Git Básico

Arturo Silvelo

Try New Roads

Buenas Prácticas

Flujos de Trabajo en Git

¿Por qué necesitamos flujos de trabajo?

Sin un flujo definido, los equipos enfrentan:

- **Conflictos constantes** entre desarrolladores
- **Código inestable** en ramas principales
- **Releases caóticas** sin control
- **Falta de trazabilidad** de cambios

Un flujo de trabajo define **cómo y cuándo** se integran los cambios. No hay una estrategia perfecta, simplemente se usa la que mejor se adapte al proyecto:

- Git Flow
- Github flow
- GitLab flow
- Trunk Based
- Shop/Shop/Ask

Git Flow

El flujo más tradicional y estructurado para equipos grandes.

Ramas principales:

- `main`: Código en producción (estable)
- `develop`: Rama de desarrollo principal

Ramas de soporte:

- `feature/*`: Nuevas funcionalidades
- `release/*`: Preparación de releases
- `hotfix/*`: Correcciones urgentes

Git Flow - Características

Ventajas:

- Control total sobre releases
- Historial muy organizado
- Ideal para software con versiones

Desventajas:

- Muy complejo para equipos pequeños
- Releases lentos
- Muchas ramas que mantener

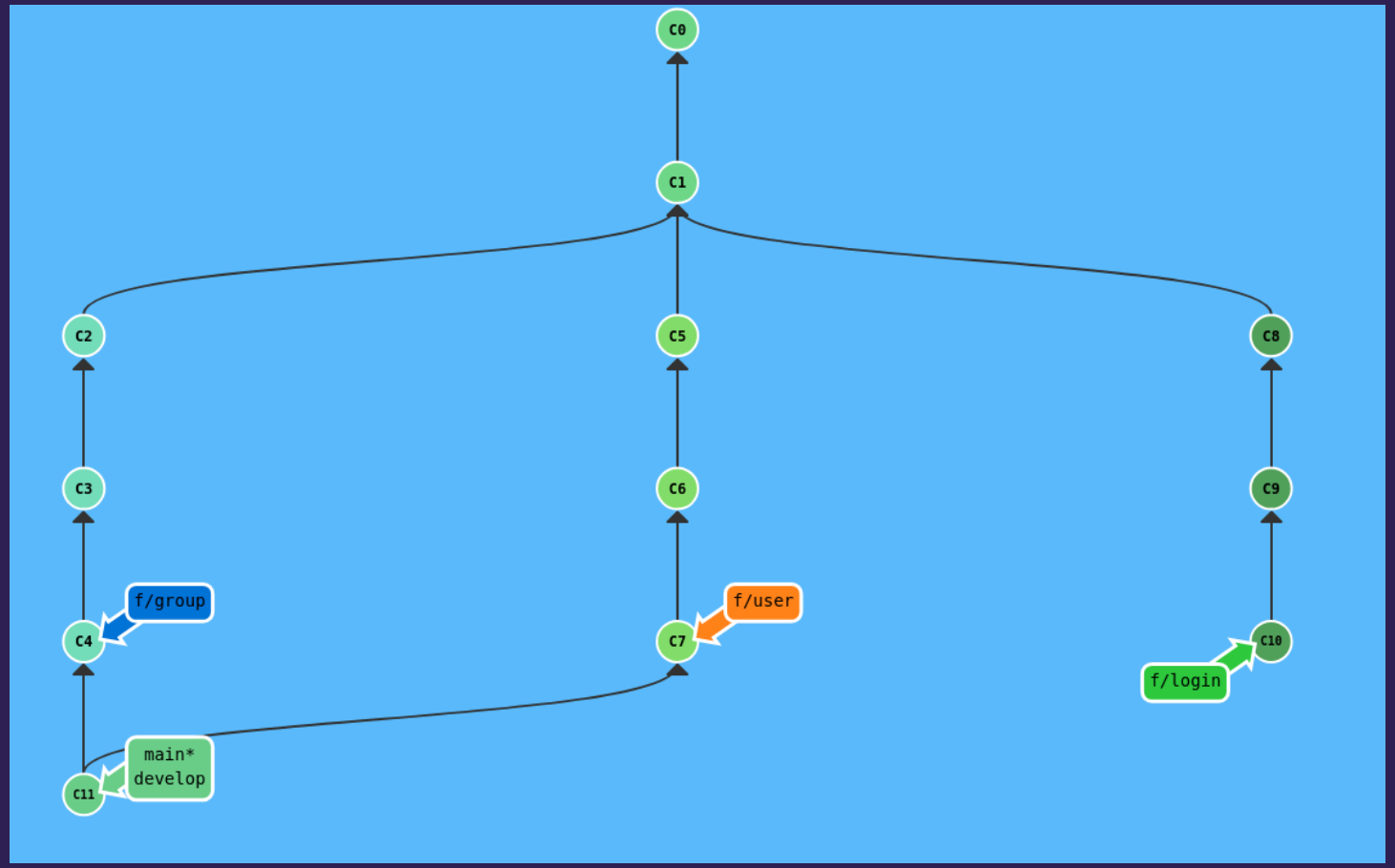
Git Flow - Flujo Feature

```
# Crear feature desde develop
git switch -c develop
git switch -b feature/login

# Desarrollo y commits
git commit -m "feat: add login form"
git commit -m "feat: add validation"

# Merge a develop
git switch develop
git merge feature/login-usuario
```

learngitbranching



GitHub Flow

El flujo más simple y directo para desarrollo continuo.

Rama principal:

- `main`: Siempre desplegable y estable

Ramas de trabajo:

- `feature/*`: Nuevas funcionalidades
- `bugfix/*`: Corrección de errores
- `hotfix/*`: Correcciones urgentes

Principio: Todo sale y entra de `main`

GitHub Flow - Características

Ventajas:

- Muy simple de entender y aplicar
- Deploy continuo natural
- Ideal para equipos ágiles
- Menos overhead de gestión

Desventajas:

- Requiere CI/CD robusto
- `main` debe estar siempre estable
- No ideal para releases planificados

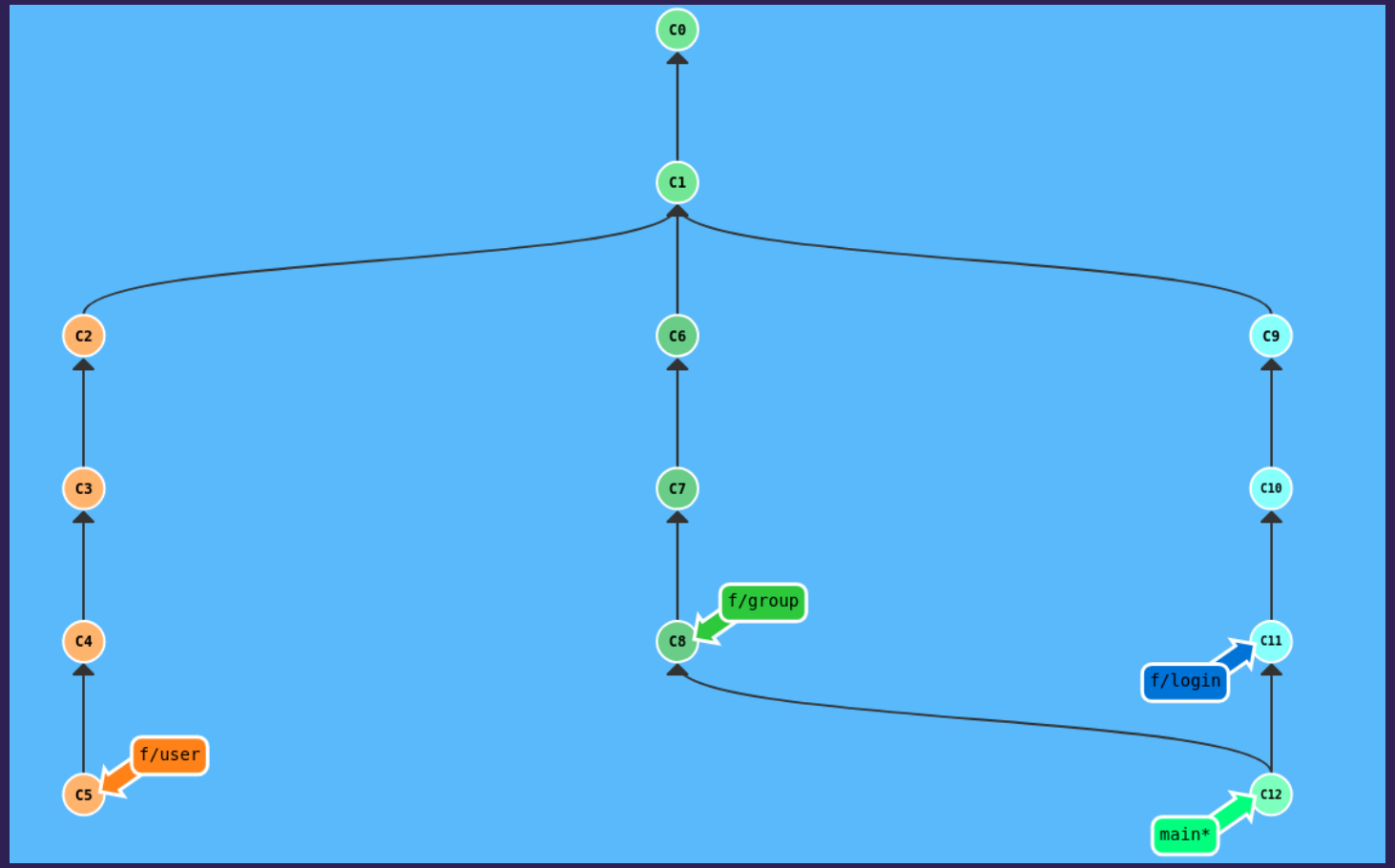
GitHub Flow - Flujo básico

```
# Crear feature desde main
git switch main
git pull origin main
git switch -c feature/user-profile

# Desarrollo
git commit -m "feat: add user profile page"
git push -u origin feature/user-profile

# PR → Review → Merge → Deploy
```

learngitbranching



Trunk-based Development

Un tronco principal donde todos integran frecuentemente.

Rama principal:

- `main` o `trunk`: Única rama principal

Ramas de trabajo:

- Muy cortas (< 1 día) o commits directos
- `hotfix/*`: Solo para correcciones urgentes

Principio: Integración continua real

Trunk-based - Características

Ventajas:

- Integración continua real
- Sin merge hell
- Deploy frecuente y rápido
- Feedback inmediato

Desventajas:

- Requiere desarrolladores senior
- CI/CD muy robusto obligatorio
- Feature flags necesarios
- Tests automáticos críticos

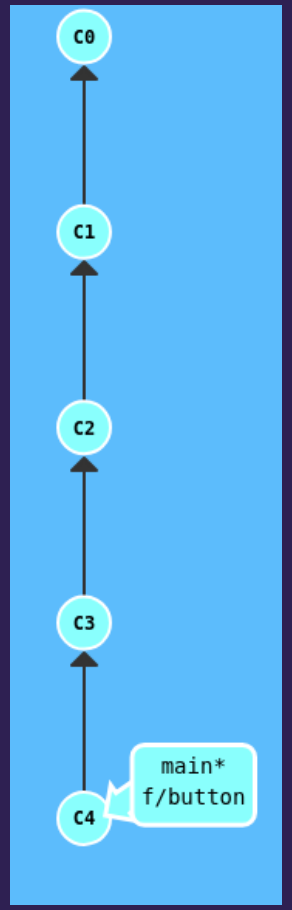
Trunk-based - Flujo básico

```
# Actualizar trunk frecuentemente
git switch main
git pull origin main

# Cambios pequeños directos
git commit -m "feat: add button component"
git push origin main

# O ramas ultra-cortas
git switch -c quick-fix
git commit -m "fix: button alignment"
git push origin quick-fix
# PR inmediato → Merge → Delete
```

learngitbranching



Ship/Show/Ask

Estrategia que combina autonomía con colaboración según el tipo de cambio.

- **Ship** 🚢: Cambios que puedes hacer directamente
- **Show** 👁️: Cambios que compartes para feedback
- **Ask** ❓: Cambios que requieren discusión previa
- **Principio:** Balance entre velocidad y control

Ship/Show/Ask - Características

Ventajas:

- Autonomía para desarrolladores senior
- Velocidad en cambios menores
- Control en cambios críticos
- Cultura de confianza en el equipo

Desventajas:

- Requiere criterio maduro del equipo
- Necesita desarrolladores experimentados
- Puede crear inconsistencias si no hay criterio
- Difícil para equipos nuevos

Ship/Show/Ask - Criterios

Ship (Push directo):

- Typos en documentación
- Tests adicionales
- Refactoring menor
- Actualizaciones de dependencias

Show (PR sin esperar aprobación):

- Features pequeñas
- Cambios de UI menores
- Optimizaciones de rendimiento

Ask (PR con revisión obligatoria):

- Cambios de arquitectura
- APIs públicas
- Features complejas
- Cambios de seguridad

Ship/Show/Ask - Flujo

```
# SHIP - Fix documentación
git commit -m "docs: fix typo in README"
git push origin main

# SHOW - Feature pequeña
git switch -c small-feature
git push -u origin small-feature
# Create PR → Optional review → Merge

# ASK - Cambio grande
git switch -c big-change
# Discutir primero → PR → Required review
```

learngitbranching