

**DevOps**

# **Collaboration avec Git -- Git workflows**

**Thomas Ropars**

**Email:** [thomas.ropars@univ-grenoble-alpes.fr](mailto:thomas.ropars@univ-grenoble-alpes.fr)

**Website:** [tropars.github.io](https://tropars.github.io)

# Dans ce cours

- Collaborer sur un projet avec Git
  - Différents modèles de coopération
- Collaborer en utilisant Github/Gitlab
- Notion de Pull/Merge Request

# **Les workflows**

# Remarques introductives

- Ce cours présente plusieurs modèles de coopération
- Il n'existe pas de modèle unique, supérieur à tous les autres
- Le modèle le plus adapté dépend de:
  - La taille du projet
  - Le type de projet
  - Le cycle de développement considéré pour le projet
  - Les habitudes des collaborateurs
  - etc.

**Le modèle le plus adapté peut être un mélange de concepts introduits par plusieurs modèles existants**

# Objectifs

## **Projet impliquant plusieurs collaborateurs**

- Travailler à plusieurs en parallèle
  - Mettre à disposition ses contributions
  - Récupérer les contributions faites par les autres
- Travailler sur plusieurs choses en parallèle
- Avoir un projet toujours fonctionnel
- Avoir un historique clair

## **Exploiter les fonctionnalités de Git pour atteindre ces objectifs**

- branch
- merge
- rebase (interactif)
- etc.

# Les workflows

- Approche centralisée
- Feature branch
- Github workflow
- Gitlab workflow

# **Approche centralisée**

# Approche centralisée

## Principe

- Les développeurs travaillent sur la même branche
  - `master`

## Les étapes

- Chacun travaille sur sa branche `master` locale
- Quand le travail est fini:
  - `Fetch` de la branche distante
  - `Merge` de la branche distante et de la branche locale
  - `Push` vers le serveur

*Ici, par travail, on entend: une correction de bug, l'ajout d'une fonctionnalité, etc.*



# Les étapes (en détails)

```
# Cloner le dépôt central
$ git clone ssh://user@host/path/to/repo.git

# Après avoir fait ses modifications
# Ajout et commit des modifications (branche locale)
$ git add ...
$ git commit -m "my commit"

# !!! Essayer de push directement va probablement échouer
# (si la branche locale n'est pas à jour par rapport à l'upstream)
$ git push origin master
```

Crédit: <https://www.atlassian.com/git/tutorials/comparing-workflows#centralized-workflow>

# Les étapes -- gestion des conflits

```
# Incorporer les changements de l'upstream dans la branche locale
# --rebase pour garder un historique linéaire
$ git pull --rebase origin master

# Message en cas de conflit (le rebase est interrompu):
  CONFLICT (content): Merge conflict in

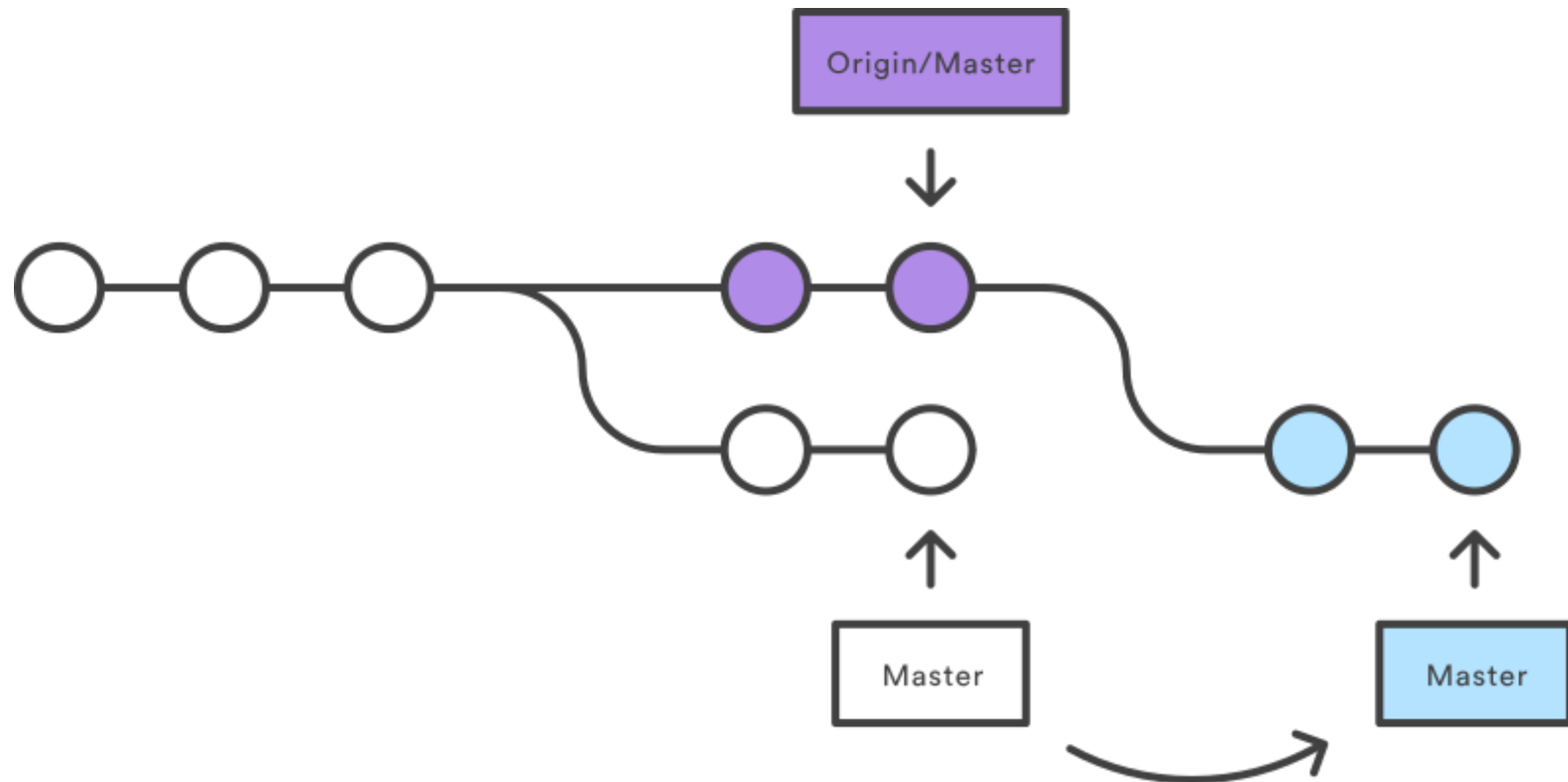
# Résoudre les conflits (par exemple avec un mergetool)
$ git mergetool

# Terminer le rebase
$ git rebase --continue

# Envoyer son travail vers le serveur
$ git push origin master
```

# Le résultat en image

## Historique après rebase



# Bilan

## Points positifs

- Modèle simple
  - Mécanisme de base utilisé par les autres modèles

## Points négatifs

- Rend difficile le travail sur plusieurs fonctionnalités en parallèle
  - Beaucoup de merges *inutiles*
- N'exploite pas les fonctionnalités de Git

# **Le workflow Feature Branch**

# Le workflow Feature Branch

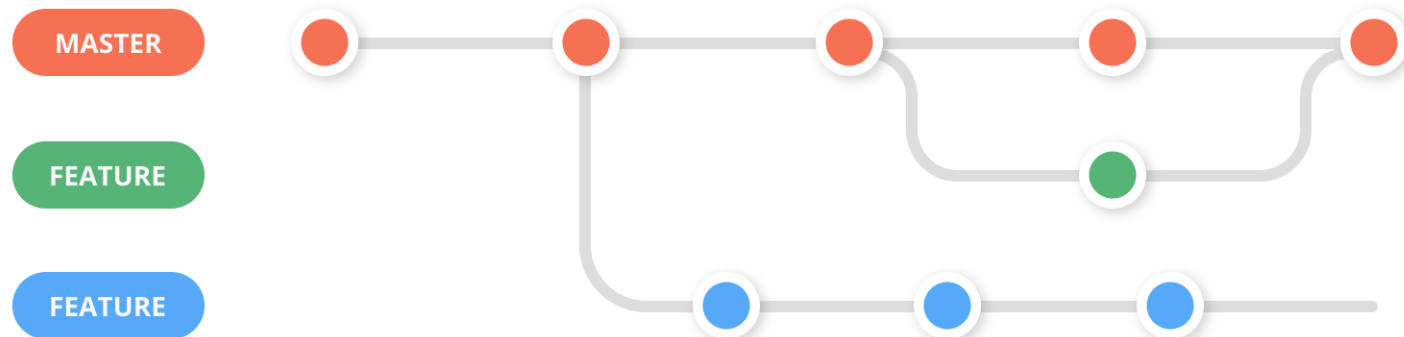
## Principe

- Chaque nouvelle fonctionnalité (ou bugfix) est développée dans une branche séparée
  - Le nom de la branche reflète l'objectif de la branche
- Les branches sont poussés vers le dépôt serveur
  - Permet à plusieurs développeurs de collaborer sur la même feature
  - L'approche *centralisée* est appliquée au sein de chaque branche

## Les étapes

- Création d'une nouvelle branche
- Travail sur la branche
- Merge de la branche avec `master`

# Illustration



Crédits: <https://zepel.io/blog/5-git-workflows-to-improve-development/>

# Les étapes en détails

```
# Partir du dernier commit sur master
$ git pull origin master

# Créer une nouvelle branche locale à partir de master
# et s'y déplacer
$ git switch -c new-feature

# Après avoir fait ses modifications
# Ajout et commit des modifications (branche locale)
$ git add ...
$ git commit -m "my commit"
```



# Les étapes en détails

```
# Créer la branche distante correspondante
# et envoyer ses modifications
$ git push -u origin new-feature

# Les commits suivants peuvent être poussés plus simplement
$ git push

# Le modèle centralisé s'applique pour la suite du dev
$ ...
```

# Les étapes en détails

## Merge avec master

```
# Récupérer les dernières modifications sur master
$ git switch master
$ git pull origin master

# Merger au sein de la branche -- résoudre les conflits si besoin
$ git switch new-feature
$ git merge master

# Fast forward de master (intégrer les changements dans master)
$ git switch master
$ git merge new-feature

# (Pousser les modifications) et supprimer la branche
$ git branch -d new-feature
$ git push origin --delete new-feature
```

# Revue de code et pull request

## Revue de code

- Procédure de relecture du code d'une autre personne
- Vérification de la qualité du code ajouté
- Peut être introduit dans le workflow *Feature Branch* avant intégration des modifications dans la branche `master`

## Pull/Merge request

- Mécanisme fourni par les services de gestion de code source pour mettre en place une procédure de revue/validation de code avant intégration dans la branche `master`

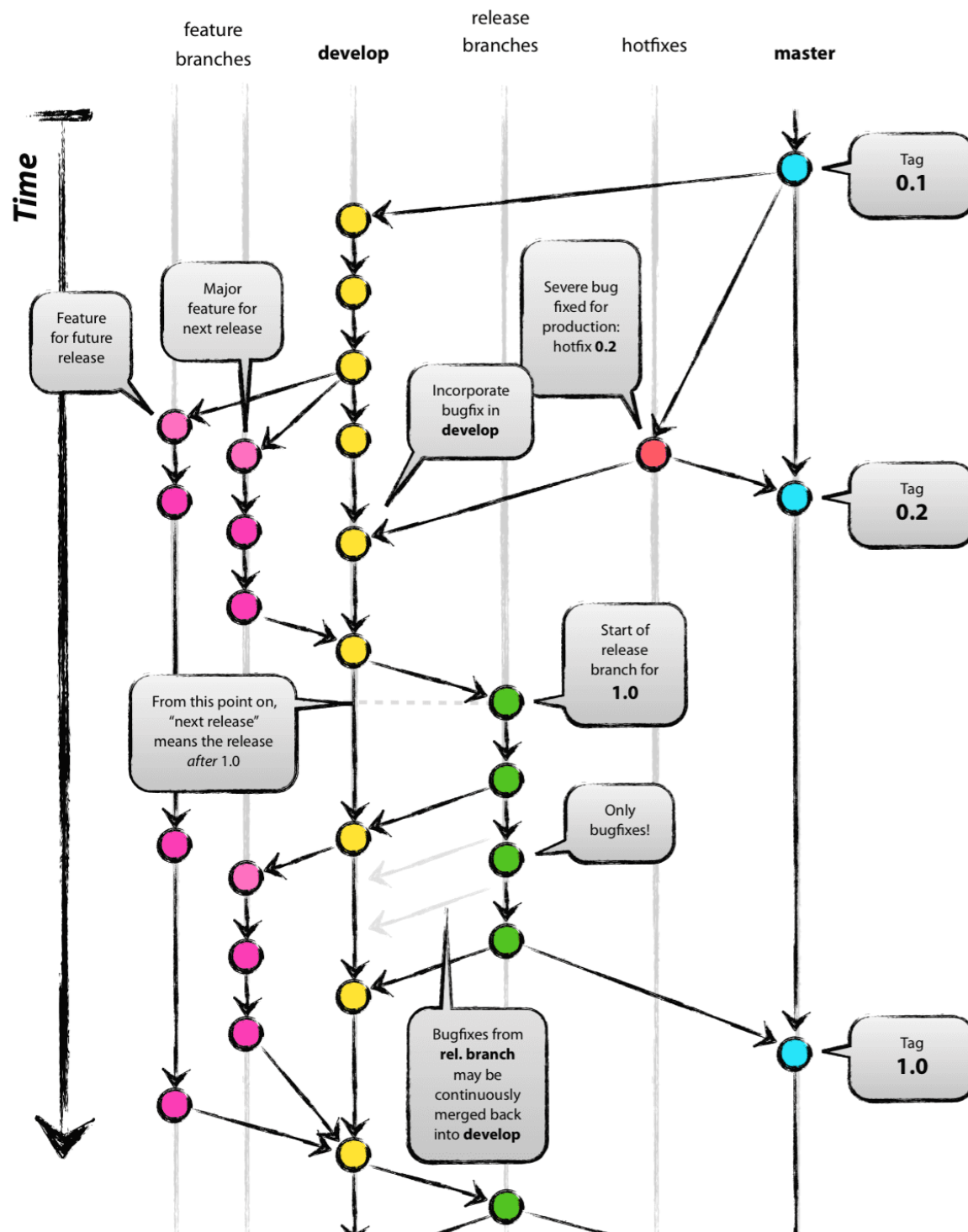
**Plus de détails à venir**

# Gitflow Workflow

# GitFlow Workflow

## Qu'est ce que c'est?

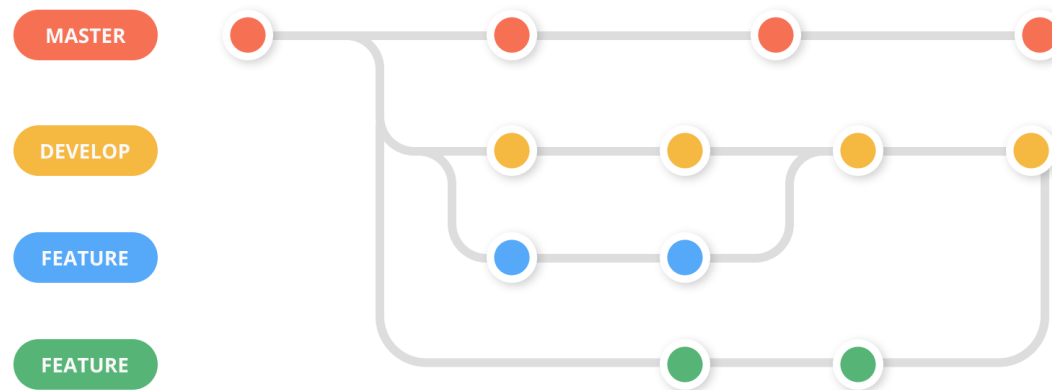
- [Un post de Blog](#) par Vincent Driesen
  - Devenu très populaire
  - Considéré par certains comme un standard
- Décrit un modèle de workflow avec Git visant des projets complexes
  - Projet pour lesquels plusieurs versions d'un logiciel doivent être maintenues
  - Projet basé sur la création de releases
  - Souvent trop complexe pour d'autres types de projets
- Lire la discussion ajoutée par l'auteur au début de son post



# Nouveaux concepts

## Une nouvelle branche principale: `develop`

- Branche intégrant les dernières fonctionnalités
- Les branches `feature` sont créés à partir de cette branche
- La branche `master` contient la dernier version *stable*
  - Les nouvelles contributions de la branche `develop` sont intégrées dans `master` lors de la création d'une nouvelle version du logiciel



# Nouveaux concepts

## Des branches en plus

- Branche **release**
  - Branche créée à partir de la branche `develop`
  - Nettoyage du code avant de faire une release
  - Branche intermédiaire avant de merger dans `master`
    - A merger dans `master` et `develop`
- Branche **hotfix**
  - Branche créée à partir de la branche `master`
  - Correction de bug pour du code déjà présent dans `master`
  - A merger dans `master` et `develop`



# **Autres workflows**

# D'autres workflows existent

Parmi les principaux, des workflows liés/proposés par les services de gestion de code source:

- Workflow Github
- Workflow fondé sur Fork
- Workflow Gitlab (voir [ici](#) et [là](#))
  - Version simplifiée du workflow GitFlow
  - Suppression de la branche `develop`
  - Un ensemble de recommandations en plus de la seule description du workflow

# Workflow Github

Workflow bien adapté pour des projets fondés sur de la livraison continue (c.a.d non basé sur des releases) - Workflow utilisé en interne chez Github

## Les principes

1. La branche `master` doit toujours être déployable
2. Utiliser le concept de *feature branch* pour les nouveaux développements
3. Poussez régulièrement les modifications vers une branche sur le serveur
4. Créer une *pull request* quand vous avez besoin de feedback et avant de merger avec `master`
5. Ne merger avec `master` qu'une fois votre code relu par quelqu'un d'autre
6. Tester un déploiement avant de merger avec `master`

Voir [l'article original](#) et la [doc actuelle](#)

# Workflow fondé sur Fork

## Notion de Fork

- Concept introduit par les services de gestion de versions
- Création d'un nouveau projet à partir d'un projet existant
  - Permet d'introduire des modifications sans affecter le projet original
  - Permet de travailler sur un projet qui ne nous appartient pas

## Les étapes

1. Fork d'un projet existant à partir de l'interface graphique du service de gestion de versions
2. Travail sur le nouveau projet
  - Typiquement selon le modèle *feature branch*
3. Création d'une pull/merge request pour demander l'intégration des nouvelles contributions dans le projet original

# **Pull/Merge request**

# Pull/Merge request

## Présentation

- Mécanisme fournit par les services de gestion de code source pour mettre en place une procédure de revue/validation de code avant intégration dans la branche de développement principale
- Pull request = Merge request
  - **Pull request:** Nom utilisé par Github
  - **Merge request:** Nom utilisé par Gitlab
  - Les mêmes fonctionnalités sont offertes
- Une *Merge request* peut aussi servir pour demander des retours sur son travail

# Description

## Les étapes

1. Travailler avec une *Feature branch*
  - Une *Pull request* se fait toujours à partir d'une nouvelle branche
2. Initialiser la *Merge request* à partir de l'interface Web
  - Par défaut, merge dans `master` (peut être modifié)
  - Désigner une personne responsable d'accepter la *Merge request* (*assignee*)
  - Demander à des personnes de relire votre contribution (*reviewer*)
  - etc.
3. Itérer sur la requête en fonction des commentaires jusqu'à validation de la requête
  - La branche peut être supprimée automatiquement une fois la requête acceptée

# **Quelques commentaires en plus**



# Bonnes pratiques (ou pas)

Quelques commentaires en plus sur *feature branch* et *merge request*.

## Committer souvent

- L'utilisation de *feature branch* permet de committer souvent sans perturber le travail des autres contributeurs
  - En particulier quand on travaille seul sur la branche

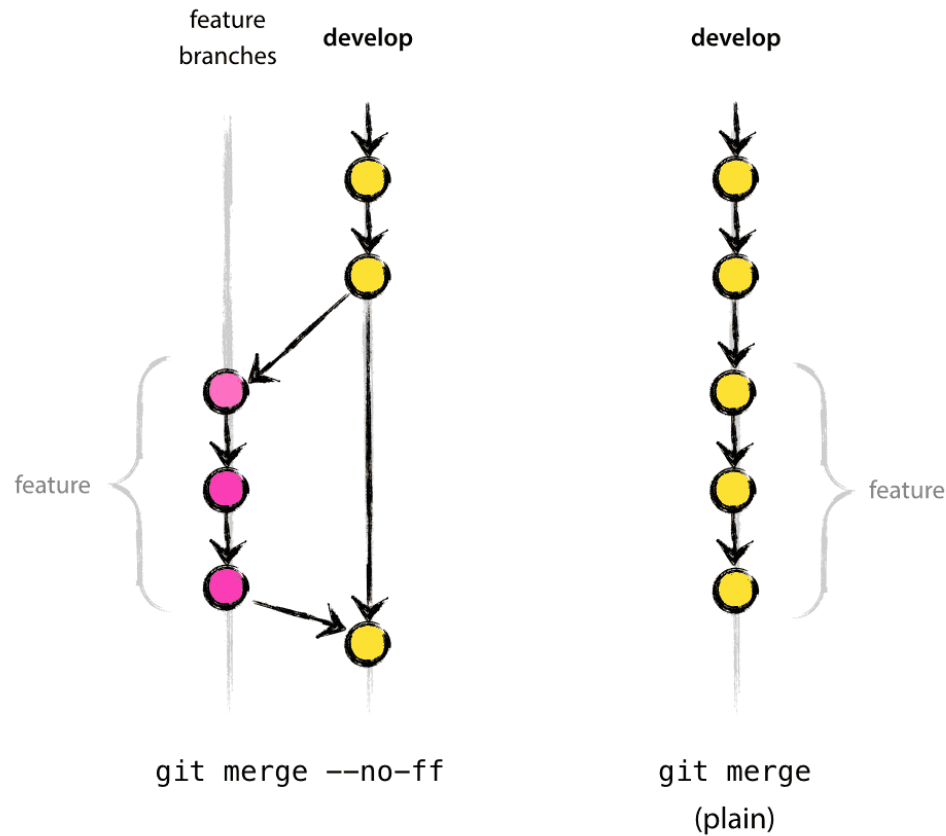
## Agrégation de commits

- Une bonne pratique peut être d'agréger (*squash* avec un rebase interactif) ses commits avant de créer une *merge request* (ou une fois la requête accepté)
  - Objectif: obtenir un historique simple à lire
  - Contre-arguments:
    - Conserver l'ensemble des commits peut permettre de mieux comprendre le cheminement du contributeur
    - L'historique peut être simplifiée à l'affichage avec des filtres (par ex: option `--first-parent` de `git log`)

# Merge avec l'option --no-ff

L'option `--no-ff` permet de créer un nouveau commit même lorsque le `merge` implique simplement un *fast-forward*

- Objectif: améliorer la lisibilité de l'historique



# Références

- [Git workflows](#) par Atlassian
- [GitFlow](#) par Vincent Driesen
- [Github workflow](#) par Scott Chacon