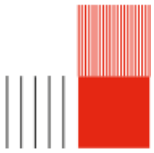
Rapport Thématique

Automatisation et DevOps dans l'industrie spatiale : Focus sur GitLab-CI

Robin Marin–Muller
Maître d'apprentissage : Gabriel Brusq
Tuteur INSA : Thierry Monteil

Table des matières



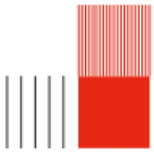
Chapitre 1

Introduction

Ce rapport intitulé "Rapport Thématique" à plusieurs objectifs pédagogiques. Dans un premier, temps il est l'occasion de m'entraîner à faire des recherches sur un sujet et en faire une synthèse. D'autre part, c'est un excellent exercice de rédaction, compétence cruciale dans bien des situations. Enfin, il est l'occasion de réaliser un état de l'art, exercice que je n'avais jamais eu l'occasion d'effectuer.

Le développement logiciel a toujours été au cœur de l'innovation technologique dans l'industrie spatiale et inversement l'industrie spatiale à joué un rôle crucial dans l'avancement des techniques logicielles et matérielles liées à l'informatique. Les missions et programmes spatiaux dépendent intrinsèquement de logiciels robustes et performants capables de gérer des données complexes efficacement. Dans ce contexte, il a fallu au cours du temps inventer et perfectionner les pratiques et méthodologies de développement afin de soutenir les exigences des systèmes critiques employés. Nous verrons plus en détail la méthodologie DevOps, qui s'impose aujourd'hui comme une révolution méthodologique permettant de livrer des logiciels très fiables et rapidement.

Ce rapport a pour objectif de dresser un portait complet de ces problématiques tout en étant abordable. Pour ce faire, il a été divisé en plusieurs axes, dans un premier temps, nous débuterons par une perspective historique du développement logiciel. Par la suite, nous étudierons en détail le cas de GitLab-CI, en s'appuyant sur des exemples pratiques et notamment de son utilisation au sein du CNES. Nous pouvons dès lors dégager trois grandes problématiques : "D'où venons-nous ?", "Ou sommes nous ?" Et "Vers ou allons nous ?".



Chapitre 2

Perspective historique du développement logiciel

Afin de comprendre la suite de ce rapport, une vue historique de la problématique s'impose comme nécessaire afin de retracer les différentes étapes et changement de paradigme du développement logiciel. Elle permettra de répondre à la question "D'où venons nous ?" Dans le triptyque qui compose cet état de l'art.

2.1 Brève Introduction à l'histoire du développement logiciel

L'histoire du développement logiciel, telle que l'on l'entend aujourd'hui commence lors des efforts de guerre de la Seconde Guerre mondiale, période de progrès technique intense ou le besoin urgent d'effectuer des calculs balistiques et de déchiffrer des codes à conduit des avancées importantes dans la conception des premiers ordinateurs et des formes de programmation associées. Alan Turing est souvent présenté comme un précurseur dans le domaine de l'informatique théorique pour son rôle déterminant dans la conception et la théorisation de la machine de Turing, concept clé de l'ordinateur programmable. Les travaux de Turing on permis de poser les bases des premiers ordinateurs modernes et la compréhension théorique de ce que signifie "programmer".

Dans le même temps, l'ENIAC (Electronic Numerical Integrator and Computer [1]), souvent considéré comme le premier ordinateur électronique généraliste et surtout Turing Complet. Il a initialement été développé pour calculer des tables de tir d'artillerie pour l'armée américaine, mais à permis d'inspirer des personnes comme John Von Neumann, qui a théorisé l'architecture Von Neuman [2] ou la programmabilité joue un rôle essentiel.

L'ENIAC ne se programmait pas dans le sens moderne du terme car il n'offrait pas de possibilité d'enregister de programme (du moins dans ses débuts). La programmation de l'ENIAC se faisait en reconfigurant manuellement un réseau complexe de câbles, un processus long et laborieux qui a néanmoins permis une avancée majeure de l'informatique moderne.

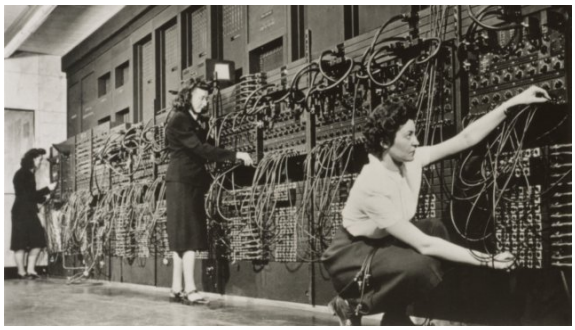


FIGURE 2.1 – L'ENIAC en fonctionnement

Ces progrès ont été le catalyseur de la création de langages de programmation plus abstraits et haut niveau dans les années 1950. Par exemple FORTRAN[3] (1957), a été développé par IBM et est devenu l'un des premiers langages de programmation largement adoptés, conçu pour être compréhensible par les humains sans nécessiter une connaissance approfondie de l'architecture matérielle sous-jacente.

Ces nombreuses avancées ont ouvert la voie à une ère de développement logiciel où la programmation devenait de plus en plus accessible et éloignée des détails de l'implémentation matérielle, permettant ainsi une explosion de l'innovation logicielle dans les décennies suivantes.

2.2 L'aube du développement logiciel dans l'industrie spatiale

Les débuts de développement logiciel, comme on l'entend aujourd'hui dans l'industrie spatiale, sont intrinsèquement liés à la course à l'espace se déroulant durant la Guerre Froide. La complexité grandissante des missions rendait nécessaire l'adoption de méthodes numériques, les logiciels et leurs méthodes de développement firent leur apparition à ce moment-là.

Il était absurde de parler de logiciel sans parler de l'hardware qui l'accompagne, en effet avant l'ère des langages permettant un niveau d'abstraction élevé, la manière dont on développait les logiciels étaient absolument dépendante du matériel sur lequel il devait être implémenté.

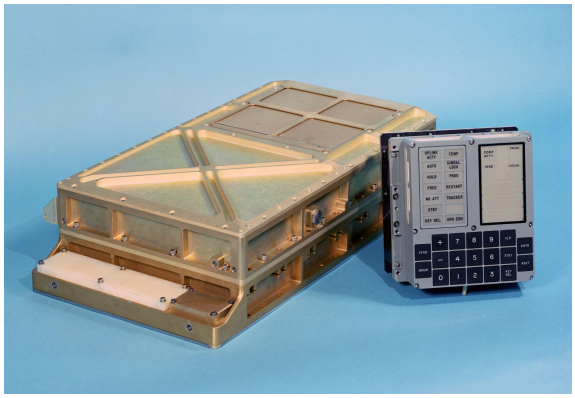


FIGURE 2.2 – Vue de l'Apollo Guidance Computer (AGC)

Nous pouvons prendre comme exemple les missions Apollo qui représentent un tournant décisif dans l'histoire du développement logiciel (entre d'innombrables d'autre avancées). La programmation se faisait principalement en langage d'assemblage étroitement couplé à l'architecture matérielle associée. Margaret Hamilton, à la tête de la division Software Engineering du MIT et pour le compte de la NASA, a développé le logiciel embarqué pour l'Apollo Guidance Computer. Le ; ou plutôt les logiciels puisqu'ils sont séparés en deux parties (une pour le module lunaire et une pour le module de commandement) ; devaient fonctionner sur un système extrêmement critique et très limité en ressources. Les méthodes de développement employées et mises au pont lors de cette mission étaient déjà en accord avec les pratiques modernes notamment au niveau des tests rigoureux et des délais serrés à tenir.

FIGURE 2.3 – Margaret Hamilton (Responsable du logiciel du programme Apollo au MIT) aux côtés du code source embarqué sur l'AGC

Fait insolite, le code source de l'AGC d'Apollo 11 est disponible publiquement sur ce référentiel : https://github.com/chrislgarry/Apollo-11

2.3 L'ère des méthodes traditionnelles

Durant les années 1970, les logiciels devenaient de plus en plus volumineux et complexes, rendant leur gestion très compliquée. C'est pour donner une réponse à ces problématiques que Winston W. Royce publia une méthode appelée méthode de développement en cascade ("Waterfall"). C'est un modèle basé sur la complétion d'étapes séquentielles, de la conception à la maintenance. Cette méthode était particulièrement adaptée aux projets d'envergure tels que ceux engagés par l'industrie aéronautique et spatiale. En effet, ces projets et missions avaient des exigences et spécifications très détaillées et peu susceptibles d'être modifié "à la volée" au cours du projet. Ainsi, la rigidité apportée par la méthodologie en cascade offrait un cadre adapté, mais ayant toutefois des limites quant à l'adoption de ce système par des équipes voulant faire évoluer les spécifications du projet en cours de route.

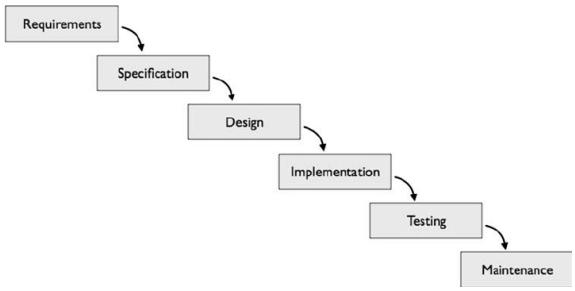


FIGURE 2.4 – Méthodologie de gestion de projet en cascade

2.4 Transition vers des méthodes agiles

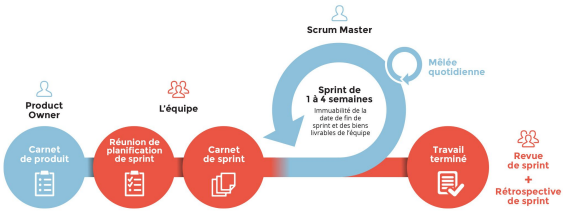


FIGURE 2.5 – Méthodologie de gestion de projet agile type SCRUM

La méthodologie Agile a émergé comme une réponse aux limitations des méthodes traditionnelles (cf. Modèle Cascade). Elle a été développée dans les années 2000 par un consortium d'expert qui ont rassemblé toutes les pratiques allant dans le sens de la philosophie Agile au sein d'un manifeste (https://agilemanifesto.org/). Elle peut être décrite en deux mots : adaptabilité et collaboration. La méthode Agile introduit des cycles de développement itératifs qui permettent des mises à jour régulière en fonction de feedback des utilisateurs. On peut citer les méthodes Scrum ou eXtreme Programming (XP) qui appliquent toutes deux la méthodologie Agile. L'agilité a apporté une nouvelle dynamique au développement logiciel, encouragée par une culture de collaboration et d'amélioration continue.

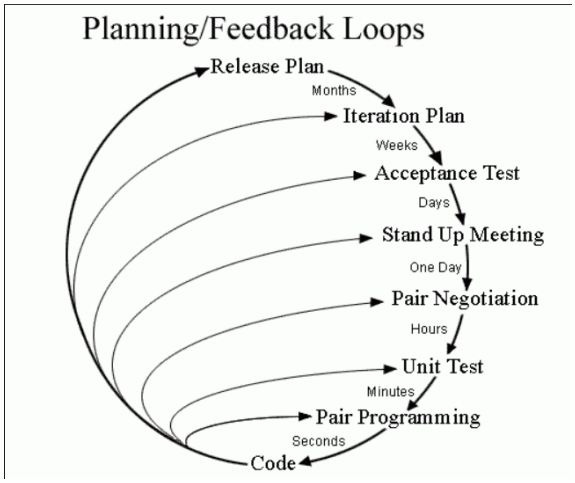
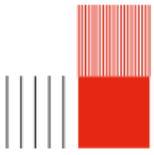


FIGURE 2.6 – Méthodologie XP

2.5 Réflexion sur l'historique du développement logiciel

L'évolution du développement logiciel montre une tendance évidente à l'accélération, l'automatisation des processus de développement et une abstraction toujours plus forte du matériel.



Chapitre 3

Principes et pratiques du DevOps

3.1 Introduction aux principes fondamentaux du DevOps

CI (Continuous Integration)

La CI est une pratique de développement logiciel où les développeurs intègrent régulièrement leur code au sein d'un référentiel partagé, et où les builds et les tests automatisés sont exécutés à chaque intégration, permettant ainsi une détection précoce des erreurs.

CD (Continuous Deployment)

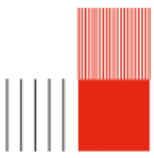
Le CD est une extension de la CI, où le code prêt à être déployé est automatiquement livré à un environnement de production ou de pré-production, réduisant ainsi le temps et les risques associés au déploiement manuel. Si le déploiement est également automatisé, on parle alors de Continuous Deployment.

Le DevOps est une philosophie relativement récente, qui a pris de l'ampleur dans les années 2010 (d'où "relativement"). En un mot, elle vise à réduire d'avantage le cycle de vie des systèmes de développement tout en offrant une meilleure qualité et en favorisant une collaboration étroite entres les équipes de Développement (Dev) et d'Opération (Ops).

Un aspect crucial du DevOps se caractérise par l'utilisation de l'intégration continue (CI) et du déploiement continu (CD). Mis ensemble, ces mécanismes permettent des mises à jour rapides et fiables des application en favorisant un cycle d'amélioration continue s'adaptant rapidement aux changements.

3.2 Les pratiques du DevOps dans l'industrie logicielle classique

Dans l'industrie logicielle, le DevOps se manifeste par des pratiques telles que l'intégration continue, où le code est régulièrement fusionné dans un dépôt partagé, ou le déploiement continu, qui assure une mise en production rapide et fiable. Ces pratiques sont appuyées par une automatisation poussée, des tests systématiques et une gestion dynamique des infrastructures (load balancing par exemple).



3.3 Défis et adaptation du DevOps dans les systèmes embarqués spatiaux

L'application stricte du DevOps aux systèmes embarqué, et notamment dans le domaine du spatial, introduit une complexité supplémentaire et nécessite une adaptation conséquente des principes. En effet, il est difficile d'imaginer effectuer plusieurs dizaines de déploiements par jour comme c'est le cas dans l'industrie classique. Les cycles de développement sont beaucoup plus longs et nécessitent des processus des tests adaptés aux contraintes matérielles (OBC réel ou représentatif). La sécurité et la durabilité sont des paramètres qui sont beaucoup plus prévalent et propre à l'industrie spatiale.

3.3.1 Adaptation des principes du DevOps à l'industrie spatiale

Pour intégrer les principes du DevOps dans ce contexte exigeant, il est souvent nécessaire d'adopter des cycles d'intégration continue plus élaborés et des environnements de test qui reproduisent les conditions spatiales de manière précise. Cela implique l'utilisation de simulations avancées et de bancs d'essai représentatifs du matériel final (serveurs de cartes). La réussite du DevOps dans l'industrie spatiale repose sur une planification minutieuse, une automatisation poussée et une certaine souplesse quant à l'impossibilité d'appliquer certains principes (notamment les difficultés d'intégrer le CD à la chaîne DevOps).

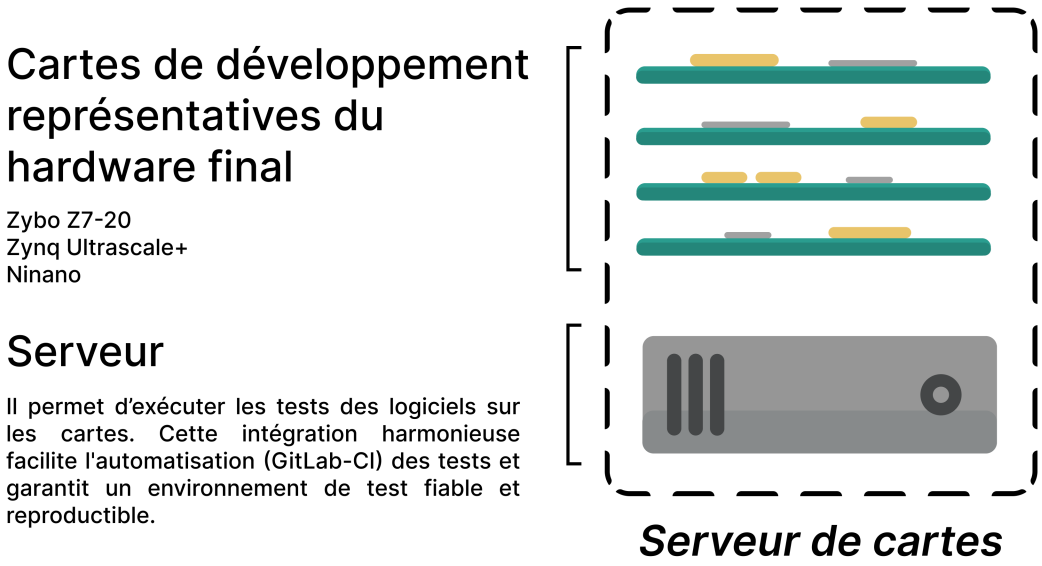


FIGURE 3.1 – Schéma du Banc d'essai du service Logiciel de Vol permettant d'effectuer du CI

En prenant en compte ces aspects, le DevOps dans l'industrie spatiale permet d'apporter une agilité et l'aspect amélioration continue que le new space ne cesse d'utiliser.

3.4 Étude de cas : Application du DevOps dans les projets spatiaux : l'exemple de SpaceX

SpaceX est régulièrement saluée pour son approche innovante et agitatrice dans l'industrie spatiale, et c'est également vrai pour ses méthodes de développement logiciel. En appliquant des principes grandement inspirés du DevOps SpaceX à réussi à considérablement augmenter l'efficacité et l'harmonie générale de son cycle de développement se propulsant en quelques années de manière phénoménale sur le devant de la scène spatiale internationale.

3.4.1 Progrès incrémentaux, data-driven design et conception modulaire

L'approche de SpaceX[4] en matière de conception de fusées repose sur le progrès incrémental et la modularité. Leur philosophie peut être résumé par l'adage : "Pour faire une omelette, il faut casser des œufs, *et vite !*". Cette philosophie a permis à elle seule de changer la face du monde de l'industrie spatiale en cassants les codes de l'industrie traditionnelle pour toujours. En effet, jusqu'à là, les méthodes conventionnelles de développement des lanceurs, comme celles utilisées dans le programme Ariane, ont été caractérisées par une approche plus prudente et graduelle motivée par une planification détaillée à long terme.

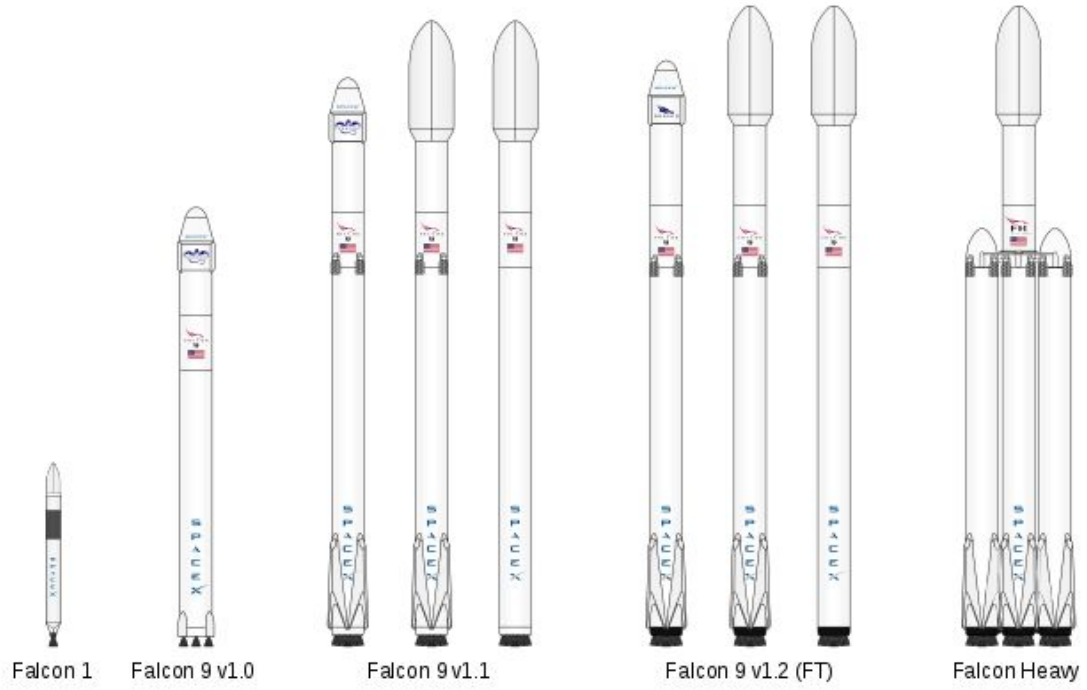


FIGURE 3.2 – Conception itérative chez SpaceX

De plus, en intégrant une stratégie de conception "data-driven", SpaceX optimise continuellement ses fusées en s'appuyant systématiquement sur les données recueillies lors de chaque test et mission. Chez SpaceX ; comme pour les bigtech ; les données valent de l'or. Elles permettent d'affiner grandement les modèles, d'améliorer la précision des simulations et d'accélérer d'avantage les cycles de développement.

3.4.2 Redondance triple et tolérance aux radiations

Dans l'environnement spatial hostile, la fiabilité des systèmes est cruciale. SpaceX utilise un système de redondance triple (triplex redundancy [5]) dans ses systèmes de contrôle de vol, augmentant la tolérance aux fautes sans nécessiter des composants spécifiquement durcis contre les radiations (pour les systèmes destinés aux orbites basses). Cette approche augmente significativement la fiabilité des missions spatiales sans recourir à des composants très coûteux limitant la prise de risque possible sur chaque test et itération.

L'utilisation de composants non-rad-hard (non durci) était dans un premier temps spécifique au new space, mais voyant les bénéfices apportés (notamment par la possibilité de développer des constellations de satellites), les institutions traditionnelles se sont elles aussi mis à penser et développer des projets avec des composants off-the-shelf. L'utilisation de ce type de composants reste tout de même à nuancer, en effet, ils apportent certes une flexibilité accrue et une réduction des coûts, mais la durabilité et la fiabilité des systèmes les embarquant peut être affectée plus facilement. De plus, leur démocratisation entraîne une augmentation drastique du nombre d'appareils sur les orbites

basses (Orbites moins soumises aux rayonnements [8]) on parle même d'hyper trafic augmentant le risque de collision.

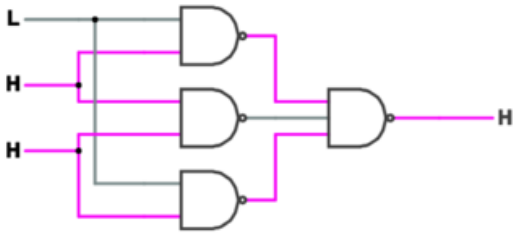


FIGURE 3.3 – Modèle simplifié de la triple redondance

3.4.3 Tests continus et simulation

SpaceX est renommé pour ses méthodes de simulations très précises et performantes leur permettant de réduire considérablement les besoins de tests en réel. Les simulations sont incluses dans des batteries de tests continus permettant de valider les logiciels (en autres). En utilisant des composants standardisés, l'entreprise peut simuler chaque contrôleur et processeur, permettant des tests d'intégration et de résistance approfondis. Cette stratégie permet d'anticiper et de gérer un large éventail de défaillances potentielles.

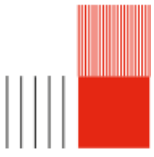
3.4.4 DevOps et amélioration continue

SpaceX adopte une approche DevOps, caractérisée par des cycles de développement très courts et itératifs. Cette méthodologie permet une amélioration constante du logiciel, essentielle dans un domaine où les exigences peuvent changer rapidement et où la fiabilité est cruciale.

3.4.5 Perspectives

L'approche de SpaceX illustre l'efficacité des méthodologies modernes de développement logiciel dans des environnements exigeants. En alliant conception modulaire, tests rigoureux et une philosophie d'amélioration continue, SpaceX ne se contente pas de relever les défis du vol spatial, mais redéfinit également les standards du développement logiciel dans l'industrie aérospatiale. À mesure que l'entreprise poursuit ses innovations, ses pratiques de développement logiciel continueront d'évoluer, offrant des enseignements précieux pour l'ingénierie logicielle en général.

En résumé, les pratiques de développement chez SpaceX montrent l'importance de l'adaptabilité, de la rigueur dans les tests et de l'engagement envers l'amélioration continue, toutes cruciales pour gérer des systèmes complexes et critiques. En adoptant ces méthodes, SpaceX à inciter toute l'industrie à changer de paradigme et adopter une méthode résolument plus agile et novatrice.



Chapitre 4

GitLab-CI et l'innovation en automatisation

Runner

Un runner Gitlab est un agent qui exécute les jobs CI définis dans les pipelines. Les runners peuvent être partagés ou project-specific.

Orchestrateur

L'orchestrateur est la pièce centrale d'une chaîne d'intégration CI/CD, il permet de simplifier, automatiser et coordonner l'ensemble du processus de développement logiciel, depuis l'intégration du code source jusqu'à son déploiement. Jenkins, GitLab-CI ou encore Travis CI sont des orchestrateurs.

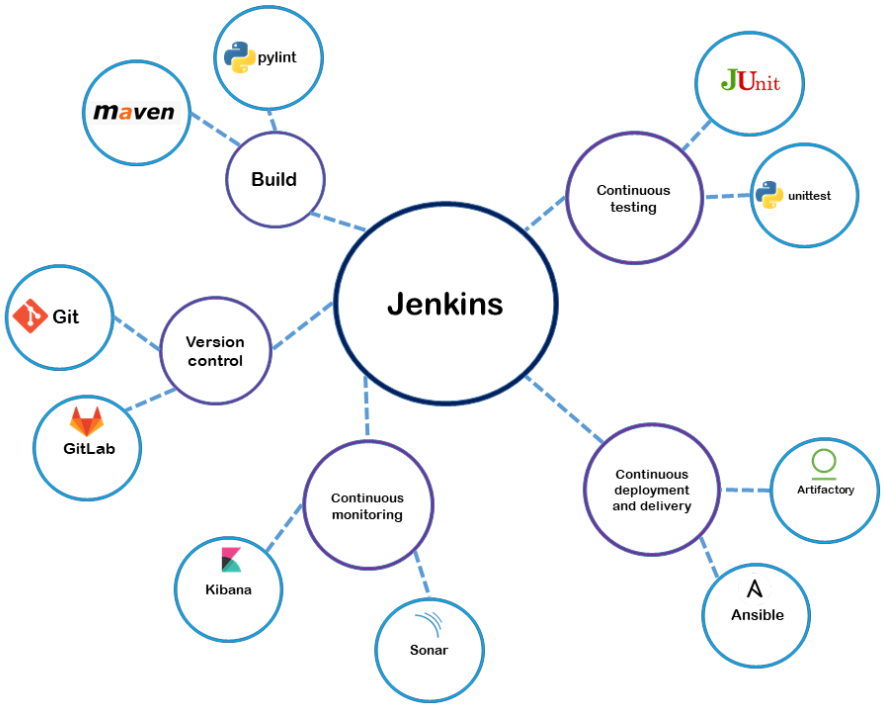


FIGURE 4.1 – Orchestrateur Jenkins et intégrations avec outils

GitLab

GitLab est un outil open-source de gestion de dépôt git très similaire à GitHub. L'avantage par rapport à celui-ci étant qu'il peut être hébergé localement (self-host).

4.1 Introduction à GitLab-CI

GitLab-CI est une fonctionnalité directement intégrée à GitLab permettant l'automatisation des phrases de test et de déploiement des logiciels, elle permet de couvrir complètement le CI et le CD, facilitant ainsi le développement rapide, fiable et harmonieux de projets en automatisant toute sorte de tâches.

4.2 Les principes de GitLab-CI

GitLab-CI repose sur plusieurs principes clés :
— **Automatisation :** Toutes les étapes de la construction, du test et du déploiement sont automatisées, permettant des livraisons plus rapides et fiables.
— **Configuration en tant que code (CaC) :** Les pipelines sont définis dans un fichier `.gitlab-ci.yml`, ce qui rend les processus transparents, simples, réutilisables et versionnés.

4.3 Déploiement et intégration continue avec GitLab-CI

GitLab-CI permet de mettre en place des pipelines d'intégration et de déploiement continu en définissant des séquences d'actions à exécuter à chaque modification du code source. Ci-dessous un exemple :

1. Un développeur push le code dans le dépôt GitLab.
2. GitLab détecte les changements et lance les tâches prédéfinies dans le ou les fichiers de configuration de pipeline. *Le développeur peut alternativement déclencher le pipeline manuellement.*
3. Les tâches sont très variées et peuvent inclure des tests, des analyses de qualité de code, des compilations, des analyses de sécurité et des déploiements.
4. Les résultats sont directement visibles sur l'UI.

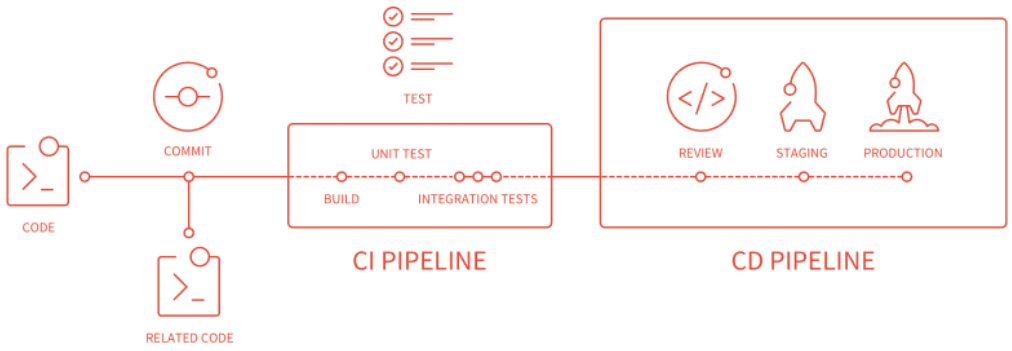


FIGURE 4.2 – Process CI/CD dans GitLab-CI

4.4 Conteneurisation et GitLab-CI

4.4.1 Qu'est-ce que la conteneurisation ?

La conteneurisation ; à ne pas confondre avec les machines virtuelles ; est une méthode de "virtualisation" très légère permettant d'offrir un environnement de développement identique et cohérent entre les parties impliquées dans le développement. Un conteneur encapsule une application ainsi que ses dépendances dans un *conteneur* isolé. Ce process assure une exécution uniforme peu importe l'environnement.

Les avantages de la conteneurisation peuvent être résumés comme suit :

— **Portabilité** : Exécution identique de l'application sur différents systèmes.
— **Reproductibilité** : L'environnement d'exécution et le build seront toujours identiques. Garanti que les binaires seront toujours identiques quelque soit la machine sur lequel il est compilé.
— **Isolation** : Chaque conteneur fonctionne de manière isolée les uns des autres.
— **Légèreté** : Beaucoup moins demandeur en ressources que les machines virtuelles : démarre très rapidement.

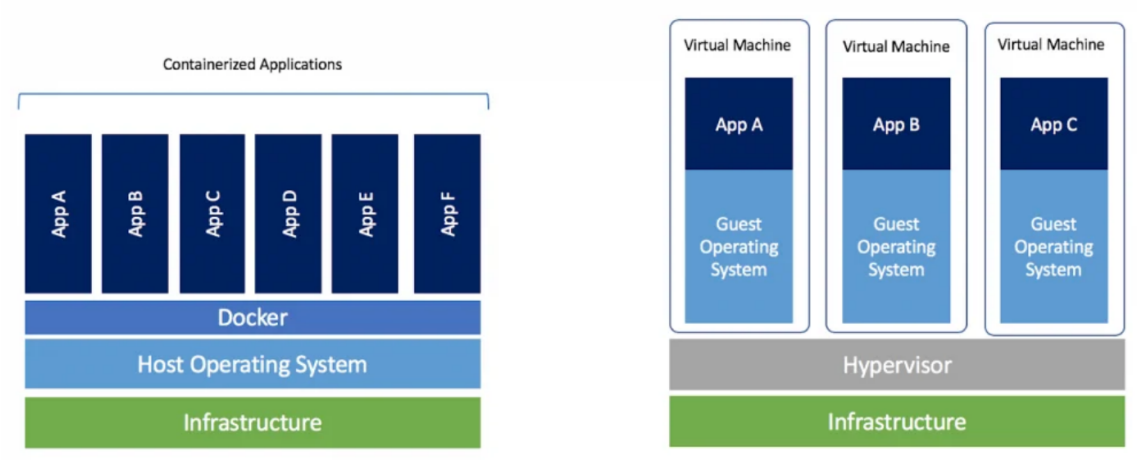


FIGURE 4.3 – Conteneur vs Machine virtuelle

4.4.2 Docker et GitLab-CI

Il est très courant d'utiliser l'outil Docker en combinaison avec GitLab-CI. Avec Docker, on peut créer des images personnalisées contenant un environnement de développement, de test et d'exécution cohérent. Cela permet notamment de grandement fluidifier le déploiement continu.

4.4.3 Docker au CNES

Le CNES et le service Logiciel de Vol utilisent extensivement la conteneurisation et Docker afin de fournir des environnement de développement cohérents intégrants tous les outils nécessaires. Par exemple lorsque l'on teste un logiciel de vol, il faut intégrer l'IDE nécessaire au chargement, à l'exécution et au debuggage du logiciel sur la carte de développement.

4.5 Avantages et Inconvénients de GitLab-CI face à Jenkins

L'avantage principal de GitLab-CI face à Jenkins réside dans son intégration avec GitLab permettant de limiter le nombre d'outils utilisés et donc de réduire la complexité globale. En effet, en utilisant Jenkins on doit mettre en place tout un système afin de synchroniser les actions sur le référentiel partagé (en l'occurrence GitLab) avec le déclenchement conditionné de pipelines d'automatisation. Ce processus est largement simplifié avec GitLab-CI.

	GitLab-CI	Jenkins
Facilité de configuration	Bonne	Modérée
Intégration avec d'autres outils	Excellente	Bonne
Flexibilité	Haute	Haute
Communauté et support	Large	Très large
Gestion des pipelines	Intuitive	Complexe
Gestion de librairie partagée	Possible [7]	Native
Language	YAML	GROOVY (syntaxe Java)
Type de fichier de conf.	.gitlab-ci.yml	Jenkinsfile

TABLE 4.1 – Comparaison entre GitLab-CI et Jenkins

Ci-dessous un exemple d'un pipeline Jenkins et GitLab-CI utilisant une image docker (gcc) pour compiler un fichier c.

Jenkinfile

```
pipeline {
    agent {
        docker {
            image 'gcc:latest' // Utilisation de l'image docker gcc
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'gcc -o foo main.c' // Compilation du fichier c
            }
        }
        stage('Test') {
            steps {
                sh './foo' // Exécution
            }
        }
    }
}
```

.gitlab-ci.yml

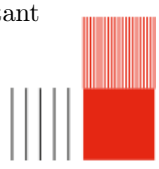
```
image: gcc:latest

stages:
  - build
  - test

build_job:
  stage: build
  script:
    - gcc -o foo main.c

test_job:
  stage: test
  script:
    - ./foo
```

4.6 Création et utilisation de bibliothèques GitLab-CI

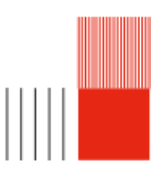
4.6.1 Possibilité de créer une bibliothèque GitLab-CI

GitLab-CI offre la possibilité de créer des libraries CI, un peu à la manière des Jenkins Shared Libraries[6]. Ces Librairies permettent de réutiliser des scripts, des templates de jobs et des configurations au sein de plusieurs projets "tirant" cette librairie. La librairie étant elle-même hébergée sur un dépôt GitLab, elles peuvent être versionnées, partagées et maintenues de manière centralisée ce qui confère une qualité constante dans les pipelines utilisés. Dans le service Logiciel de Vol du CNES, son utilisation est très utile car les pipelines de CI sont communs à tous les projets modulo quelques éléments de configuration. Cela évite les duplicata de code inutiles et simplifie les processus dans leurs globalité.

GitLab-CI présente néanmoins quelques désavantages face à Jenkins concernant la gestion de librairie CI partagée. En effet, Jenkins possède le concept de "Jenkins Shared Librarie" facilitant



grandement le développement de ce type de librairie, alors que pour développer une librairie sur GitLab-CI il faut utiliser le concept d'include permettant d'utiliser des snippets de configuration depuis d'autres fichiers.



Chapitre 5

GitLab-CI et DevOps au CNES

Dans l'objectif de suivre les tendances de l'industrie spatiale actuelle, le CNES s'est tourné il y a quelques années sur une approche DevOps favorisant l'efficacité et la robustesse de ses développements. Ma connaissance se porte principalement sur ce qui est fait au sein du service Logiciel de Vol, mais d'autres instances utilisent également extensivement ces principes et méthodes. Ce chapitre explore l'état actuel, les motivations et les défis liés à l'intégration d'une philosophie CI dans les projets du service Logiciel de Vol.

5.1 Migration Stratégique : De Jenkins à GitLab-CI

5.1.1 Contexte et Environnement Actuels

Depuis quelques années le service Logiciel de Vol développe en adoptant une philosophie CI basée sur l'orchestrateur Jenkins comme noyau central. Bien que suffisamment performant Jenkins possède néanmoins quelques inconvénients, notament une intégration non-optimale avec GitLab et quelques autres problématiques spécifiques au CNES (Plus de mises à jour ni nouvelles fonctionnalités car le service IT à choisi GitLab-CI en baseline). Ces problématiques seraient résolues par l'adoption de GitLab-CI en tant qu'orchestrateur.

L'environnement actuel repose sur une Jenkins Shared Librairy développée spécifiquement pour le développement de logiciel de vol, elle comporte des pipelines permettant de build et de tester ces logiciels (build - analyse qualité code - tests unitaire - tests intégration). L'objectif consiste à migrer cette librairie vers GitLab-CI en préservant toutes ses fonctionnalités afin d'éviter toute interruption lors de la transition. La transition sera effectuée selon un ordre de priorité : Build - Tests - Qualité.

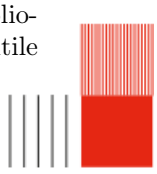
5.1.2 Procédure

La structure de la bibliothèque Jenkins actuelle est organisée comme suit. Elle se base sur le modèle indiqué dans la documentation de Jenkins.

```
+- resources                                # Contains files other than Groovy code (Parsers, useful python scripts...)
+- src
|    +- ci
|      +- Pipeline-1.groovy
|      +- Pipeline-2.groovy
|      +- Pipeline-3.groovy
+- vars                                     # Contains definition of global steps (which are elementary Jenkins function)
|   +- cleanWorkspace.groovy               # Defines step cleanWorkspace()
|   |   +- cleanWorkspace                  # Removes all files and directories in the workspace
|   +- clone.groovy                        # Defines step clone()
|   |   +- clone                           # Execute a simplified git clone over SSH or HTTPS.
...
```

FIGURE 5.1 – Arborescence de la bibliothèque Jenkins

Afin d'assurer une transition cohérente, la bibliothèque GitLab-CI suivra le même modèle dans les grandes lignes en ajoutant une fonctionnalité majeure : la possibilité d'auto-tester la bibliothèque au travers de tests unitaires exhaustifs. Cette fonctionnalité se révèle particulièrement utile



quand le service IT du CNES procède à des montées en version sur certains outils utilisés par les chaînes CI. Ces mises à jour peuvent potentiellement perturber le déroulement de certains processus et pipelines. En ayant des scénarios de test intégrés à la bibliothèque, le service IT est en mesure de tester, notifier les besoins d'évolution de la bibliothèque et corriger de manière autonome les problèmes liés aux mises à jour sans nécessiter l'intervention du personnel du service Logiciel de Vol. Cette approche contribue à rendre le processus d'amélioration plus fluide et transparent en réduisant les échanges inter services inutiles et les points bloquants. C'est en quelque sorte une philosophie CI dans CI.

```
lv-gitlabci-library/                    # Contains predefined pipeline configuration
├── pipelines/
│   ├── pipeline-1.yml
│   ├── pipeline-2.yml
│   └── pipeline-3.yml
│
├── includes/                           # Directory for reusable components
│       ├── docker-build.yml
│       ├── test.yml
│       ├── sonarqube-analysis.yml
│       ├── deploy.yml
│       └── ...
│
├── scripts/                            # Script directory
│   ├── prepare-environment.sh
│   ├── analyse_regression.py
│   └── ...
│
└── tests/                              # Tests unitaires de la library
    ├── scripts/                        # Tests des scripts
    ├── pipelines/                      # Tests des pipelines
    └── includes/                       # Tests des composants génériques
```

FIGURE 5.2 – Arborescence hypothétique de la bibliothèque GitLab-CI

GitLab-CI ne disposant pas de mécanisme de Shared Library natif, la migration de la bibliothèque Jenkins se basera sur le concept d'*include* expliqué précédemment.

5.2 Évolution des Tests : De TCL à Python

Les tests sont un aspect crucial du développement de logiciel de vol, ils permettent d'éprouver le logiciel sur une plage de contraintes prédéfinies et des scénarios d'exécution mockant une exécution réelle. Les tests intègrent bien souvent du matériel réel dans la boucle de test : HSVF (Hardware Software Verification Facility). Cette approche permet une vérification plus précise du bon fonctionnement du logiciel, assurant une performance optimale et une sécurité accrue. La volonté d'évolution des tests de vaildation (Validation des Chaines Fonctionnelles) de TCL vers Python illustre le mouvement continu du CNES vers des méthodologies et technologies plus modernes, efficaces et démocratisées, essentielles pour répondre aux exigences actuelles.

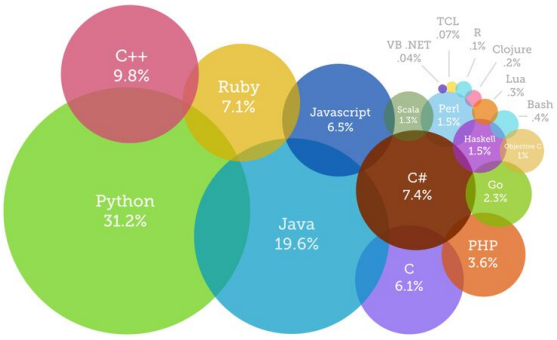


FIGURE 5.3 – Popularité de Python face à TCL : Il est plus facile de trouver des talents ayant la compétence Python

5.2.1 Environnement de Test Actuel

Les tests actuels sont développés en TCL (Tickle) et les résultats sont exportés au format jUnit [10] permettant de les importer automatiquement dans Jira (gestionnaire de tickets/bugs). Ci-dessous un exemple du déroulement d'un simple test ping HSVF :

- Initialisation de l'environnement de test et chargement du logiciel de vol sur la carte
- Déclaration des variables pour suivre le nombre de test, les échecs éventuels et différentes métriques.
- Exécution du test ping
  - Envoi de la TC(17,1)
  - Attente de la TM(17,2)
  - Comparaison des résultats du test avec les résultats attendus et génération d'un test report.
- Affichage d'un résumé des tests
- Sauvegarde les résultats des tests en format XML en utilisant le framework jUnit.

5.2.2 Utilisation de Pytest

Pytest [9] est un framework de test pour Python qui est utilisé pour écrire des tests. Il est très facile à utiliser et permet d'écrire des tests unitaires et fonctionnels. Ci-dessous une version simplifiée du test ping en utilisant le framework pytest :

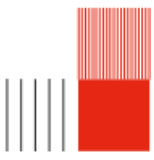
```
import pytest

@pytest.fixture # Définition de l'environnement de test
def setup():
    # Setup de l'environnement et chargement du LV
    yield # Permet de créer un point de départ pour test_ping
          # après le test le programme retournera ici
    # Opérations de nettoyage après test

def test_ping(setup):
    assert send_command("TC(17,1)"), "Echec de l'envoi de la TC"

    telemetry = wait_for_telemetry("TM(17,2)")
    assert telemetry["status"] == "OK", "Telemetry status not OK"
```

Il est toujours possible d'exporter les résultats au format jUnit en utilisant le module python pytest-junitxml il suffit ensuite d'exécuter le test avec les arguments suivants :

```
python3 -m pytest --junitxml=./report.xml
```

5.2.3 Nécessité de Généricité des Tests

Les tests doivent être conçus de manière à être facilement compréhensibles et modifiables afin de pouvoir modifier ou ajouter des nouveaux tests sans efforts de compréhension de l'environnement en place. Cela implique d'une part l'utilisation de frameworks puissants et bien structurés mais aussi d'une philosophie DRY prônant la réusabilité, la modularité et la généricité. L'objectif est de réduire un maximum la charge de travail liée à la maintenance des tests et de permettre une exécution rapide et régulière des batteries de tests. Ces efforts devraient contribuer à la réduction du cycle de développement tout en améliorant la fiabilité et en facilitant les mises à jour futures.



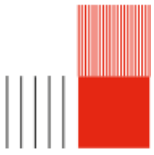
Chapitre 6

Conclusion

Au travers de ce rapport, nous avons étudié en détail l'évolution du développement logiciel en mettant en lumière la transition des méthodes traditionnelles vers des pratiques agiles. Nous avons observé la place de plus en plus importante que le DevOps prend dans les chaînes de développement de tous les secteurs, et son impact sur les logiciels : plus fiables, plus rapidement.

En conclusion, nous pouvons observer que le développement logiciel et ses techniques ont parcouru un long chemin depuis leurs balbutiements et que l'industrie à continuellement changé de forme au fil des méthodologies et outils. L'adoption des méthodes prônant la rapidité et l'agilité a agi comme un coup de pied dans la fourmilière des agence spatiales traditionnelle et à permis de rebattre les cartes. Ces méthodes nécessitent un changement drastique de paradigme et d'infrastructure pour les appliquer dans un domaine éminemment critique. En fin de compte, l'avenir du développement logiciel dans le new space semble être définitivement axé sur l'agilité, l'automatisation et l'amélioration continue pour répondre aux besoins croissants de fiabilité et de rapidité. Les institutions traditionnelles, tout en s'inspirant de certains aspects du DevOps, conservent une approche hybride, mêlant innovation et méthodes éprouvées, afin de répondre au mieux aux exigences spécifiques de leurs projets.

Pour finir, ce rapport a ainsi contribué à répondre aux trois grands axes offrant un aperçu de l'histoire, de la situation actuelle et des perspectives futures.



Bibliographie

[1] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=476557

[2] https://en.wikipedia.org/wiki/Von_Neumann_architecture

[3] https://fr.wikipedia.org/wiki/Fortran

[4] https://www.youtube.com/watch?v=t705r8ICkRw

[5] https://www.coderskitchen.com/spacex-software-development-and-testing/

[6] https://www.jenkins.io/doc/book/pipeline/shared-libraries/

[7] https://dev.to/anhtm/a-comprehensive-guide-to-creating-your-own-gitlab-ci-template-library-5b3

[8] https://www.arcep.fr/actualites/les-prises-de-parole/detail/n/interview-ademe-arcep-cnes-satellites-environnement-la-tribune-201123.html

[9] https://docs.pytest.org/en/7.4.x/

[10] https://junit.org/junit5/docs/current/user-guide/