

UNIVERSITÉ PARIS DAUPHINE

MÉMOIRE DE RECHERCHE

Septembre 2016

Améliorer la qualité des logiciels avec l'Intégration Continue

Gaëtan Meynier

Supervisé par

Khalid BELHAJJAME, Maître de conférence à l'Université Paris Dauphine,
Arthur SZUMOVICZ, Consultant en système d'information à AXA.

Table des matières

1	Introduction	3
1.1	Motivation	4
1.2	Scope	4
1.3	Problèmes de recherche	4
1.4	Methodes	4
1.4.1	Méthodes de recherche	4
1.4.2	Collecte de données	4
1.5	Structure du mémoire	4
2	DevOps	5
2.1	« The Wall of Confusion »	5
2.2	La Culture	5
2.3	« Infrastructure As Code »	5
2.4	Cloud Computing	7
2.4.1	Le Cloud public	8
2.4.2	Le Cloud privé	9
2.4.3	Les avantages du Cloud Computing	9
2.4.4	Quel Cloud choisir ?	10
3	Intégration Continue	11
3.1	Histoire	11
3.2	L'Intégration Continue comme un processus	12
3.2.1	Les bénéfices de l'Intégration Continue	13
3.2.2	Le cycle de travail de l'Intégration Continue	14
3.2.3	Comment l'Intégration Continue s'appuie-t-elle sur d'autres pratiques de développement	15
3.3	L'Intégration Continue est un mélange de personnes et de systèmes	16
3.3.1	Le développeur	16
3.3.2	Le référentiel de contrôle de version	16
3.3.3	Le serveur d'Intégration Continue	17
3.3.4	Les scripts de construction (build)	19
3.3.5	Les mécanismes de feedback	20
3.3.6	Les machines de build d'intégration	20
3.4	Caractéristique de l'Intégration Continue	21

3.4.1	Compilation du code source	21
3.4.2	Tests	21
3.4.3	Qualimétrie	24
3.4.4	Base de données d'intégration	24
3.4.5	Inspection Continue	24
3.5	Déploiement Continu	26
3.5.1	Les bonnes pratiques du Déploiement Continu	26
3.5.2	La dockerisation	29
3.5.3	La « bêta perpétuelle »	31
3.5.4	Le « Zero Downtime Deployment »	33
3.5.5	Le « Flipping Feature »	36
4	Etude de l'art	37
4.1	Logiciel en développement	37
4.2	Processus de développement logiciel	37
4.2.1	Scrum dans l'AgilLab	37
4.2.2	Comment l'Intégration Continue s'intègre-t-elle dans le processus de développement logiciel	37
4.2.3	Les équipes de développement	37
4.3	La sécurité à l'esprit	37
5	Implémentation	38
5.1	Choisir ses outils	39
5.1.1	Scaling	39
5.1.2	Le choix du serveur d'Intégration Continue	39
5.2	Logiciels et outils utilisés	39
5.2.1	Serveur d'Intégration Continue	39
5.2.2	Logiciel de gestion de configuration	39
5.2.3	Outils de build	39
5.2.4	Automatisation des tests et de la couverture de code . . .	39
5.2.5	Documentation Continue	39
5.2.6	Déploiement Continu	39
5.2.7	Feedback Continue	39
5.3	Architecture	39
5.3.1	Serveur d'Intégration Continue	39
5.3.2	Logiciel de gestion de configuration	39
5.3.3	Réseaux et déploiement	39
5.4	Configuration	39
5.4.1	« Build jobs »	39
5.4.2	Reporting	39
5.5	Délivrables et artefacts de build	39

Chapitre 1

Introduction

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 agiles à la livraison et au déploiement du projet.

- 1.1 Motivation
- 1.2 Scope
- 1.3 Problèmes de recherche
- 1.4 Methodes
 - 1.4.1 Méthodes de recherche
 - 1.4.2 Collecte de données
- 1.5 Structure du mémoire

Chapitre 2

DevOps

Le mouvement DevOps, contraction de « Development » et « Operational » tente de répondre à la problématique, certainement aussi ancienne que les DSI, qu'est la frontière qui sépare les développeurs (ceux qui écrivent le code source) et l'exploitation (ceux qui déploient et exploitent).

Les attentes et perspectives autour de Devops sont nombreuses :

- **Des cycles de déploiement plus courts** : les DevOps jouent un rôle clé dans la réduction du temps du cycle de déploiement des logiciels, passant de quelques semaines à seulement quelques heures, permettant une plus grande flexibilité quant aux nouvelles fonctionnalités et changements à apporter au produit initial.
- **Mise à disposition de nouveaux services plus rapidement** : des déploiements fréquents associés à des délais de livraison plus rapides permettent une agilité opérationnelle.
- **Une satisfaction client améliorée** : grâce à des applications ciblées et de qualité, conformes aux retours clients end to end.
- **Des coûts réduits** : L'automatisation permet aux équipes de réaffecter des ressources précieuses à des tâches à plus haute valeur.
- **Conformité et Gouvernance** : Automatisation du tracking et reporting end-to-end sur les phases de livraison/déploiement contenu.

2.1 « The Wall of Confusion »

2.2 La Culture

2.3 « Infrastructure As Code »

<http://blog.octo.com/et-si-devops-nous-emmenait-vers-tdi-test-driven-infrastructure/>

TODAY'S DEVOPS

DEV + OPS

PART DEV



Application Performance

Modern developers use APM tools to decrease latency, have complete visibility into code, databases, caches, queues, and third-party services.



End User Analytics

A great developer understands end users have the best feedback and analytics play an enormous part of understanding users. Developers are constantly monitoring end user latency and checking performance by devices and browsers.



Quality Code

Developers need to ensure their deployments and new releases don't implode or degrade the overall performance.



Code-Level Errors

When you have a large distributed application it is vital to lower MTTR by finding the root cause of errors and exceptions.

PART OPS



Application Availability

The applications need to be up and running and it's Ops responsibility to ensure uptime and SLAs are in order.



Application Performance

Classic Ops generally rely on infrastructure metrics - CPU, memory, network and disk I/O, etc. Modern Ops correlate all of those metrics with application metrics to solve problems 10x faster.



End User Complaints

The goal is to know about and fix problems before end users complain, reduce the number of support tickets, and eliminate false alerts.



Performance Analytics

Automatically generated baselines of all metrics help Ops understand what has changed and where to focus their troubleshooting efforts. Alerts based upon deviation from observed baselines improve alert quality and reduce alert noise.

FIGURE 2.1 – DEV + OPS

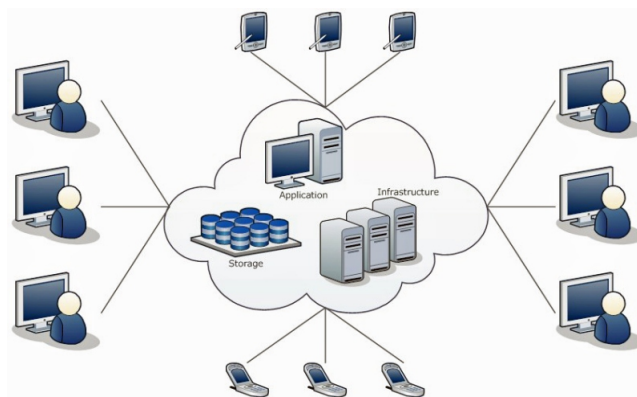


FIGURE 2.2 – Schématisation du Cloud Computing

2.4 Cloud Computing

Définition

Il existe autant de réponse à « Qu'est-ce que le Cloud Computing ? » que de distribution linux. J'ai pour ma part décider de baser mon analyse sur la définition fournie par NIST (National Institute of Standards and Technology) :

« Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable Computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This Cloud model is composed of five essential characteristics, three service models, and four deployment models. »

« Le Cloud Computing est un modèle qui permet un accès omniprésent, pratique et à la demande à un réseau partagé et à un ensemble de ressources informatiques configurables (comme par exemple : des réseaux, des serveurs, du stockage, des applications et des services) qui peuvent être provisionnées et libérées avec un minimum d'administration. »

Pour simplifier, le Cloud Computing est la dématérialisation de l'informatique. C'est le fait de déporter toutes les opérations normalement effectuées sur nos ordinateurs sur des ordinateurs distants, autrement dit sur internet. L'ensemble de ses serveurs constituent le Cloud (Voir Figure 2.2).

Un service à la demande et une flexibilité des ressources.

Avec les applications Cloud, nous adaptons le flux de données à l'évolution de nos besoins : nous ne payons que pour ce que nous consommons. Et nous n'avons plus à nous soucier de manquer de capacité.

Accès universel par le réseau

Le terme de nuage représente bien le concept. Imaginons que nos ordinateurs nous suivent partout, que nous puissions y accéder n'importe où, n'importe quand depuis n'importe quel terminal. Imaginons un accès universel aux ressources stockés sur nos ordinateurs.

Une mise en commun des ressources

L'application de correctifs, la mise à niveau et le test d'applications peuvent occuper nos équipes informatiques plusieurs jours par mois. Grâce aux applications Cloud, cette problématique disparaît.

Cependant ce n'est pas si simple que cela. Il existe trois types de Cloud : le Cloud public (loué par des prestataires), le Cloud privé (créer et gérer par notre entreprise) et le Cloud hybride (un mélange des deux précédents).

2.4.1 Le Cloud public

Le Cloud public appartient à des prestataires de service qui louent l'utilisation de leurs serveurs en facturant leur utilisation à la demande. Nous payons le flux de données échangé. Il dispose principalement de quatre modèles qui correspondent à des besoins bien distincts : le IAAS, le PASS, le SAAS et le STAAT.

Le Cloud IAAS ou Infrastructure As A Service (infrastructure en tant que service)

Le IAAS correspond à une infrastructure informatique (serveurs, réseaux, stockage) hébergé chez le prestataire de service. Nous n'avons pas besoin d'acheter le matériel nécessaire à notre infrastructure, nous nous contentons de la louer mais la gérons comme bon nous semble. Nous installons les serveurs que nous souhaitons utiliser et nous gérons l'ensemble des OS installés sur ces derniers. IAAS est utile si l'on désire créer ses propres plateformes pour ensuite y déployer des applications comme sur des serveurs locaux.

Le Cloud PAAS ou Platform As A Service (plateforme en tant que service)

Le PAAS constitue l'utilisation par le biais d'internet de plateformes sur lesquelles nous pouvons déployer nos propres applications. C'est le niveau d'exé-

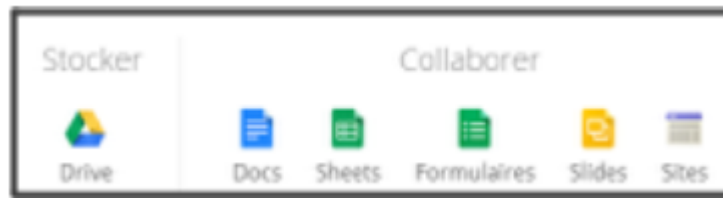


FIGURE 2.3 – Exemple : la suite bureautique proposée par Google

cution des logiciels. Nous louons une plateforme, c'est à dire une machine avec un OS, le tout prêt à l'emploi.

Le Cloud SAAS ou Software As A Service (logiciel en tant que service)

Il s'agit d'utiliser des logiciels directement sur le Cloud, la plupart du temps via un navigateur web. Nous n'avons pas d'achat de licence, pas d'installation sur nos propres terminaux, pas d'entretiens. Le logiciel « tourne » sur des serveurs dans des datacenters (Voir Figure 2.3).

Le STAAT ou Storage As A Service (stockage en tant que service)

Le STAAT permet simplement de stocker nos données.

2.4.2 Le Cloud privé

Le Cloud privé est un Cloud propre à notre entreprise. A l'aide d'outils de virtualisation il est possible de transformer les serveurs existants au sein de l'entreprise. Cela crée un réseau puissant disponible, comme le Cloud public, à tout moment et à tout endroit mais géré entièrement par notre entreprise.

2.4.3 Les avantages du Cloud Computing

Les avantages du Cloud Computing sont nombreux. Si l'accessibilité est l'avantage le plus visible, il faut remarquer que l'utilisation du Cloud Computing permet à l'entreprise d'avoir une grande agilité, allant du démarrage rapide de son activité à un déploiement tout aussi rapide de nouvelles applications et donc un gain de temps et d'argent pour l'entreprise.

De plus avec l'utilisation d'un Cloud public l'entreprise n'a plus d'investissement lourd en capital informatique, plus de dépendance en perte de temps et en entretien des infrastructures. Gains de places et de performances sont au rendez-vous.

Cependant il faut soulever un petit bémol. En effet la contrepartie de tous cela est une dépendance totale envers internet et les prestataires de service. Ainsi qu'un coût croissant à mesure que les besoins de l'entreprises grandissent.

2.4.4 Quel Cloud choisir ?

Si nous disposons déjà de serveurs et que nos besoins sont constants nous opterons pour le Cloud privé. Nous virtualiserons notre infrastructure et profiter de notre propre Cloud. Nos investissements sont ainsi préservés et nous gagnerions en productivité.

En revanche, si nous ne disposons pas de serveurs ou que nous commençons tout juste notre activité et que nous ne souhaitons pas investir dans un parc informatique de serveur complet nous opterons pour un Cloud public. Economie et flexibilité sont au rendez-vous avec les services à la demande.

Enfin si nous possédons des serveurs mais que notre activité fluctue beaucoup, nous demandant parfois des capacités plus fortes nous opterons pour le Cloud hybride. Nous concernerons ainsi notre autonomie avec notre Cloud privé et étendrons notre capacité ponctuellement avec le Cloud public.

Pour conclure, si le Cloud Computing offre une grande disponibilité et une adaptabilité exemplaire, qui en font un service formidable, il n'en reste pas moins dépendant d'internet et les risques en matières de sécurité sont encore mal évalués.

Chapitre 3

Intégration Continue

3.1 Histoire

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 entraîne 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'accroît 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-événement.

« 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 [Duv07].

L'idée d'Intégration Continue a été développée par la communauté Extreme Programming (XP) et a été décrite par Kent Beck dans son livre « Extreme programming explained » [Bec09], 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 [Fow00]. Plus tard, les travaux ont été poursuivis par Paul Duvall qui a écrit tout un livre à propos de l'intégration [Duv07].

Martin Fowler [Fow00] 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.

3.2 L'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 séparée. 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 terminée 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. »

3.2.1 Les bénéfices de l'Intégration Continue

Selon Paul Duvall [Duv07], l'intégration logiciels n'est pas un problème dans les petites équipes, un à deux développeurs, mais quand le nombre de collaborateurs 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 fournit 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.

3.2.2 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 3.1). Il est basé sur les idées présentées par Martin Fowler [Fow00] 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.

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, refactorisation 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

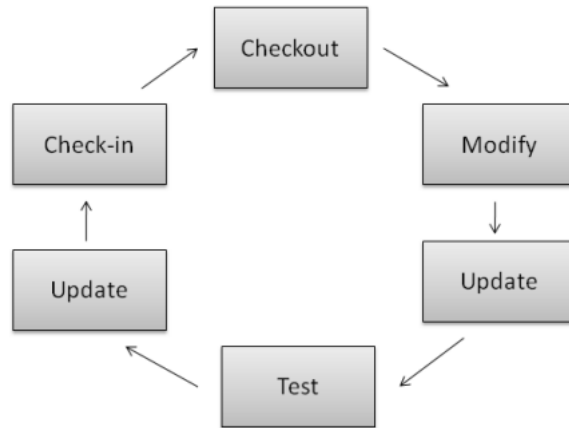


FIGURE 3.1 – Cycle de travail de l'Intégration Continue

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.

3.2.3 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 :

1. effectuer régulièrement des commits,
2. ne pas commit des codes buggés,
3. régler les problèmes et builder immédiatement,
4. écrire des tests de développement automatisés,
5. tous les tests et métriques doivent être valides,
6. exécuter en local ses builds,
7. éviter de travailler avec du code buggés.

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.

3.3 L'Intégration Continue est un mélange de personnes et de systèmes

3.3.1 Le développeur

!!!!!!!!!!!!!!!!!!!! You build it, you run it!!!!!!!!!!!!!!!!!!!!

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.

3.3.2 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

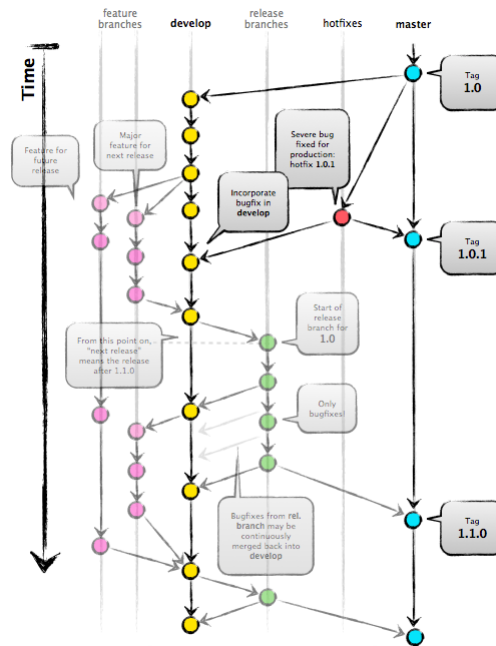


FIGURE 3.2 – Ligne de temps d'un référentiel de contrôle de version

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 « trunk » 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.

3.3.3 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 plu-

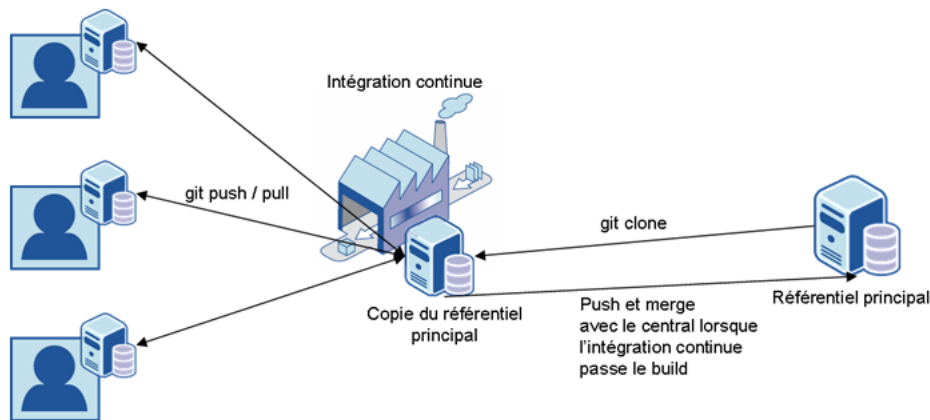


FIGURE 3.3 – Organisation d’un serveur d’Intégration Continue

seurs 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 3.3. 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 apportées 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 »¹ et afficher leurs résultats en temps réel configurés (Voir Figure 3.4). 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 effectuées 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 connaître l’état des builds d’Intégration.

Toutes les fonctionnalités d’un serveur d’Intégration Continue ne sont pas

1. Build jobs : cela correspond aux différentes tâches du processus de build.

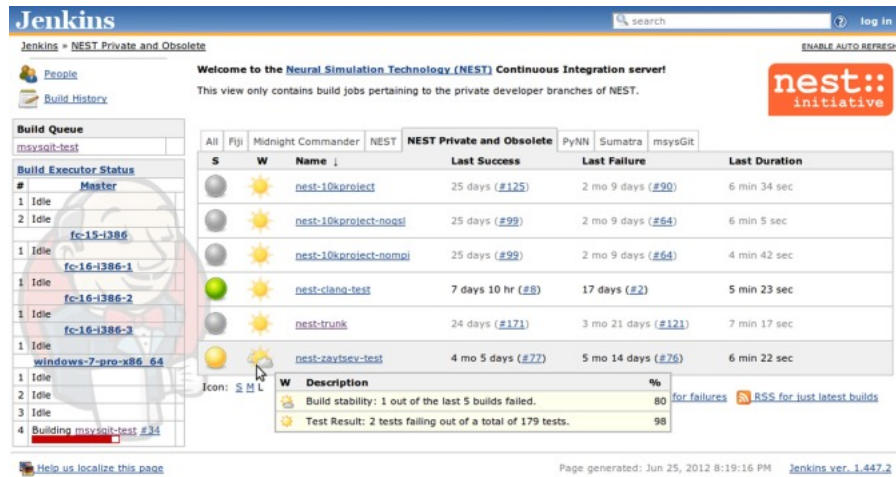


FIGURE 3.4 – Tableau de bord d'un serveur d'Intégration Continue Jenkins

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 source² 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 ».

3.3.4 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

2. Open source : logiciel libre redistribution, d'accès au code source et de création de travaux dérivés.

(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].

3.3.5 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 différents 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 continu » 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 [Swa04].

3.3.6 Les machines de build d'intégration

Un serveur d'Intégration Continue a 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 bibliothèques 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 augmenter la rapidité de la rétroaction il peut être nécessaire de paralléliser les tâches sur plusieurs machines (scalabilité horizontale). 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 apporte la réponse à ce problème. En utilisant une infrastructure de virtualisation bien établie, 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 offrent 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 [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.

3.4 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.

3.4.1 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].

3.4.2 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 framework³ 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 3.5).

3. Framework : ensemble d'outils et de composants logiciels.

```
testchoice (__main__.TestSequenceFunctions) ... ok
testsample (__main__.TestSequenceFunctions) ... ok
testshuffle (__main__.TestSequenceFunctions) ... ok

-----
Ran 3 tests in 0.110s

OK
```

FIGURE 3.5 – xUnit output

Niveaux de tests

Le test peut être effectué à différents niveaux 3.6. 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. » Microsoft [Mic16].

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 sta-

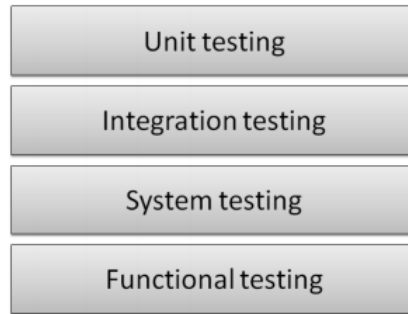


FIGURE 3.6 – Les principaux niveaux de test

bilité 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 leur de la build d'intégration. Si l'ensemble des tests est exécuté en une seule fois cela peut prendre du temps et ainsi faire perdre le bénéfice de rétroaction rapide. 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 infallible [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.

3.4.3 Qualimétrie

3.4.4 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 intégrées 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, 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.

3.4.5 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 efficaces 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 où aucun ne modifie 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 [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 petites 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'afférent coupling » et « l'efférent 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.

$$Instability = \frac{EfférentCoupling}{EfférentCoupling + AfférentCoupling}$$

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.

3.5 Déploiement Continu

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 Continu. Le Déploiement Continu constitue l'ensemble des outils, méthodologies et bonnes pratiques permettant le déploiement de l'application sur un environnement de production en un clic.

3.5.1 Les bonnes pratiques du Déploiement Continu

Pour cela Paul Duvall définit six bonnes pratiques de haut niveau afin de réaliser convenablement un bon Déploiement Continu :

- étiqueter 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,

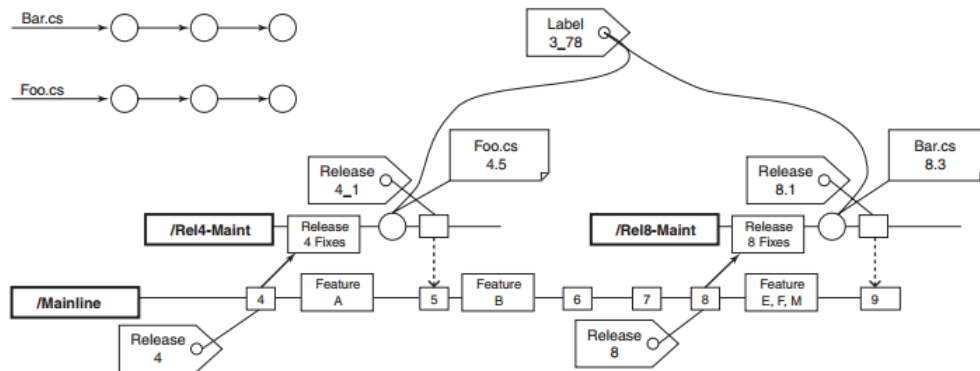


FIGURE 3.7 – Exemple de référentiel labélisé contenant Foo.cs et Bar.cs

- 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 3.7.

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 rapports de build

Les rapports de build fournissent des informations à propos des actions effectuées 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 antérieure de l'application est simple (rollback). Par ailleurs si l'ensemble de la chaîne de déploiement est automatisé le temps nécessaire au rollback est minime.

3.5.2 La dockerisation

Le site officiel de Docker définit son outils comme tel :

« Docker est un outil qui peut emballer une application et ses dépendances dans un conteneur virtuel, qui pourra être exécuté sur n'importe quel serveur Linux »

Le mot clé de cette définition est conteneur. Qu'est ce qu'un conteneur ?

Le conteneur

Le conteneur n'est pas une notion nouvelle, ce n'est pas une notion inventé par Docker. Linux a un système de conteneur qui s'appelle LXC (LinuX Container) qui permet de gérer cet emballage.

Un conteneur est globalement une sorte de boîte, un peu comme une machine virtuelle, qui va être complètement isoler du système d'exploitation dans laquelle nous allons pouvoir installer toutes les bibliothèques dont a besoin notre application pour fonctionner ainsi que notre application. Le conteneur étant complètement isolé du reste nous allons pouvoir la distribuer un peu partout, indépendamment du système d'exploitation.

Les avantages

Actuellement nos applications ont besoin de plus en plus de technologies pour fonctionner. Prenons un cas concret avec une application PHP.

Une application PHP a besoin d'une certaine version de PHP, d'images magiques (pour la conversion d'images), d'une base de données, d'un système de cache, d'un serveur web (Nginx ou Apache), d'un indexeur (Elasticsearch)... Pour au final nous retrouver avec une application nécessitant de nombreuses dépendances.

Le problème est que lorsque nous travaillons avec un administrateur système ou un hébergeur tier le déploiement de notre application peut rapidement s'avérer fastidieuse. Ces derniers devront installer sur chaque serveur de déploiement

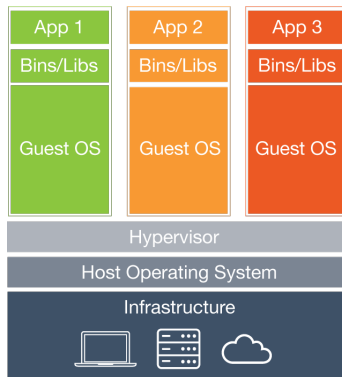


FIGURE 3.8 – Schéma de fonctionnement d’une machine virtuelle

les dépendances requises pour le bon fonctionnement de notre application. En tant que développeur nous ne sommes pas forcément sensible à toutes les problématiques liées aux serveurs ce qui crée un climat hostile entre les développeurs et les administrateurs systèmes.

Le gros avantage du conteneur est que ce dernier va pouvoir être livré avec l’intégralité des dépendances liées à notre application.

Conteneur VS Machine virtuelle (VM)

Ce que nous venons de décrire est exactement le principe de fonctionnement d’une VM. Pour comprendre la différence entre ces deux technologies voici un petit schéma issu du site officiel de Docker (Voir Figure 3.8).

Voici la structure actuelle d’un serveur que l’on a avec des machines virtuelles. Nous nous retrouvons avec une infrastructure et par dessus un système d’exploitation qui va ensuite faire fonctionner diverses machines virtuelles. Pour schématiser nous avons un ordinateur dans un ordinateur. Le problème de cette structure est la redondance des système d’exploitation. Si notre serveur héberge x machine(s) virtuelle(s), nous aurons x système(s) d’exploitation(s) installer sur notre serveur, ce qui consomme beaucoup de ressources mémoires et processeurs.

Le système de conteneur nous permet de nous absoudre de cette contrainte et de faire fonctionner nos application directement sur le système d’exploitation du serveur hôte. Nous avons ainsi plus besoin de virtualiser les différents systèmes d’exploitations de nos applications. Ce qui va alléger notre structure serveur (Voir Figure 3.9).

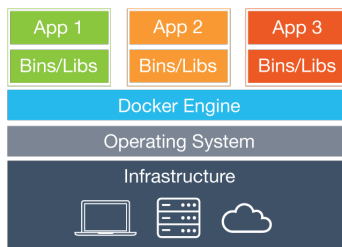


FIGURE 3.9 – Schéma de fonctionnement d'un conteneur

Les avantages

Les avantages de l'utilisation de conteneur sont nombreux. Nous allons nous intéresser aux quatre principaux, les gains en performance, la probabilité des conteneurs, leur scalabilité et les facilités de déploiement.

Un système d'exploitation s'appuie sur de nombreux processus coûteux en mémoire et en CPU (temps de calcul d'un ordinateur). La réduction du nombre de systèmes d'exploitation à un unique OS tournant sur notre serveur hôte augmente considérablement les performances de ce dernier.

Le système de boîte isolée rend la technologie des conteneurs beaucoup plus portable. Si nous voulons transférer une machine virtuelle d'un serveur A à un serveur B nous devons effectuer un snapshot intégral de notre 1 VM et le transférer. Le problème est qu'une machine virtuelle pèse lourd. Du coup le transfert d'une VM prend beaucoup de temps.

Le troisième avantage du conteneur est qu'il est beaucoup plus facilement scalable. Nous allons pouvoir bouger nos petites boîtes très rapidement d'un serveur à l'autre, et même les faire évoluer en terme de performance.

Le dernier avantage que nous allons aborder est lié à la faculté de portable et scalable de nos conteneurs est le déploiement de ces derniers. Déployer un conteneur va être très simple. Vu qu'un conteneur est léger et qu'il embarque l'intégralité des dépendances nécessaires au bon fonctionnement de notre application nous allons pouvoir envoyer à notre administrateur système ou notre hébergeur l'intégralité de notre environnement de production.

3.5.3 La « bêta perpétuelle »

Le principe de la « bêta perpétuelle » a été introduit pour la première fois par Tim O'Reilly dans le manifeste du Web 2.0, qui expose le principe selon lequel :

« Les utilisateurs doivent être considérés comme des co-développeurs, en suivant les principes Open Source (...). Avec le Web 2.0, ce principe évolue vers une position plus radicale, la bêta perpétuelle, dans laquelle le produit est développé de manière ouverte, avec de nouvelles fonctionnalités offertes sur une base mensuelle, hebdomadaire, ou même quotidienne. »

Le terme « bêta perpétuelle » désigne le fait que notre application n'est jamais finalisée. Elle s'absout des contraintes habituelles liées au cycle de développement en « release » au profit d'une livraison en continu des nouvelles fonctionnalités.

Release early, release often

Derrière ce nouveau concept se cache un concept déjà bien en place chez les agilistes (du point de vue de l'itération courte) et dans le monde de l'Open Source (du point de vue de la récolte continue du feedback), le « Release early, release often », traduit en français par « Publiez tôt, publiez souvent ». Cette pratique a été décrite par Eric Steven Raymond dans “La cathédrale et le bazar” où il formulait explicitement :

« Publiez tôt. Publiez souvent. Et écoutez vos clients »

Cette méthodologie vise à réduire les temps de développement et améliorer l'implication de l'utilisateur dans la conception du logiciel afin de créer un produit correspondant à ces attentes et ainsi éviter la création d'un logiciel que personne n'utilisera.

Les services en ligne (Software As A Service)

Le concept de « bêta perpétuelle » a été rendu possible grâce au Cloud Computing et plus particulièrement à la généralisation du service en ligne ou « Software As A Service ». L'hébergement de l'application par l'éditeur permet d'absoudre ce dernier au traditionnel cycle de déploiement d'un logiciel et de ne gérer qu'une seule version de son application. Les services en ligne sont continuellement mise à jour sans pour autant en informer l'utilisateur. Les nouvelles fonctionnalités, découverte au fur et à mesure par l'utilisateur, permettent un apprentissage progressif des nouveautés applicatives.

La « Customer driven roadmap »

L'hébergement de l'application sur serveur offre à l'éditeur une maîtrise totale de sa plateforme de production. Il peut ainsi mettre en place des sondes analytiques afin de récolter des informations sur l'usage de notre application et l'accueil réservé à nos nouvelles fonctionnalités par l'utilisateur.

Les pré-requis

La mise en place d'une stratégie de « bêta perpétuelle » requiert certains pré-requis pour en garantir le succès :

- une intégration continu,
- une livraison continue,
- un déploiement continu,
- une stratégie de type « One click deployment / Rollback » pour une restauration rapide de l'application au dernier état stable.

Conclusion

Le concept de la « bêta perpétuelle » est présent chez de nombreux géants du Web tel que Google, Facebook, Amazon, Twitter, Flickr... Peu en font mention dû à la mauvaise image du terme « bêta », qui pour la conscience collective, se réfère à un produit non fini et peu fiable. Prenons en exemple Gmail, la boîte aux lettres mails développé par Google, qui jusqu'en 2009 intégrait la mention « bêta » dans son logo. De petites fonctionnalités unitaires sont fréquemment proposés aux utilisateurs. En fonction de leur niveau d'adoption Google les intègre ou non à la version standard de son service.

3.5.4 Le « Zero Downtime Deployment »

Nous avons vu précédemment comment améliorer le « Time To Market » tout en garantissant la qualité des développements. L'étape suivante est de garantir que ces déploiements de plus en plus fréquents n'impactent pas la disponibilité de l'application. Le « Zero Downtime Deployment » (ZDD) offre une approche qui permet de déployer une nouvelle version issus de la build sans interruption de service. Pour cela le ZDD repose sur trois grands patterns, le « Blue/Green Deployment », le « Canary Release » et le « Dark Launch ».

Les patterns

Le « Blue/Green Deployment » est le pattern de base du ZDD. L'application, hébergée sur au moins deux chaînes applicatives, déploie sa version N+1 sur une des chaînes (ou plusieurs) tandis que le service en ligne est maintenu en version N sur les autres chaînes applicatives (Voir Figure 3.10).

Une fois le déploiement effectué, notre nouvelle version doit être testée par une population restreinte d'utilisateurs. Le « Canary Release » confronte la version N+1 à une niche d'utilisateurs cibles tandis que la version N reste accessible à la majorité des utilisateurs (Voir Figure 3.11).

La dernière étape du ZDD est le test de charge de notre nouvelle application. Le « Dark Launch » stimule progressivement le trafic généré par l'utilisateur

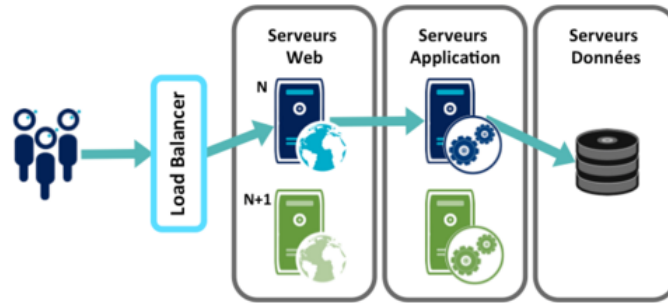


FIGURE 3.10 – Schéma de pattern « Blue/Green Deployment »

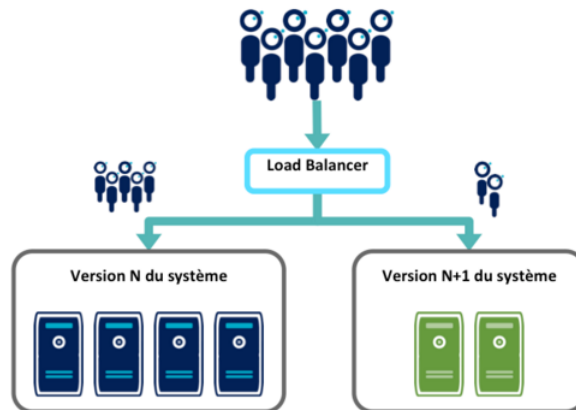


FIGURE 3.11 – « Schéma de pattern Canary Release »

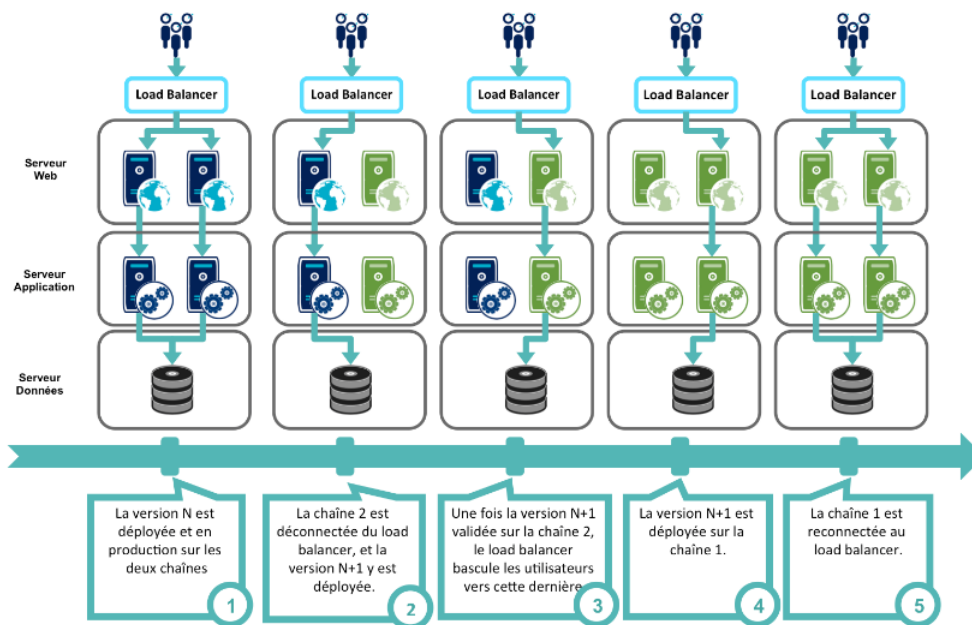


FIGURE 3.12 – « Schéma de la mise en oeuvre du Zero Downtime Deployment »

afin de valider les performances et la scalabilité de notre plateforme. La stimulation progressive du trafic permet de préparer et d'optimiser au mieux notre plateforme afin que la mise en production finale de notre application se déroule sans problème.

La mise en oeuvre

Le point clé du ZDD est d'associer le mécanisme de répartition de charge (Load Balancer) à la cinématique de déploiement. Cette mise en oeuvre s'articule autour de trois grands temps (Voir Figure 3.12) :

- le load balancer déconnecte de une à x chaînes de production sur laquelle est déployée la version N+1,
- une fois cette migration effective, le load balancer dirige une partie de ses utilisateurs vers cette nouvelle chaîne contenant la version N+1,
- si la version N+1 est validée, la chaîne de production en version N est déconnectée, mise à jour et reconnectée à la répartition de charge.

3.5.5 Le « Flipping Feature »

Les organisations appuyant leur service de livraison de code sur le Déploiement Continu se sont rapidement confronté à une limite ; comment commiter fréquemment sur le référentiel de source tout en garantissant la stabilité de l'application, toujours prête pour la production dans le cadre de fonctionnalités longues et complexes ? Deux possibilités s'offrent au développeur, créer une branche parallèle au niveau du référentiel de code et la maintenir ou créer une branche au niveau du code, le « Flipping Feature ».

Le « Flipping Feature » permet d'activer ou de désactiver des fonctionnalités de l'application au chaud, ie directement en production sans relivraison de code. Le mécanisme est très simple, il suffit de conditionner l'exécution du code de la fonctionnalité avec un « if » qui ira regarder dans un fichier de configuration ou d'interroger une base de données afin de savoir si la fonctionnalité est activée ou non.

Ce pattern offre quatre axes d'amélioration aux applications :

- **sécuriser le déploiement** : l'activation et la désactivation d'une fonctionnalité à chaud permet d'éviter un processus de rollback,
- **expérimenter pour améliorer le produit** : il est possible d'étendre le « Flipping Feature » à une sous population d'utilisateur, et en fonction de leur retour d'expérience activé la fonctionnalité ou non à l'intégralité des utilisateurs,
- **offrir un produit customisable** : il peut s'avérer intéressant à un moment du développement de laisser le choix à l'utilisateur entre deux modes de fonctionnement pour une même fonctionnalité.

Chapitre 4

Etude de l'art

4.1 Logiciel en développement

4.2 Processus de développement logiciel

4.2.1 Scrum dans l'AgilLab

4.2.2 Comment l'Intégration Continue s'intègre-t-elle dans le processus de développement logiciel

4.2.3 Les équipes de développement

4.3 La sécurité à l'esprit

Chapitre 5

Implémentation

5.1 Choisir ses 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 Automatisation des tests et de la couverture de code

5.2.5 Documentation Continue

5.2.6 Déploiement Continu

5.2.7 Feedback Continue

5.3 Architecture 39

5.3.1 Serveur d'Intégration Continue

5.3.2 Logiciel de gestion de configuration

5.3.3 Réseaux et déploiement

5.4 Configuration

Table des figures

2.1	DEV + OPS	6
2.2	Schématisation du Cloud Computing	7
2.3	Exemple : la suite bureautique proposée par Google	9
3.1	Cycle de travail de l'Intégration Continue	15
3.2	Ligne de temps d'un référentiel de contrôle de version	17
3.3	Organisation d'un serveur d'Intégration Continue	18
3.4	Tableau de bord d'un serveur d'Intégration Continue Jenkins . .	19
3.5	xUnit output	22
3.6	Les principaux niveaux de test	23
3.7	Exemple de référentiel labélisé contenant Foo.cs et Bar.cs	27
3.8	Schéma de fonctionnement d'une machine virtuelle	30
3.9	Schéma de fonctionnement d'un conteneur	31
3.10	Schéma de pattern « Blue/Green Deployment »	34
3.11	« Schéma de pattern Canary Release »	34
3.12	« Schéma de la mise en oeuvre du Zero Downtime Deployment »	35

Bibliographie

- [Bar03] Paul Barham. *Xen and the Art of Virtualization*. ACM SIGOPS Operating Systems Review, 164-177, 2003.
- [Bec09] Cynthia Andres Kent Beck. *Extreme programming explained*. Addison Wesley, 2009.
- [Duv07] Paul M. Duvall. *Continuous Integration : Improving Software Quality and Reducing Risk*. Addison Wesley, 2007.
- [Fow00] Martin Fowler. Continuous integration, 2000. <http://www.martinfowler.com/articles/originalContinuousIntegration.html>.
- [Fow06] Martin Fowler. Continuous integration, 2006. <http://www.martinfowler.com/articles/continuousIntegration.html>.
- [Kan03] Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2003.
- [McC96] Steve McConnell. *IEEE Software*. 1996.
- [Mic16] Microsoft. Test d'intégration, 2016. [https://msdn.microsoft.com/fr-fr/library/aa292128\(v=vs.71\).aspx](https://msdn.microsoft.com/fr-fr/library/aa292128(v=vs.71).aspx).
- [Swa04] Michael Swanson. Automated continous in-tegration and the ambient orb, 2004. <http://blogs.msdn.com/b/mswanson/archive/2004/06/29/169058.aspx>.
- [Wat96] McCabe Watson. *Structured Testing : A Testing Methodology Using the Cyclomatic Complexity Metric*. NIST Special Publication, 1996.