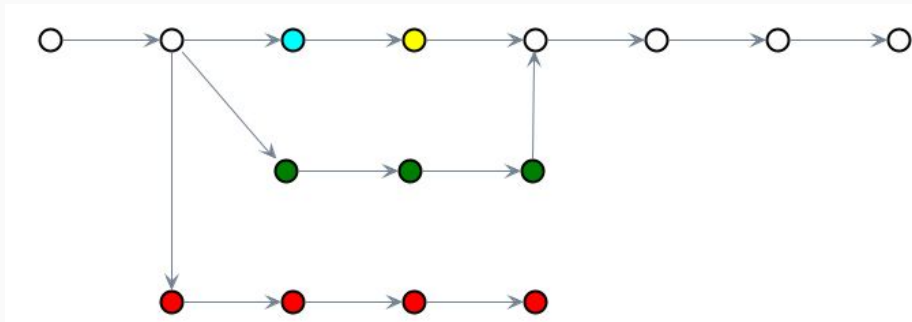
Relación con el buen diseño:



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

Pull Requests

También 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**

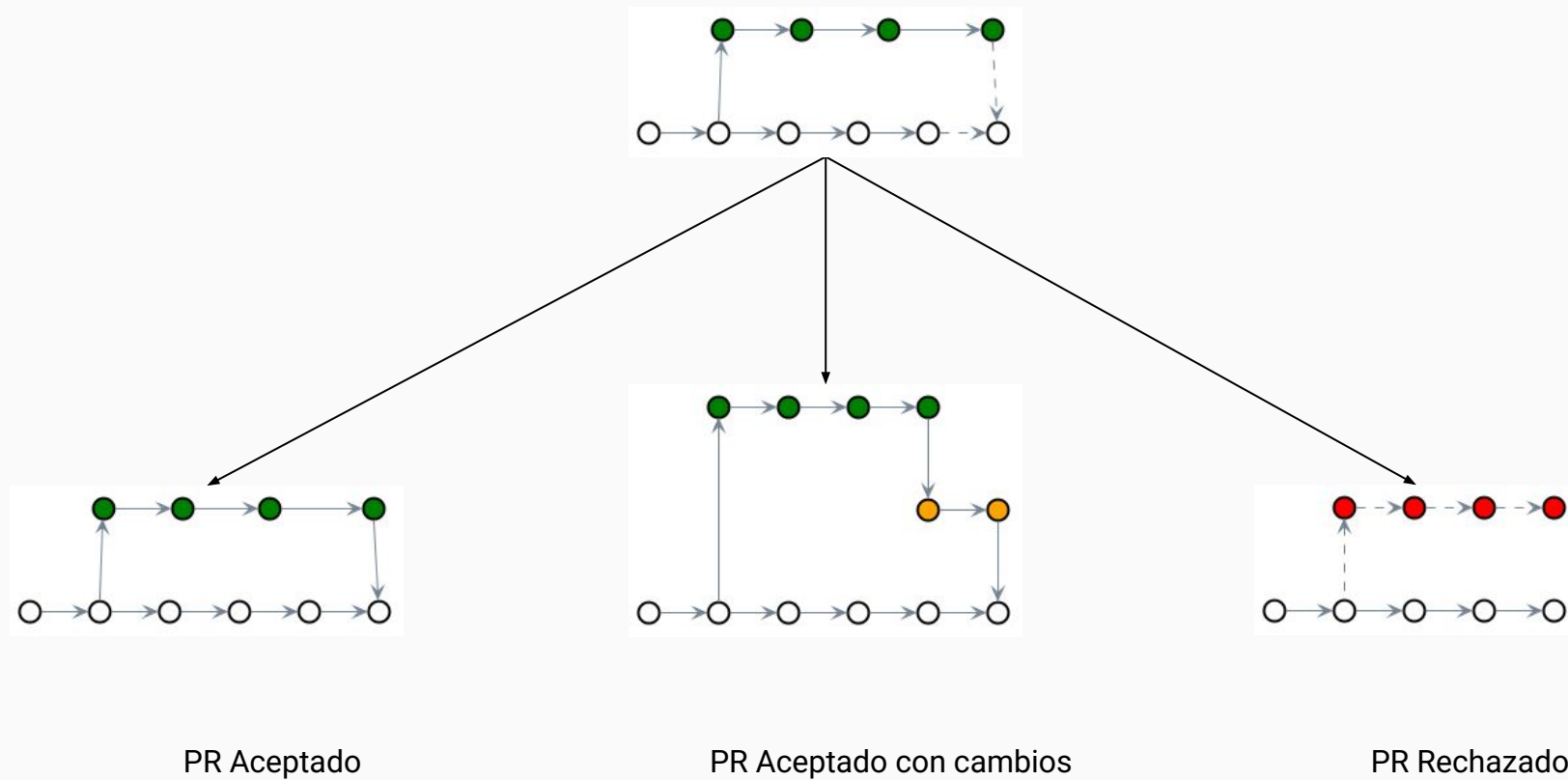
Pull Requests

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



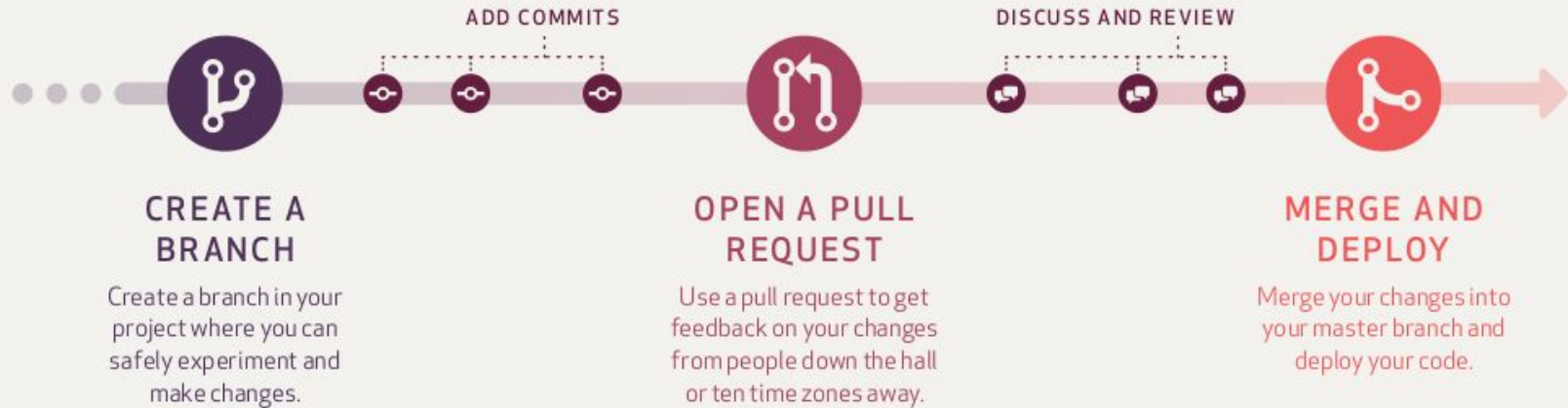
Un pull request da la oportunidad de hacer revisión de código.

Para que sea exitosa, es recomendado que sea:

- Cooperativo
- Sin ego
- Prioridad
- Repartido

Obtenemos:

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



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

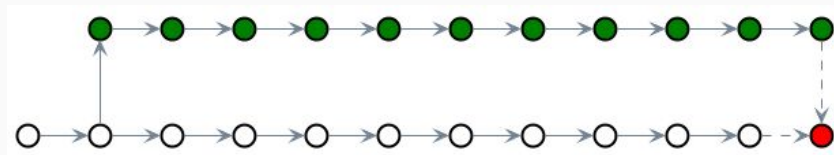
Para compartir con el resto del equipo, se puede cambiar el directorio:

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

Branches a Largo Plazo

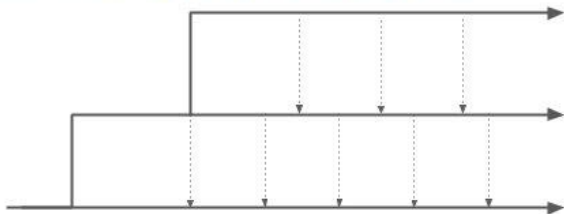
Si mantenemos un branch separando durante demasiado tiempo:

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

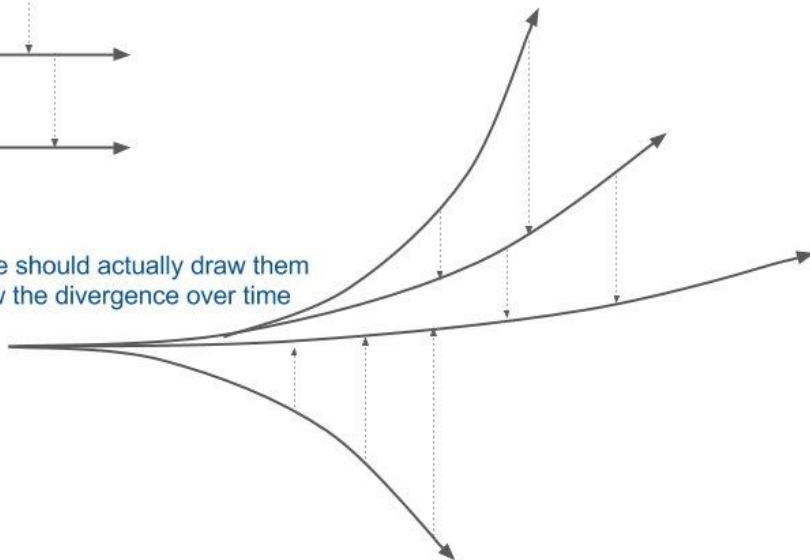


Branches a Largo Plazo

How most people draw branching diagrams



How we should actually draw them to show the divergence over time



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

Branches a Largo Plazo

No hacer branches a largo plazo

- Modularizar mejor: Separar variantes por archivo en lugar de por branch
- Feature flags: Activar una u otra sección de código por configuración
- Excepción: Branches de Entorno

Branches de Entorno

Branch que:

- Representa un entorno de despliegue
- Despliega automáticamente cambios de código al entorno

Algunos nombres comunes:

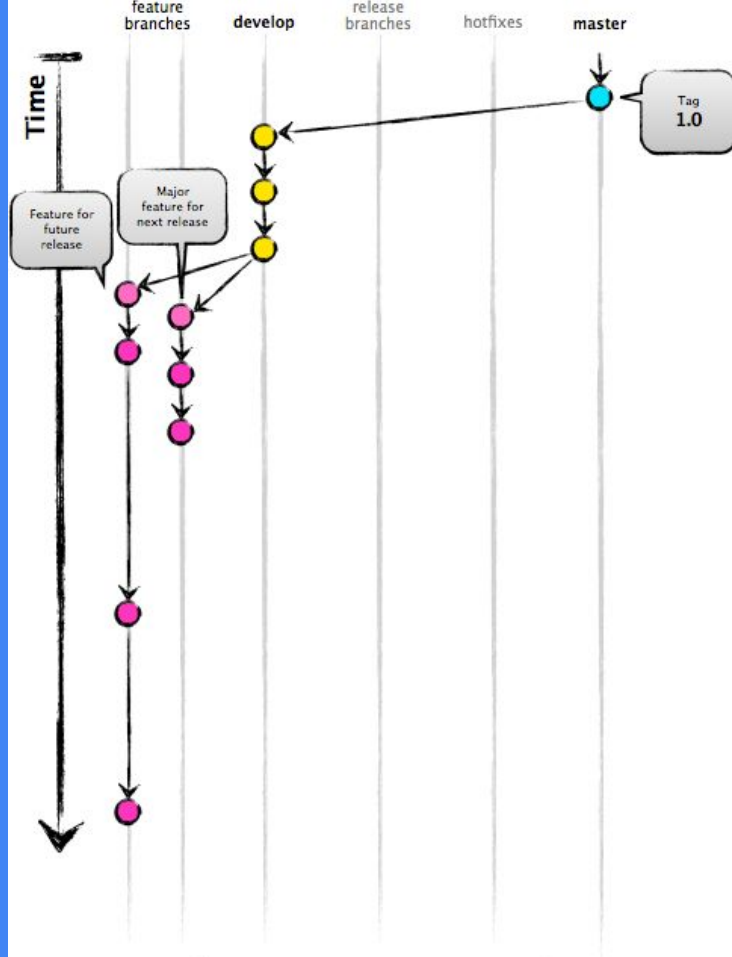
- production, main, master
- uat
- qa
- development, dev

Branches de Entorno

production	uat	qa	development
------------	-----	----	-------------

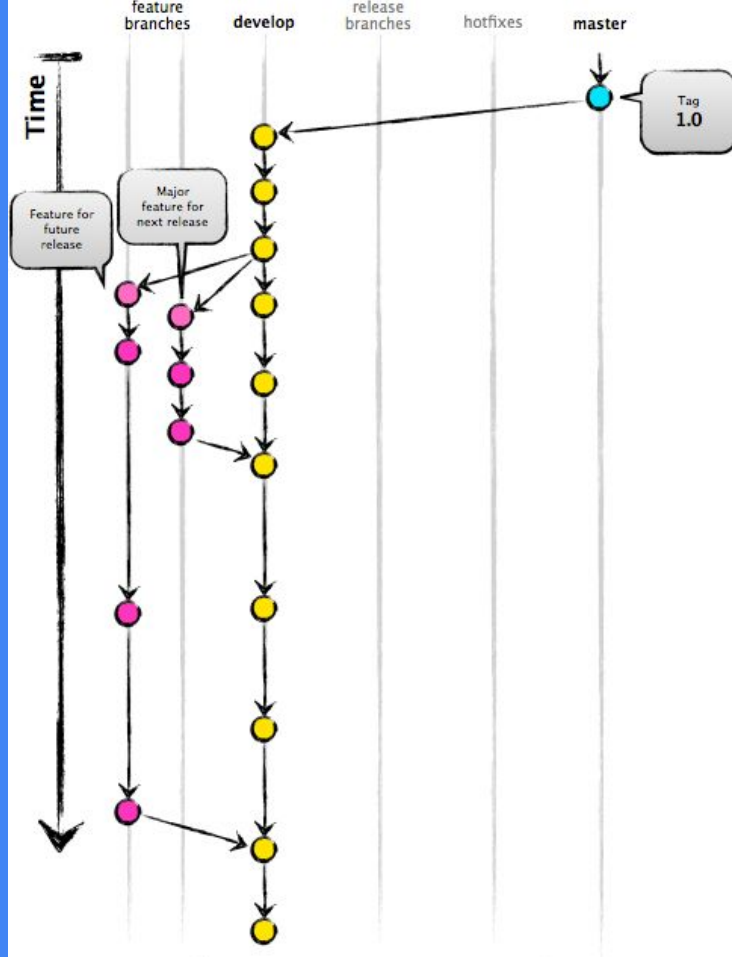
- Definimos un orden de más a menos estables
- Desarrollo
 - Sucede en entorno menos estable
 - Se estabiliza y propaga hacia entornos más estables
- Hotfixes
 - Arreglos de problemas urgentes
 - Sucede en entorno más estable
 - Se propaga a entornos menos estables

Git Flow



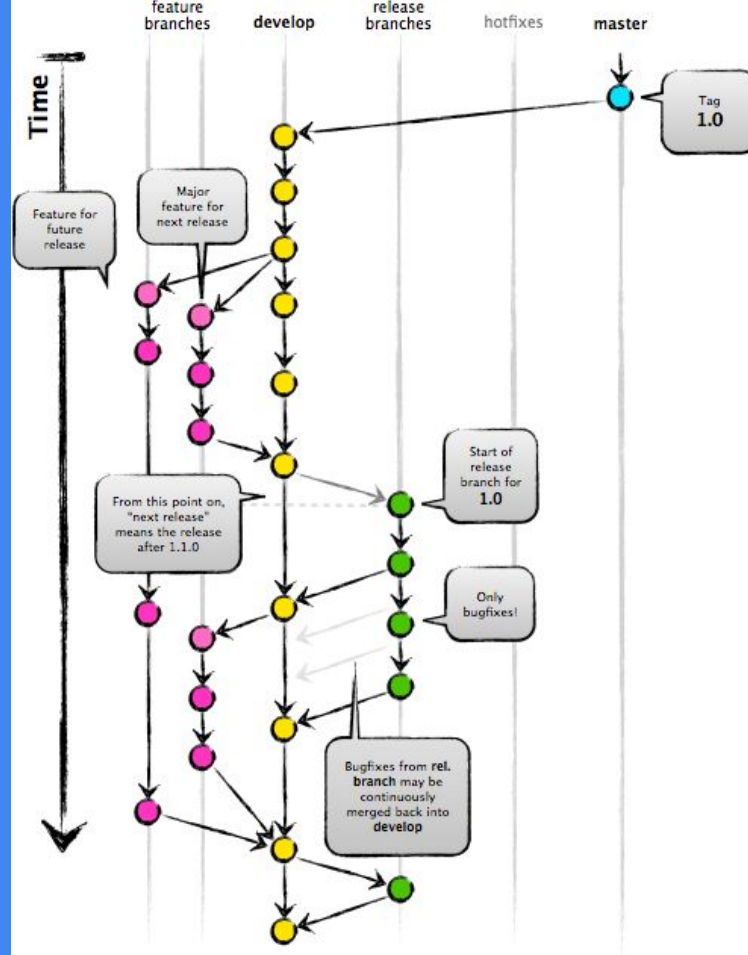
Author: Vincent Driessen 
Original blog post: <http://nvie.com/>

Git Flow



Author: Vincent Driessen 
Original blog post: <http://nvie.com/>

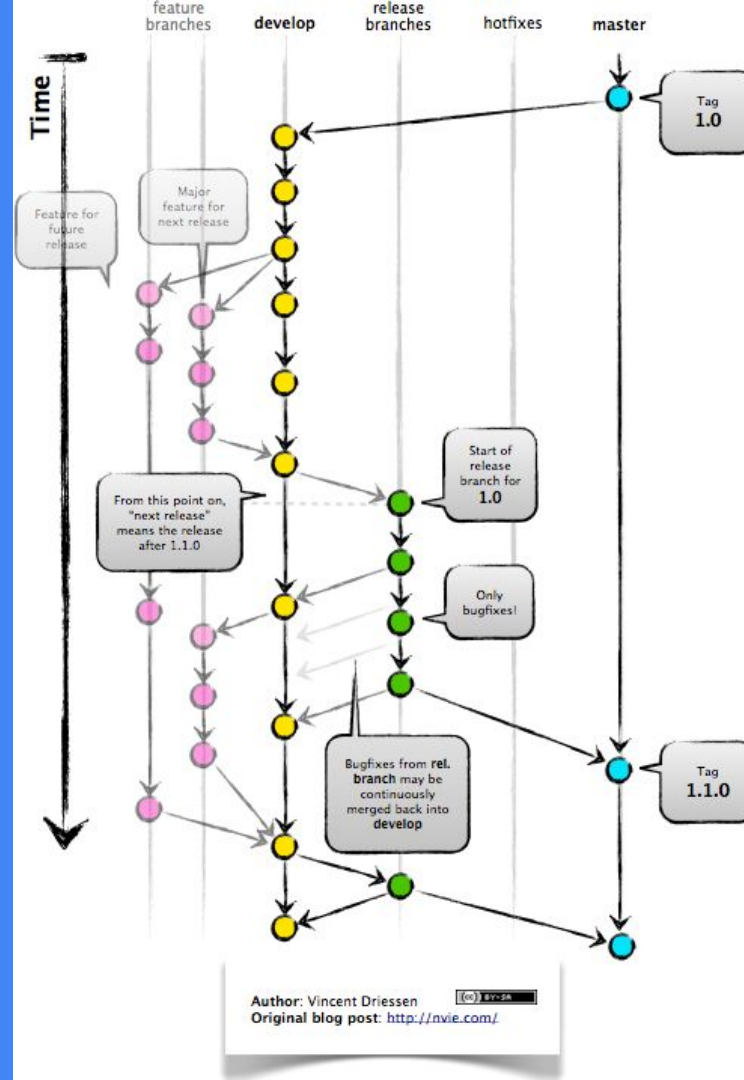
Git Flow



Author: Vincent Driessen
Original blog post: <http://nvie.com/>

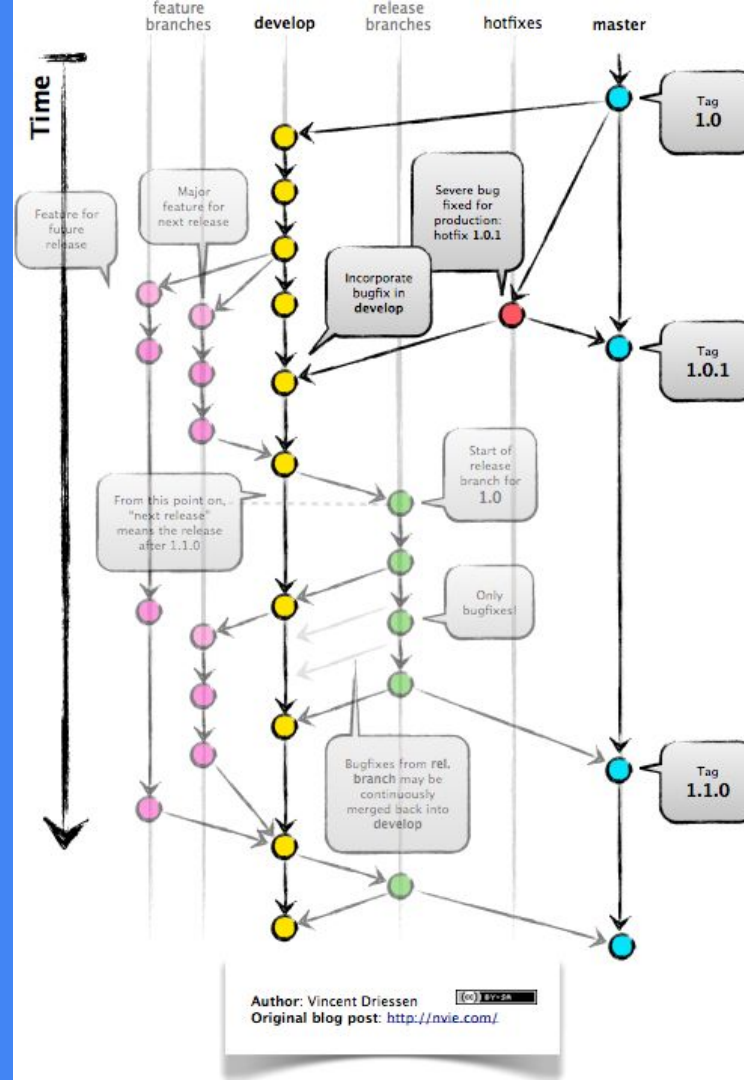
CC BY-SA

Git Flow



Author: Vincent Driessen
Original blog post: <http://nvie.com/>

Git Flow



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

Características:

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

Polirepo

Un repositorio por componente del proyecto

Características:

- 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 automática

Continuous Deployment

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

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
```

Artefacto: Algún archivo que deseamos usar fuera del build

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


Otras etapas pueden reusar artefactos:

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

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

Runners

Dos opciones para ejecutar pipelines:

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

```
test:
```

```
  tags:
```

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

Portabilidad

Indicamos que las mismas tareas se ejecuten 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}"]
```

CI/CD

Dos formas de usar docker para determinar el entorno en que ejecuta un build:

- Build crea una imagen docker
docker push a un registro
- Ejecutar build dentro de un contenedor
Extraer artefactos del contenedor

En ambos casos, docker hace a los builds independientes del runner

Mejores despliegues:

- Producir imágenes en build
- Subir imágenes a registro privado (GitLab, GitHub, etc)
- Usar mismas imágenes en diversos entornos
 - ◆ Variables de entorno para controlar variaciones
- Ops puede manejar componentes de manera uniforme
- Mínimas dependencias en servidor: Docker (+ orquestador)

GitOps

Combinando:

- Branches de entorno
- CI/CD
- Infraestructura declarativa

- Cada commit representa un cambio en el contenido de un entorno
- ◆ 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](#)