

Introduction au CI/CD

ENSG - Décembre 2017



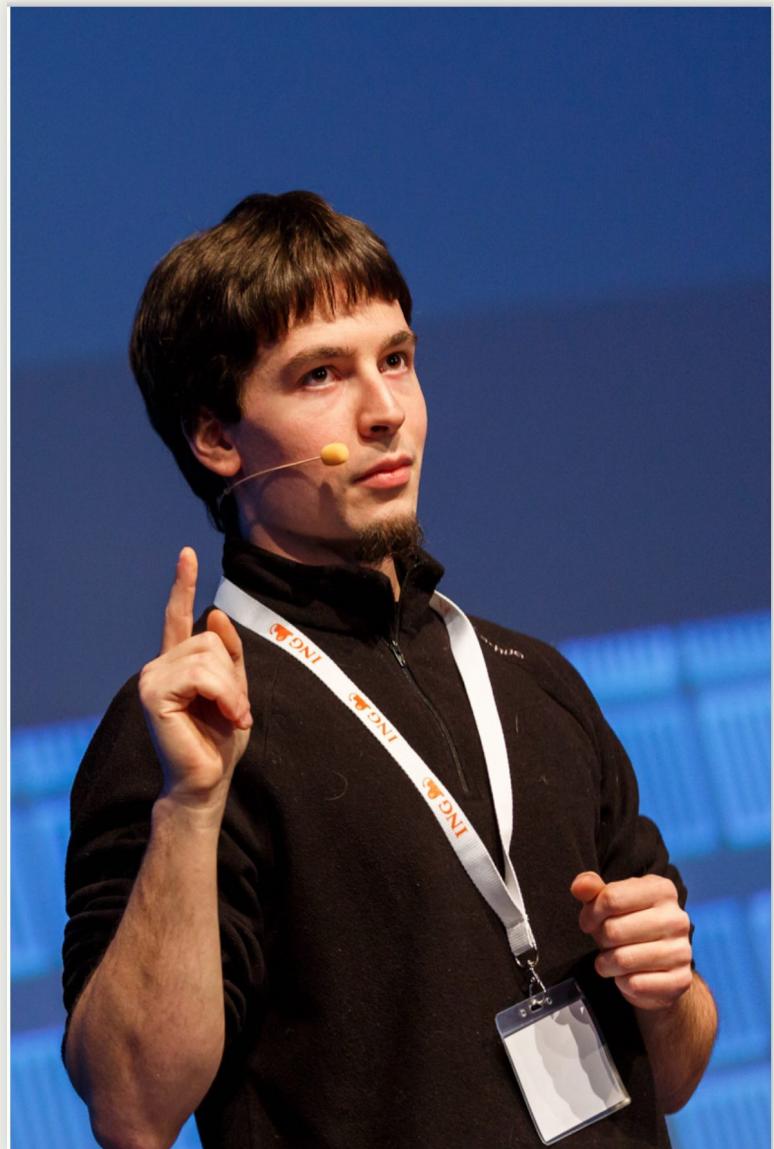
- Présentation disponible à l'URL <https://dduportal.github.io/ensg-ci-cd/2019-2020>
- This work is licensed under a Creative Commons Attribution 4.0 International License
- Source Code

Comment utiliser cette présentation ?

- Pour naviguer, utilisez les flèches en bas à droite (ou celles de votre clavier)
 - Gauche/Droite: changer de chapitre
 - Haut/Bas: naviguer dans un chapitre
- Pour avoir une vue globale : utiliser la touche "o" (pour "**Overview**")
- Pour voir les notes de l'auteur : utilisez la touche "s" (pour "**Speaker notes**")

Bonjour !

Damien DUPORTAL



- **Training Engineer @ CloudBees**
 - Docker & Apple fanboy.
 - Human stack focused
 - Rock climber
- Contact:
 - Twitter: @DamienDuportal
 - Github: dduportal
 - eMail: damienduportal@gmail.com

Et vous ?



Organisation du Cours

Objectifs

- Découvrir (ou redécouvrir) les bases des éléments de la "chaîne de production logicielle"
 - SCM, Tests, Sécurité, etc.
- Découvrir les concepts de l'Intégration Continue et du Déploiement Continu
- Découvrir Jenkins
- Mettre en oeuvre Jenkins en tant que développeur

Temps

- 2 jours ensemble
- Matin:
 - 09:00 → 10:30
 - 11:00 → 12:30
- Après-Midi:
 - 13:30 → 15:00
 - 15:30 → 17:00

Evaluation

Evaluation

- QCM: 25 % de la note
- Rendu de TP: 75 % de la note

QCM

- But : évaluer l'acquisition des concepts théoriques autour du CI/CD
- Format :
 - 10 questions à choix multiples portant sur les 2 jours
 - Pas de points négatifs
- Temps limité: 15 minutes

Rendu de TP

- But : évaluer l'acquisition de compétences de mise en oeuvre basiques autour du CI/CD
- Format:
 - Rendu public dans GitHub : valorisable sur vos CVs
 - Réutilisation d'un de vos projets
- Temps:
 - Estimation du travail personnel: 4h +- 1h
 - Dates de remise des consignes et rendus à définir ensemble
 - 3 semaines entre les deux

Source Code Management (SCM)

"Gestion de Code Source"

Qu'est ce que le SCM ?

Les Gestionnaires de Code Source, également connus comme "Version Control Systems" (VCS):

- Sont des systèmes **logiciels**
- Conservent **toutes** les modifications apportées à une collection de fichiers, dans le temps
- Permettent de **partager** ces changements
- Fournissent des fonctionnalités de **merge** et de **suivi** des modifications

Pourquoi les SCM ?

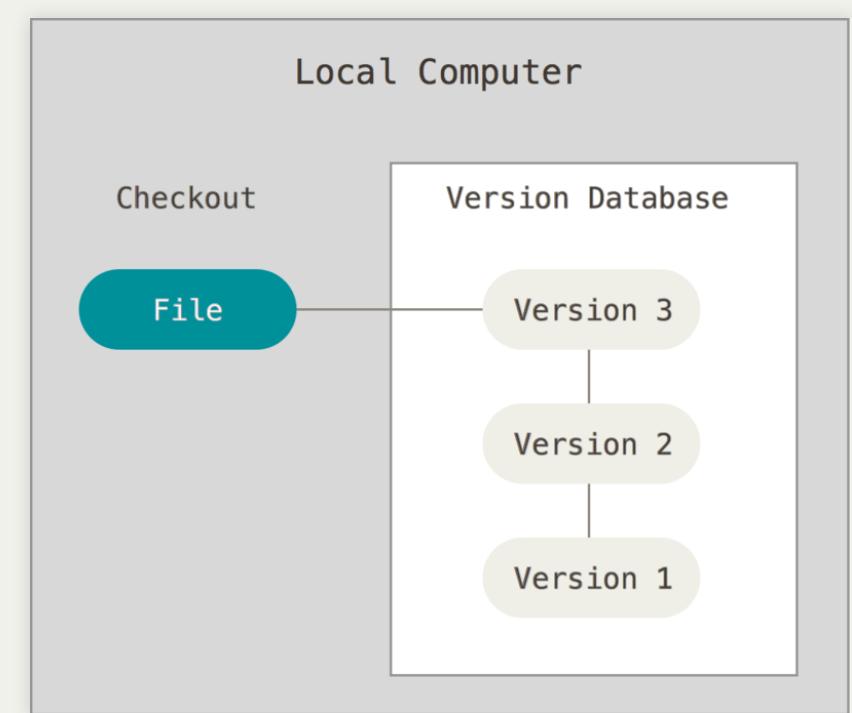
- Pour **collaborer** efficacement sur un même référentiel de code source
 - Aide à la résolution de conflits
 - Partage de contenu facile
- Pour conserver une trace de **tous** les changements : On parle de source unique de vérité (*Single Source of Truth*)
 - Historique complet des modifications
 - Possibilité de retour arrière à tout moment

Quels sont les types de SCM ?

- Locaux
- Centralisés
- Distribués

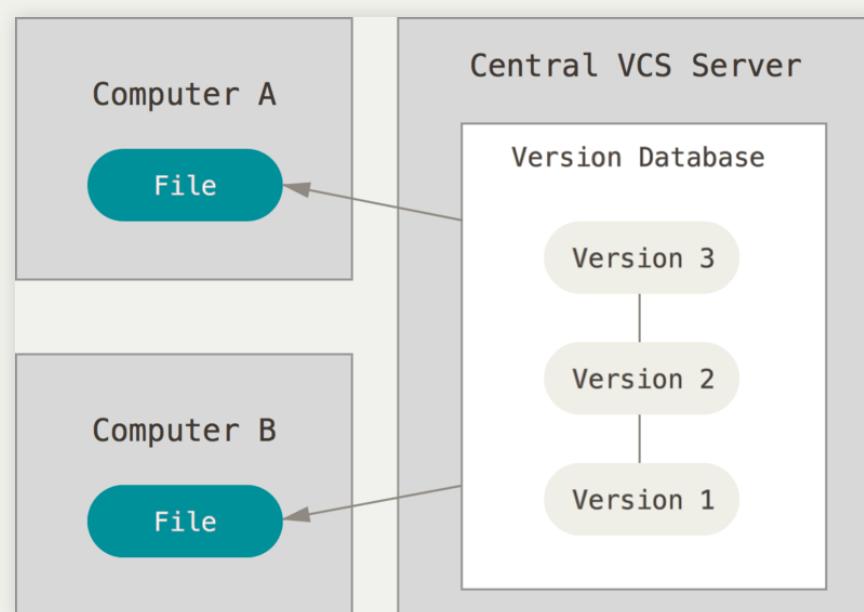
SCM Locaux

- Plus vieux type, ancêtre de tous les autres
- **Uniquement** historique de modification
 - Utilise une "*base de donnée de versions*" des fichiers
 - Stockage uniquement des différences ("*diff*")
- **Pas** de partage
- Exemple: *rcs* (toujours dans Apple XCode Tools)



SCM Centralisés (CVCS)

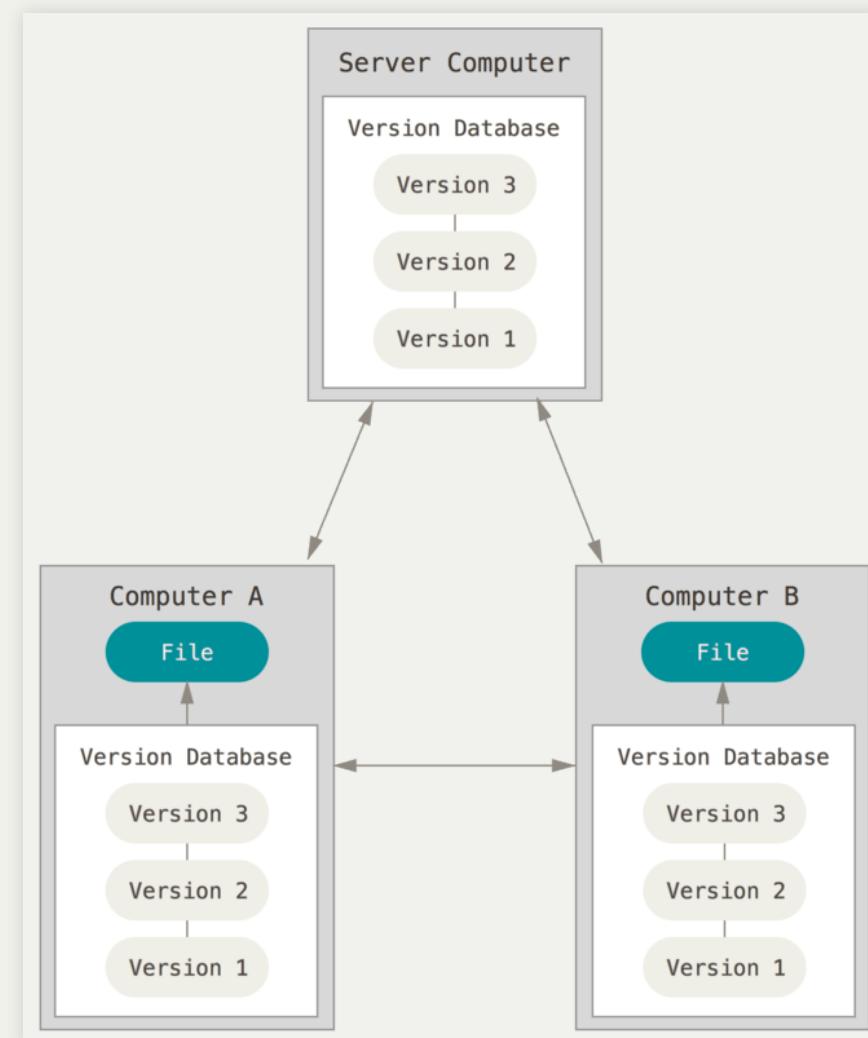
- Couvre **historique ET partage**
 - La "base de données de versions" est stockée sur un serveur central
- Chaque client ne possède qu'**une seule** version du code



- Apprentissage très facile, limité sur la résolution de conflits
- Exemples: CVS, SVN, Perforce, TFS

SCM Distribués (DVCS)

- La "base de données de versions" est distribuée par **duplication sur chaque noeud**
- Exemples: Git, Mercurial, Bazaar, Monotone



Comment héberger son SCM ?

- Hébergés dans le Cloud
- Hébergé "à la maison"

SCM "Hébergés dans le Cloud"

- **SCM as a Service**
- Le serveur centralisé est un service hébergé par un fournisseur
- Avantages:
 - Pas de temps/énergie passés sur la gestion
 - Associent au SCM d'autre services : gestionnaire de tickets, wiki, éditeur de texte online, etc.
- Risque: Votre code est hébergé par un tiers
- Exemples: GitHub, Bitbucket by Atlassian, Amazon CodeCommit, Visual Studio Online by Microsoft, SourceForge, GitLab.com, etc.

SCM "Hébergés dans le Cloud"

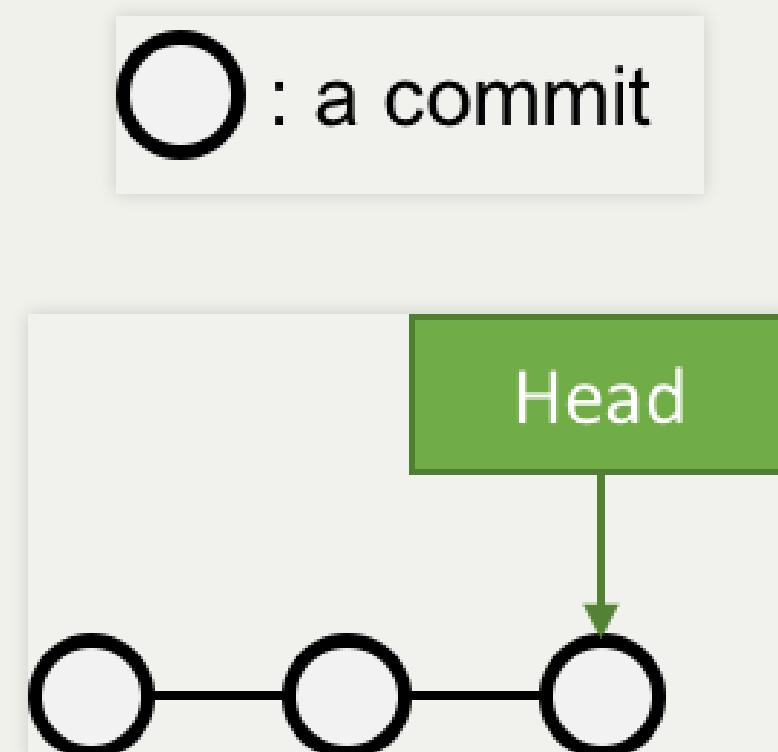
- Pour pallier au risque précédent, on trouve des versions "On-Premide" (généralement payantes)
- Le monde de l'Open Source fourni également des solutions à héberger soit-même
 - Très souvent gratuit et on peut le corriger
 - Temps et énergie à consacrer
- Exemples: Gitlab, Gitea, Gogs, Bazaar server, VisualSVN Server, etc.

Terminologie des SCM : Basiques

- **diff:** un ensemble de lignes "changées" sur un fichier donné
- **changeset:** un ensemble de "diff" (donc peut couvrir plusieurs fichiers)
- **commit:** Action de sauvegarder un changeset dans la base de données des versions.

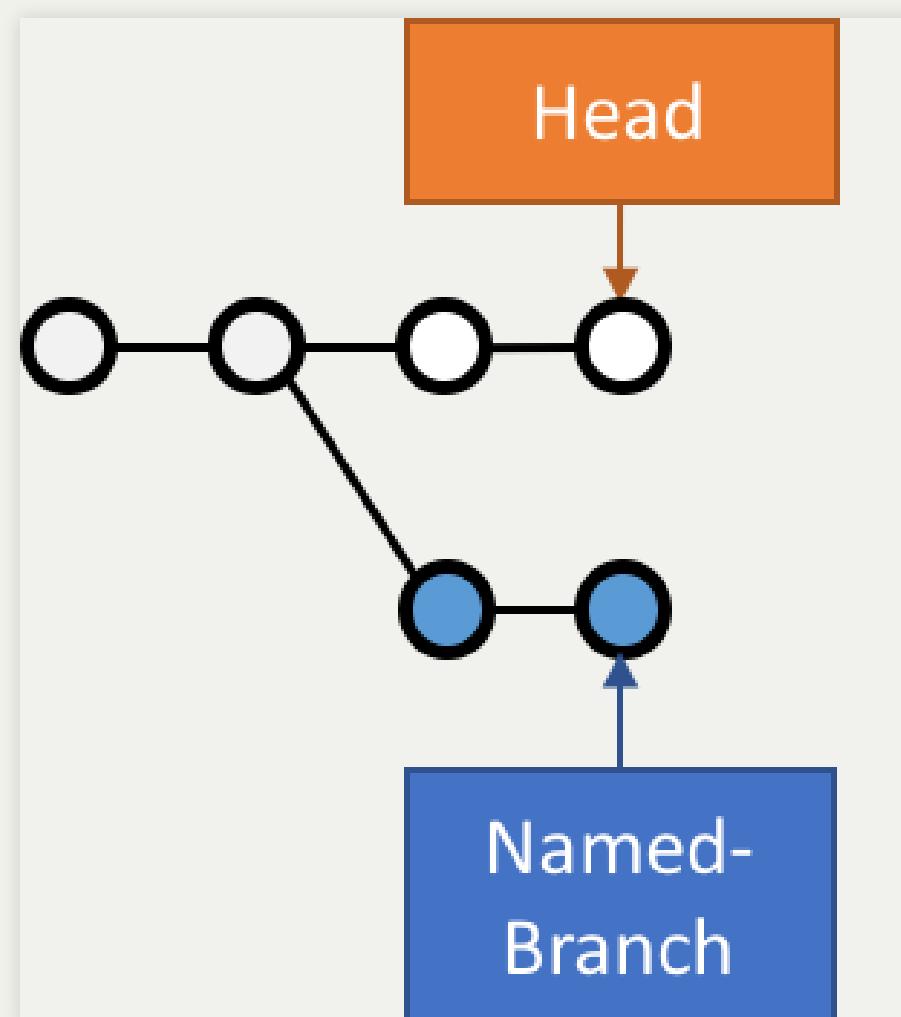
Terminologie des SCM : Représentation

- Le dernier commit dans l'historique est aliasé comme "*HEAD*"



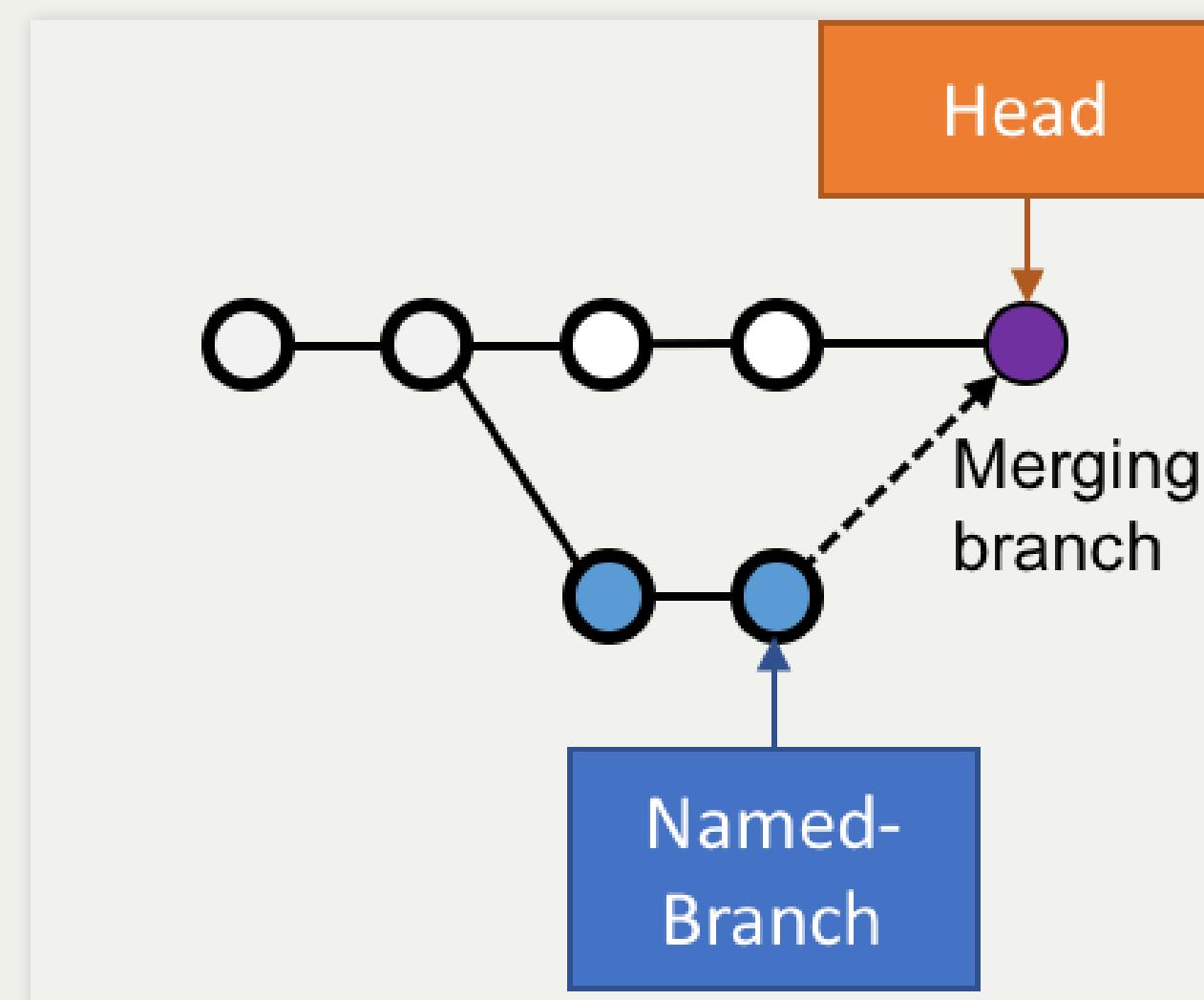
Terminologie des SCM : Branches

- Abstraction d'une version "isolée" du code
- Concrètement, une **branche** est un alias pointant vers un "commit"



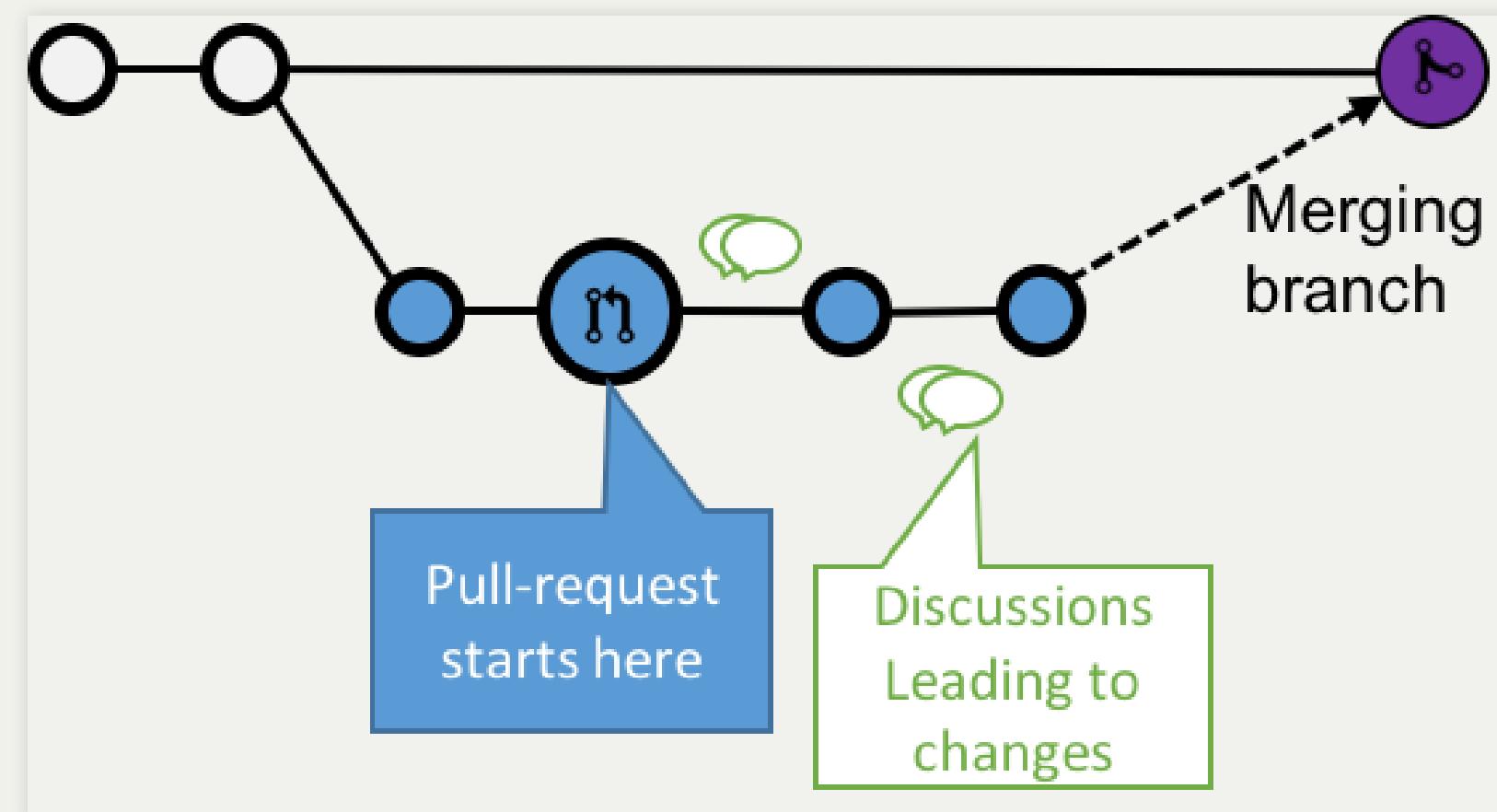
Terminologie des SCM : Merge

- On intègre une branche dans une autre en effectuant un **merge**
 - Un nouveau commit est créé, fruit de la combinaison de 2 autres commits



Terminologie des SCM : Pull Request

- Une **Pull Request** (ou "Merge Request") est une procédure de revue de code avant intégration

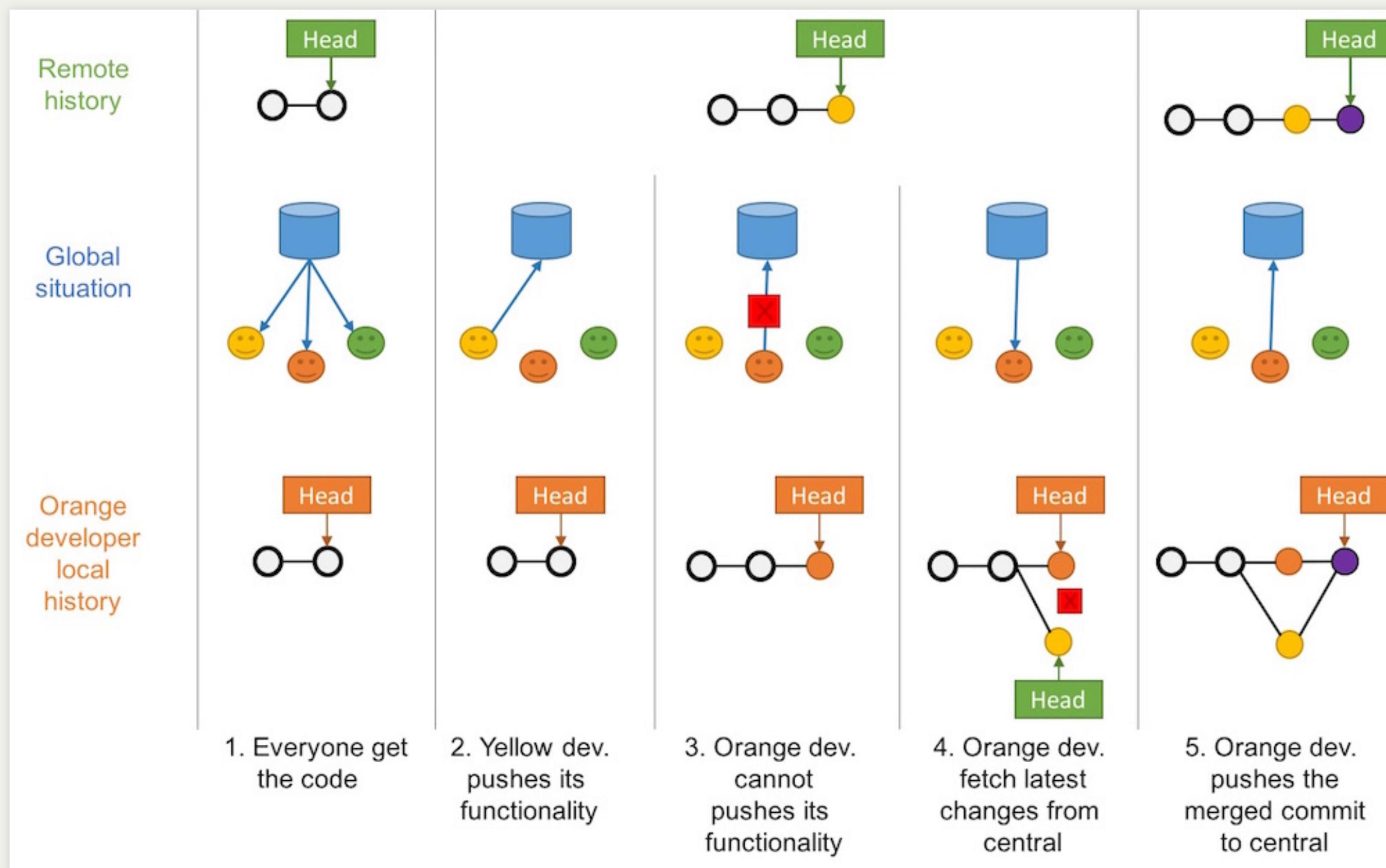


Motifs d'utilisation des SCMs ?

Voici quelques motifs d'utilisation des SCMs :

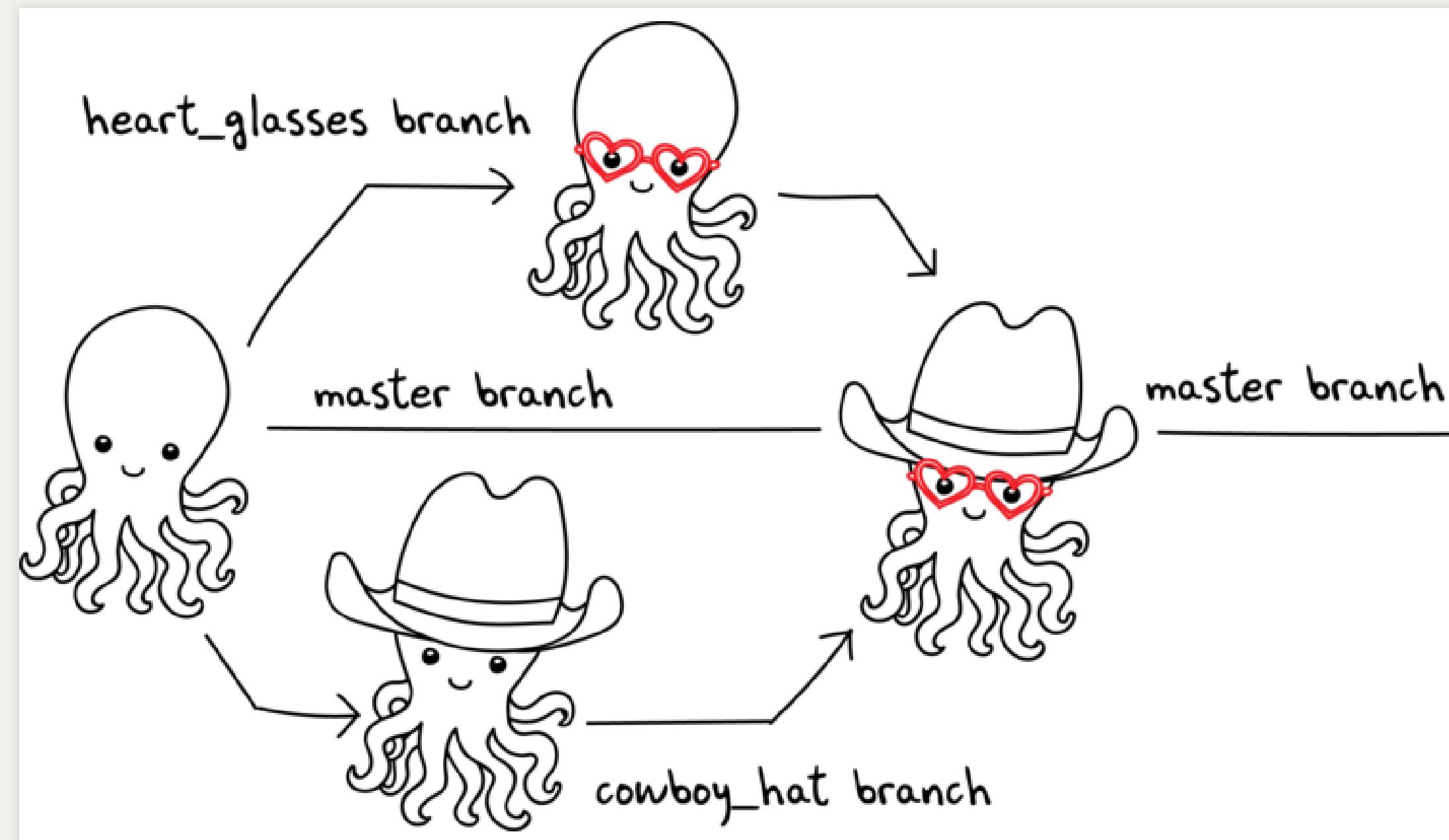
- "Centralized" Flow
- "Feature Branch" Flow
- "Git" Flow
- "GitHub" Flow

Centralized Flow

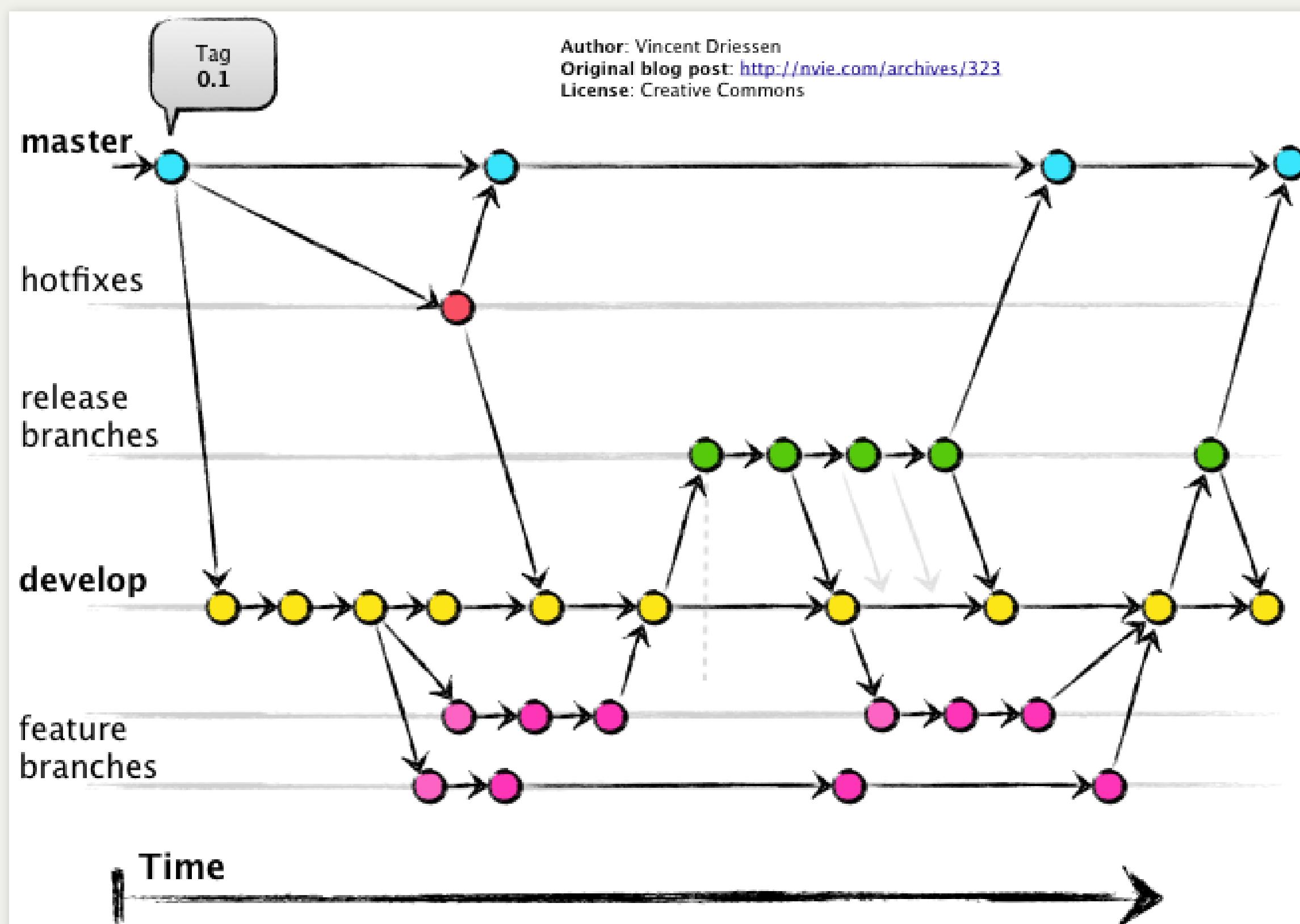


Feature Branch Flow

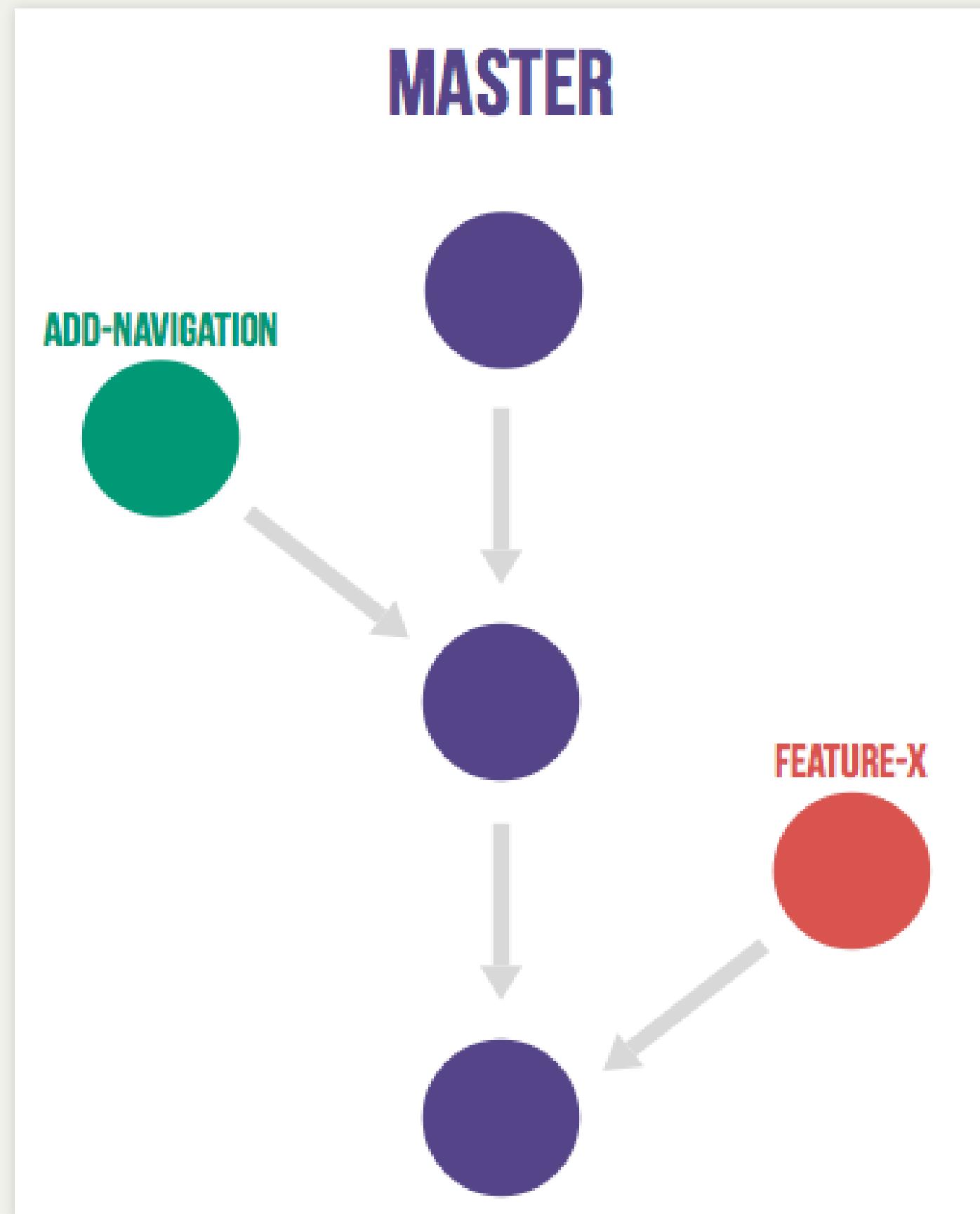
- Une seule branche par fonctionnalité



Git Flow



GitHub Flow



Résoudre des problèmes avec le SCM

- "Infrastructure as Code" :
 - Besoins de traçabilité, de définition explicite et de gestion de conflits
 - Collaboration requise pour chaque changement (revue, responsabilités)
- Code Civil:
 - <https://github.com/steeve/france.code-civil>
 - <https://github.com/steeve/france.code-civil/pull/40>

Pour aller plus loin...

Un peu de lecture :

- <http://martinfowler.com/bliki/VersionControlTools.html>
- <http://martinfowler.com/bliki/FeatureBranch.html>
- <https://about.gitlab.com/2014/09/29/gitlab-flow/>
- <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>
- <https://www.atlassian.com/git/tutorials/comparing-workflows>
- <http://nvie.com/posts/a-successful-git-branching-model/>

Tests Logiciels

Pourquoi tester un logiciel ?

- Les Humains font des erreurs
- Les Humains écrivent du code
- Le code est donc sujet à erreurs: **Conséquences?**
- Les **tests** identifient ces erreurs, dans un **but** de correction

Qu'est ce que le "test logiciel" ?

- Le test logiciel est une pratique suivant 2 piliers :
 - Valider que le logiciel remplisse les rôles qui lui sont confiés
 - Rechercher les **fautes** pour les corriger, améliorant la qualité du système

Tests Automatisé ou Manuels ?

- Automatiser : **répétition et reproductibilité**
- Test Manuel à considérer dans **peu de cas**, quand :
 - Coût de l'automatisation dépasse sa valeur
 - Automatisation impossible

Terminologie du Test Logiciel

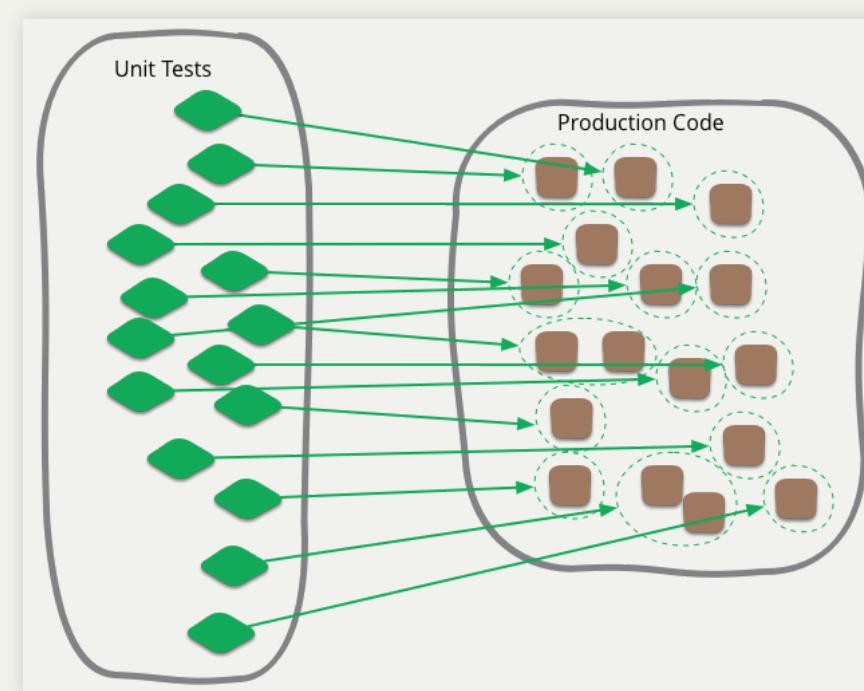
- **SUT:** "System Under Testing". Défini les frontières du système.
- **Test Double:** Terme générique désignant un sous-ensemble simplifié du "S.U.T.". Exemples: Mock, Stub, Spy, etc.
- **Boîte Blanche:** Tester avec une vue interne du SUT
- **Boîte Noire:** Tester le SUT sans connaissance préalable de ses mécanismes internes

Comment tester le logiciel ?

- La question **primordiale** est: "Que voulez-vous tester ?"
- En fonction de la réponse, différent types de tests peuvent être utilisé (liste NON exhaustive) :
 - Unit testing
 - Integration testing
 - Smoke testing
 - Functional Testing
 - Non-Regression testing
 - Acceptance testing

Test Unitaire

- Focalisé sur le plus petit sous système possible du SUT, en "boîte blanche"
- Tests **indépendants** les uns des autres
 - Ordre d'exécution non important
 - Utilisation de **Test Doubles** pour simuler le "reste" en bon fonctionnement



Tests d'Intégration

- Vérifier l'intégration entre différents sous-systèmes
- Le SUT est en "boîte blanche"



Smoke Testing



- But : **Fail Fast** en "boîte blanche"
- Valide les fonctions "de base" du système
- On parle parfois de "Sanity Checking"

If it smokes, it's bad

— Anonymous Electrician

Tests Fonctionnels

- Vérifie que le logiciel se comporte comme prévue par **les personnes en charge de la fabrication**
- Pas de **biais** d'interprétation
- Le SUT est en "boîte noire"

Tests de Régression

- Vérifie que le SUT a un comportement stable dans le temps
- Focalisation sur bug qui ne doit pas revenir
- Le SUT est en "boîte noire"



Correcting a single bug may introduce several more.

— Any developer

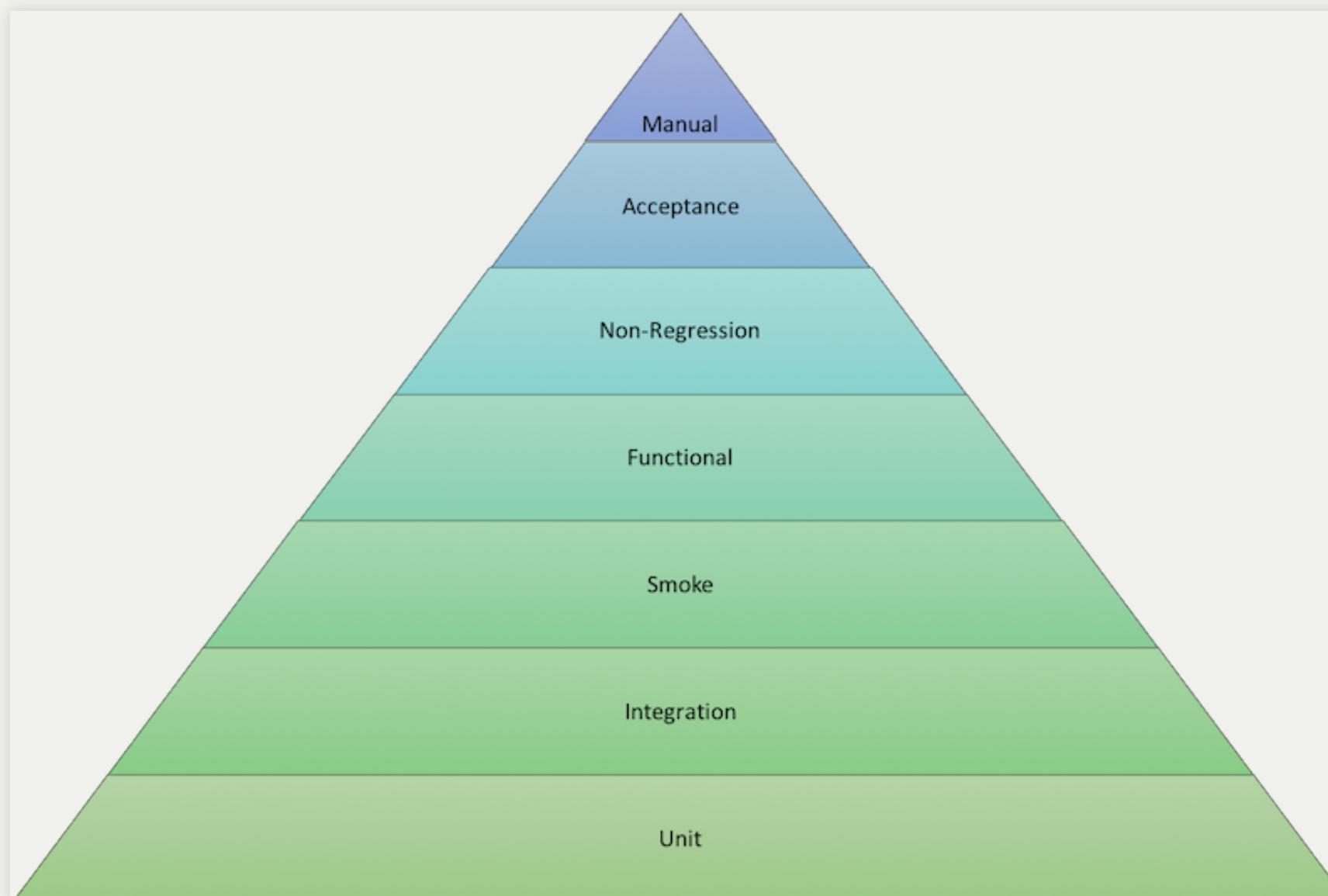
Tests d'Acceptation

- Également connu sous l'acronyme "UAT" User Acceptance Testing
- Vérifie que le logiciel se comporte comme attendu par **l'utilisateur**
- Biais de communication inclus
- Le SUT est en "boîte noire"



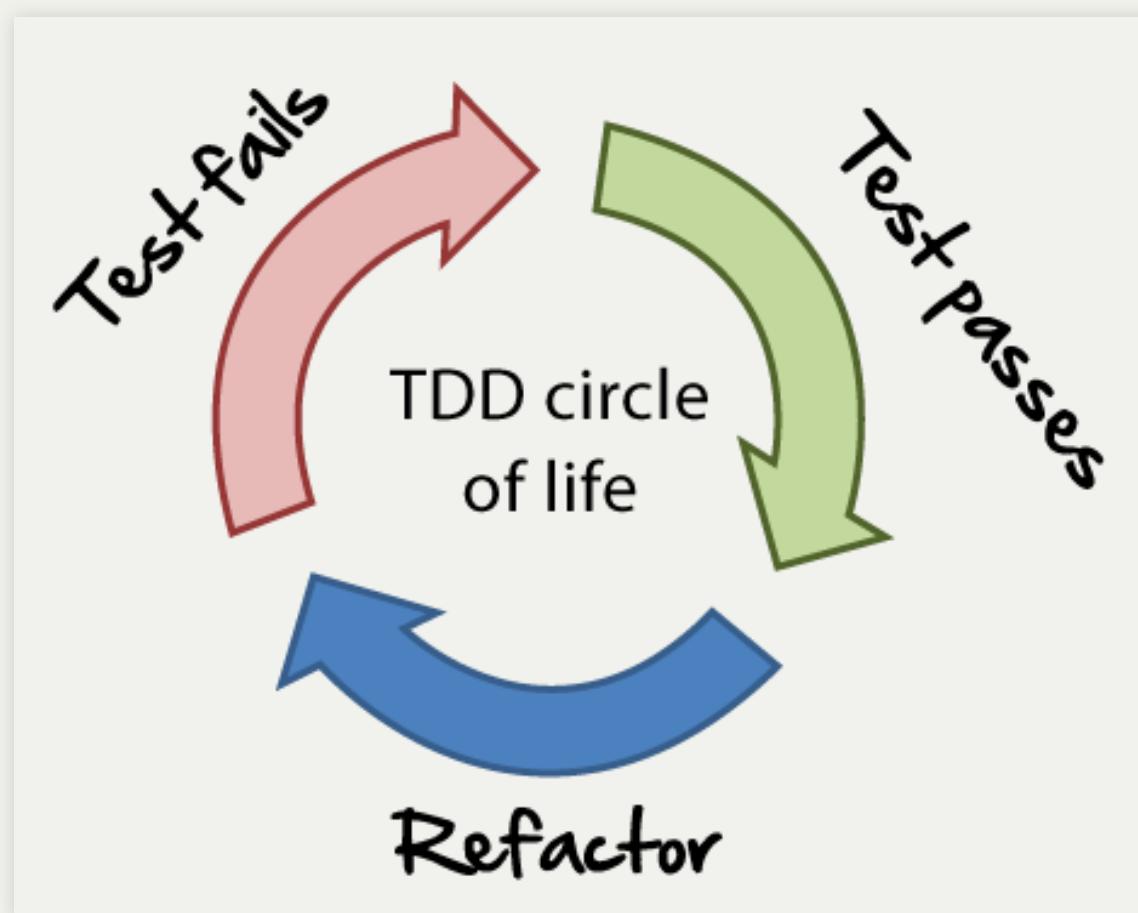
Ordre des Tests

- Fonction des temps d'exécutions, des coûts de corrections, et des valeurs ajoutées. **Contextuel.**



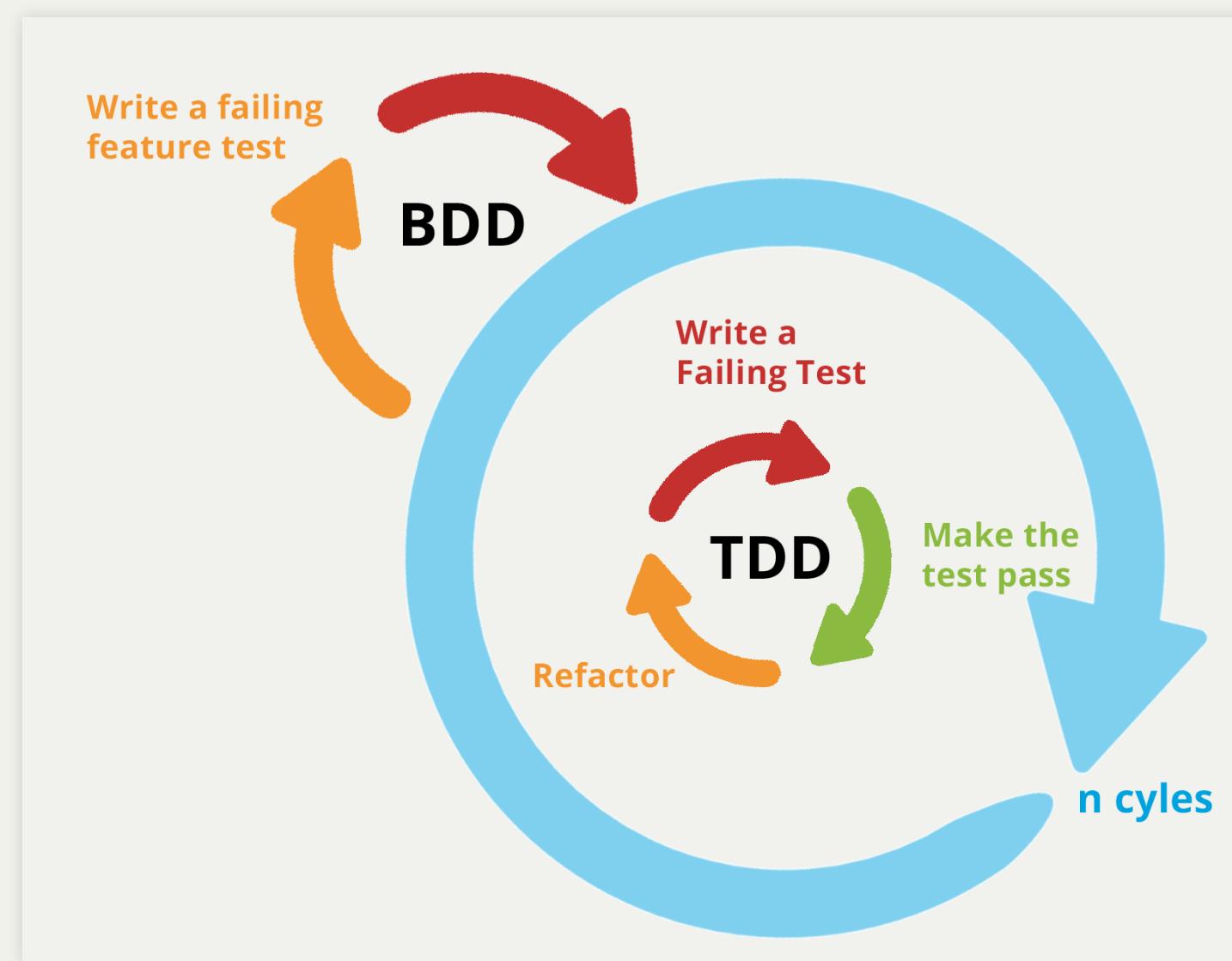
Test Driven Development

- TDD: Écrire les tests unitaires **avant** le code



Behaviour Driven Development

- BDD: Privilégier language naturel et interactions
 - "Given, When, Then"
 - Moins de technique. Valeur ajoutée pour l'utilisateur.



Pour aller plus loin...

- (FR) <http://douche.name/blog/nomenclature-des-tests-logiciels/>
- <http://martinfowler.com/bliki/UnitTest.html>
- https://en.wikipedia.org/wiki/Software_testing
- <http://martinfowler.com/tags/testing.html>
- <http://martinfowler.com/bliki/TestCoverage.html>
- <http://martinfowler.com/bliki/TestDrivenDevelopment.html>

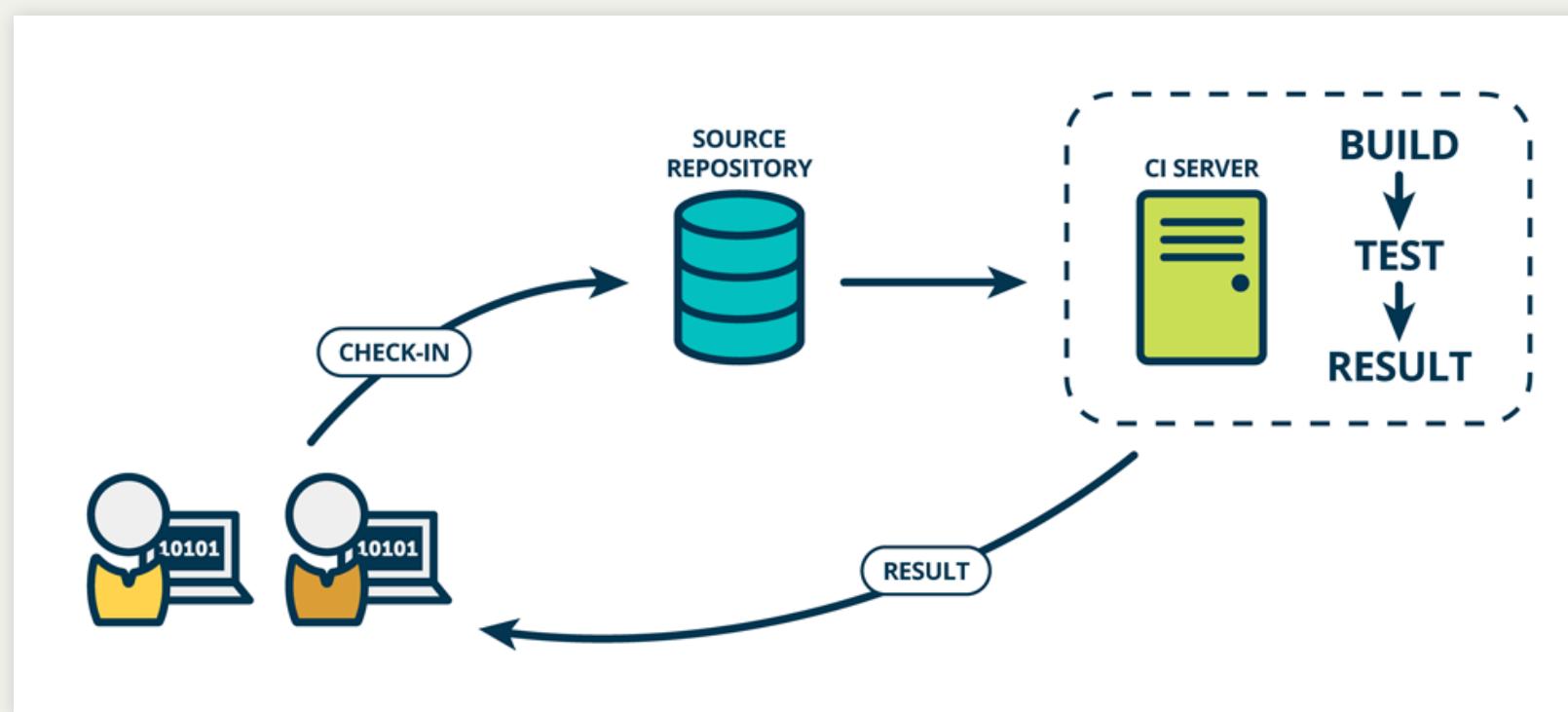
"Continuous Everything"

Qu'est ce que l'Intégration Continue ?

Continuous Integration is a software development practice where members of a team integrate their work frequently. Usually each person integrates at least daily - leading to multiple integrations per day.

— Martin Fowler - Continuous Integration

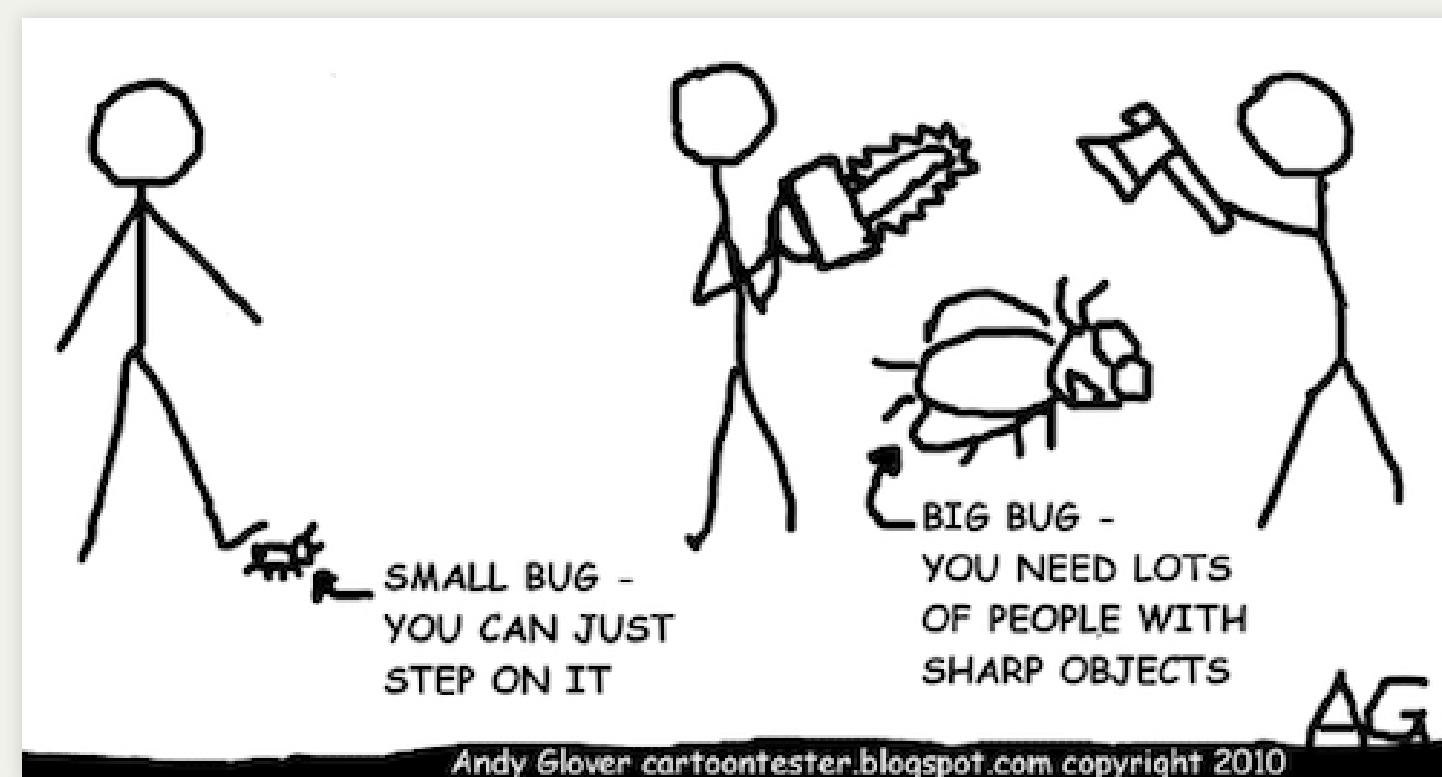
Intégration Continue (CI)



- Construire et intégrer le code **en continu**
- Le code est intégré **souvent**, au moins quotidiennement pour que l'intégration soit un *non-événement*
- Chaque intégration est validée par une construction **automatisée** avec tests

Pourquoi la CI ?

But : Déetecter les fautes au plus tôt



Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove.

— Martin Fowler

Livraison Continue

Continuous Delivery (CD)

Pourquoi la Livraison Continue ?

- Diminuer les risque liés au déploiement
- Permettre de récolter des retours utilisateurs plus souvent
- Rendre l'avancement visible par **tous**

How long would it take to your organization to deploy a change that involves just one single line of code?

— Mary and Tom Poppendieck

Qu'est ce que la Livraison Continue ?

- Suite logique de l'intégration continue:
 - Chaque changement est **potentiellement** déployable en production
 - Le déploiement peut donc être effectué à **tout** moment

*Your team prioritizes keeping the software **deployable** over working on new features*

— Martin Fowler

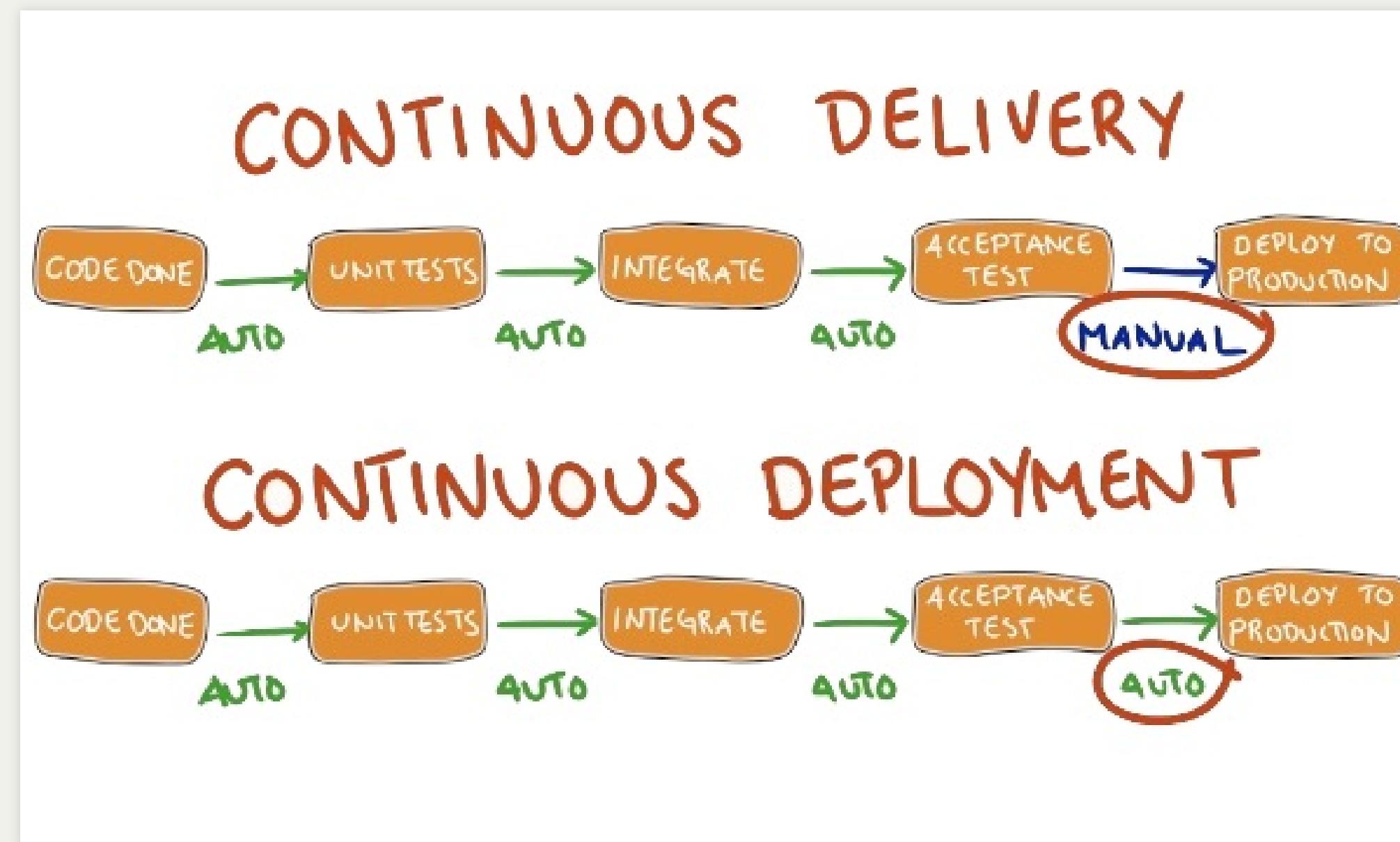
Déploiement Continu

Continuous Deployment

Qu'est ce que le Déploiement Continu ?

- Version "avancée" de la livraison continue:
 - Chaque changement **est** déployé en production, de manière **automatique**
- Question importante: En avez-vous besoin ?
 - Avez-vous les mêmes besoin que Amazon Google ou Netflix ?

Continuous Delivery versus Deployment



Pour aller plus loin...

- <http://martinfowler.com/articles/continuousIntegration.html>
- <http://martinfowler.com/bliki/ContinuousDelivery.html>
- <https://jaxenter.com/implementing-continuous-delivery-117916.html>
- <https://technologyconversations.com/2014/04/29/continuous-delivery-introduction-to-concepts-and-tools/>
- <http://blog.arungupta.me/continuous-integration-delivery-deployment-maturity-model>
- <http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment>

Chaîne Logistique du Logiciel

"Software Supply Chain"

La boucle de rétroaction

"Feedback loop" / "Boucle de feedback"

Pourquoi de la rétroaction ?

- *Problématique* : réagir **rapidement** pour corriger une faute
 - "Au plus tôt, au moins cher"
- Problème #1: Avoir un retour
- Problème #2: Réagir **systématiquement** sur un retour
- Problème #3: Avoir **confiance**

Qu'est ce qu'une boucle de Feedback ?

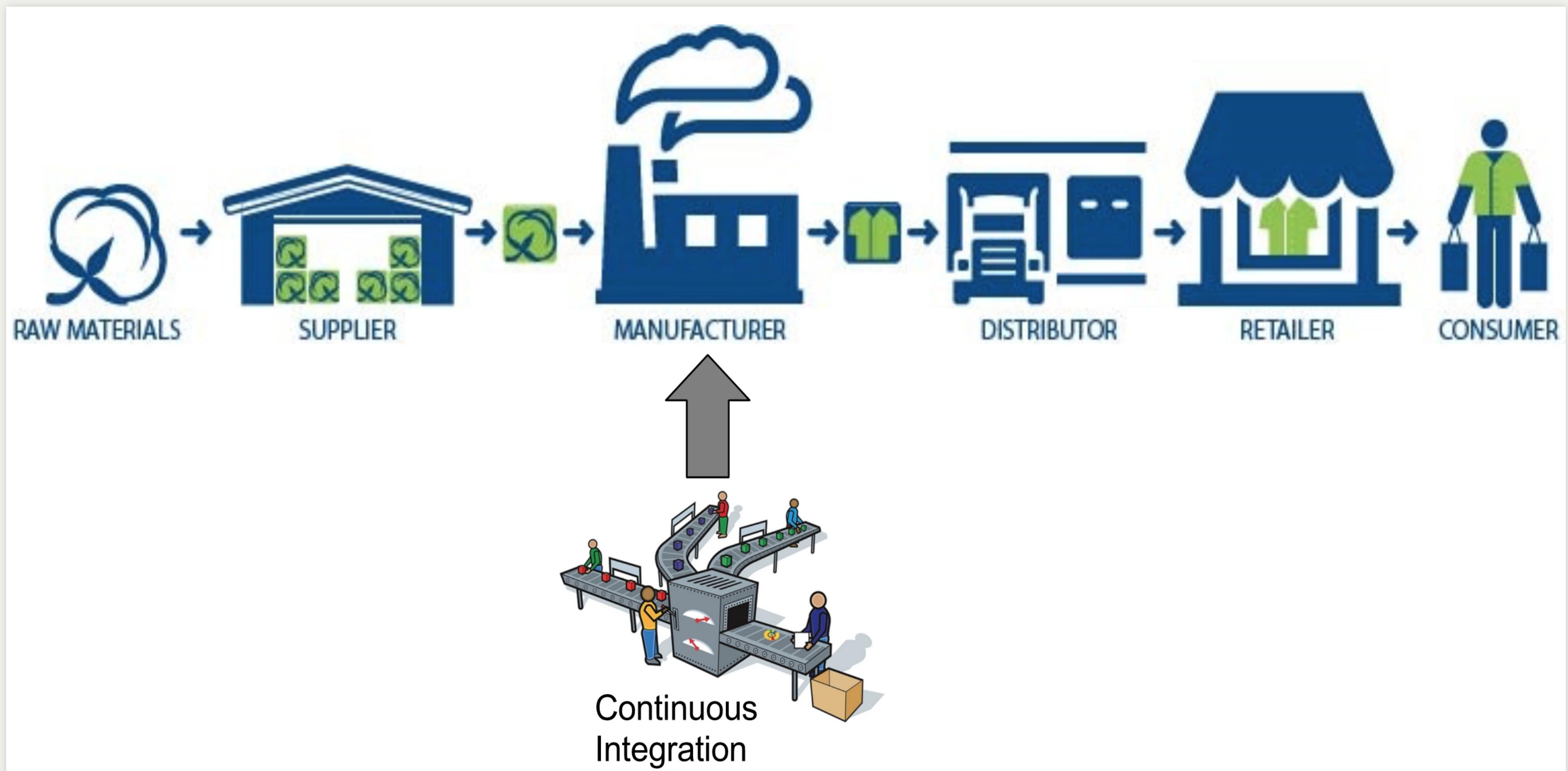


Comment implémenter la rétroaction

?

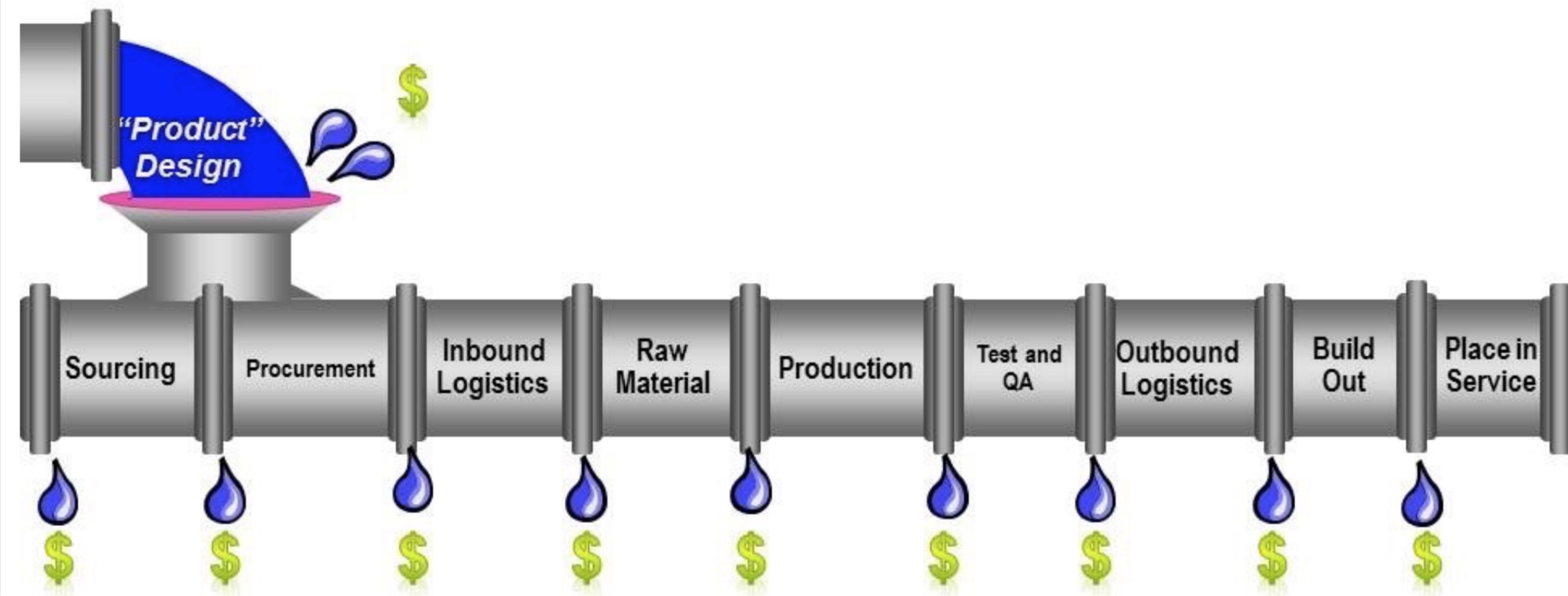
- Quels **acteurs** du système ?
- Quel médium de **communication** ?
- Quel déclencheurs et quelles limites ?
- **Culture à construire**, les outils suivent facilement

Chaîne Logistique du Logiciel



The Pipeline

- 1. Do We Have a Leaky Pipe?**
- 2. Where Are the Leaks?**
- 3. Let's Plug the Leaks.**



Qu'est ce qu'un Pipeline ?

- **Industrialisation du logiciel**
- Modélisation de la chaîne de valeur ("Value Stream Mapping")
- "Fast is cheap": Piloté par le concept de la défaillance rapide ("fail fast")

Anatomie d'un "Pipeline" 1/2

- **Stage** ("étape"): Élément de base
 - Abstraction **atomiques** d'un ensemble d'actions
 - Exemple: "Build", "Run Unit Tests"
 - Possibilité de parallélisation
- **Gate** ("Porte"): Transition entre 2 étapes
 - Manuel ou automatique
 - Peuvent être conditionnelles

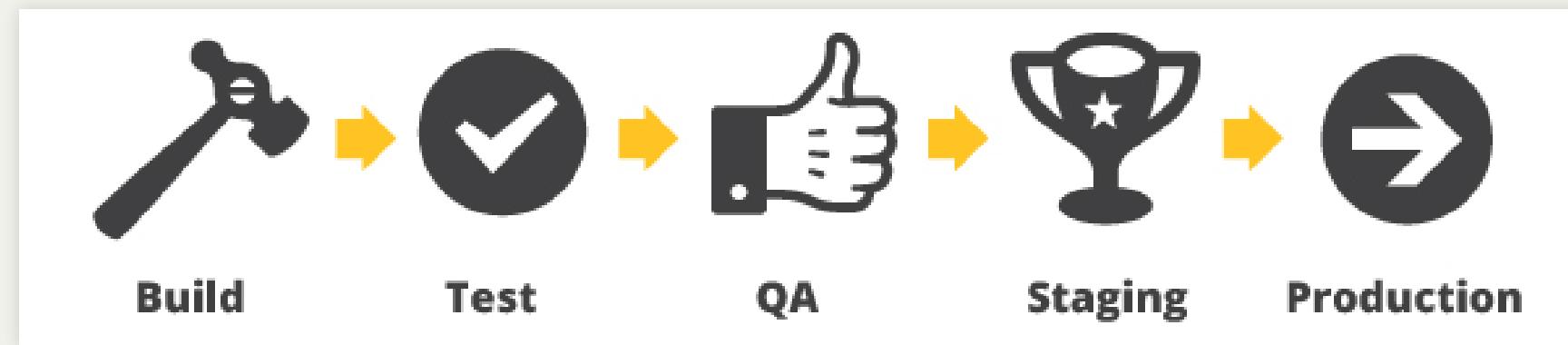
Anatomie d'un "Pipeline" 2/2

- **Déclenchement initial** : un changement dans la base de code
- Chaque étape *peut* produire des livrables: on parlera d'**Artefacts** dans ce cours

Etapes de "Déploiement"

- Le **déploiement** est ce qui permet de rendre le logiciel prêt à l'usage
- Un "déploiement" est exécuté vers un **environnement**
 - Production
 - Préproduction ("staging") / recette ("qualification")
 - Tests
 - "Disaster Recovery Environment"

Un exemple de Pipeline



Comment faire des "bons" Pipelines ?

- Commencer par un "Produit Minimum Viable" (MVP) puis itérer
- S'efforcer d'appliquer les bonne pratiques
- Optimiser le Pipeline (lors des itérations)

Bonnes Pratiques

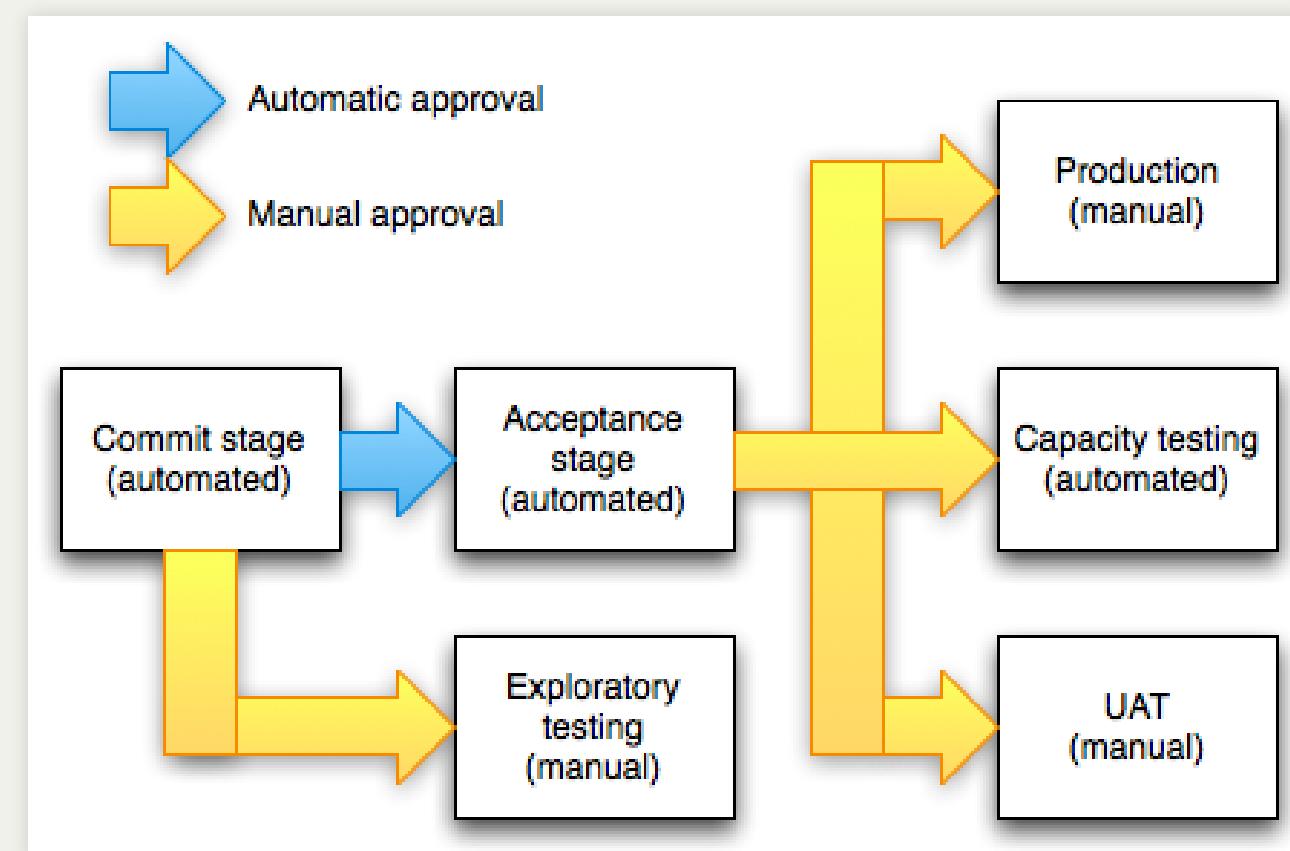
- Réutilisation des artefacts: "*Only Build Your Binaries Once*"
- Arrêt du Pipeline dès qu'une faute est identifiée: "*Fail Fast*"
 - **Identifier** si un artefact n'est pas déployable (tests...)
- S'assurer qu'une même version de la base de code est utilisée à tout moment pour un Pipeline donnée

Optimiser le Pipeline



- Paralléliser les étapes
 - Arrêt du Pipeline si une "branche" est en erreur
 - Sinon: étape inutile à supprimer
- Les "gates" manuelles peuvent également être paralleliser
 - relation "1-N": N "gates" manuelles déclencheront N étapes parallèles

Exemple de Pipeline optimisé



Pour aller plus loin...

Un peu de lecture :

- <http://devops.com/2014/07/29/continuous-delivery-pipeline/>
- <http://martinfowler.com/bliki/DeploymentPipeline.html>
- <http://www.informit.com/articles/article.aspx?p=1621865>
- <https://www.thoughtworks.com/insights/blog/architecting-continuous-delivery>

Sécurité: Basiques

Pourquoi la Sécurité ?

- Votre organisation utilise l'*information* pour créer de la valeur
- L'information doit donc être:
 - Confidentielle
 - Intègre
 - Disponible

Qu'est ce que la Sécurité ?

- C'est l'ensemble des pratiques et des outils permettant de prévenir et combattre les menaces sur l'organisation
- 4 piliers:
 - **Connaissance du système**
 - **Least Privilege**
 - **Défense en profondeur**
 - **Mieux vaut prévenir que guérir**

AAA

- AAA signifie :
 - Authentification
 - Authorisation
 - Accounting (comptabilisation)

Authentification

- C'est l'ensemble des procédures et outils pour **identifier** un acteur avec une **confiance suffisante**



Authorisation

- Une fois l'acteur identifié avec confiance, il faut contrôler ses droits en terme de manipulations
- Nomenclature :
 - **Ressources:** Tâches ou objets manipulables et accessibles
 - **Rôles:** Ensemble de droits regroupés par commodité
 - **Requêteurs:** Acteur souhaitant manipuler des ressources

Accounting (Comptabilisation)

- Etre *autorisé* à manipuler des ressources ne garantie pas l'effectuation à 100%
 - Limites du système (mémoire, disque, consommation, temps, etc.)
 - Erreurs, pannes et fautes
- L'"accounting" permet de mesurer et contrôler les manipulations
 - Respect des limites
 - Reprises sur erreur
 - **Capacity planning**

Pour aller plus loin...

Un peu de lecture :

- https://www.owasp.org/index.php/Main_Page
- <https://danielmiessler.com/study/infosecconcepts/>
- <http://searchsecurity.techtarget.com/definition/authentication-authorization-and-accounting>
- <http://www.nap.edu/read/1581/chapter/1>
- <https://cryptome.org/2013/09/infosecurity-cert.pdf>
- <https://danielmiessler.com/study/infosecconcepts/>

Démarrer le Lab

Pré-requis

- VirtualBox (5.1.30+)
 - Check: VboxManage -v
- Vagrant (2.0.1+)
 - Check: vagrant -v
- Pour le proxy HTTP:
 - <https://github.com/tmatilai/vagrant-proxyconf>
 - export VAGRANT_HTTP_PROXY=... && vagrant ...
 - et/ou <https://github.com/AlbanMontaigu/docker-transparent-proxy>

Obtenir Le template de la VM

```
vagrant box add jenkins-lab-demo http://bit.ly/2Bl1hfI
```

Initialiser l'environnement

- Créer un dossier de travail nommé `jenkins-lab-demo` dans votre HOME utilisateur
- Se positionner dans ce dossier
- Exécuter `vagrant init`

```
mkdir ~/jenkins-lab-demo  
cd ~/jenkins-lab-demo  
vagrant init -m -f jenkins-lab-demo
```

Démarrer / Arrêter le Lab

```
vagrant up # Démarrer la VM  
vagrant suspend # Suspendre la VM  
vagrant resume # Relancer la VM  
vagrant halt # Arrêter "proprement" la VM  
vagrant destroy # Détruire la VM (BRUTAL)
```

Accéder au Lab

- Dans votre navigateur, ouvrir <http://localhost:10000>
- Bienvenue sur la page d'accueil du Lab:
 - Le lien **Workshop Slides** vous permettra d'accéder au slides

Accéder au Code Source

- Cliquer sur le lien **Git Server** sur la page d'accueil du Lab
 - Lien direct
 - S'authentifier en tant que **butler** (mot de passe `butler`)
- Cliquer sur **Explore** (en haut)
 - Cliquer sur **butler/demoapp**
 - Lien direct

Demo Application

Pourquoi ?

- **But :** Illustrer un exemple de "Software Supply Chain"
- **Problématique :** Quel language/frameworkoutil choisir ?
- **Solution:**
 - *"Opinionated"* demo application
 - Tout le monde sur le même pied

Demo Application

- Application Web
- Page d'accueil affichant "Greetings from Spring Boot!"

Demo Application : Technical Stack

- C'est un des exemples de Spring Boot Starter
- Langage: **Java** (OpenJDK 8)
- Toolchain: **Maven** (Maven \geq 3.3)
- Code source stocké dans un dépôt **Git**

Demo Application : Check it

- Configuration Maven: pom.xml
- Code source de l'application: src/main/java/
- Code source des tests: src/test
- Script utilitaires: scripts

Web Command Line

- Ouvrir la console **DevBox**
 - Lien direct
 - WebSockets doit être autorisé

Command Line Tricks

- Clean the window: `clear`
- Show command history: `history`
- CTRL + R: search the command history interactively
- CTRL + C: cancel current command and clean line buffer
- CTRL + A: jump to beginning of line
- CTRL + E: jump to end of line

Quelques Commandes

- cat (concatenate)
- ls (list)
- cd (change directory)
- pwd (print working directory)
- man (manual)
- rm (remove)
- mkdir (make directory)
- touch (create an empty file)

Obtenir le code de la demo

- Obtenir l'adresse HTTP du dépôt depuis le **GitServer**

```
# Get the git repository
git clone http://localhost:10000/gitserver/butler/demoapp
# Browse to the local repository
cd ./demoapp
# Check source code
ls -l
```

Demo Application : Maven

- Maven TL;DR:
 - Workflow standardisé
 - pom.xml décrit l'application (Project Object Model)
- Maven Command line : mvn, attend des **goals** en argument

```
mvn dependency:list
```

- Accepte des **flags**

```
mvn dependency:list -fn
```

Demo Application : Compiler

- Goal **compile**
 - Résolution des dépendances
 - Pré-traitement du code source
 - Compilation des classes
- Résultats dans le dossier **./target**:

```
mvn compile  
ls -l ./target
```

Demo Application : Tests Unitaires

- Goal **test**
 - Exécute le goal compile
 - Compilation des tests unitaires
 - Exécution des tests unitaires
- Rapports de tests dans **./target/surefire-reports** :

```
mvn test
ls -l ./target/surefire-reports
```

Demo Application : Construire

- goal **package**
 - Exécute les goals compile et test
 - Paquetage de l'application
- Résultat dans ./target

```
mvn package  
ls -l rh ./target/
```

Demo Application : Exécution

- Spring Boot demo exécutée comme un "**Über-Jar**"
- Exécution avec la commande java:

```
java -jar ./target/demoapp-1.0.0.jar
```

- Ouvrir une autre instance de **DevBox** et vérifier <http://localhost:8080> avec curl

Demo Application : Tests d'intégration

- Goal **verify**
 - Exécute compile, test et package
 - Compile et exécute les tests d'intégration sur l'application empaquetée
- Rapport de tests dans **./target/failsafe-reports**:

```
mvn verify # 1 test failure expected
ls -l ./target/failsafe-reports
```

Demo Pipeline

Discussion

- Définissons ensemble un pipeline

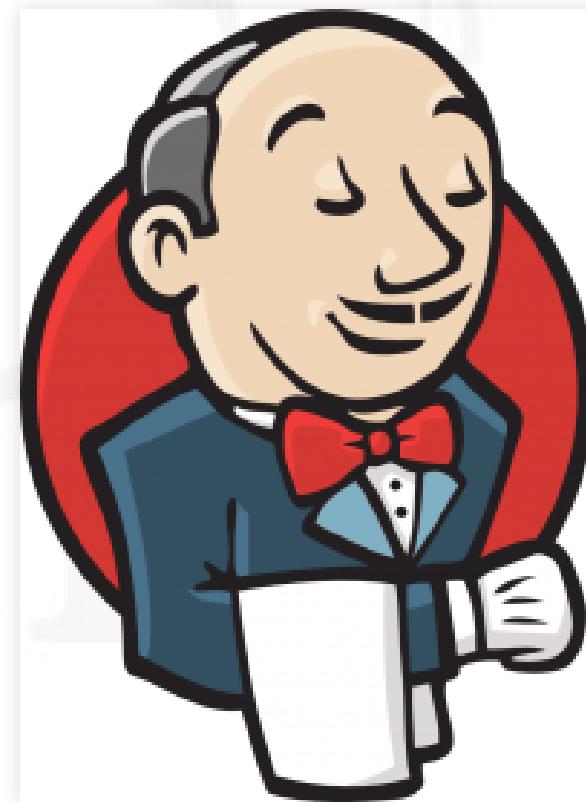
Scripts

- Etudier le contenu du dossier scripts

Jenkins

Meet Jenkins

Jenkins is an open source automation server which enables developers around the world to reliably build, test, and deploy their software.



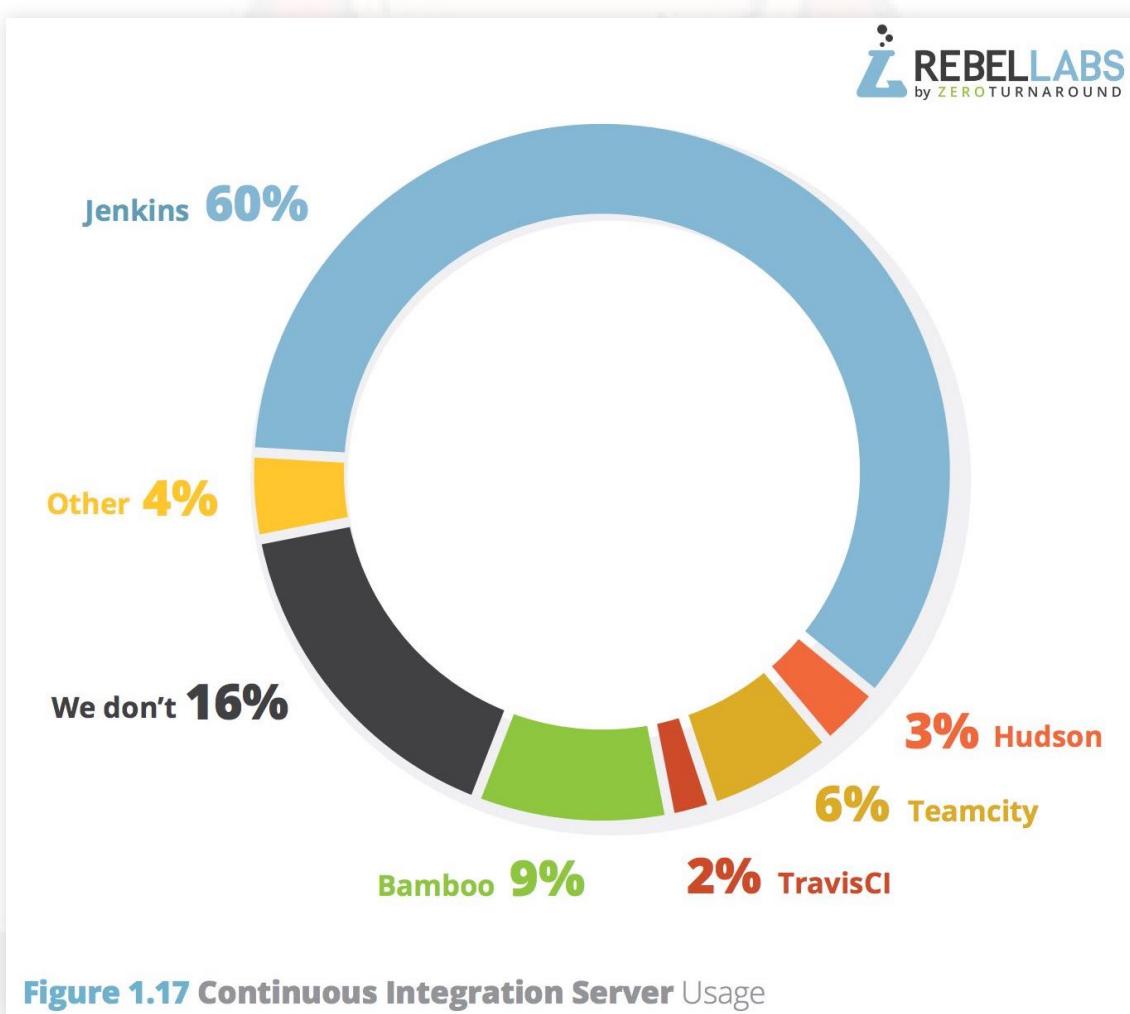
Jenkins

- Orchestrateur de tâches Open Source
- #1 Serveur Intégration Continue Integration
- Un des tout premiers moteur d'intégration continue
 - Crée par Kohsuke Kawaguchi en 2006
- Architecture orientée plugins

Communauté Jenkins

- Une communauté **indépendant et active** (jenkins.io)
 - Projet original: "Hudson", renommé "Jenkins" en 2011
 - 500+ releases
 - 150,000+ installations actives
 - 1,200+ plugins

Un Outil Populaire



Source : RebelLabs Tools and Technologies Leaderboard 2016

Jenkins en 2016 : Jenkins 2

- **But : CI → CD**
- Pas de "cassure" depuis Jenkins 1
- Expérience "premier démarrage" améliorée
- **Pipeline-as-Code:**
 - Syntaxe "scriptée" : "Code your Pipeline"

Jenkins en 2017



Introducing Jenkins
Blue Ocean



Blue Ocean

- Pierre angulaire de l'évolution de Jenkins
- Visualisation et manipulation de "Pipelines"
- GUI moderne, se concentrant sur les actions principales

Jenkins Pipeline

- Un outil pour définir votre "Pipeline" dans Jenkins
- Le Pipeline est décrit dans un fichier texte: le `JenkinsFile`
 - DSL spécifique
 - Stocké dans un SCM

Débuter avec les Pipelines

- "Pipeline-as-code" : Nous avons besoin d'un Jenkinsfile
- Par où commencer ?
 - Getting Started with Pipeline
 - Pipeline "Handbook"
 - Pipeline Syntax Reference
 - Pipeline Steps Reference

Syntaxe Déclarative ou Scriptée ?

- Declarative
 - Syntaxe par défaut
 - S'utilise avec Blue Ocean
- Scripted
 - Syntaxe originale (~3 ans)
 - "Great Power == Great Responsibility"
 - À utiliser lorsque le Déclaratif commence à être **bizarre**

Blue Ocean Pipeline Editor

- Fourni le cycle ("round trip") **complet** avec le SCM
- Pas de Pipeline ? "Suivez le guide".
- Le Pipeline existe déjà ? Edit, commit, et exécutez le

Simple Jenkins Pipeline

Accéder à Jenkins

- Cliquez sur le lien "Jenkins" sur la page d'accueil du lab
 - Lien direct
 - Authentifiez vous en tant que **butler** (le mot de passe est butler)

Blue Ocean

- Passez à l'interface Blue Ocean:
 - Lien direct
 - *Ou cliquez sur le lien "Open Blue Ocean" à gauche*

Exercice : Configurer le projet dans Jenkins

- Créer un nouveau Pipeline, avec les réglages suivants :
 - Stocké dans **Git**
 - Utilisez l'URL en **SSH**
 - Configurez Gitea pour que Jenkins puisse accéder au code :
 - Dans Gitea, cliquez en haut à droite sur le drop-down
 - Allez dans "Settings" → "SSH/GPG Keys"
 - Ajoutez la clef SSH publique générée par Jenkins

Exercice : Votre Premier Pipeline

- Cliquez sur le bouton **Create a Pipeline**
- Utilisez **Blue Ocean Pipeline Editor** et le serveur **Gitea**
- Créez un pipeline avec 3 "stages" : **Build, Test et Deploy**
- Chaque "stage" doit avoir 1 "step" qui affiche un message
 - "Building..." pour **Build**, "Testing..." pour **Test** ...

Solution : Votre Premier Pipeline

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Build'
            }
        }
        stage('Test') {
            steps {
                echo 'Test'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploy'
            }
        }
    }
}
```

Exercice : Exécuter des tâches

- En utilisant **Blue Ocean Pipeline Editor**:
 - Modifiez les 3 stages du pipeline actuel
 - Les scripts sont stockés dans le dossier `./scripts`
 - Utilisez la "step" **Shell Script** (mot clef `sh`)
 - Supprimez les steps `echo`

Solution : Exécuter des tâches

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh './scripts/build.sh'
            }
        }
        stage('Test') {
            steps {
                sh './scripts/test.sh'
            }
        }
        stage('Deploy') {
            steps {
                sh './scripts/deploy.sh'
            }
        }
    }
}
```

Exercice : Archives et Tests

- En utilisant le **Blue Ocean Pipeline Editor**:
 - Modifiez le pipeline actuel
 - La "stage" **Build** doit archiver tous les fichier ".jar" générés
 - *Indices* : Dans le dossier target, motif * .jar
 - La "stage" **Test** doit publier les rapports de test junit
 - *Indices* : Dans le dossier target, motif ** / * .xml
- Le Pipeline doit être **UNSTABLE**

Solution : Archives et Tests

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh './scripts/build.sh'
                archiveArtifacts 'target/*.jar'
            }
        }
        stage('Test') {
            steps {
                sh './scripts/test.sh'
                junit 'target/**/*.xml'
            }
        }
    }
}
```

Amélioration du Pipeline

Exercise: Réparer le build

- Le build est en état UNSTABLE (jaune)
- **Priorité:** réparer le build
- Utiliser **Gitea git server**
- Les tests d'intégration sont dans `src/test/java/hello`
- Indices:
 - *Integration Tests: == IT*

Solution: Réparer le build

- Depuis src/test/java/hello/HelloControllerIT.java
 - Cliquer sur "Edit"
- Commenter la ligne 39
- Dé-commenter la ligne 40
- Commit avec un message (push automatique)
- Lancer le build manuellement dans Blue Ocean
- Le build doit être vert (**Stable**)

Webhooks pour un retour plus rapide

- Nous avons dû lancer le build manuellement
- IC: Retours **rapides** !
 - Lancer le build dès que le code est poussé

Exercise: Webhooks

- Configurons un "**Webhook**" :
 - Depuis **Gitea git server** → **Settings** → **Webhooks**
 - Lien direct vers la configuration de Webhooks
- Ajouter un nouveau webhook:
 - Type: **Gitea**
 - When should this webhook be triggered?: **I need everything**
 - Payload URL: `http://localhost:10000/jenkins/job/demoapp/build?`
`delay=0`

Solution: Webhooks

- Ajoutez un commentaire dans le Jenkinsfile depuis **Gitea git server**
 - Un build va démarrer
 - Validez dans l'onglet "Changes"

Aller plus loin...

- Dans l'éditeur **Blue Ocean**, voir la version textuelle:
 - Combinaison CTRL + S (On Mac: CMD + S)
 - Bi-directionnel: essayez de charger une solution de pipeline
- Le **Pipeline Syntax Snippet Generator** comme acolyte:
 - Génération dynamique en fonction de vos plugins
 - Depuis l'interface "ancienne" de Jenkins
 - Menu de gauche de votre job "Pipeline" (ou MultiBranch)
 - <http://localhost:10000/jenkins/job/demoapp/pipeline-syntax/>

Exercise - Réutilisation Binaire

- But: réutiliser les binaires générés dans **Build**
- Action: "mise sur étagère": **Stash / Unstash**
- Modifier le Pipeline pour :
 - "Stasher" le dossier `target`, à la fin de la phase **Build**
 - "Unstasher" au début de la phase **Test**
- *Attention*, une limite de l'éditeur **Blue Ocean** va être atteinte

Solution - Réutilisation Binaire

- L'éditeur Blue Ocean ne supporte pas encore le ré-ordonnancement de "stages"
 - Mode textuel et/ou **Snippet Generator** à utiliser

```
pipeline {  
    agent any  
    stages {  
        stage('Build') {  
            steps {  
                sh './scripts/build.sh'  
                archiveArtifacts 'target/*.jar'  
                stash(name: 'build-result', includes: 'target/**/*')  
            }  
        }  
        stage('Test') {  
            steps {  
                unstash 'build-result'  
                sh './scripts/test.sh'  
                junit 'target/**/*.xml'  
            }  
        }  
    }  
}
```

Post-Stage

- Section post :
 - Contient des "steps" à exécuter à la fin du Pipeline **ou** après une "stage"
 - Divisé en "condition d'états": always, success, failure, changed
- Chaque condition contient ses propres "steps"
- Pas *encore* intégré dans l'éditeur Blue Ocean

Exercice - Rapport de Tests Unitaires

- Si le "stage" **Build** échoue, alors la tâche "archiveArtifacts" ne devrait pas être exécutée
 - Même chose pour stash
- Les rapports de tests unitaires doivent être publiés dans **tous** les cas après la phase **Build**
 - Format Junit
 - Stockés dans target/**/*.xml
- Utiliser la documentation:
 - Post section on docs.jenkins.io

Solution - Rapport de Tests Unitaires

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh './scripts/build.sh'
            }
            post {
                success {
                    archiveArtifacts 'target/*.jar'
                    stash(name: 'build-result', includes: 'target/**/*')
                }
                always {
                    junit 'target/**/*.xml'
                }
            }
        }
    }
}
```

Agents

Qu'est ce qu'un Agent?

- Un noeud (ou **node**) est une machine prête à recevoir des *builds*
- Step agent spécifie sur quel "noeud" exécuter des "stages".
- Une section agent globale **doit** être définie (au niveau du block pipeline)
- On peut aussi définire des sections agent par "stage"

Exercice : Agents

- Exécuter l'étape **Build** sur un agent configuré avec le label maven-jdk8
- Exécuter l'étape **Test** sur un agent configuré avec le label java8
- L'éditeur Blue Ocean est utilisable

Solution : Agents

```
pipeline {
    agent any
    stages {
        stage('Build') {
            agent {
                node {
                    label 'maven-java8'
                }
            }
            steps {
                sh './scripts/build.sh'
            }
            post {
                always {
                    junit 'target/**/*.xml'
                }
            }
        }
    }
}
```

Exercice : Tests Parallèles

- **But:** Tester en parallèle l'application sur java 7 et 8
- Mot clef `parallel` définissant un block contenant des "stages"
- Agent `java7` pour le Test Java 7
- L'éditeur Blue Ocean est utilisable (et recommandé)

Solution : Tests Parallèles

```
pipeline {
    agent any
    stages {
        stage('Build') {
            agent {
                node {
                    label 'maven-java8'
                }
            }
            steps {
                sh './scripts/build.sh'
            }
            post {
                always {
                    junit 'target/**/*.xml'
                }
            }
        }
    }
}
```

Agents avec Docker

- **But:** Usage de Docker pour faciliter la définition des environements de build
- Le mot clef `agent` permet d'exécuter les "stages" dans un container Docker, depuis une "image Docker", ou depuis un `Dockerfile` (recette maison d'image Docker)
 - Directive `Agent` sur jenkins.io

Exercice : Agent Docker

- Exécuter le **Build** dans un conteneur basé sur le fichier `Dockerfile.build`
- Exécuter le **Test Java 8** dans un conteneur basé sur les images maven : `3-jdk-8-alpine`
- Trick: documentation manquante sur `filename`, dans un block `dockerfile`

Solution : Agent Docker

```
pipeline {
    agent any
    stages {
        stage('Build') {
            agent {
                dockerfile {
                    filename 'Dockerfile.build'
                    label 'docker'
                }
            }
            steps {
                sh './scripts/build.sh'
            }
        post {
            always {
```

Pipeline Avancés

Input

- Jenkins permet de "mettre en pause" l'exécution d'un pipeline, en attendant une validation humaine
- Continuous Delivery / Deployment
- Mot-clef `input`, c'est une "step" Pipeline
 - Hautement configurable: Documentation Step Input
- Généralement utilisé dans un "stage" dédié.
 - Obligatoirement avec `agent none` (pas d'exécuteur, pas de workspace)

Exercice : Validation Humaine

- Ajouter une stage "approval" avant le déploiement

Solution : Validation Humaine

```
pipeline {
    agent any
    stages {
        stage('Build') {
            agent {
                dockerfile {
                    filename 'Dockerfile.build'
                    label 'docker'
                }
            }
            steps {
                sh './scripts/build.sh'
            }
        post {
            always {
```

Conditionnal Stage

- Définir l'exécution conditionnelle d'une "stage"
- Mot clef `when`
- Doit contenir au moins une condition parmi:
 - `branch`
 - `environment`
 - `expression`
 - Logique "built-in": `allOf`, `anyOf`, etc.
- When Documentation

Variables d'environnements

- Définir des collections de clé-valeurs, en tant que variables d'environnement:
 - Définition globale ou par "stage"
 - Ou en utilisant `withEnv` dans un `block steps` pour une instruction spécifique

Exercice : Déploiement Conditionnel

- N'exécuter les "stages" **Approval** et **Deploy** que:
 - Si on se trouve sur la branche master
 - Ou si la variable FORCE_DEPLOY est à true

Solution : Déploiement Conditionnel

```
pipeline {
    agent any
    environment {
        FORCE_DEPLOY = 'false'
    }
    stages {
        stage('Build') {
            agent {
                dockerfile {
                    filename 'Dockerfile.build'
                    label 'docker'
                }
            }
            steps {
                sh './scripts/build.sh'
            }
        }
    }
}
```

Paramètres

- Directive parameters : l'utilisateur fournit des arguments au pipeline
- Disponible en tant que variables accessibles dans l'objet params
- Oeuf & Poule : Jenkins ne peut accéder au paramètre lors du 1er build

Exercice : Paramètres

- Ajouter un paramètre DEPLOY_MESSAGE dont la valeur par défaut est Deploy ?
- Ce paramètre est utilisé dans le input

Solution : Paramètres

```
pipeline {
    agent any
    parameters {
        string(name: 'DEPLOY_MESSAGE', defaultValue: 'Deploy ?', description: 'Message de déploiement')
    }
    environment {
        FORCE_DEPLOY = 'false'
    }
    stages {
        stage('Build') {
            agent {
                dockerfile {
                    filename 'Dockerfile.build'
                    label 'docker'
                }
            }
        }
    }
}
```

Options & Configurations

- **Options:** Configurer un "job" depuis le Pipeline
- **Configuration:** Même chose dans la GUI "Legacy"
- Mot clef options
- Documentation des options:
<https://jenkins.io/doc/book/pipeline/syntax/#options>
- Certaines options comme `timeout` peuvent être appliquée dans un bloc `step`

Exercice : Options & Timeout

Approval

- On souhaite que la "stage" **Approval** attende 3 minute avant d'arrêter le pipeline
- On ne veut conserver que les 5 derniers builds d'un pipeline: limiter l'usage disque

Solution : Options & Timeout

Approval

```
pipeline {
    agent any
    options {
        buildDiscarder(logRotator(numToKeepStr: '5'))
    }
    parameters {
        string(name: 'DEPLOY_MESSAGE', defaultValue: 'Deploy ?', description: 'Message de déploiement')
    }
    environment {
        FORCE_DEPLOY = 'false'
    }
    stages {
        stage('Build') {
            agent {
                dockerfile {
                    ...
                }
            }
        }
    }
}
```

Shared Libraries

Exercice : Shared Libraries

- Create a repository jenkins-libs
 - Init with a file vars/runPipeline.groovy:

```
def call(String MESSAGE) {  
    echo "Lib: ${MESSAGE}"  
}
```

- Switch to legacy UI, configure the Multibranch job
 - Add the shared library
- Adapt the pipeline to run the function in Build stage

Solution : Shared Libraries

```
pipeline {
    agent any
    libraries {
        lib('jenkins-libs@master')
    }
    options {
        buildDiscarder(logRotator(numToKeepStr: '5'))
    }
    parameters {
        string(name: 'DEPLOY_MESSAGE', defaultValue: 'Deploy ?', description: 'Message de déploiement')
    }
    environment {
        FORCE_DEPLOY = 'false'
    }
    stages {
        stage('Deploy') {
            steps {
                echo "Deploying ${DEPLOY_MESSAGE}..."
                sh "echo ${DEPLOY_MESSAGE} | ./deploy.sh"
            }
        }
    }
}
```

Aller plus loin

Industrie CI/CD

- Industrialisation du logiciel : secteur balbutiant et très dynamique
- Beaucoup de solutions !

Analyse du besoin

- Hébergement du Service ?
- Existant ?
- Profil des utilisateurs ?
- Criticité ?
- CI seul ou "plus large" ?
- Topologie de facturation ?

De nombreux outils 1/2

- SaaS (modèles gratuits et/ou Enterprise) :
 - TravisCI
 - CircleCI
 - CodeShip
 - GitLab.com
 - Microsoft VSTS
 - Amazon Pipeline
 - BitBucket Pipeline

De nombreux outils 2/2

- A héberger soit-même :
 - Jenkins
 - CloudBees Jenkins Enterprise
 - GitLab
 - Microsoft VSTS
 - Atlassian Bamboo (BitBucket Pipeline)

Le Futur

- "Serverless": <https://github.com/lambci>
- Glisser vers la production (OpenShift Pipelines)