Intégration Continue

Une intégration continue ou de l'intégration continue ?

```
"There are only two hard things in Computer Science: cache invalidation and naming things", Phil Karlton
```

Marc Loupias, BBL AccorHotels du 20/06/2019

Quiproquo ?

Intégration Continue ⟶ Quiproquo ?

Une intégration continue désigne un outil :

Intégration Continue ⟶ Quiproquo ?

Une intégration continue désigne un outil :

- Jenkins, TeamCity, ...

Intégration Continue ⟶ Quiproquo ?

Une intégration continue désigne un outil :

- Jenkins, TeamCity, ...
- Version cloud : TravisCI, CircleCI, ...

Intégration Continue ⟶ Quiproquo ?

Une intégration continue désigne un outil :

- Jenkins, TeamCity, ...
- Version cloud : TravisCI, CircleCI, ...

L'intégration continue désigne une pratique.

Intégration Continue ⟶ Quiproquo ?

Une intégration continue désigne un outil :

- Jenkins, TeamCity, ...
- Version cloud : TravisCI, CircleCI, ...

L'intégration continue désigne une pratique.

Origine

Intégration Continue ⟶ Origine

Selon les sources :

- Grady Booch dans Object Oriented Design: With Applications (1991)
- Kent Beck dans Extreme Programming Explained (1999)
- ...

Intégration Continue ⟶ Origine

Vient de l'Extreme Programming :

- frequents releases
- code review
- pair programming
- TDD
- ...

Intégration Continue ⟶ Origine

Influence fortement Agile Manifesto (2001).
Kent Beck fait parti des auteurs avec Martin Fowler, Oncle Bob, Ward Cunningham, etc ...

Définitions

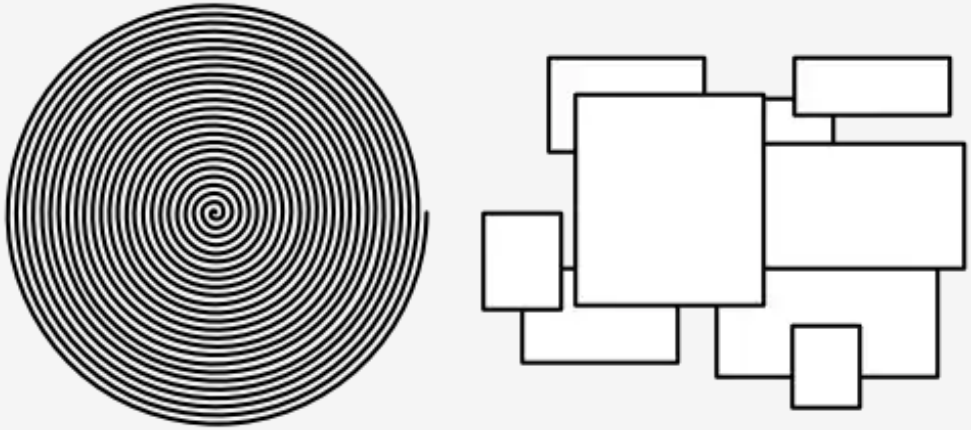
Intégration Continue ⟶ Définitions

```
Continuous Integration (CI) is a development practice that requires developers
to integrate code into a shared repository several times a day.

Each check-in is then verified by an automated build,
allowing teams to detect problems early.
```

www.thoughtworks.com

Intégration Continue ⟶ Définitions



C'est la mise en oeuvre connue la plus proche de l'idée qu'un logiciel ne se construit pas, il se grandit.

cf: Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering," Computer, Vol. 20, No. 4 (April 1987) pp. 10-19.

Intégration Continue ⟶ Définitions

Prérequis

- > *You should be able to walk up to the project with a virgin machine, do a checkout, and be able to fully build the system.*
- > *Continuous Integration assumes a high degree of tests which are automated into the software.*
- > *You must put everything required for a build in the source control system.*

Prérequis

Traduit pour le monde JavaScript :

```
$ git clone https://github.com/robert/killer-app.git && cd killer-app/
$ npm ci
$ npm build
$ npm start
$ npm test
```

- `npm ci` utilise strictement le `package-lock.json` (`npm >= 5.7`).
- les tests unitaires (uniquement) peuvent être exécutés avant le build.

Intégration Continue ⟶ Définitions

Etapes :

- commit

  écrire une révision en local
- build

  build un livrable en local et exécuter tous les tests
- pull master

  tirer les modifs des autres devs du trunk et rejouer build + tests
- push trigger CI

  pousser ce qui trigger l'outil de CI qui fait de son côté build + tests
- merge dans le trunk

  si job OK alors la révision est mergée.
- deploy

  La révision peut être déployée dans un environnement

Ces étapes sont répétées plusieurs fois par jour pour chaque développeur.

Intégration Continue ⟶ Définitions

Quels impacts sur la gestion du code source ?

Intégration Continue ⟶ Définitions

Quels impacts sur la gestion du code source ?

On va parler Git.

- *Comment intégrer les contributions ?*
- *Comment organiser l'intégration des contributions ?*

Comment intégrer les contributions ?

Avec une branche d'intégration.

- Via une opération de fusion (`merge`) de la branche de travail vers la branche d'intégration.
- La branche d'intégration est désignée sous le terme de "trunk" (tronc) sous SVN.
- Sous Git elle est nommée par défaut `master`, parfois `develop` selon le workflow utilisé.
- C'est le tronc de l'arbre des commits.

Intégration Continue ⟶ Définitions

Comment organiser l'intégration des contributions ?

Avec un workflow.

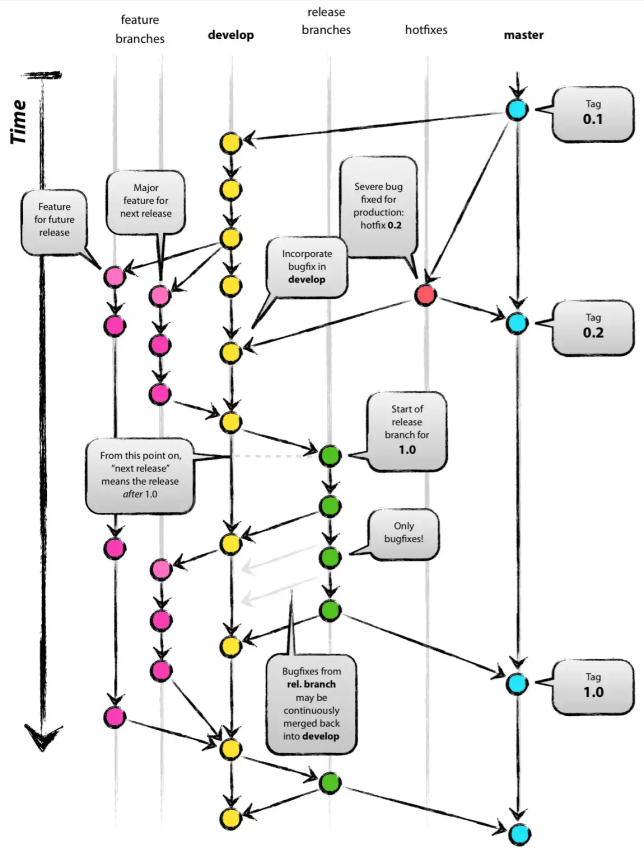
Les plus courants :

- Le feature branching à la papa (GitFlow)
- Continuous Integration Oriented (trunk-based dev)
- Continuous Delivery Oriented (GitHub/GitLab Flow)

Feature Branching (GitFlow)

Principes de base

GitFlow



- 1 branche = 1 feature
- la branche de collaboration est `develop`, elle n'est pas déployable
- on merge dans `develop` quand PR terminé
- release candidate sur release branch (bugfix QA only)
- `master` accueille les tags des livrables (hoftfix prod only)

Intégration Continue ⟶ Feature Branching (GitFlow)

Gestion du scope fonctionnel

- `develop` ne contient que le scope de la future release
- on freeze les PR jusqu'à création branche de release
- la branche de release prépare la MEP (QA)

Intégration Continue ⟶ Feature Branching (GitFlow)

Gestion des MEP

- tags de prod créés sur `master`
- hotfix branch créées depuis les tags de prod
- on déploie donc depuis `master` et pas depuis la branche d'intégration (`develop`)

Intégration Continue ⟶ Feature Branching (GitFlow)

Avantages

- on est pas exposé au WIP des autres
- on expose pas son WIP
- gestion facile pour le dev de base jusqu'au merge
- gestion débutants facile

Inconvénients

- confusion dans les targets des merges fréquentes (vers `develop` ? `release/*` ? `hotfix/*` ? `master` ?)
- faible fréquence d'intégration :
  - code review plus longues et difficiles
  - conflits plus fréquents et plus complexes
  - régressions visibles plus tard
  - embouteillages en fin de sprint
- gestion du scope fonctionnel rigide :
  - peut entrainer période freeze des PR avant création branche de release
  - difficile de dev plusieurs itérations en parallèle (plusieurs `develop`)
  - très difficile de changer de scope fonctionnel rapidement
- ne scale pas
- gestion lourde pour le Repo Owner (nombreux reports de code)

Intégration Continue ⟶ Feature Branching (GitFlow)

Cercle vicieux

Certains inconvénients entrainent d'autres inconvénients :

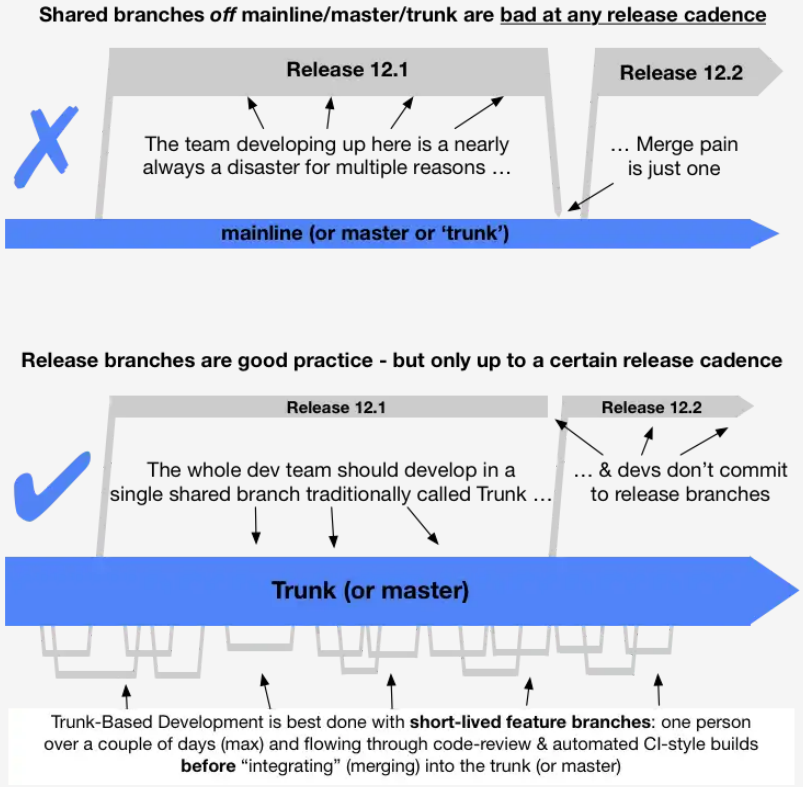
- conflits fréquents et difficiles entrainent peur de la refacto entraine baisse qualité
- régressions détectées plus tard entrainent baisse qualité et baisse productivité
- embouteillages fin sprint entrainent rush et stress
- difficultés à dev en parallèle des versions mal comprises du métier
  - incompréhension des causes techniques
  - incompréhension des conséquences des choix de scopage des releases par le métier (cf Concurrent development of consecutive releases - trunkbaseddevelopment.com)
- etc ...

Continuous Integration Oriented (Trunk-based Dev)

Intégration Continue ⟶ Continuous Integration Oriented (Trunk-based Dev)

Principes de base

Trunk-Based Development



- 1 branche `!=` 1 feature, 1 branche = 1 petite révision (2h de taf, 2j maxi)
- on merge ASAP
- une seule branche de collaboration (le trunk / `master`)
- `master` est toujours stable et déployable
- chaque révision est déployée en env de test, recette, QA, ...

Intégration Continue ⟶ Continuous Integration Oriented (Trunk-based Dev)

Gestion du scope fonctionnel :

Via feature toggle (ou flipping, ou flags)

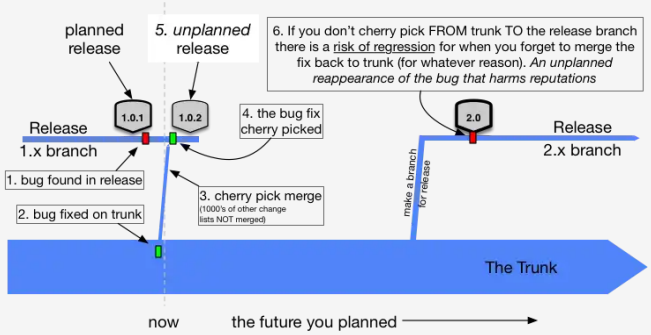
- 1er commit d'une feature implémente le toggle
- généralement au runtime mais peut aussi être réalisé au build
- peuvent servir pour A/B testing
- on merge avec tous les toggles activés
- on peut scoper les releases via toggle

Quand une feature est OK, le toggle est supprimé de la codebase. Sinon c'est plus du toggle mais de la configuration, donc une feature en soi.

La complexité de la gestion des branches est transférée dans la complexité applicative.

Intégration Continue ⟶ Continuous Integration Oriented (Trunk-based Dev)

Gestion des MEP



- MEP a échéances fixes
  - on crée release branch depuis `master` pour préparer les MEP (QA)
  - on fix les bugs levés par QA sur `master`
  - on cherry-pick de `master` vers release-branch
- hotfix
  - on crée une release branch depuis le dernier tag en prod
  - on dev le hotfix sur `master`
  - on cherry-pick de `master` vers release-branch
- on peut MEP quand on veut (Continuous Delivery)
- on peut MEP à chaque révision (Continuous Deployment)
  - stade ultime du chemin DevOps
  - un commit mergé dans `master` va automatiquement en prod si pipeline OK

Intégration Continue ⟶ Continuous Integration Oriented (Trunk-based Dev)

Prérequis

- suppose automatisation totale des tests (acceptance inclus)
- suppose une infra solide
- nécessite provisionning d'environnement
- feature toggle (ou flipping, ou flags) obligatoire

Intégration Continue ⟶ Continuous Integration Oriented (Trunk-based Dev)

Avantages

- intégration continue effective
- facile de pousser/récupérer du travail transverse
- code review rapides et faciles
- conflits petits et faciles si pas inexistants
- régressions visibles immédiatement
- fluide (pas d'emboutillage, pas de période de freeze)
- scalabilité
- très facile de changer de scope fonctionnel à la demande

Intégration Continue ⟶ Continuous Integration Oriented (Trunk-based Dev)

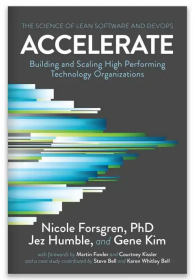
Inconvénients

- historique dégueux (impossible de générer un CHANGELOG lisible)
- feature flipping pénible à gérer
  - augmente la complexité
  - augmente la combinatoire des tests
  - gérer les différents environnements
- nécessite une équipe majoritairement senior sur la pratique

Intégration Continue ⟶ Continuous Integration Oriented (Trunk-based Dev)

DevOps

Point de vue DevOps c'est aujourd'hui l'état de l'art en matière de gestion de codebase.



Cf "Accelerate, The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations"

27 mars 2018

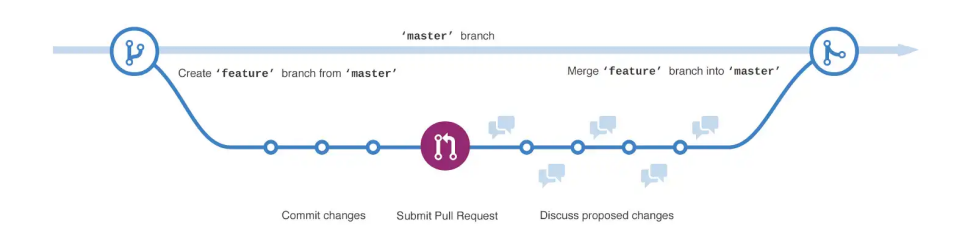
https://itrevolution.com/book/accelerate/

Continuous Delivery Oriented (GitHub/GitLab Flow)

Intégration Continue ⟶ Continuous Delivery Oriented (GitHub/GitLab Flow)

Principes de base

GitHub Flow / GitLab Flow



- 1 branche = 1 feature
- une seule branche de collaboration (le trunk / `master`)
- `master` est toujours stable et déployable
- on ouvre la PR dès le début
- on pousse régulièrement sur `origin`
- intégration continue inversée (on tire depuis `master` vers PR)
- quand review ok :
    - on verrouille `master` via un bot sur slack
    - on merge `master` dans feature branch
    - on déploie en prod
    - on teste, on évalue les métriques, etc ...
- quand test OK on merge la PR et on libère `master`
- si test KO on rollback la prod et on libère `master`

Intégration Continue ⟶ Continuous Delivery Oriented (GitHub/GitLab Flow)

Gestion du scope fonctionnel

- si MEP scopée les features non voulues restent en PR
- pas de scope fonctionnel, on pousse en prod au fil de l'eau

Les features sont seulement priorisées dans le backlog, et : *It's done when it's done*.

Intégration Continue ⟶ Continuous Delivery Oriented (GitHub/GitLab Flow)

Gestion des MEP

- pas de releases définies à l'avance, on déploie ASAP en prod
- si problème rollback

Intégration Continue ⟶ Continuous Delivery Oriented (GitHub/GitLab Flow)

Prérequis

- suppose automatisation totale des tests (acceptance inclus)
- suppose une infra solide
- nécessite provisionning d'environnement
- GitLab Flow propose des variances (branches d'environnements pour déploiement en fast-forward only, permet QA/recette manuelle avant passage prod)
- GitHub Flow suppose que les devs aient la main sur les outils de monitoring et de déploiement de prod

Intégration Continue ⟶ Continuous Delivery Oriented (GitHub/GitLab Flow)

Avantages

- très simple à gérer pour les devs
- code review au coeur du workflow
- scalabilité
- pas de feature flipping complexe à gérer

Intégration Continue ⟶ Continuous Delivery Oriented (GitHub/GitLab Flow)

Inconvénients

- historique dégueux si merge policy > rebase policy
- infra très avancée + beaucoup de tooling obligatoire
- en rupture avec la culture d'entreprise classique
  - suppression des silos dev / qa / fonctionnels / managers / ops
  - pas de scope fonctionnel
  - c'est top mais mission impossible à faire accepter

Conclusion

Intégration Continue ⟶ Conclusion

- L'intégration continue (la pratique) a une définition très précise

  On merge tout le temps, même quand pas terminé

- Difficile à mettre en oeuvre

  Nécessite

  → un gros investissement d'infrastructure (ou de dépendre de services tiers)

  → un haut degré d'automatisation des tests (pas de test manuel)

  → de gros changements de culture d'entreprise

- Au coeur de la culture DevOps

  Les GitHub/GitLab flow sont plus orientés CD que CI au sens strict.

  Les GAFAM et les licornes de la Silicon Valley l'utilisent tous : Google, Amazon, Microsoft, GitHub, etc ...

  En France des sociétés comme BlablaCar, LesFurets, Doctolib, ...

  En France aucun grand compte à ma connaissance (sauf SoGé en finance de marché)

Intégration Continue ⟶ Conclusion

- Souvent des projets web BtoC

  Releases fixes scopées pas utiles.
  Prio TTM le plus court possible.
- Les projets opensource sont peu concernés

  Se tournent plutôt vers du GitHub/GitLab Flow réadapté.
  Aucun sens de livrer 35 versions d'une lib par jour.
  Travail effectif rarement en fulltime.
- Tout démarre à partir du workflow Git

Sources

- www.thoughtworks.com
- Continuous Integration, Martin Fowler - 01/05/2006
- A successful Git branching model, Vincent Driessen - 05/01/2010
- Trunk-Based Development, Alexander Grebenyuk - 23/03/2018
- trunkbaseddevelopment.com - 2017
- Continuous Integration, LeSS Company - 2014-2019
- GitHub Flow - Scott Chacon - 31/08/2011 / Voir aussi / En vidéo, Alain Hélaïli@Devoxx2016
- Introduction to GitLab Flow - Documentation GitLab