Git: *gitworkflows* et bonnes pratiques

Comment utiliser les branches ?

Git vous *encourage* à utiliser les *branches au maximum*.

Comment gérer ces branches lorsque l'on travaille seul·e ? À plusieurs ?

C'est à ça que servent les *gitworkflows* (manière recommandée de travailler avec git)

> Situation de départ: un dépôt distant (*Decentralized but centralized*), une équipe de dévelopeur·ses

Beaucoup de stratégies possibles

Beaucoup de choix possibles pour le modèle de développement d'un projet :

- faire une branche par feature,
- faire une branche par équipe,
- faire une branche par personne,
- utiliser uniquement des merges, ou uniquement des rebase,
- créer des tags pour identifier des releases,
- utiliser des forks pour développement donc un fork par personne,
- etc.

> Pas de solution parfaite ! À vous de trouver celle qui convient à votre équipe, votre projet

gitworkflows visités

- Gitflow
- Githubflow
- Workflow alternatif
- Workflow adapté à une agence web

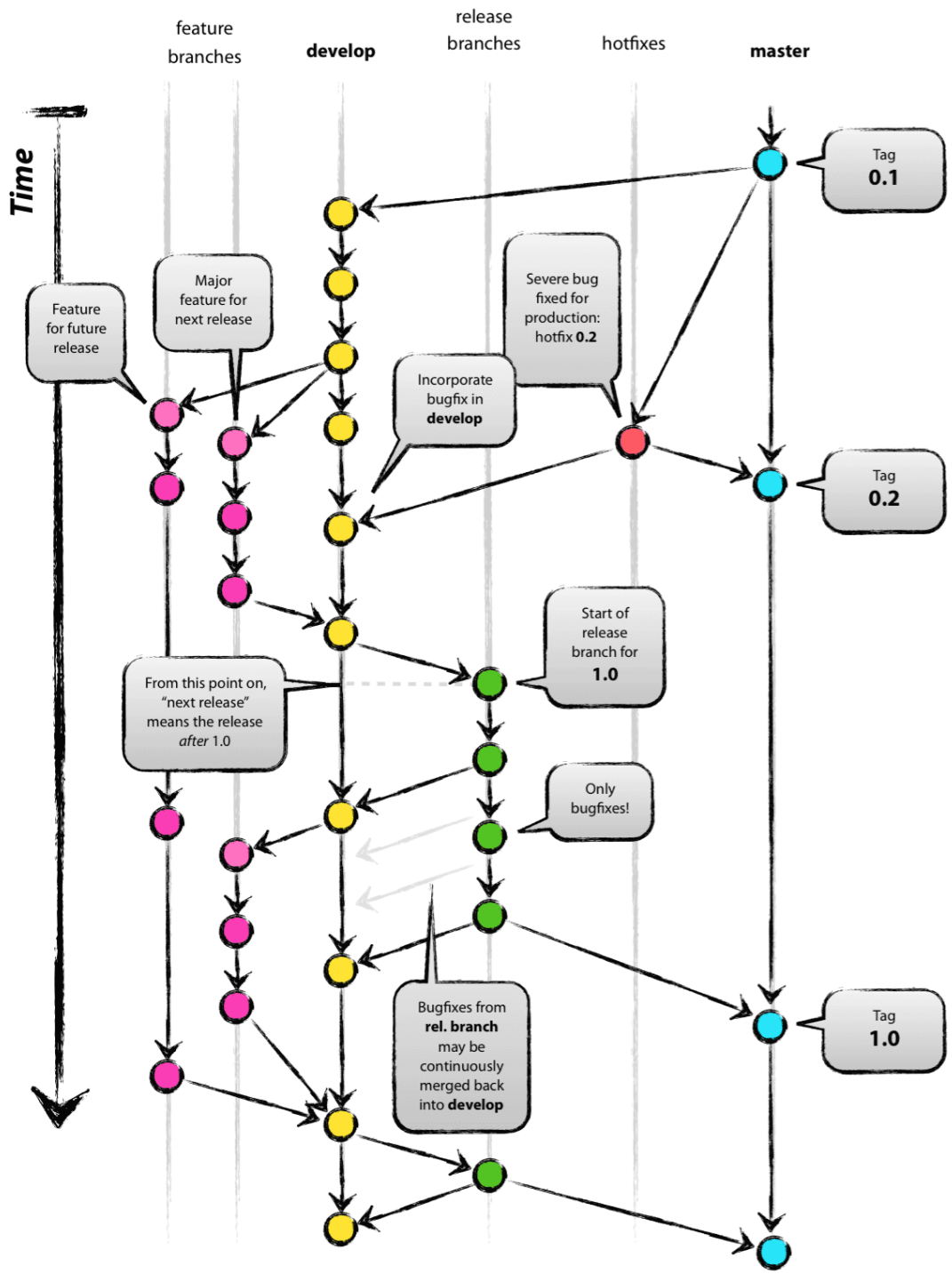
Gitflow

Publié en 2010 par Vincent Driessen (il y a plus de 10 ans)

Philosophie:

- quelques branches principales (`main`, `develop`)
- des branches secondaires
- basé sur des merge

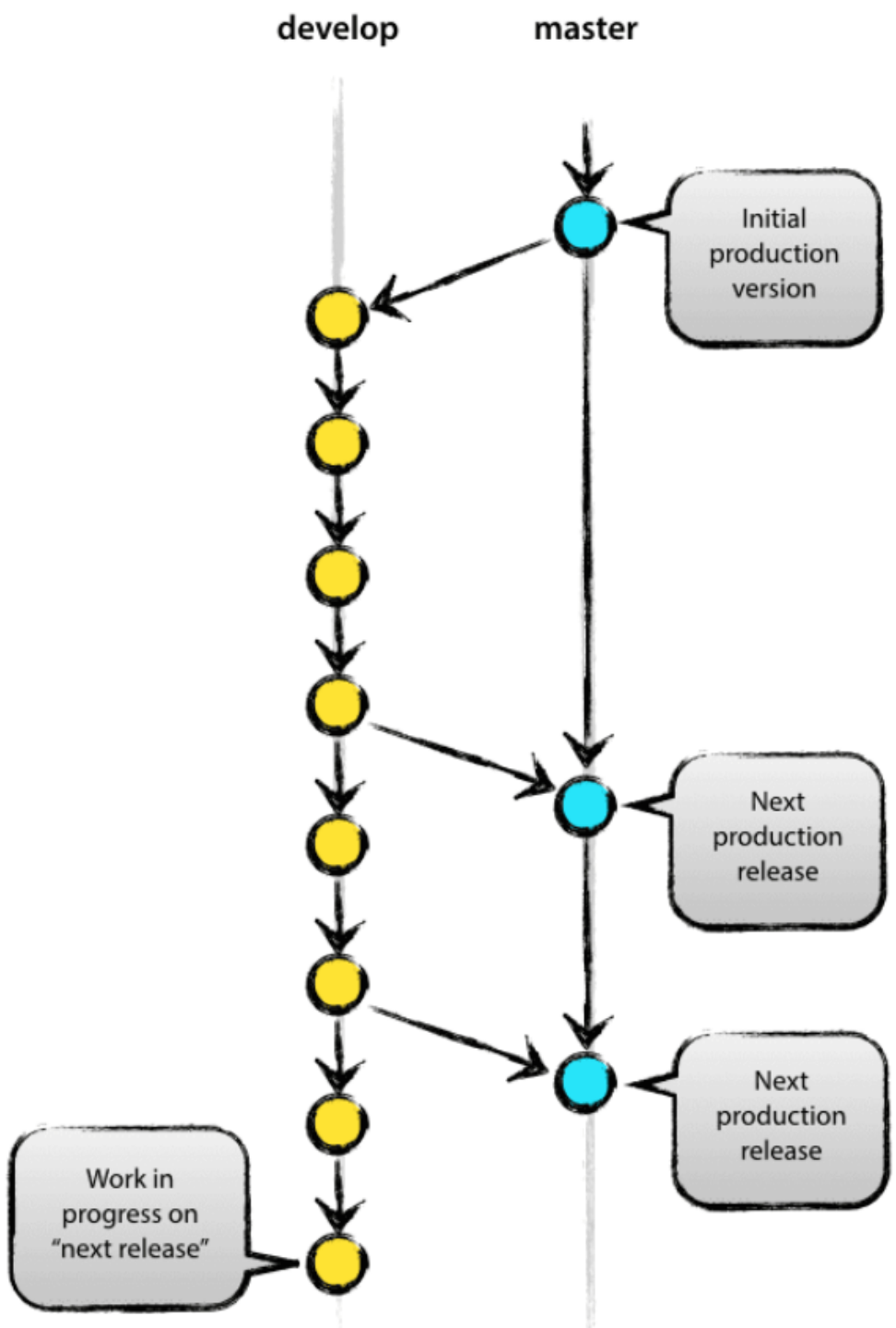
> On a abandonné la convention maître-esclave (*master*-slave), et tant mieux ! On parle aujourd'hui de branche *main*



Gitflow: branches principales

- la branche `main` est toujours dans un état *production-ready*
- la branche `develop` reflète toujours les derniers développements (branche d'intégration)

Par définition, chaque merge de `develop` dans `main` **est une nouvelle version de production** (`release`).



Gitflow: branches de support (*feature*, *release*, *hotfix*)

Aux côtés des branches principales, on trouve d'autres branches pour faciliter le travail parallèle.

Trois types de branches secondaires:

- **feature** branches
- **release** branches
- **hotfix** branches

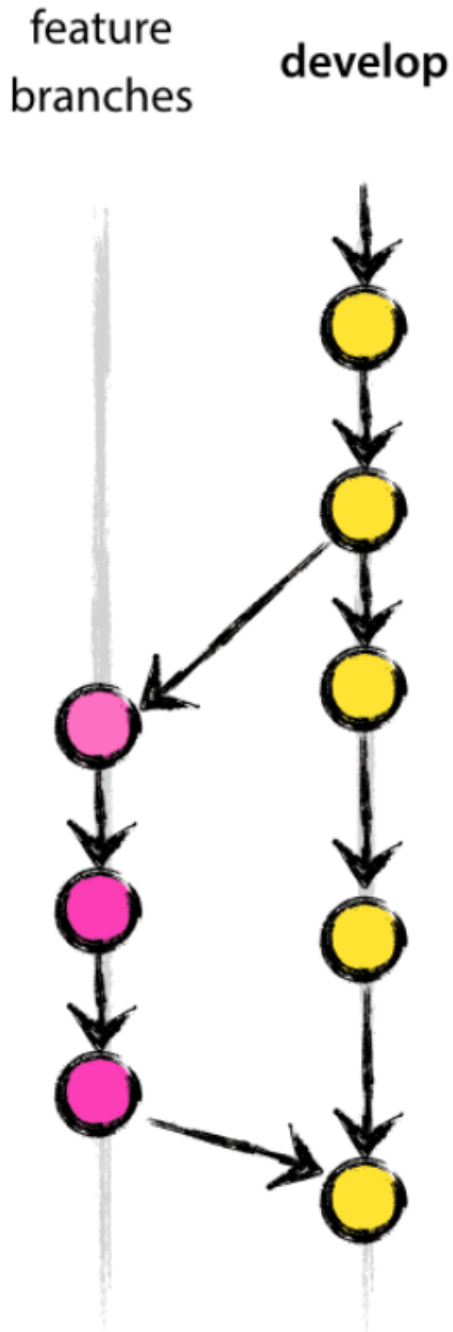
Chacune de ces branches **a un but précis** !

Feature (ou *topic*) branch

But: développer de nouvelles fonctionnalités pour la prochaine version.

- Doit être *tirée* depuis `develop`
- Doit être mergée dans `develop`
- Convention de nom: tout sauf `main`, `master`, `develop`, release-* ou hotfix-*

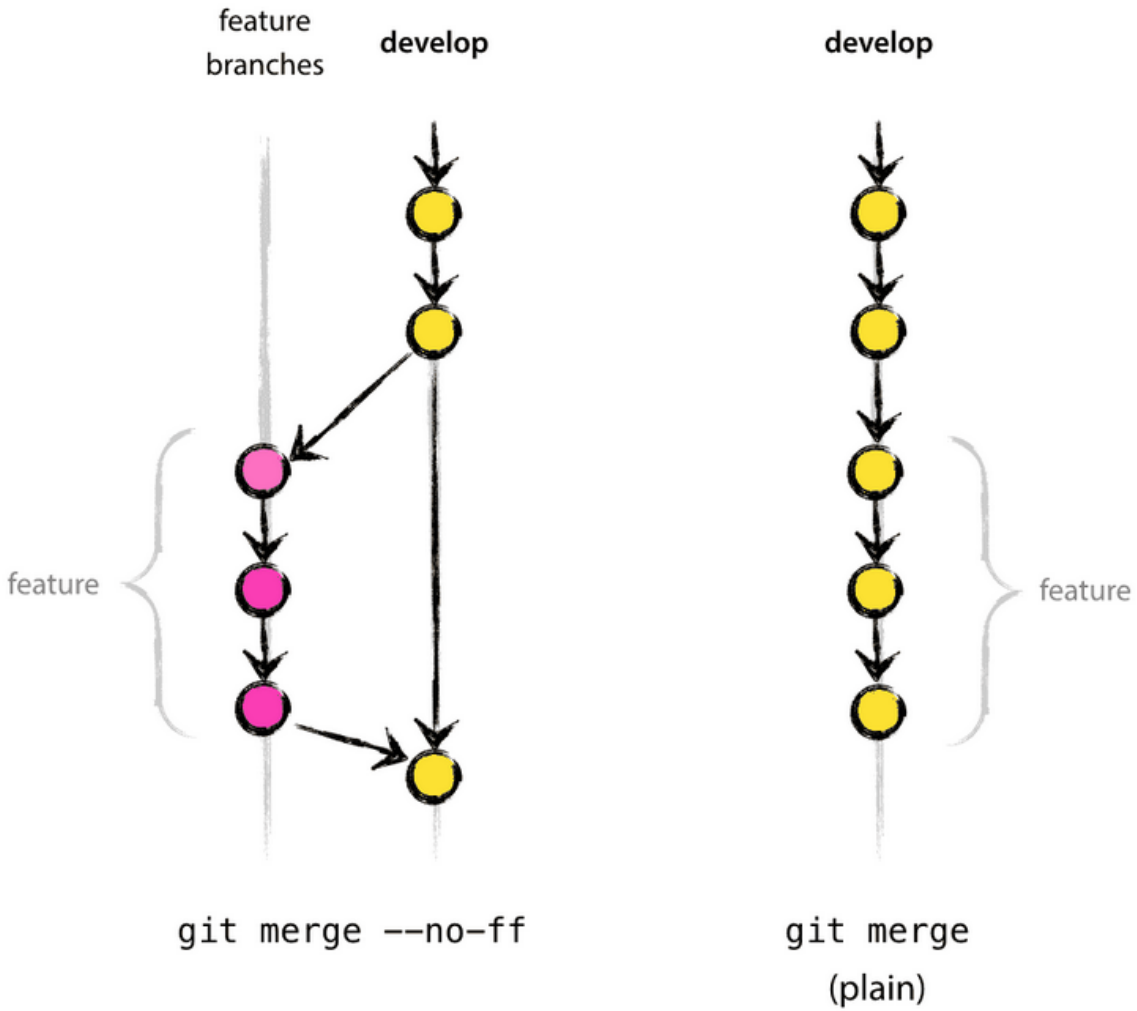
> Les feature branch se trouvent typiquement dans le repo **local** de la personne qui développe, *pas sur origin* (dépôt remote)



Feature (ou *topic*) branch

Attention sur le merge

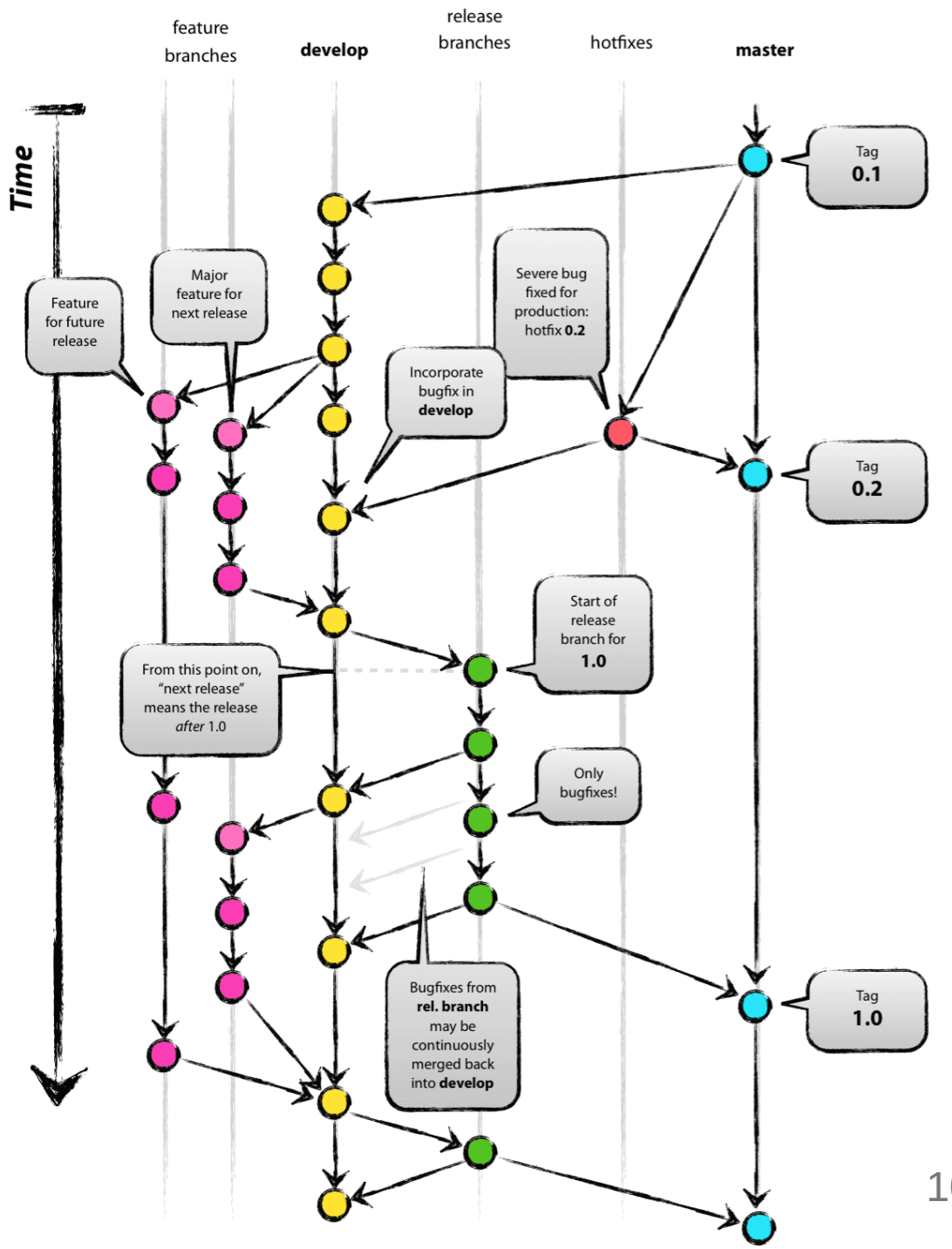
`git merge --no-ff` (no fast-forward): crée **toujours** un commit de merge.



Release branch

But: stocker le travail pour préparer la future release. Permets de garder la branche develop propre (réservée aux développements de features, non au fixs de dernières minutes comme version number, build dates, etc.)

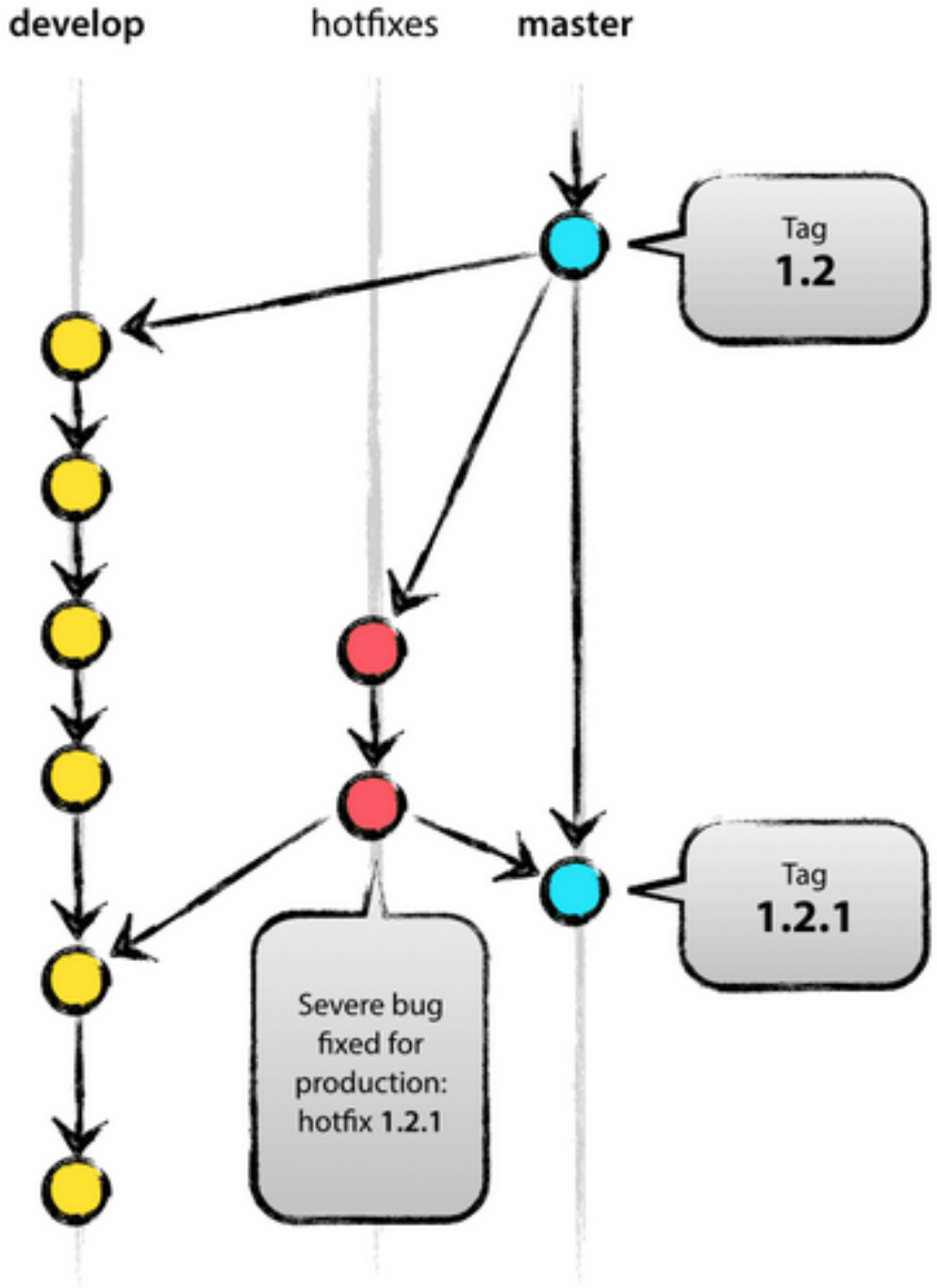
- Doit être *tirée* depuis `develop`
- Doit être mergée dans `develop` et `main`
- Convention de nom: release-*



Hotfix branch

But: agir immédiatement pour corriger un état indésirable en production (*hot* fix)

- Doit être *tirée* depuis `main`
- Doit être mergée dans `develop` et `main`
- Convention de nom: hotfix-*



Gitflow, le bilan

Avantages:

- **ça marche !**
- général
- tentative de *standardisation* du workflow sur git
- des outils existent pour forcer le workflow
- bien pour maintenir **une version d'un projet** (par exemple le site web de la boîte)
- `main` n'a que du code stable

Inconvénients:

- beaucoup, parfois trop de branches !
- basé sur les *merge*: historique parfois compliqué et *merge hell*
- branches parfois courtes, parfois *longues* (dur à merger)
- pas toujours adapté aux pratiques dans l'entreprise/groupe (agilité) et aux nouvelles pratiques de CI/CD

Démo

GitHub flow

GitHub flow est un flot plus **simple** que Gitflow, basé sur des branches.

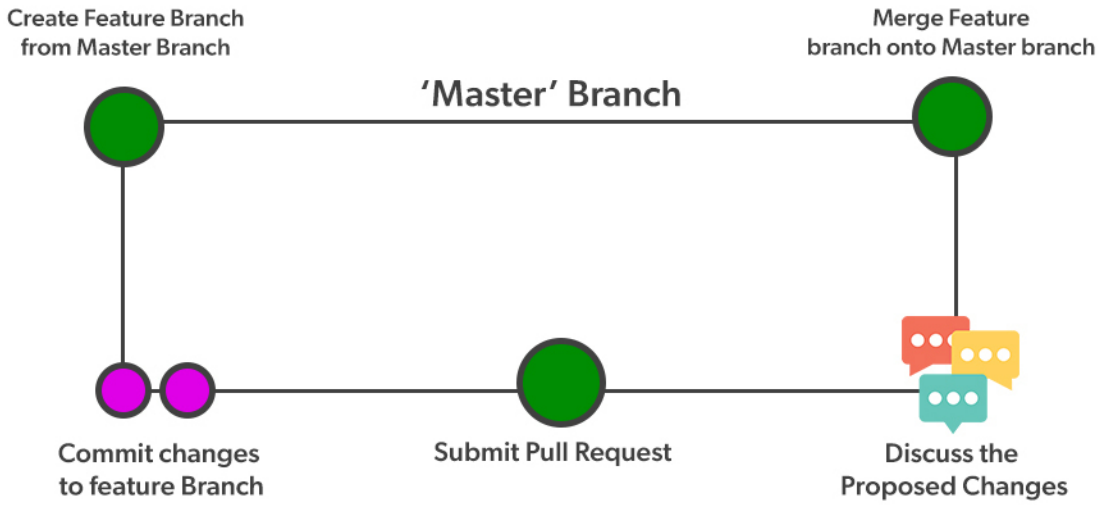
Adapté pour déployer en production tous les jours.

2 types de branches:

- la branche principale `main`
- des branches de développement secondaires

Pull Request au centre du flot

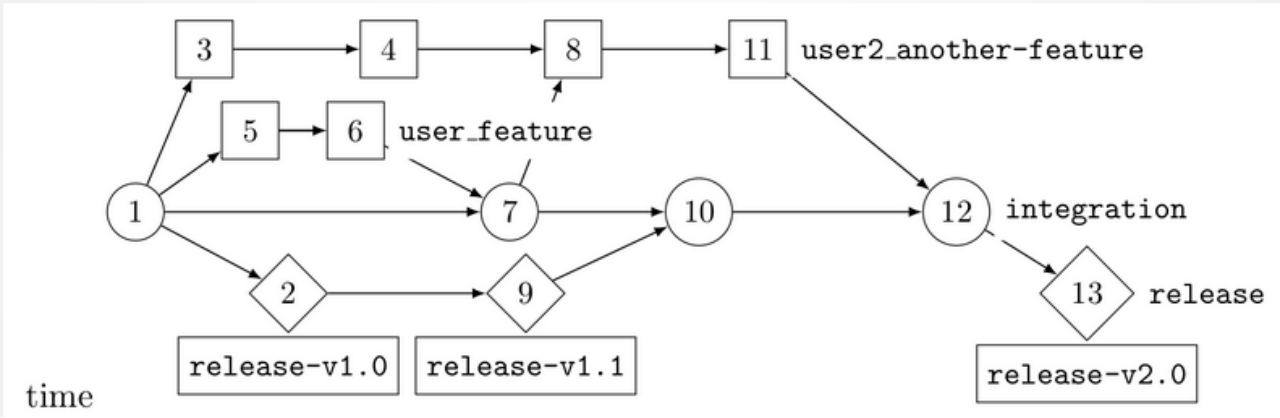
> Pas de gestion de releases (de versions)



Workflow alternatif

Workflow publié dans le cadre de la maintenance d'un code de simulation numérique, pour améliorer le suivi des défauts.

- branche `integration` : branche principale, tout y est stable (comme `main`)
- branches `releases` : contient les différentes versions du projet
- merge les branches `releases` à `integration` quand elles sont stables



Workflow alternatif

Avantages:

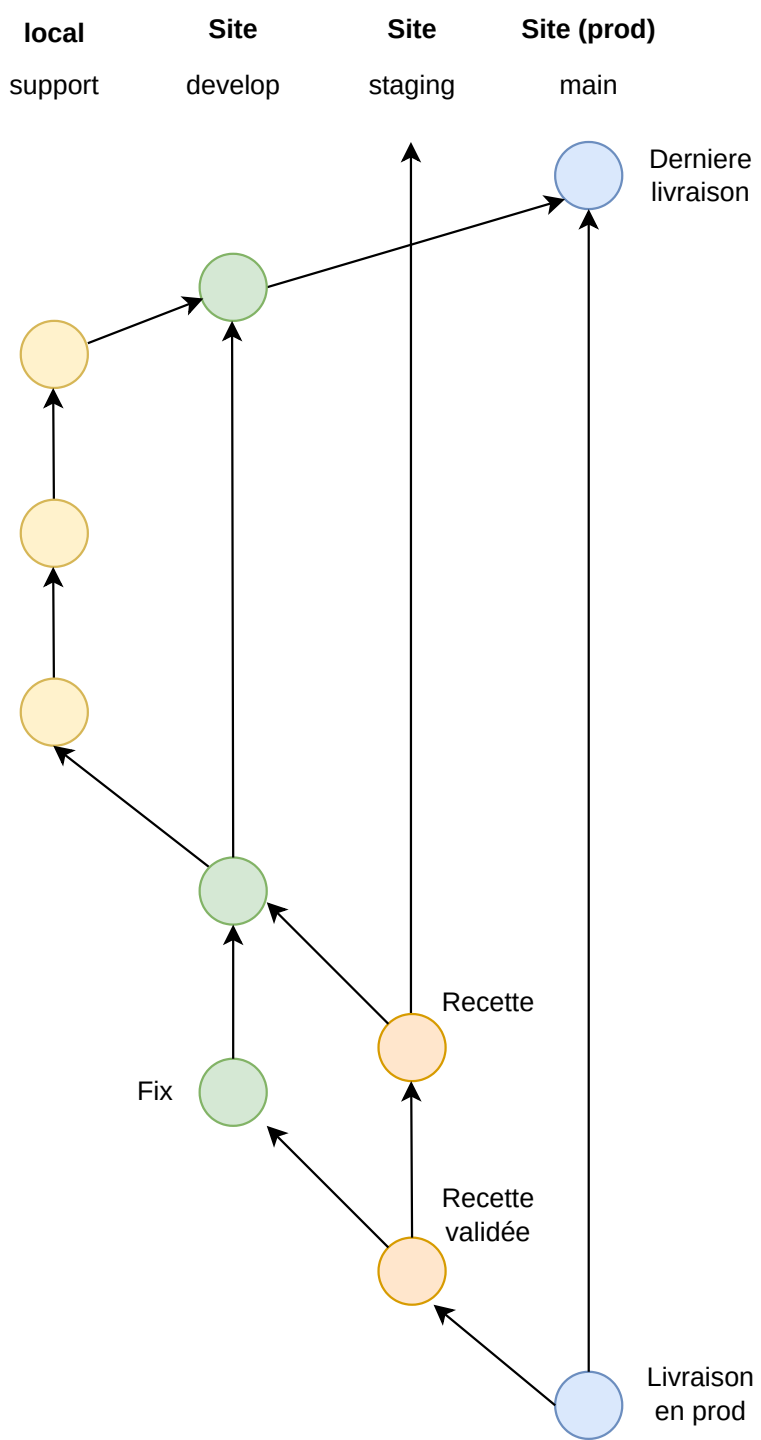
- branches courtes
- branche de développement est centrale
- gestion naturelle des versions concurrentes du logiciel

Exemple de workflow adapté à une agence web

Une variation de gitflow, adaptée à la livraison d'applications/sites web.

3 branches principales:

- `main` (site en production)
- `staging` (site de recette)
- `develop` (site en développement)
- branches support



Conseils généraux

- les branches doivent être *courtes* (dans le temps et en commit)
- plus une branche est isolée longtemps, plus elle est difficile à merger (récupérer le travail)

Des commit bien faits

- chaque commit contient une modification isolée et complète (*atomique*): il doit concerner une tâche ou un fix
- ne commit que lorsque votre travail est terminé

Le message d'un commit

Très important ! Ce sont les *logs* du projet. Améliore le process de reviewing.

Donner du *contexte* (*why*) à chaque changement apporté (pour les autres et son futur soi)

> Source: xkcd

Le message d'un commit, quelques guides à suivre

- Message *concis* et de *taille consistante* (sujet < 50 caractères, corps < 72 caractères)
- Séparer le *sujet* du *corps* par un saut de ligne
- Capitaliser la première lettre de la première ligne, pas de point à la fin de la première ligne

- Utiliser le *mode impératif* à la 2e personne (présent). Par exemple: *Parle*, *Met à jour*, *Corrige*
- Écrire le sujet pour qu'il complète la phrase suivante: *Si appliqué, ce commit* `<votre sujet>`. Par exemple, *Si appliqué, ce commit* `Met à jour la section getting started de la doc`
- Ne racontez pas *comment* (le code le fait) mais *pourquoi* vous avez changé quelque-chose, qu'est-ce qui n'allait pas *avant*, comment ça marche *à présent*, pourquoi vous avez décidé de le régler *comme ça*.

Références

- Recommandations sur les gitworkflows sur le site officiel de git
- A successful Git branching model, le post original de Vincent Driessen présentant le modèle Gitflow
- Comparing Git Workflows: What You Should Know, un article sur les gitworkflows publié par Bitbucket
- Adopt a git branching strategy, article publié par Microsoft sur les stratégies à mettre en place
- Opinionated Git, opinions sur la manière d'utiliser git par Johan Abildskov
- How to Write a Git Commit Message, quelques conseils sur comment écrire un bon message de commit
- Merge vs Rebase: Do They Produce the Same Result?, un article d'Edward Thomson sur la différence entre merge et rebase et de l'impact de chaque approche sur l'historique du projet
- Developer Tip: Keep Your Commits “Atomic”
- An approach to increase readability to HPC simulation, application to the GISEL5D code, publié par Julien Bigot, Guillaume Latu, Thomas Cartier-Michaud, Virginie Grandgirard, Chantal Passeron et Fabien Rozar, dans la revue ESAIM: PROCEEDINGS AND SURVEYS, 2016. Voir l'abstract et la section 2.2