Développement > Intégration > Déploiement

Intégration continue

# Qu’est-ce que l’intégration ?

L’intégration est un ensemble de processus, tels que la compilation, l’exécution des tests, la génération de la configuration, la mise à jour de la base de données, la génération de métrique de qualité… permettant

En génie logiciel l’intégration est l’ensemble des processus permettant la mise en commun de différentes versions de code utilisées par le développeur. Tout comme pour les performances serveurs, les développeurs se sont rapidement rendu compte que la « scalabilité verticale »[[1]](#footnote-1) de leurs compétences était limité. Hypothétiquement même si nous nous plaçons dans un univers parfait un développeur ne peut pas être plus productif indéfiniment. Il éteindra forcément, un jour ou l’autre, son seuil maximal de rentabilité. Les développeurs ont alors pensé à travailler en équipe afin d’accroitre non pas leur propre rendement mais plutôt le rendement du projet informatique.

Le premier obstacle rencontré a été de s’organiser fonctionnellement afin de tirer un maximum avantage de la puissance du travail en équipe. Pour cela les développeurs ont découpé leurs projets initiaux en une multitude de tâches (plus ou moins importantes) qu’ils se sont départagés selon les profils de chacun. Nous parlons alors de « scalabilité horizontale »[[2]](#footnote-2).

# Pourquoi dite « continue » ?

Plus l’on met de temps à faire deux intégrations, plus nous allons ajouter des fonctionnalités, corriger des bugs, modifier les fichiers de configurations donc potentiellement l’intégration sera plus compliquée avec le temps. Donc l’idée de « continue » est de faire régulièrement des intégrations pour minimiser les risques.

L’objectif de l’intégration continue est de transformer le processus d’intégration en un non-événement. Avec pour objectif final le « push button », c’est-à-dire que les développeurs n’auront qu’à appuyer sur un bouton (ou taper une ligne de commande) pour compiler/packager, pour déployer, les testeurs pour lancer les scénarios de tests…

# Composition de l’intégration continue

Qu’est ce qui va composer à minima l’intégration continue ? Il nous faut déjà un mécanisme de surveillance du changement - le développeur a changé une valeur dans un fichier de configuration, une icône, une librairie – un mécanisme pour monitorer (surveiller par le biais de métrique) l’environnement, un gestionnaire de version (de code source), les scripts et les outils pour implémenter les processus dont nous avons parlé plus haut (compilation, packaging …) et un mécanisme de notification parce qu’un workflow qui fonctionne tout seul c’est bien mais si nous ne savons pas ce qu’il fait c’est déjà beaucoup moins intéressant.

# Le cloud

Le cloud nous permet de mettre à disposition rapidement une VM en temps réel.

Jenkins

Serveur d’intégration continue. Jenkins peut effectuer des builds distribués (100 esclaves) et offre un mécanisme de plugins. Jenkins est un produit open source, nous pouvons créer nos propres plugins afin de répondre aux formats de packaging spécifique à notre entreprise.  Ce mécanisme de plugin nous permet d’envisager l’intégration continue sous un angle différent et voir l’intégration continue en tant que fonction.

# Une fonction ?

Nous pouvons imaginer que l’intégration va prendre un objet source (pas nécessairement du code) et nous renvoyer un artefact. Ce qui est important c’est que la fonction n’ait pas d’effet de bord. Mais cette fonction peut-être très couteuse, donc l’idéal est de travailler en asynchrone. Nous allons donc avoir une première phase qui va packager le livrable, et dans un second temps, une fois le livrable packagé, nous pouvons commencer les tests sur le livrable et en parallèle effectuer le calcul des métriques qualités qui pourront-elles mêmes être découpées.

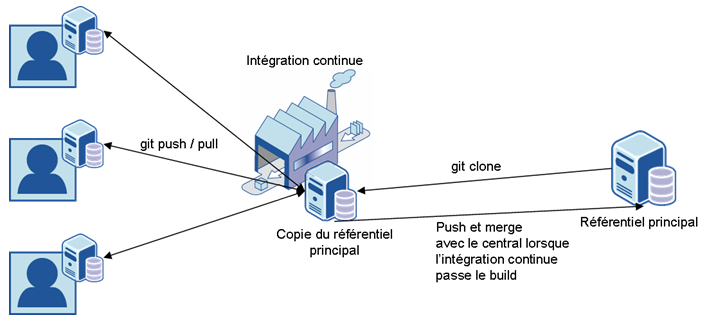
L’objet source de notre intégration continue sera le commit. Il est le candidat naturel comme entrée de la fonction car :

- il détermine le contenue d’une arborescence source,

- il permet une copie simple et rapide d’une machine à l’autre, du fait qu’il ne récupère que le différentiel d’une copie à l’autre. Nous imaginons pour des builds distribuées, même si nous avons notre repository central qui a changé, la récupération par l’ensemble des machines esclaves va être très rapide.

Nous allons utiliser l’intégration continue en tant que Revue de Code automatique. Nous allons avoir une structure avec un repository central partagé (master), un repository local pour chaque chaque développeur (sur son poste) et entre les deux en repository dédié à Jenkins.

Le développeur code, fait un commit – le commit n’est qu’en local chez le développeur – et décide de pousser son commit sur le repository de Jenkins. Jenkins lance alors les différentes tâches qui lui sont allouées et si tous les tests passent Jenkins va pousser le commit sur le repository central. Ce qui nous assure un repository central toujours validé.



Contenu

[Qu’est-ce que l’intégration ? 1](#_Toc442802851)

[Pourquoi dite « continue » ? 1](#_Toc442802852)

[Composition de l’intégration continue 2](#_Toc442802853)

[Le cloud 2](#_Toc442802854)

[Une fonction ? 2](#_Toc442802855)

[2.1 Histoire 6](#_Toc442802856)

[2.2 Intégration Continue comme un processus 6](#_Toc442802857)

[2.1.1 Les bénéfices de l’Intégration Continue 7](#_Toc442802858)

[Réduire les risques 7](#_Toc442802859)

[Générer des logiciels déployables 7](#_Toc442802860)

[Permettre une meilleure visibilité du projet 7](#_Toc442802861)

[Une plus grande confiance du produit 7](#_Toc442802862)

[2.2.2 Le cycle de travail de l’Intégration Continue 8](#_Toc442802863)

[2.2.3 Comment l’Intégration Continue s’appuie-t-elle sur d’autres pratiques de développement 10](#_Toc442802864)

[2.3. L’Intégration Continue est un mélange de personnes et systèmes 10](#_Toc442802865)

[2.3.1 Les développeurs 10](#_Toc442802866)

[2.3.2 Les répertoires de contrôle de version 10](#_Toc442802867)

[2.3.3 Les serveur d’Intégration Continue 10](#_Toc442802868)

[2.3.4 Les scripts de construction (build) 10](#_Toc442802869)

[2.3.5 Les mécanismes de feedback 10](#_Toc442802870)

[2.3.6 Les machines de construction d’intégration 10](#_Toc442802871)

[2.4 Caractéristique de l’Intégration Continue 10](#_Toc442802872)

[2.4.1 Compilation du code source 10](#_Toc442802873)

[2.4.2 Tests 10](#_Toc442802874)

[Niveaux de tests 10](#_Toc442802875)

[Exécuter les tests plus rapides en première 10](#_Toc442802876)

[Ecrire des tests d’échecs 10](#_Toc442802877)

[2.4.3 Base de données d’intégration 10](#_Toc442802878)

[2.4.4 Inspection Continue 10](#_Toc442802879)

[Différences entre inspection et test 10](#_Toc442802880)

[Rapport d’inspection 10](#_Toc442802881)

[2.4.5 Deployment 10](#_Toc442802882)

[Etiqueter les actifs d’un référentiel 10](#_Toc442802883)

[Produire un environnement propre 11](#_Toc442802884)

[Etiqueter chaque build 11](#_Toc442802885)

[Exécuter tous les tests 11](#_Toc442802886)

[Créer des build feedback reports 11](#_Toc442802887)

[Capacité à effectuer un rollback 11](#_Toc442802888)

[2.4.6 Documentation 11](#_Toc442802889)

[2.4.7 Feedback Continue 11](#_Toc442802890)

Devops

Depuis maintenant quelques années (il est difficile de donner une date précise), les DSI s’appuient sur la mouvance agile afin de mener à bien leurs projets. Aujourd’hui, les patterns agiles arrivent à maturité et offrent un éventail de méthodologies adaptables à tous les contextes. Les méthodes agiles garantissent la satisfaction du client et non la conformité aux termes d’un contrat de développement. Elles sont centrées sur la satisfaction de besoin du client et non sur les termes contractuels du projet. Nous n’allons pas aborder en profondeur le concept de l’agilité, ceci n’est pas le propos de ce mémoire, mais nous allons tout de même faire un petit rappel des idées fortes de cette méthodologie. Il faut des **cycles courts**, quelques semaines tout au plus, et découper le projet en **petites tâches** puis les **hiérarchiser** **en fonction du besoin**. Cela permet d’éviter le superflu et de se concentrer au début de chaque cycle sur ce qui a de la valeur pour l’utilisateur final. Le **feedback** permanent devient la règle d’or, avec des validations à chaque étape et des techniques ludiques d’évaluation de l’utilité des fonctions. L’agilité offre une meilleure visibilité et permet d’éviter les dérives observées lorsque les développeurs sont isolés. Le **changement** est autorisé voir encouragé, même tardivement,  car c’est un avantage décisif pour le client. Cela permet de ne pas se priver des bonnes idées en cours de route et surtout d’éliminer les mauvaises idées lancées au début du projet. Les méthodes agiles favorisent la **co-construction**, en intégrant l’annonceur lui-même dans le travail quotidien et en responsabilisant la totalité de l’équipe de développement, créant ainsi un véritable **esprit collaboratif** et l’ensemble du projet en gagne en qualité.

Cependant l’agilité, lorsqu’elle est exclusivement cantonnée au développement, se trouve néanmoins freinée par les tâches d’exploitation. Le mouvement DevOps a pour objectif d’étendre les pratiques agilistes à la livraison et au déploiement du projet.

# Scrum

# Devops

Intégration Continue

# 2.1 Définition

L'idée de l'Intégration Continue a été développée par la communauté Extreme Programming et a été décrit par Kent Beck dans son livre « Extreme programming explained »[[3]](#footnote-3). Martin Fowler a également été l'un des premiers contributeurs ayant écrit sur l’intégration continue[[4]](#footnote-4). Plus tard, les travaux ont été poursuivi par Paul Duvall qui a écrit tout un livre à propos de l’intégration[[5]](#footnote-5).

Dans l’industrie logicielle l’intégration d’un projet logiciel est souvent un moment lourd et douloureux. La mise en commun des différents modules composants l’application entraine de grave problème d’intégration. Les modules fonctionnent correctement individuellement mais se confrontent à des problèmes de synergie. La résolution de ces problèmes demande un effort important qui s’accroisse avec la complexité du système. L’introduction des techniques et méthodologie de l’Intégration Continue, estompent les problèmes d’intégration jusqu’à les réduire en un non-évènement.

« Continuous integration is the practice of making small well-defined changes to a project’s code base and getting immediate feedback to see whether the test suites still pass. »

« L'intégration continue est la pratique de faire de petits changements bien définis à la base du code source d'un projet et d'obtenir une rétroaction immédiate pour voir si les suites de test passent toujours» Paul Duvall.

L’idée d'Intégration Continue a été développée par la communauté Extreme Programming (XP) et a été décrit par Kent Beck dans son livre « Extreme programming explained », articulé autour de douze pratiques de développement agile. Son but étant de prévenir les problèmes décrit ci-dessus désignés comme « integration hell » (l’enfer de l’intégration) par Ron Jeffries en 2001. Martin Fowler a également été l'un des premiers contributeurs ayant écrit sur l’intégration continue[[6]](#footnote-6). Plus tard, les travaux ont été poursuivi par Paul Duvall qui a écrit tout un livre à propos de l’intégration[[7]](#footnote-7).

Martin Fowler désigne 10 principes clés pour réussir un Intégration Continue efficace :

* Maintenir un référentiel de source unique,
* Automatiser la construction de l’application (build),
* Automatiser les tests,
* Valider quotidiennement les modifications au niveau de la branche principale du contrôle de version (commit),
* Créer une build d’intégration à chaque commit sur la branche principale,
* Assurer une build rapide,
* Effectuer les tests dans un environnement clone de la production,
* Assurer la disponibilité pour tous des derniers livrables,
* Assurer une visibilité pour tous,
* Automatiser le déploiement.

# 2.2 Intégration Continue comme un processus

L’Intégration Continue est un processus où le logiciel est construit à chaque changement. Cela signifie que lorsqu’une modification apportée par un développeur est détectée dans le code source, une construction automatique est déclenchée sur une machine de construction séparé. La construction contient plusieurs étapes prédéfinies comme la compilation, les tests, la génération de métrique du code source et le déploiement - entre autres. Une fois la construction terminé un rapport est envoyé aux membres du projet spécifié. Le rapport de compilation indique le résultat de chaque étape de la construction avec des informations détaillées sur les erreurs possibles qui ont pu survenir.

Martin Fowler [Fow00] décrit l’Intégration Continue comme :

« 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. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.”

« L'Intégration Continue est une pratique de développement logiciels où les membres d'une équipe intègre leur travail régulièrement, chaque développeur intègre au moins quotidienne une version - conduisant à de multiples intégrations journalière. Chaque intégration est vérifiée par une construction automatique (y compris le test) pour détecter les erreurs d'intégration le plus rapidement possible. De nombreuses équipes trouvent que cette approche conduit à réduire considérablement les problèmes d'intégration et permet une équipe pour développer des logiciels cohésifs plus rapidement. »

## 2.1.1 Les bénéfices de l’Intégration Continue

Selon Paul Duvall [Duv07], l'intégration logiciels n’est pas un problème dans les petites équipe, un à deux développeurs, mais quand le nombre de collaborateur se multiplie ou que diverses équipes commencent à travailler ensemble sur un projet, l'intégration logiciels devient un problème, car plusieurs personnes modifient simultanément des morceaux de code devant fonctionner ensemble. Vérifier que les divers composants logiciels interdépendants continuent de fonctionner correctement ensemble soulève la nécessité d'intégrer plus tôt et plus souvent. Les points suivants décrivent les différents effets bénéfiques que Paul Duvall a été en mesure d'identifier.

### Réduire les risques

En intégrant plusieurs fois par jour, les risques de dysfonctionnement sont considérablement réduits. Les problèmes sont remontés dès leur intégration et peuvent même être la cause d’un rejet d’intégration. Ceci étant possible par l’intégration et l’exécution de tests et l’inspections automatiquement du code source après chaque modification.

### Générer des logiciels déployables

L'un des objectifs du développement logiciel agile est de déployer tôt et souvent. L’Intégration Continue aide à atteindre cet objectif en automatisant les étapes de production des logiciels déployables. Des logiciels déployables et fonctionnels est l'avantage le plus évident de l’Intégration Continue du point de vue extérieur, car le client ou l'utilisateur final est généralement pas intéressés par le fait que l’Intégration Continue ait été utilisé dans le cadre de l'assurance qualité. Il est aussi l'atout le plus tangible, étant le résultat final de l’Intégration Continue.

### Permettre une meilleure visibilité du projet

Le fait que le processus d’Intégration Continue s’exécute régulièrement fournit la capacité à remarquer les tendances et à prendre des décisions sur la base d’informations réelles. Sans Intégration Continue, les informations doivent être recueillies manuellement requérant du temps et des efforts. Le processus d’Intégration Continue fourni en temps réel les informations sur l'état de la construction et de la qualité des dernières mesures tels que la couverture de test ou le nombre de codage violations de la convention.

### Une plus grande confiance du produit

En ayant une Intégration Continue en place, l'équipe projet se protège contre certaines actions négatives portées aux codes sources. L’Intégration Continue agit comme un filet de sécurité, il repère les erreurs tôt et régulièrement. Ce qui se traduit par une plus grande confiance à l'équipe. Même des changements importants peuvent être faits avec confiance.

## 2.2.2 Le cycle de travail de l’Intégration Continue

Alors que l’Intégration Continue est généralement orchestrée par un serveur d’Intégration Continue, c’est aussi une façon de travailler. L'idée principale est d'intégrer les changements au code source aussi souvent qu’il y ait du nouveau code ajoutant de la valeur à l’application, à savoir plusieurs fois par jour. Cela rend l'intégration du code plus facile parce qu'il y a moins de code à intégrer à la fois. Il contribue également dans une situation où plusieurs personnes modifient la même base de code, car les développeurs auront ainsi les modifications apportées par d'autres.

L'intégration des changements au code source de base comporte plusieurs étapes. Le cycle de travail est illustré dans le schéma suivant (voir Figure 2.1). Il est basé sur les idées présentées par Martin Fowler [Fow06] avec une phase de mise à jour supplémentaire, après l'exécution des tests. Cette phase de mise à jour est nécessaire dans la situation où le test prend du temps de sorte qu'il pourrait déjà y avoir de nouvelles modifications intégrées pendant notre phase de test.

1. Récupérer le code source du référentiel (Check-out),

2. Modifier le code et créer les tests pour les modifications,

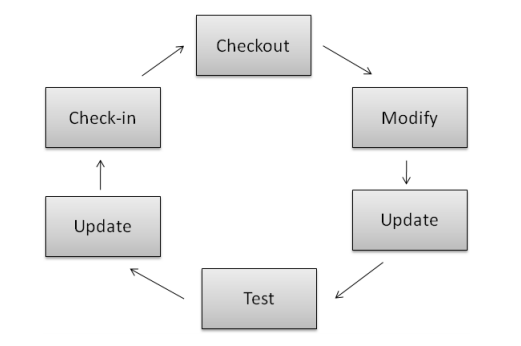
3. Vérifiez si quelqu'un d'autre a modifié le code source,

4. Exécuter tous les tests afin de vérifier que les modifications n’aient pas altéré le bon fonctionnement de l’application,

5. Vérifiez à nouveau si quelqu'un d'autre ont modifié le code source,

6. Valider les modifications dans le référentiel (Check-in),

7. Démarrer un nouveau cycle.



**Figure 2.1 : Cycle de travail de l’Intégration**

**Le cycle de travail commence créer une copie du code source hébergé sur le serveur de gestion de code qui est sur le point d'être modifié. Si le code avait précédemment récupéré, une mise à jour est faite à la place. Le développeur effectue les changements appropriés à sa tâche (nouvelle fonctionnalité, réfactorisation du code…) et implémente les tests appropriés afin de vérifier la conformité de son développement. Un test de développement vérifie que la sortie d’une méthode, en fonction de ses entrées, soit en adéquation avec le résultat attendu. Il est recommandé d’écrire les tests avant le code réel de l’application, ceci est appelé Test Driven Development (TDD).**

**Dans un projet de logiciel, il est courant que de nombreux développeurs travaillent simultanément sur une même partie de code. Par conséquent avant d’intégrer notre partie en code source, il est nécessaire de vérifier qu’aucune modification n’ait été faite par un développeur tierce sur par le biais d’une nouvelle mise à jour. Dans certains cas, il est possible que d’autres développeurs aient modifié exactement les mêmes lignes de code que nous produisant un ou plusieurs conflits. Dans ce cas, les conflits doivent être immédiatement résolus et tous les tests ré-effectués. Il est possible que dans cet intervalle une nouvelle version soit disponible sur le référentiel, l'étape précédente doit être répétée jusqu'à ce qu'aucuns nouveaux changements ne soient détectés. Une fois cette synchronisation effectué le check-in peut être fait.**

## ****2.2.3 Comment l’Intégration Continue s’appuie-t-elle sur d’autres pratiques de développement****

**L’Intégration Continue comprend un ensemble de règles qui tous développeurs devraient suivre :**

**- Effectuer régulièrement des commits,**

**- Ne pas commit du codes cassés,**

**- Régler les problèmes et builder immédiatement,**

**- Ecrire des tests de développement automatisés,**

**- Tous les tests et métriques doivent être valides,**

**- Exécuter en local ses builds,**

**- Eviter de travailler avec du code cassé.**

**Ces règles ne sont pas nouvelles dans le monde du développement logiciel et sont aussi adopté d'autres pratiques de développement. Par conséquent, si des pratiques telles que les tests de développeur, les normes de codage, le refactoring et la propriété collective sont déjà mises en place au sein du projet, il est facile de commencer à utiliser l’Intégration Continue. Ces pratiques doivent être rigoureusement appliquées au travers de l’Intégration Continue sous peine d’empêcher les autres collaborateurs du projet de travailler. Prenons par exemple le cas d’un développeur ayant remanié une partie du code source et cassé quelques tests. Ne l’ayant pas remarqué, faute de n’avoir pas exécuté les tests, il valide ses changements dans le référentiel. La build éclate du faite que la règle « tous les tests doivent être au vert » n’est pas suivie. Maintenant si un autre développeur commence à travailler avec le code du référentiel, la première chose qu'il doit faire est de fixer ce qui avait été cassé par son collègue. Et cela peut prendre un beaucoup de temps, si cette personne ne connaît pas la partie de code qui provoque l’échec des tests.**

# 2.3. L’Intégration Continue est un mélange de personnes et systèmes

## 2.3.1 Les développeurs

Pratiquer l’Intégration Continue exige de la discipline de la part des développeurs. Ils devront appliquer avec rigueur les pratiques de développement vus précédemment. Une fois le développement de la tâche effectué, le développeur doit exécuter une build sur sa propre machine de développement. On appelle cela une build privée. Cette étape permet de vérifier que les modifications apportées n’ont pas endommagé l'intégration avec le reste du code source. Il est important d'exécuter la build privée avant de valider les changements dans le référentiel de contrôle de version, car soumettre un code erroné peut empêcher les autres développeurs de travailler. Une fois l’exécution de la build privée avec succès, le développeur peut valider les modifications, et les tests. Si l'intégration de la build échoue malgré ses précautions, réparer cette build est la priorité numéro un.

## ****2.3.2 Le référentiel de contrôle de version****

L'intégration continue ne peut pas se faire sans référentiel de contrôle de version. Le référentiel de contrôle de version, également connu sous le nom de gestionnaire de code source (SCM), est un système utilisé pour stocker le code source et d'autres aspects du logiciel (comme la documentation) de manière centralisée. Il assure également le suivi de l'historique des versions et modifications effectuées au cours du développement. Les développeurs de logiciels ont la possibilité de revenir à une version antérieure ou la révision d'un logiciel, ou de prendre connaissances des changements apportés sur toute révision donnée. Ce référentiel fournit un point d'accès unique au code source pour les développeurs et le système d’Intégration Continue. Il peut être constitué de différentes branches du logiciel stocké. Une branche peut être créée pour la réécriture majeure d’un morceau de code ou pour le prototypage d’une idée intéressante qui pourrait ne pas se retrouver dans le produit final. La build d’intégration est exécuté sur la branche principale du référentiel de contrôle de version [Duv07]. « Master » est la branche de code source où la plupart de la mise au point a lieu. Certains systèmes de contrôle de version appellent cela aussi « tunk » ou « head ». La ligne principale se doit d’être toujours stable et la build d’intégration ne doit jamais échouée quand elle est intégrée au référentiel.

## ****2.3.3 Le serveur d’Intégration Continue****

Le serveur d’Intégration Continue est l'orchestrateur de l'ensemble du processus. Il exécute la build d'intégration lorsqu'une modification a été apportée au référentiel. Quatre approches sont à prendre en compte.

La première est la configuration d’un « post commit hook » au niveau du gestionnaire de code source. Le référentiel de contrôle de version peut-alors avertir immédiatement le serveur d’Intégration Continue qu’une modification a été ajoutée et validée. De cette façon une build d’intégration est exécutée pour chaque commit.

Une autre approche, dénommée « polling approach »  [Duv07] est de vérifier les changements à intervalles réguliers (de l’ordre de la minute). De ce fait plusieurs changements peuvent être effectués entre chaque build.

Une troisième option est d’exécuter une build d’intégration à intervalles réguliers, mais si l'intervalle est trop long le bénéfice de la rétroaction est rapidement perdu.

La quatrième et dernière option, consiste à intégrer une copie du référentiel principal, accessible uniquement par le serveur d’Intégration Continue, au niveau du serveur lui-même. Les développeurs n’ont ainsi accès qu’au référentiel du serveur d’Intégration Continue. Ce dernier peut alors être configuré afin de rejeter les modifications apporté au référentiel ne respectant pas les tests ou les métriques qualités prédéfinies. Garantissant ainsi la qualité de l’application.

Le serveur d’Intégration Continue fournit également une vue, généralement une page web, qui expose l'état de santé de tous les build jobs[[8]](#footnote-8) configurés (voir Figure 2.2). Ce tableau de bord peut être affiché, par exemple, sur un grand écran dans la salle de l'équipe de développement pour donner un aperçu rapide et en temps réel des tâches effectués sur le serveur et voir ainsi s'il y a des constructions en cours d'exécution. De nombreux serveurs d’Intégration Continue proposent également un type de visualisation basé sur les principes des feux de circulation ou de météo de connaitre l’état des builds d’Intégration.

Toutes les fonctionnalités d’un serveur d’Intégration Continue ne sont pas nécessaires pour faire de l’Intégration Continue. De nombreux scripts personnalisés peuvent effectuer les mêmes tâches, mais avoir un serveur conçu à cet effet aide beaucoup [Duv07]. De plus en plus de solutions propriétaires ou open source[[9]](#footnote-9) performe le marché et offre un environnement d’Intégration Continue stable et complet.

Dans sa forme la plus simple, l’Intégration Continue pourrait être mise en place avec un seul ordinateur dédié exécutant des scripts afin de vérifier le code source du référentiel, lancer une build d’Intégration et envoyer des rapports une fois la build terminée. Le serveur d’Intégration Continue offre une autre possibilité en fournissant une interface utilisateur afin de configurer les multiples « build jobs » et afficher leurs résultats en temps réel.

## ****2.3.4 Les scripts de construction (build)****

Généralement la plupart des étapes d’une build sont définies en utilisant un script de compilation. Un script de compilation peut être constitué d'un ou plusieurs scripts et il est utilisé, par exemple, pour compiler, tester, contrôler et déployer des logiciels. Toutes les mesures qui peuvent être automatisées pour construire et déployer des logiciels doivent être automatisées. Cela économise du temps et les nerfs des développeurs. Il existe de nombreuses techniques disponibles comme Ant (Java), Make (C/C++) ou Scons (Python).

Certains développeurs utilisent leur environnement de développement intégré (IDE) pour builder leurs logiciels. Dans ce cas la build ne pourra être encadré par l’Intégration Continue. L’intégration Continue nécessite que la build puisse être exécutée indépendamment de tout IDE [Duv07].

## ****2.3.5 Les mécanismes de feedback****

Lorsque la build d'intégration est terminée, les résultats doivent être accessibles dès que possible. La capacité à fournir un feedback rapide est l'un des avantages de l’Intégration Continue. La rétroaction est disponible immédiatement une fois la build terminée. La rétroaction peut être diffusée par différent canaux ; tableau de bord, courrier électronique, flux RSS. En cas de build défectueuse, la réparation peut démarrer immédiatement après réception de l'avis.

Certain pionnier commence même à intégrer le terme de « monitoring continue » dans l’ingénierie logicielle en complément de l’Intégration Continue. Le monitoring continu consiste à avoir un affichage visible par tous les membres de l’équipe de développement, actualisé en temps réel, donnant un feedback direct sur l’état des différentes builds simplifié et directement interprétable. Cet affichage est dans la plupart du temps un moniteur, mais d’autres solutions plus amusantes, telles que la lampe à lave ou une « ambient orb » commencent à s’imposer[[10]](#footnote-10).

## ****2.3.6 Les machines de build d’intégration****

Un serveur d’Intégration Continue à besoin d'un hôte pour fonctionner. La machine de build d’intégration (ou nœud) est une machine distincte qui doit imiter l'environnement de production. Si possible, elle doit fonctionner avec le même système d'exploitation, la même version de serveur de base de données et les mêmes versions de librairies doivent être utilisées, comme il est prévu pour en production. Chaque différence augmente le risque des tests de ne pas détecter les problèmes liés à l’environnement de la production [Fow06].

Dans le cas où de nombreux « build jobs » sont configurés pour l’application, pour réduire la durée de la build et ainsi augmenté la rapidité de la rétroaction il peut être nécessaire de paralléliser les tâches sur plusieurs machines (scalabilité horizontal). Certains logiciels de serveur d’Intégration Continue fournissent une architecture maître-esclave qui permet ainsi de diviser la charge de travail sur plusieurs hôtes.

Parfois plusieurs environnements sont nécessaires pour builder sur différentes plates-formes. La virtualisation des serveurs apportent la réponse à ce problème. En utilisant une infrastructure de virtualisation bien établis, il est assez facile d’exécuter des instances esclaves multiples sur un seul nœud physique. Ces instances pouvant fonctionner sous différents systèmes d’exploitation. Certains logiciels de serveur d’Intégration Continue offre la possibilité à la build d’intégration de fonctionner simultanément sur ces différentes instances et de collecter les résultats pour chaque environnement.

Puis virtualisation est quelque chose de valeur à l'aide de la peine de la virtualisation diminue[[11]](#footnote-11) [Bar03]. Certains logiciels de serveur de CI comme Hudson permet même de construction de travail afin de fonctionner simultanément sur plusieurs serveurs virtuels et de collecter des résultats pour chaque environnement.

# 

# ****2.4 Caractéristique de l’Intégration Continue****

Selon Paul Duvall [Duv07] seules quatre caractéristiques sont nécessaires à l’Intégration Continue :

- une connexion à un référentiel de contrôle de version,

- un script de compilation,

- un mécanisme de rétroaction,

- un processus pour intégrer les modifications au code source (manuelles ou serveur d’Intégration Continue).

Ces éléments seuls sont nécessaires à la construction d’un système d’Intégration Continue efficace, qui est expliqué plus en détail dans les sections suivantes.

## ****2.4.1 Compilation du code source****

La compilation du code source est l'une des caractéristiques de base du système d’Intégration Continue. La compilation crée des exécutables binaires à partir de source lisible (pour les développeurs). Lors de l'utilisation des langages dynamiques comme Python ou Ruby la compilation est différente. Les binaires ne sont pas compilés, à la place les développeurs ont la possibilité d'effectuer un checking strict, qui peut être considéré comme de la compilation dans le contexte de ces langues [Duv07].

## ****2.4.2 Tests****

Les tests sont la partie la plus vitale de l’Intégration Continue. Beaucoup considèrent qu’une Intégration Continue sans automatisation du contrôle continu ne peut être un Contrôle Continue [Duv07]. Il est difficile d'avoir confiance dans les changements du code source sans une bonne couverture de test. Les tests peuvent être automatisés en utilisant des outils de tests unitaires tels que JUnit (Java), NUnit (C#), ou d'autres framework[[12]](#footnote-12) de xUnit. Certains de ces frameworks peuvent également générer des rapports machines lisibles, qui peuvent être analysés et utilisés pour générer des représentations graphiques telles que des pages Web ou des tableaux (voir Figure 2.5).

### ****Niveaux de tests****

Le test peut être effectué à différents niveaux (voir Figure 2.6). Le plus bas niveau de test est appelé test unitaire. Une unité est la plus petite partie d’une application testable. Cela correspond une fonction ou une méthode dans une classe. Le but des tests unitaires est de vérifier que les différentes parties du code fonctionnent comme elles le devraient. Ils assurent la stabilité du code en testant chaque unité unitairement. Les régressions seront ainsi remontées très rapidement ce qui permet de manipuler le code source avec confiance. Les tests unitaires sont généralement écrits par le développeur qui a également écrit le code. Une bonne pratique des tests unitaires est de commencer par écrire les tests et d’ensuite les valider par le code. C’est ce qu’on appelle le « Tests Driven Development » (TDD) ou Développement Dirigé par les Tests ».

Le niveau suivant est le test d'intégration. Dans ce contexte d’Intégration nous devons vérifier que les modules individuels du logiciel fonctionnent aussi en tant que groupe.

« Le test d'intégration identifie les problèmes qui se produisent lors de la combinaison d'unités. En utilisant un plan de test exigeant que vous testiez chaque unité et que vous vérifiez la viabilité de chacune d'elles avant de les combiner, vous savez que les erreurs découvertes lors de la combinaison d'unités concernent probablement l'interface entre les unités. Cette méthode réduit le nombre de possibilités à un niveau beaucoup plus simple d'analyse. »

Le troisième niveau teste les API d’un point de vue externe, sans se préoccuper de fonctionnement interne du système. Le test système analyse le flux de retour de l’API en fonction de son flux d’entrée afin  de détecter les défauts à la fois dans les "inter-assemblages" mais également au sein du système dans son ensemble. Cette méthode est appelé « boîte noire ».

Le test fonctionnel est le quatrième niveau de test majeur. Il assure la stabilité de l’application en reproduisant le parcours d’un utilisateur sur le navigateur. Il teste le bon fonctionnement de l’application et remontent les régressions fonctionnelles.

D’autres niveaux de test existent, tel que le test applicatif qui assure la sécurité et la compatibilité, le test d’IHM qui fiabilise l’ergonomie et la visibilité, le test de charge qui veille à la performance et à la robustesse de l’application...

### Exécuter les tests plus rapides en première

Lorsque le logiciel se développe le nombre de tests augmente, ce qui se traduit par une hausse du temps d’exécution des tests lors de la build d’intégration. Si l’ensemble des tests est exécuté en une seule fois cela peut se révéler relativement long et ainsi faire perdre le bénéfice de la rétroaction rapide mise en place avec l’Intégration Continue. Pour faire face à ce syndrome, les tests doivent être classés de plus rapides au plus lents en terme d’exécution. Les tests peuvent aussi être divisés en plusieurs étapes. Les rapports seront envoyés à la fin de chaque étape garantissant un feedback rapide.

Les tests unitaires nécessitent peu de temps à mettre en place et sont les tests les plus rapides à exécuter. L’exécution d’un test unitaire ne doit pas excéder la fraction de seconde. Ces tests sont exécutés de nombreuses fois par jour par développeurs. La rapidité d’exécution est primordiale sinon les tests deviennent une méthodologie de développement à éviter ce qui est contraire au principe de l’Intégration Continue [Duv07]. Les tests unitaires sont donc de bons candidats pour être exécuter au cours de la première étape.

La configuration des tests d'intégration et des tests système nécessitent beaucoup plus de temps que celle des tests unitaires. La mise en place la (des) base(s) de données avec des données de test et de lancement réel de l’application sont des tâches relativement chronophages. L’exécution de ces tests peut être une étape longue. Les tests d’intégration et système peuvent être exécutées dans des étapes ultérieures ou à des intervalles périodiques. Par exemple nous pouvons exécuter une suite complète de tests de plusieurs heures tous les soirs. Nous appelons ce processus Daily Build [McC96].

### ****Ecrire des tests d’échecs****

L’écrire et l’exécution automatique des tests avec l’Intégration Continue diminue la fréquence des logiciels défectueux. Mais l’Intégration Continue n’est pas infaillible [Duv07]. Si un défaut est trouvé, il doit être immédiatement fixé et pour éviter qu’elle se reproduise, un test défectueux devra être implémenté. L'idée sous-jacente est d’améliorer continuellement la qualité de l’application.

## ****2.4.3 Base de données d’intégration****

L’Intégration Continue ne se limite pas à la construction du code source, elle peut également être utilisée dans le développement de base de données. Les bases de données sont des parties intégrantes des applications et ont donc besoin d’être intégré dans le processus d’Intégration Continue. La création d’une base de données fonctionnelle lors d’une build d’intégration nécessite un ensemble de scripts stocké dans le référentiel de contrôle de version. Ces scripts comprennent les définitions des tables, les procédures stockées (fonctions de base de données), le partitionnement... L’exécution des tests fonctionnels nécessite des données « réelles » afin de garantir un environnement proche de la production. Des scripts de « données », exécuter après la création de la base de données complètent la mise en place d’une base de données fonctionnelle.

Par exemple, lorsqu’un développeur ou un administrateur de base de données (DBA) apporte une modification à script de base de données et le valide au niveau du système de contrôle de version (check-in), la build d’intégration (la même que celle utilisée pour l’intégration du code source) construit la nouvelle base de données et lance les tests.

### Test unitaire sur les fonctions de base de données

Pour des raisons de sécurités et de performances les fonctions propres aux bases de données sont stockées et exécutées par elles-mêmes. On les appelle alors « procédures stockées ». Comme pour les méthodes de l’application les procédures doivent être testées et automatisées par des scripts qui seront exécutés lors de la build d’intégration.

## ****2.4.4 Inspection Continue****

Le code source peut être examiné manuellement et/ou automatiquement. La revue de code manuelle peut être effectuée selon deux principes, le « pair programming » (écriture du code en binôme) ou le « code review » (session collective de relecture du code). Elle améliore la qualité algorithmique et syntaxique du code source de l’application et permet aux développeurs d’échanger sur les bonnes pratiques. Pour une revue de code automatique, de nombreux analyseurs de code statique sont disponibles selon les langages de développement. Ces outils analysent les fichiers sources dans le but de souligner les violations de règles prédéfinies propre au langage et d’améliorer la syntaxe de nos lignes de code.

La différence entre la revue de code manuelle faite par des humains et la revue de code automatique faite par des outils d’analyses est double. Exécuter les analyseurs de code statique est peu cher et une fois automatisé ils garantissent une relative propreté au code source. De plus un ordinateur est toujours objectif et ne se lasse pas d’inspecter l’intégralité du code à chaque fois qu’un changement est engagé dans le référentiel de contrôle de version.

Les analyses statiques de code automatisé sont efficientes pour des grandes bases de code. Elles permettent aux développeurs de se concentrer sur les parties importantes. Elles offrent des métriques de qualité sous la forme de rapport d’inspection après chaque exécution. Les revues automatisées ne remplacent pas les revues manuelles, elles permettent de recentrer l’intelligence humaine là où elle est nécessaire.

De nombreux IDE (environnement de développement intégré) intègrent des fonctionnalités d’inspection pour aider les développeurs dès l’écriture du code avec une mise en forme automatisée, la mise en évidence des variables non utilisées, l’utilisation illégale de certain élément… Il est fortement encouragé de les utiliser mais ne remplace en aucun cas les revues de code.

Il existe également des outils qui proposent de réécrire certain bout de code selon une convention particulière, de détecter les blocs de code en double…

### ****Différences entre inspection et test****

Le test (vu précédemment) est dynamique et exécute l’application, ou un fragment de l’application, pour tester les fonctionnalités. L’inspection, quant à elle, analyse le code selon un ensemble de règles prédéfinies. Les deux sont des concepts similaires dans le sens ou aucun ne modifient le code source, ils ne font que remonter les problèmes résidant dans l’application.

### ****Rapport d’inspection****

Les outils d’analyse statique de code fournissent un grand nombre de mesures et de rapports, encore faut-il les interpréter. Depuis quelques décennies des chercheurs étudient ces mesures afin de trouver une corrélation entre les défauts soulignés par les analyses et le code source.

Une des principales mesures étudiée est la complexité cyclomatique, qui quantifie la complexité du code source en comptant le nombre de chemins distincts au travers d'un programme représenté sous la forme d'un [graphe](https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_graphes) [Kan03]. Les analystes suggèrent une complexité de 10 car plus grand est ce nombre, plus important sera le risque de défauts [Wat96]. Le moyen le plus efficace pour réduire la complexité cyclomatique d’une application est d'appliquer la technique de la méthode d'extraction et de distribuer la complexité en petits méthodes, plus faciles à gérer, et donc plus testable, [Duv07].

Outre les problèmes de complexités les rapports d’inspection nous fournissent des mesures sur des problèmes liés à l’architecture de notre application ; « l’afferent coupling » et « l’efferent coupling ». Ces métriques comptent les nombres de dépendances vers, où à partir d’un objet soulignant ainsi les risques de responsabilités ou de dépendances trop fortes. Elles permettent de déterminer le niveau de risque dans le maintien et l’évolutivité du code. Duvall introduit l'utilisation de ces deux valeurs combinées pour calculer une valeur d'instabilité.

La compréhension de ces mesures et de l'analyse des rapports d'inspection peut une réelle plus-value sur le temps investi. Les problèmes de maintenabilité peuvent être repérés dès le début et les risques de défauts peuvent être réduits.

## ****2.4.5 Deployment****

L'un des objectifs de l’Intégration Continue est d'avoir un logiciel prêt et fonctionnel pour le déploiement à tout moment du développement. Toutes les étapes vues précédemment sont des parties du processus de déploiement visant à générer les artefacts de logiciels fournis avec les dernières modifications de code disponibles dans un environnement de test [Duv07]. Une fois l’application packagée il est possible de l'installer automatiquement sur les serveurs de production avec le Déploiement Continue. Pour cela, nous avons six étapes de haut niveau :

* Etiqueter les actifs d'un référentiel,
* produire un environnement propre, exempt d'hypothèses,
* générer et étiqueter une version directement à partir du référentiel et l'installer sur la machine cible,
* effectuer les tests avec succès à tous les niveaux dans un clone de l'environnement de production,
* créer des rapports de rétroaction de build,
* si nécessaire, la release peut être annulée en utilisant des étiquettes dans le système de contrôle de version.

### ****Etiqueter (labéliser) les actifs d’un référentiel****

La création de label au niveau du référentiel facilite l'identification et le suivi des actifs, en définissant clairement un groupe de fichiers comme appartenant à un ensemble. De plus, les étiquettes permettent un suivi historique d'un groupe de fichiers et pas seulement comme des fichiers individuels qui peuvent être sur différentes versions à un moment donné.

### Produire un environnement propre

Afin de réduire les risques comportementaux de l’application il est important de ne pas faire d’hypothèse sur son fonctionnement au travers des différents environnements. Idéalement l’intégralité des environnements doivent être virtualisé et automatisé afin d’être redéployé à chaque déploiement et ainsi garantir un environnement « propre ».

Voici les grandes étapes de la virtualisation et l’automatisation d’un environnement :

* mise en place du système d’exploitation (OS),
* configuration de l’OS (utilisateurs, firewalls, …),
* mise en place des composants du serveur (serveur web, base de données, …),
* configuration du serveur,
* installation des outils tiers (frameworks web, librairies, …),
* configuration des divers softwares.

### ****Etiqueter chaque build****

Afin de créer un identifiant unique pour chaque build il est impératif que le code du référentiel ait été labélisé (cf. Etiqueté (labélisé) les actifs d’un référentiel). Le deuxième point clé est la mise en place d’un schéma de nommage commun aux diverses builds. Labéliser chaque build fournit un moyen simple de suivre efficacement la version du code et son environnement d’exécution. En outre, les défauts, les améliorations et les nouvelles fonctionnalités peuvent être émises contre cette instance de code source.

Notez la différence entre une étiquette de référentiel et une étiquette de build. Les étiquettes de référentiel désignent un ensemble de fichiers non compilés tandis que les étiquettes de build désignent les fichiers binaires en sortie d’une build. Les schémas de nommage sont cependant liés. Par exemple si l’étiquette du référentiel est 4-32 celle de la build sera 4-32.01.

### ****Exécuter tous les tests****

Avant le packaging et le déploiement d’une build l’intégralité des tests doivent être exécutés et validés, des tests unitaires aux tests fonctionnels. Ces tests doivent être exécutés dans un environnement aussi proche que possible que de l’environnement de production.

### ****Créer des build feedback reports****

Les rapports de build fournissent des informations à propos des actions effectués au cours de la build (tests, analyses statiques …), des fichiers modifiés, des issues impactés ainsi que des changements majeurs par rapport à la build précédente. Ces rapports sont généralement composés de :

* un rapport de test qui indique le nombre de tests effectués, le nombre de succès et d’échec et le pourcentage de code couvert par les tests,
* un fichier différentiel qui informe des changements exacts dans le code source,
* un rapport d’analyse statique qui avertit des violations des bonnes pratiques de développement,
* un Changelog qui recueille les notes du développeur (résolution de problèmes, nouvelles fonctionnalités ...).

### ****Capacité à effectuer un rollback****

Quelques fois l’inévitable se produit et le logiciel déployé en production ne fonctionne plus correctement. La capacité à annuler une mise en production est inestimable dans le Déploiement Continu. Lorsque le principe d’étiquetage est correctement utilisé la demande d’une version antérieur de l’application est simple (rollback). Par ailleurs si l’ensemble de la chaine de déploiement est automatisé le temps nécessaire au rollback est minime.

## ****2.4.6 Documentation****

## ****2.4.7 Feedback Continue****

**4. Etude de l’art**

# ****4.1 Logiciel en développement****

# ****4.2 Processus de développement logiciel****

## ****4.2.1 Scrum dans AgileLab****

## 4.2.2 Comment l’Intégration Continue s’intègre-t-elle dedans

## 4.2.3 Equipes de développement

# 4.3 La sécurité à l’esprit

5. Implémentation

# 5.1 Le choix des outils

## 5.1.1 Scaling

## 5.1.2 Le choix du serveur d’Intégration Continue

### Support du langage de programmation

### Support du Source Code Management

### Gestion des builds

### Sécurité

### Autres caractéristiques

### Extensibilité

### ****Installation et configuration****

### Autres questions à examiner

# ****5.2**** Logiciels et outils utilisés

## ****5.2.1 Serveur d’Intégration Continue****

## ****5.2.2 Logiciel de gestion de configuration****

## ****5.2.3 Outils de build****

## ****5.2.4 Analyse de code statique****

## ****5.2.5 Automatisation des tests et de la couverture de code****

## ****5.2.6 Documentation Continue****

## ****5.2.7 Déploiement Continue****

## ****5.2.8 Feedback Continue****

# ****5.3 Architecture****

## ****5.3.1 Serveur d’Intégration Continue****

## ****5.3.2 Logiciel de gestion de configuration****

## ****5.3.3 Réseaux et déploiement****

# ****5.4 Configuration****

## ****5.4.1 Build jobs****

### ****Découpage des jobs****

### ****Etablir des relations****

## ****5.4.2 Reporting****

# ****5.5 Délivrables et artéfacts de build****

Bibliography

[Duv07]   
Paul M. Duvall. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 1st edition, 2007.

[Fow00]  
Martin Fowler. Continuous Integration, 2000. http://www. martinfowler.com/articles/originalContinuousIntegration.html.

[Bec99]  
Cynthia Andres Kent Beck. Extreme programming explained. Addison Wesley, 1999.

1. Possibilité d’augmenter les ressources physiques d’un composant informatiques afin d’augmenter ses performances. [↑](#footnote-ref-1)
2. Possibilité d’ajouter des composants à une entité informatique afin d’augmenter ses performances. [↑](#footnote-ref-2)
3. [Bec99] Cynthia Andres Kent Beck. Extreme programming explained. Addison Wesley, 1999 [↑](#footnote-ref-3)
4. [Fow00] Martin Fowler. Continuous Integration, 2000.

   http://www. martinfowler.com/articles/originalContinuousIntegration.html. [↑](#footnote-ref-4)
5. [Duv07] Paul M. Duvall. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 1st edition, 2007. [↑](#footnote-ref-5)
6. [Fow00] Martin Fowler. Continuous Integration, 2000.

   http://www. martinfowler.com/articles/originalContinuousIntegration.html. [↑](#footnote-ref-6)
7. [Duv07] Paul M. Duvall. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 1st edition, 2007. [↑](#footnote-ref-7)
8. Build jobs : Cela correspond aux différentes tâches du processus de build. [↑](#footnote-ref-8)
9. Open source : Logiciel libre redistribution, d'accès au code source et de création de travaux dérivés. [↑](#footnote-ref-9)
10. [Swa04] Michael Swanson, Automated Continous Integration and the Ambient Orb, <http://blogs.msdn.com/b/mswanson/archive/2004/06/29/169058.aspx>, 2004 [↑](#footnote-ref-10)
11. [Bar03] Paul Barham, Xen and the Art of Virtualization, ACM SIGOPS Operating Systems Review, 2003 [↑](#footnote-ref-11)
12. Framework : ensemble d’outils et de composants logiciels. [↑](#footnote-ref-12)