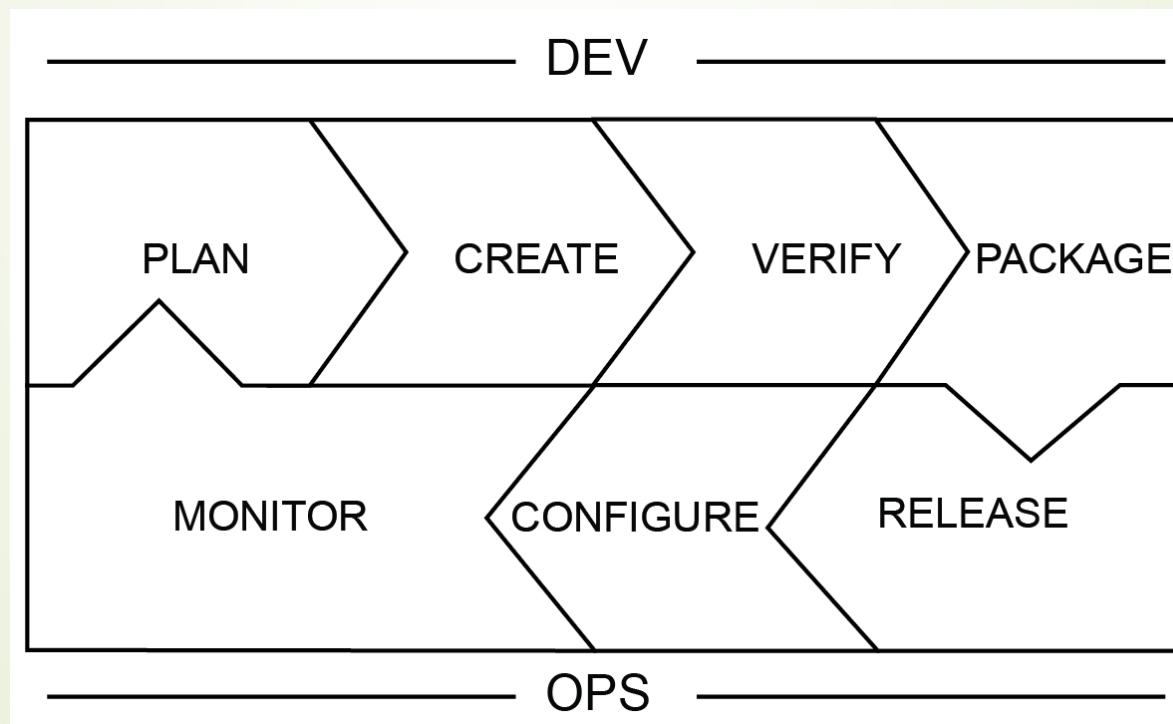


LES OUTILS DEVOPS

La chaîne d'outils DevOps

- Description

- La chaîne d'outils DevOps est une collection de solutions logicielles permettant à une équipe DevOps d'implémenter tout ou partie du flux ci dessous représentant un cycle DevOps



La chaîne d'outils DevOps

- Panorama des outils
 - **Plan** : Outils de planification
 - gestionnaires de projet : Redmine, Jira, Asana, Monday...
 - gestionnaire de backlog : Trello
 - gestionnaire de documents : Dropbox, Google Drive...
 - **Create** : Outils de développement
 - éditeurs de texte : Visual Code, Sublime Text, Pycharm, vim...
 - gestionnaire de code source :SVN, Git, Gitlab...

La chaîne d'outils DevOps

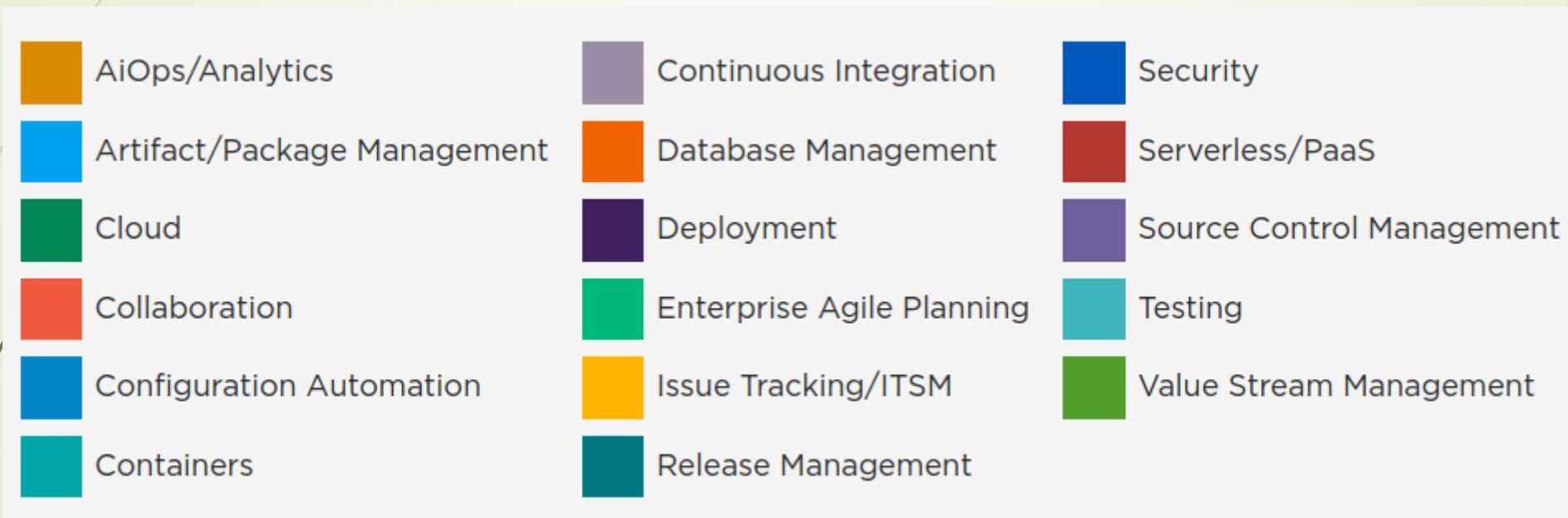
- Panorama
- **Verify** : Outils de test / validation
 - frameworks de tests unitaires : Junit et ses déclinaisons
 - frameworks de tests de validation : Robot Framework
 - serveurs de tests e2e : Selenium
 - serveurs de tests de performances : Gatling
 - gestionnaire d'assurance qualité : Sonarqube
- **Package ou Build** : Outils de préparation à la livraison
 - gestionnaire de build : Buildout (python), Apache Maven (Java), Composer (PHP), Webpack (JavaScript)...

La chaîne d'outils DevOps

- Panorama
 - **Release** : Outils en charge de la gestion des livrables
 - Gestionnaire d'intégration continu : Jenkins, gitlab CI, Team City...
 - **Configure** : Outils de configuration et d'orchestration
 - gestionnaires de configuration et de déploiement : Ansible, Chef, Puppet
 - orchestrateur (+docker) : Kubernetes, Terraform...
 - **Monitor** : Outils de surveillance de l'activité
 - serveur de monitoring : nagios, prometheus

La chaîne d'outils DevOps

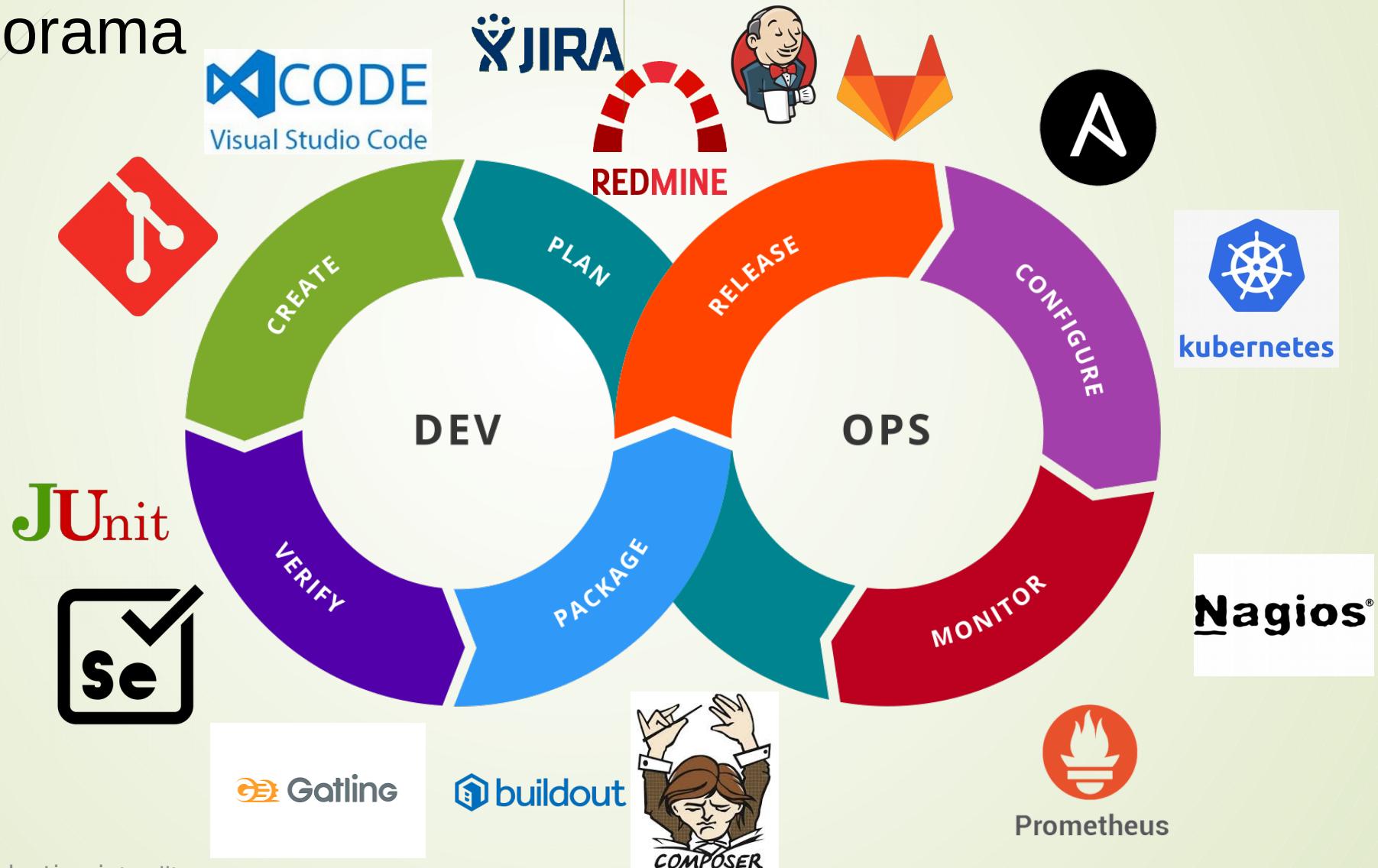
- Catégories d'outils



<https://digital.ai/periodic-table-of-devops-tools>

La chaîne d'outils DevOps

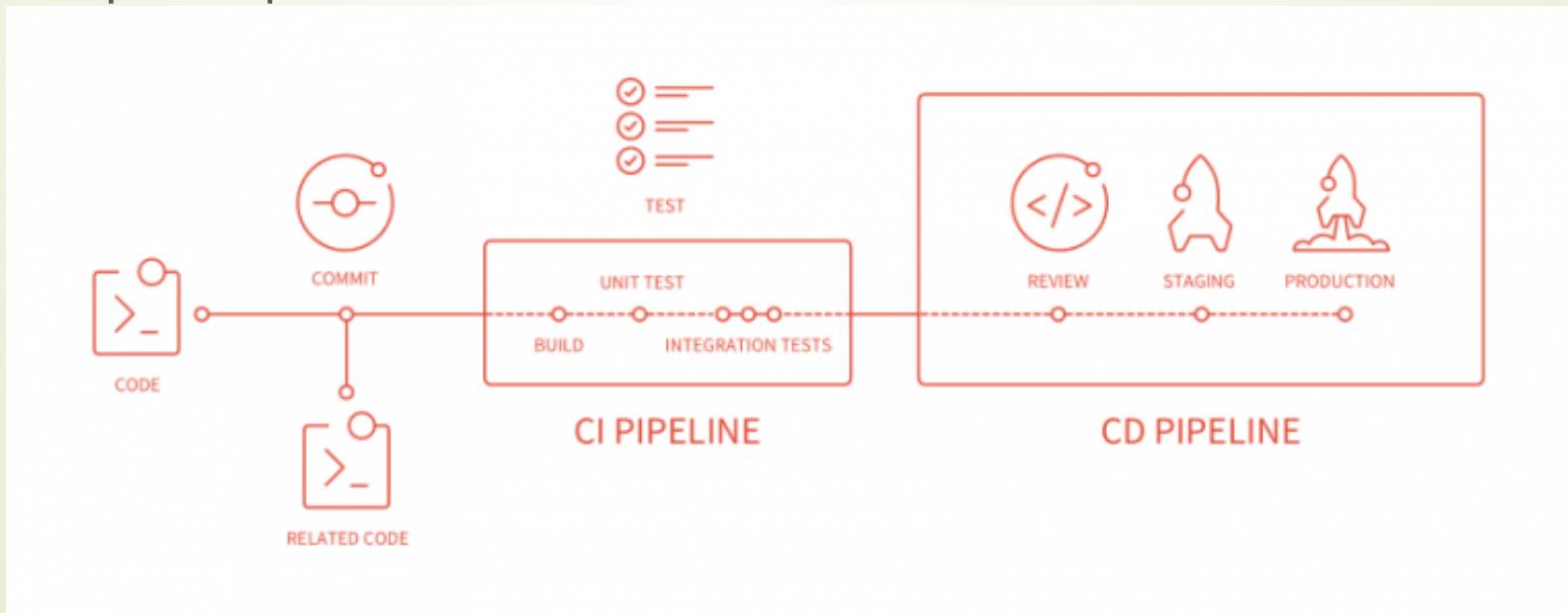
- Panorama



Les Pipelines DevOps

• Description

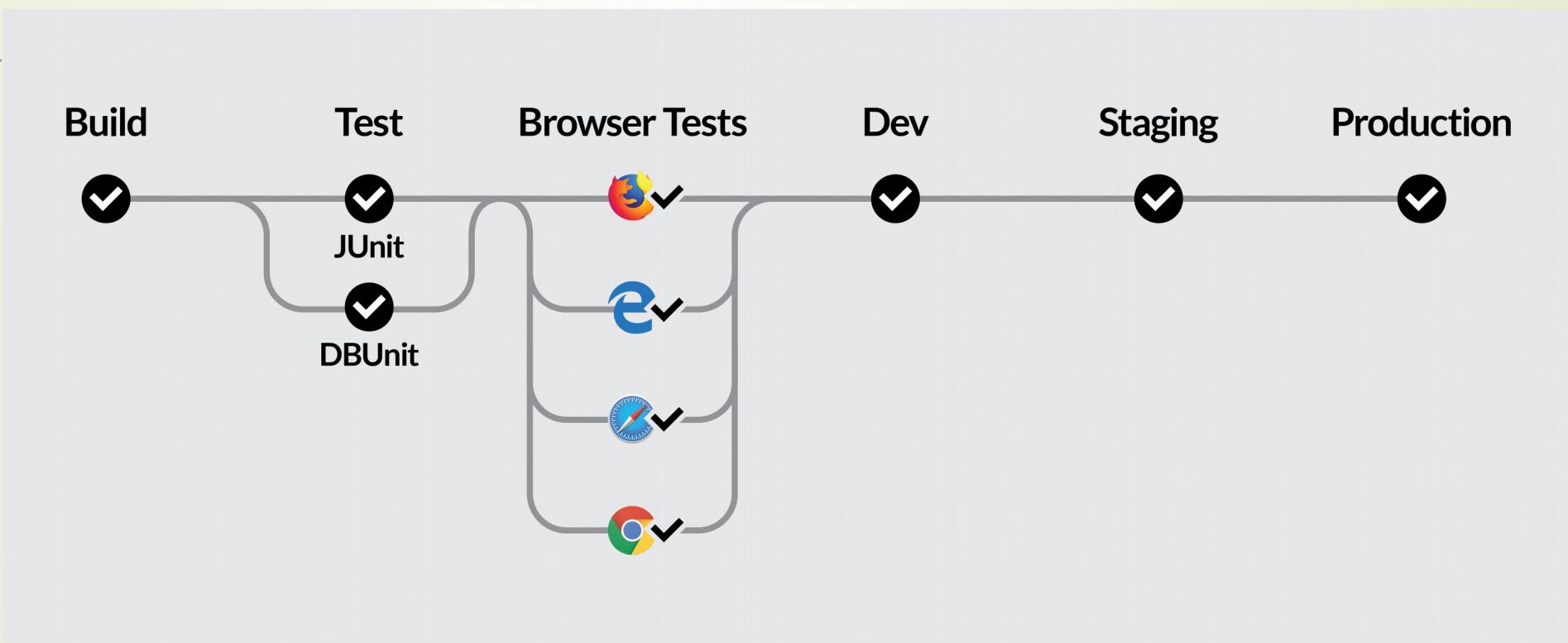
- Un pipeline DevOps est une implémentation de la chaîne d'outils DevOps
- Un exemple simple, reliant un pipeline d'intégration continue - build, tests, avec un pipeline de déploiement continu – analyse qualité et deux déploiements
- Les gestionnaires d'intégration continue (jenkins, gitlab...) sont les composants qui instancient les pipelines et en contrôle l'exécution en fonction des résultats de chaque étape



Les Pipelines DevOps

- Parallélisation

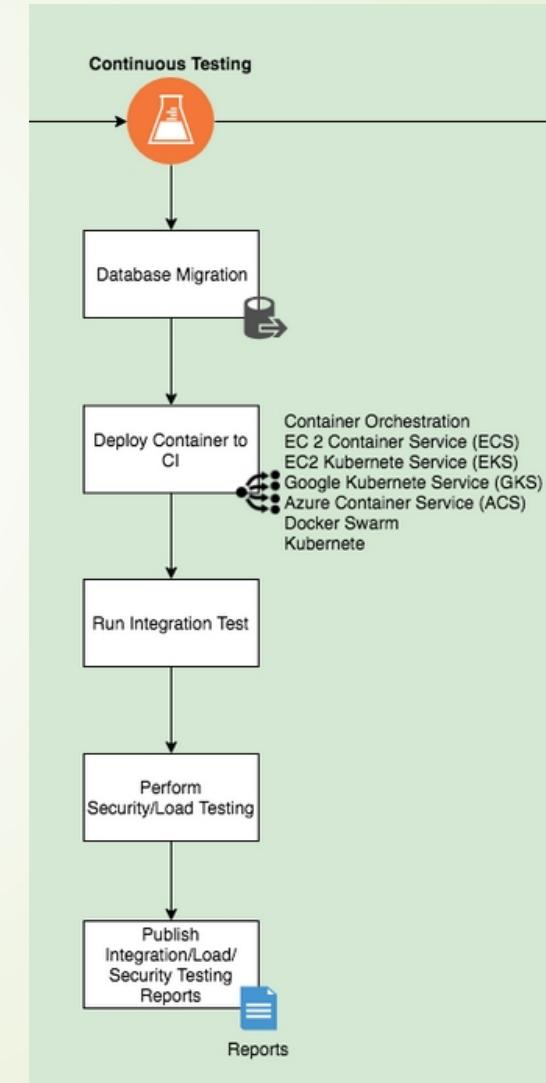
- Il est possible de paralléliser l'exécution de certaines étapes du pipeline, si l'on a préalablement déterminé les sous ensembles de code indépendants les uns des autres



Les Pipelines DevOps

• Sous tâches

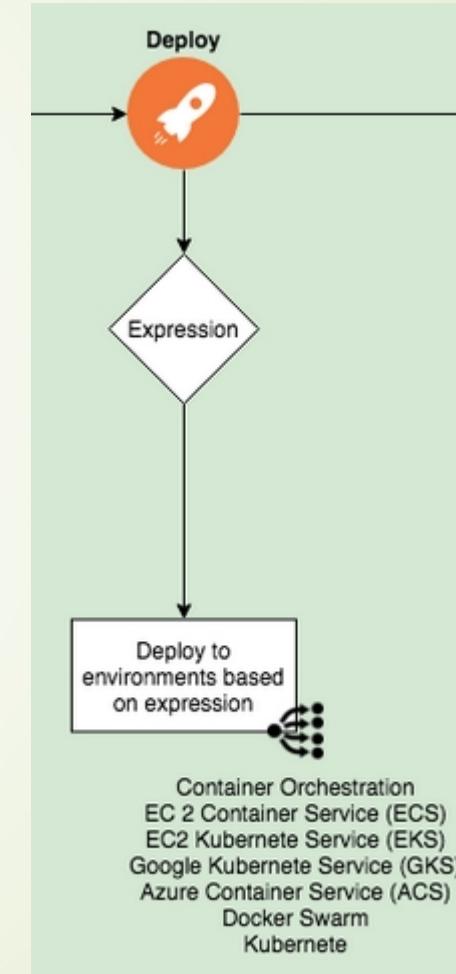
- Il est possible sur certains gestionnaires d'intégration continue, de découper une étape en plusieurs sous tâches en fonction de la complexité de celle ci.
- Ces sous tâches peuvent s'exécuter sur des environnements différents (cf exemple)



Les Pipelines DevOps

• Exécutions conditionnelles

- On peut soumettre l'autorisation d'exécution d'une étape à la résolution d'une expression booléenne prenant compte des résultats d'étapes précédentes
- Cette fonctionnalité est indispensable dans le cas d'un pipeline de déploiement continu



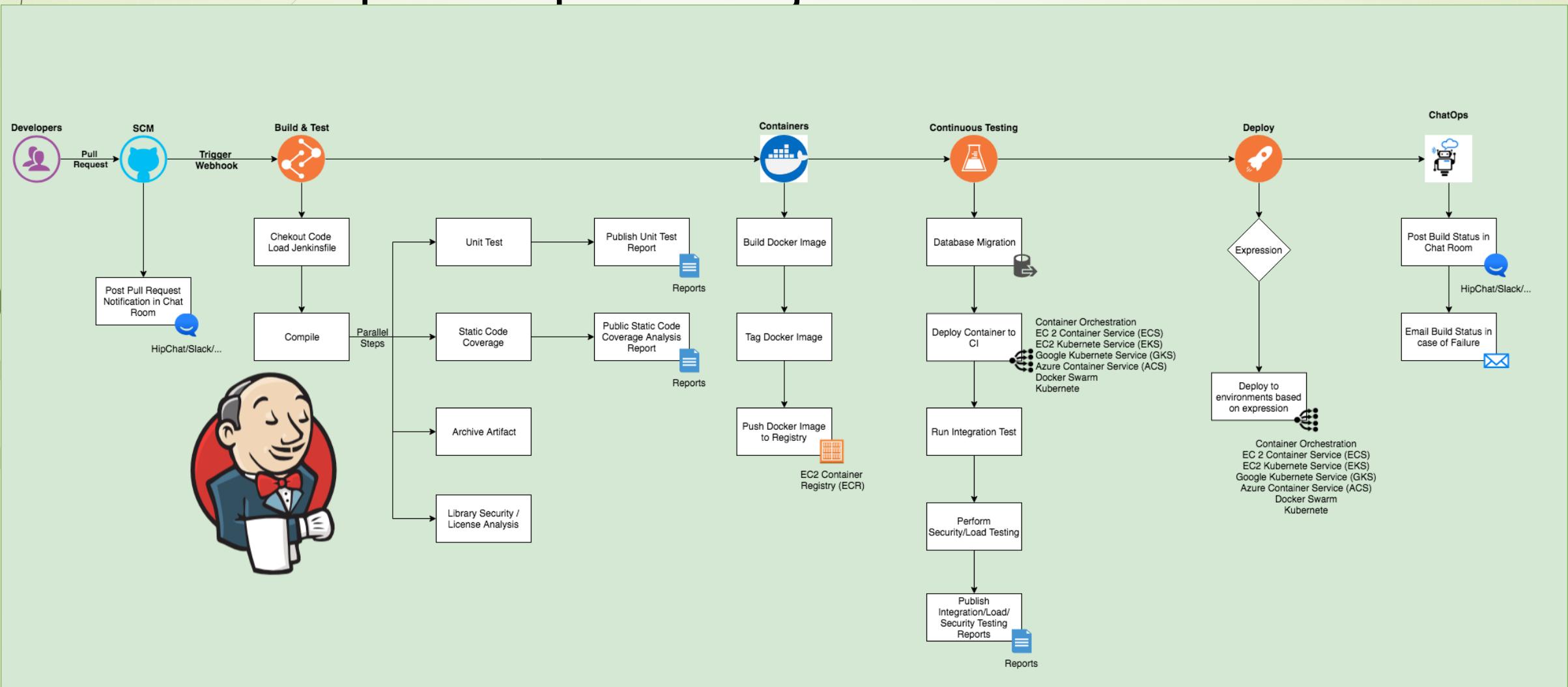
Les Pipelines DevOps

• Gestion des artefacts

- On appelle artefact tout produit intermédiaire de l'exécution du pipeline
- Il peut s'agir :
 - de paquets ou de bibliothèques logicielles issue de l'étape build, et qui doivent être déployés avec le reste du code
 - des rapports de tests ou d'analyse Qualité destinés à être consultés ou intégrés dans une base de connaissances
 - des rapports de monitoring utiles pour mettre à jour les critères de tests d'acceptance
- Les gestionnaires d'intégration continue fournissent des mécanisme de stockage et de manipulation des artefacts
- Il existe des solutions de gestion en masse et multi projets des artefacts : « artifactory » (JFrog)

Les Pipelines DevOps

- Exemple complet avec jenkins



Scalabilité de Devops

- Bonnes pratiques

- Un système est scalable s'il peut répondre sans dégradation de performances à une montée de la demande en ordre de grandeur
- En premier lieu, il faut s'assurer de toujours développer intégrer en cycles courts, ce qui limite les blocages, sachant que les étapes d'intégrations sont automatisées
- Il faut concentrer son attention sur les étapes les plus lentes (théorie des contraintes) pour les optimiser, les paralléliser et en augmenter le débit
- Plus Devops sera impliqué dans un grand nombre de process métier, plus la scalabilité sera assurée pour l'entreprise. Il est donc primordial de décloisonner les structures organisationnelles en silos

Scalabilité de Devops

- Bonnes pratiques

- Il faut mettre l'accent sur les métriques qualité et sécurité du système, donc au-delà de la livrabilité pour les devs et au-delà de la stabilité pour les ops.
- Il faut garantir la visibilité du système en temps réel par les outils de surveillance, de remontée d'information et d'alertes, afin de favoriser la réactivité des équipes
- Il faut pouvoir analyser régulièrement les comportements utilisateurs et recueillir les demandes utilisateurs, pour orienter la croissance du système
- Utiliser autant que possible des templates d'integration continue (comme auto devops de gitlab) permet d'implémenter à peu de frais les bonnes pratiques de CI/CD

Gestion de Code Source

- Définition
 - Logiciel qui permet de stocker un ensemble de fichiers en conservant la chronologie de toutes les modifications qui ont été effectuées dessus. Il permet notamment de retrouver les différentes versions d'un lot de fichiers connexes, nommés « commits ».
 - Les avantages de git :
 - le travail est sauvegardé régulièrement, permettant des rollbacks en cas de perte de données
 - l'outil fournit une série de métadonnées détaillant les commits (date, auteur, détail des modifications..)
 - git est décentralisé, chaque collaborateur possède une version du code dans son dépôt local, et travaille indépendamment des autres

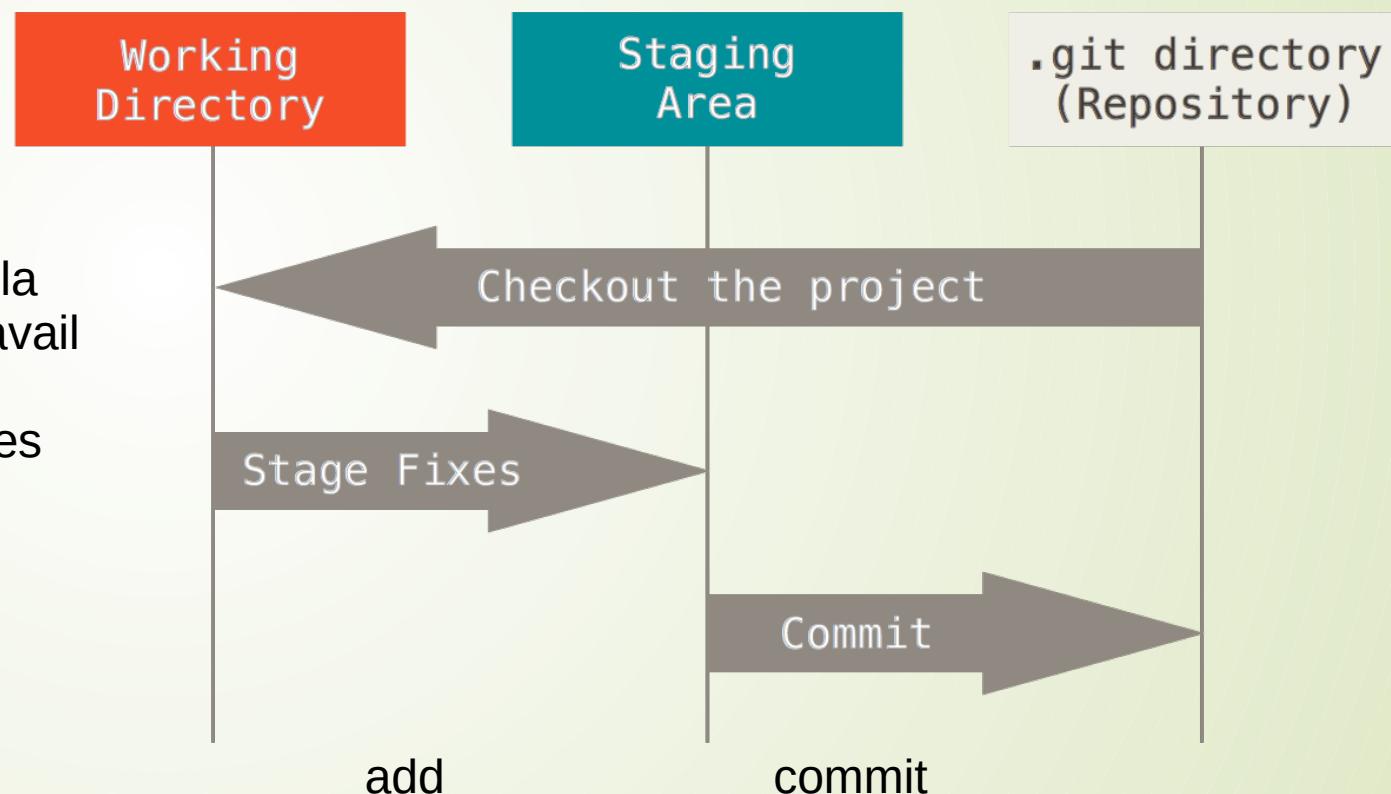
Gestion de Code Source

- Dépôt git local

git init : création du dépôt

git add : ajout des modifications à la staging area depuis la copie de travail

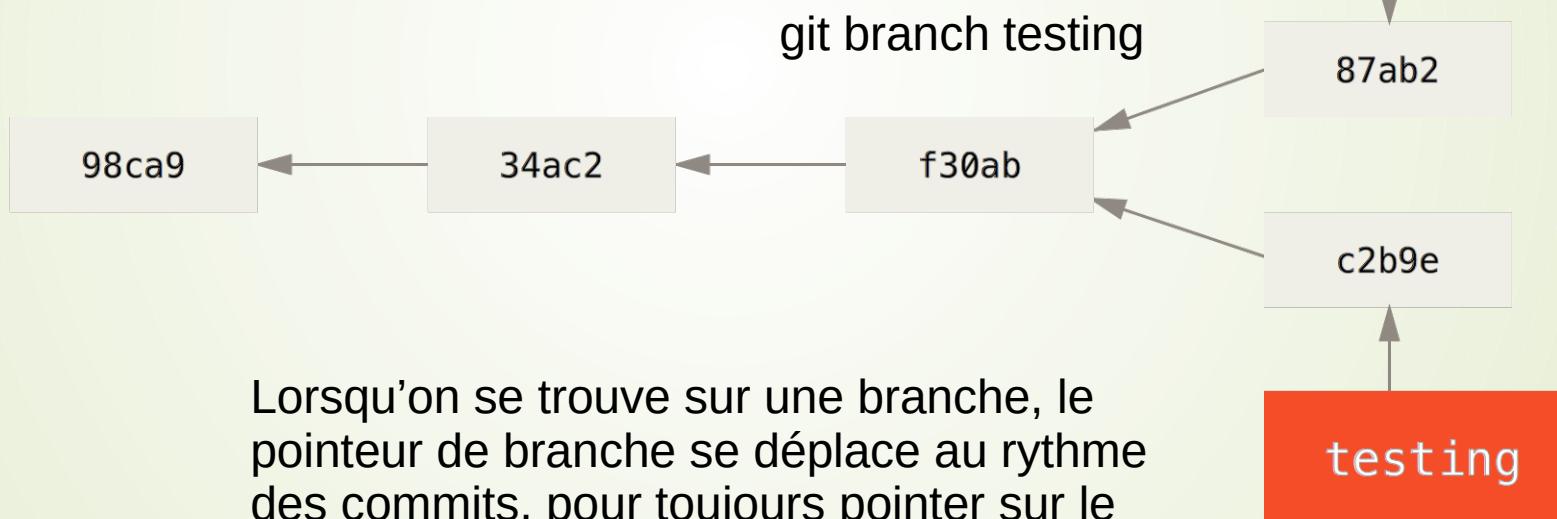
git commit -m "msg" : validation des modifications / déplacement de la staging area au dépôt avec renseignement d'un message de description du commit



Gestion de Code Source

- Description des branches

- Une branche est un pointeur créé sur un commit particulier
- La branche initiale se nomme « master »
- le pointeur Head pointe toujours sur le dernier commit de la branche courante



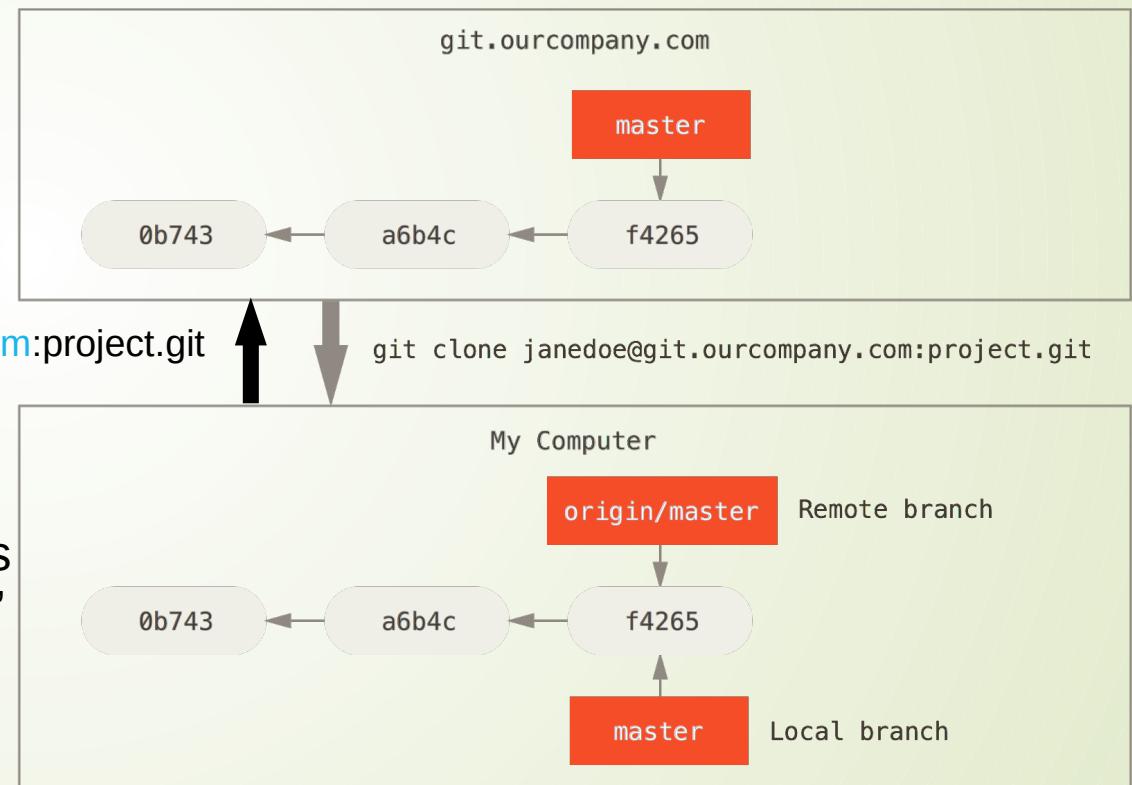
Gestion de Code Source

- Connexions à un dépôt distant

git clone : copie d'un dépôt distant avec ses branches

git remote add origin janedoe@git.ourcompany.com:project.git
git pull origin master

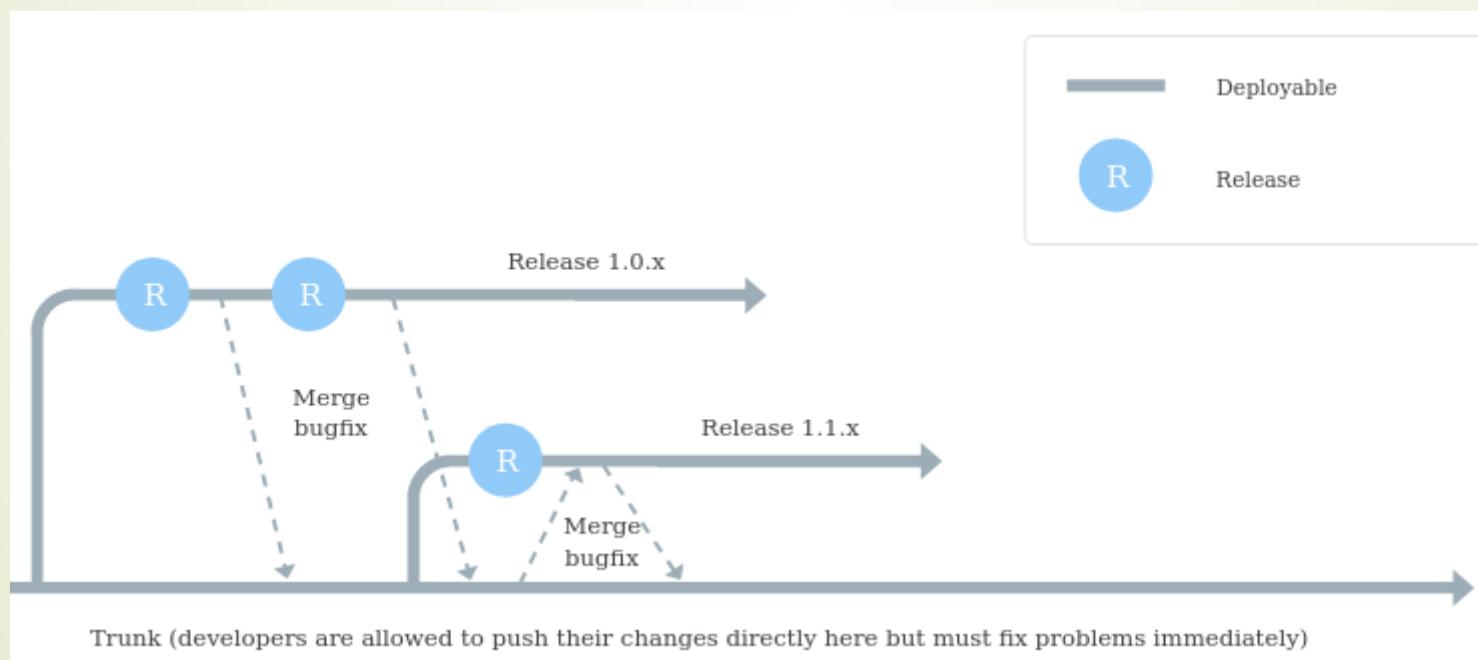
git remote : connexion à un dépôt distant "origin"
git pull origin master : rapatriement en local du des derniers commits de la branche master de "origin"



Gestion de Code Source

- Trunk Based Development

- Le Développement à branche unique, orienté DevOps : chaque développeur divise son propre travail en petits lots et fusionne ce travail avec la branche principale au moins une fois par jour (et éventuellement plusieurs fois).



JUnit

- ## Introduction

JUnit est un framework de test de regression open source à l'usage des développeurs pour écrire et exécuter des test unitaires en Java

Objectif : automatiser les tests and augmenter la qualité du code.

Junit est formé d'une collection de classes structurant les tests (POO)

Issue de la suite XUnit family (HTTPUnit, Cactus), CppUnit

JUnit

- Mécanisme

On définit une classe de Test par dérivation de la classe abstraite **TestCase**.

On Surdéfinit les méthodes **setUp()** et **tearDown()**, de préparation et de destruction du test

Chaque test unitaire correspond à l'écriture d'une méthode estampillée **testXXX()**

Les tests sont réalisés par utilisations de fonctions de types **AssertXXX()**

Definir en dehors de la classe une méthode statique **suite()**

Créer dedans un objet TestSuite de type Factory qui va consommer la classe de test.

On peut doter sa classe **TestCase** d'un bloc **main()** pour une exécution en mode batch (CI / CD).

JUnit

- Exemple : classe de test

```
public class StringTest extends TestCase {  
    protected void setUp(){ /* run before */}  
    protected void tearDown(){ /* after */ }  
  
    public void testSimpleAdd() {  
        String s1 = new String("abcd");  
        String s2 = new String("abcd");  
        assertTrue("Strings not equal",  
                  s1.equals(s2));  
    }  
  
    public static void main(String[] args){  
        junit.textui.TestRunner.run (suite());  
    }  
}
```

JUnit

- Exemple : exécution de la Suite de Test

```
public static Test suite (){
    suite = new TestSuite ("StringTest");
    String tests = System.getProperty("tests");
    if (tests == null){
        suite.addTest(new
            TestSuite(StringTest.class));
    }else{
        StringTokenizer tokens = new
        StringTokenizer(tests, ",");
        while (tokens.hasMoreTokens()){
            suite.addTest(new
                StringTest((String)tokens.nextToken()));
        }
    }
    return suite;
}
```

JUnit

- Rapport JUnit (artefact)

Unit Test Results				
Designed for use with JUnit and Ant				
Summary				
Tests	Failures	Errors	Success rate	Time
2	1	0	50.00%	2.369
Note: <i>failures</i> are anticipated and checked for with assertions while <i>errors</i> are unanticipated.				
Packages				
Name	Tests	Errors	Failures	Time(s)
org.example.antbook.ant.lucene	1	0	1	1.197
org.example.antbook.junit	1	0	0	1.172

JUnit

- Fonctions Assert

`assertEquals(String message, expected, actual)` : évalue une égalité entre deux objets de même classe ou deux variables de même type

`assertSame(Object expected, Object actual)` : évalue deux références d'un même objet

`assertNotSame(Object expected, Object actual)`

`assertNull(Object object)` : évalue si l'objet est null

`assertTrue(Boolean condition)` : évalue si le booléen vaut True

`fail(String message)`: provoque l'échec du test avec ajout du message au rapport Lance une Exception `AssertionFailedError`

JUnit

- Déclenchement de JUnit depuis Apache Ant

```
<target name="batchtest" depends="compile-tests">
  <junit printsummary="yes" fork="yes" haltonfailure="no">
    <classpath>
      <pathelement location="${build.dir}" />
      <pathelement location="${build.dir}/test" />
    </classpath>
    <formatter type="plain" usefile="yes"/>
    <batchtest fork="yes" todir="">
      <fileset dir="${test.dir}">
        <include name="**/*Test.java" />
      </fileset>
    </batchtest>
  </junit>
</target>
```

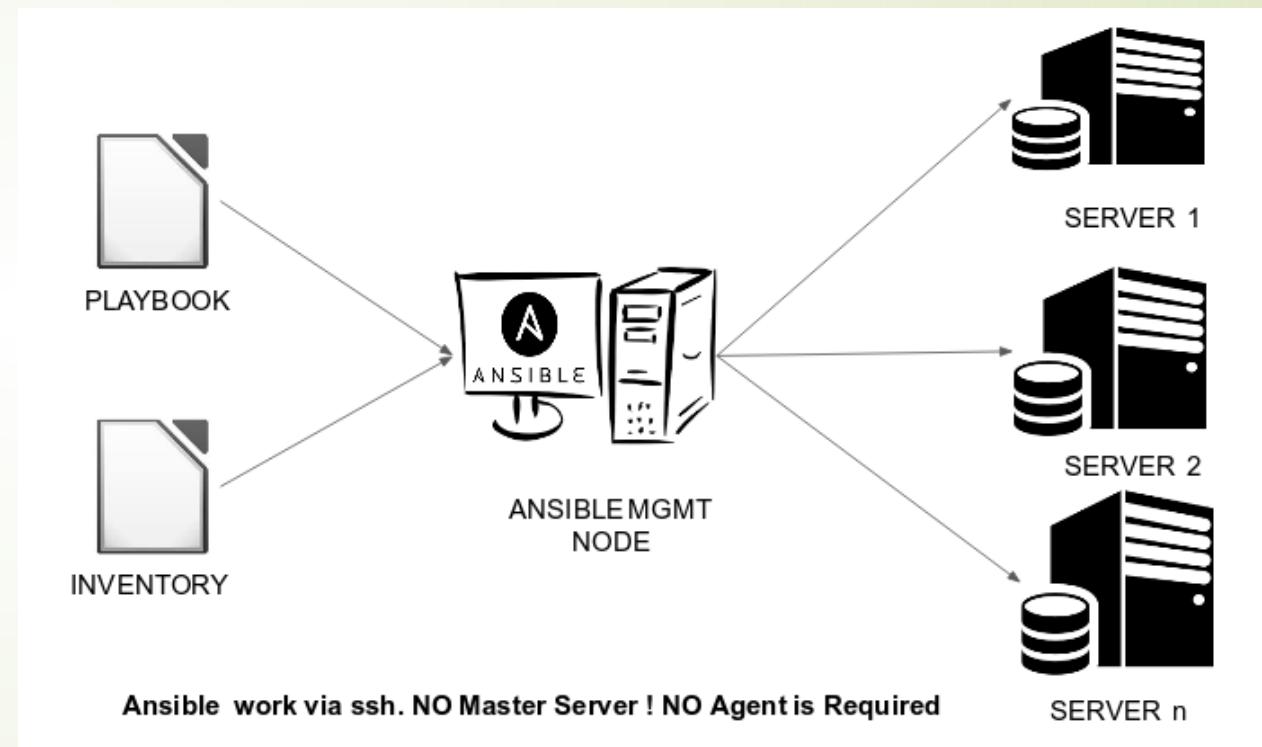
Ansible

- Présentation

Solution Open Source d'automatisation de déploiement et de configuration de logiciels sur machines distantes, dites noeuds

Solution écrite en python

Les noeuds peuvent être des machines, des Vms, des conteneurs...



Ansible

- Connexion ssh sécurisée



Ansible

- inventaire

```
production ansible_host=172.17.0.5

[dev]

[prod]
production

[prod:vars]
env_user=ansible
```

Un fichier inventaire contient les adresses des machines cibles,

Les déclarations de groupes de serveurs et leurs variables associées

Ansible

- Ligne de commande

```
ansible all -i my_inventory_file -m ping
```

all : toutes les machines, -i : fichier inventaire, -m module ping

=> effectuer un test de connexion sur toutes les machines de l'inventaire

```
ansible prod -i my_inventory_file -m command -a 'ls -ltr'
```

prod : groupe prod de l'inventaire, -a paramètre du module demandé

=> exécuter la commande 'ls -ltr' sur les machines du groupe prod de l'inventaire

```
ansible -m setup prod
```

=> afficher les variables prélevées du serveur prod (informations sur la cible)

Ansible

- Exemple de playbook Ansible

```
- name: BOOTSTRAP
  hosts: "{{ env_hosts }}"
  remote_user: "{{ env_user }}"
  gather_facts: false
  vars:
    dependencies:
      - curl
      - git

  tasks:
    - name: BOOTSTRAP | Check python install
      raw: test -e /usr/bin/python
      changed_when: false
      failed_when: false
      register: check_python

    - name: BOOTSTRAP | Install Python
      raw: test -e /usr/bin/apt && (apt -y update && apt install -y
      when: check_python.rc != 0

    - name: "Install dependencies : {{ item | capitalize }} (APT)"
      apt:
        name: "{{ item }}"
        state: "present"
        cache_valid_time: 10
        when: ansible_os_family == 'Debian'
        loop: "{{ dependencies| }}"
```

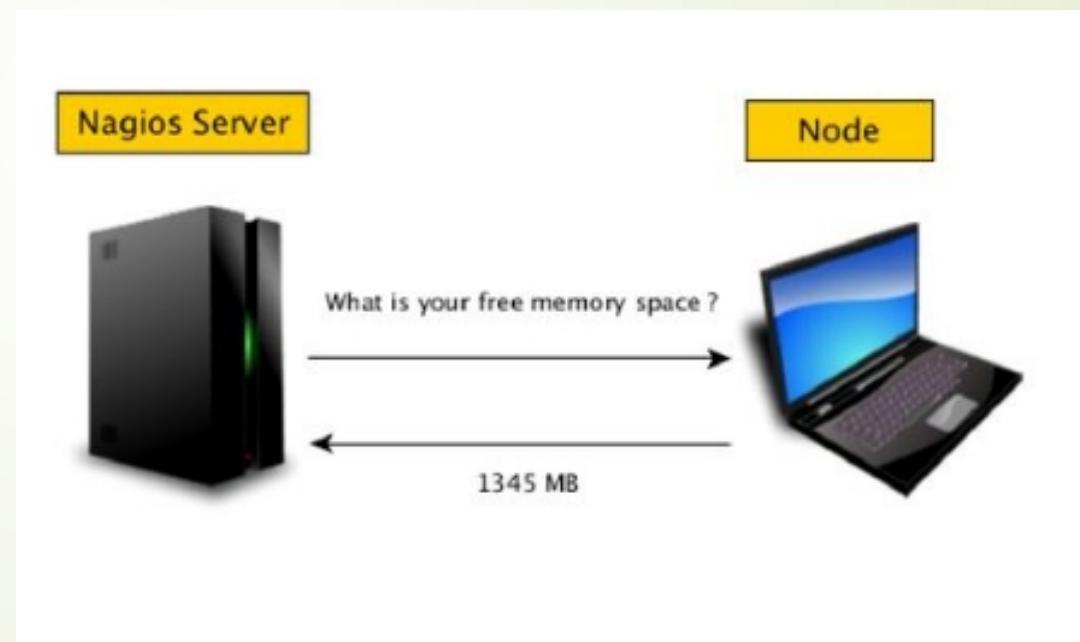
Fichier au format yaml
Décrivant les tâches à effectuer
ou contrôler sur les nœuds
distants, de manière déclarative et
idempotente

Utilisation du moteur de template
Python Jinja2 permettant de
configurer les playbooks
« {{ }} »

Infrastructure as code

Nagios

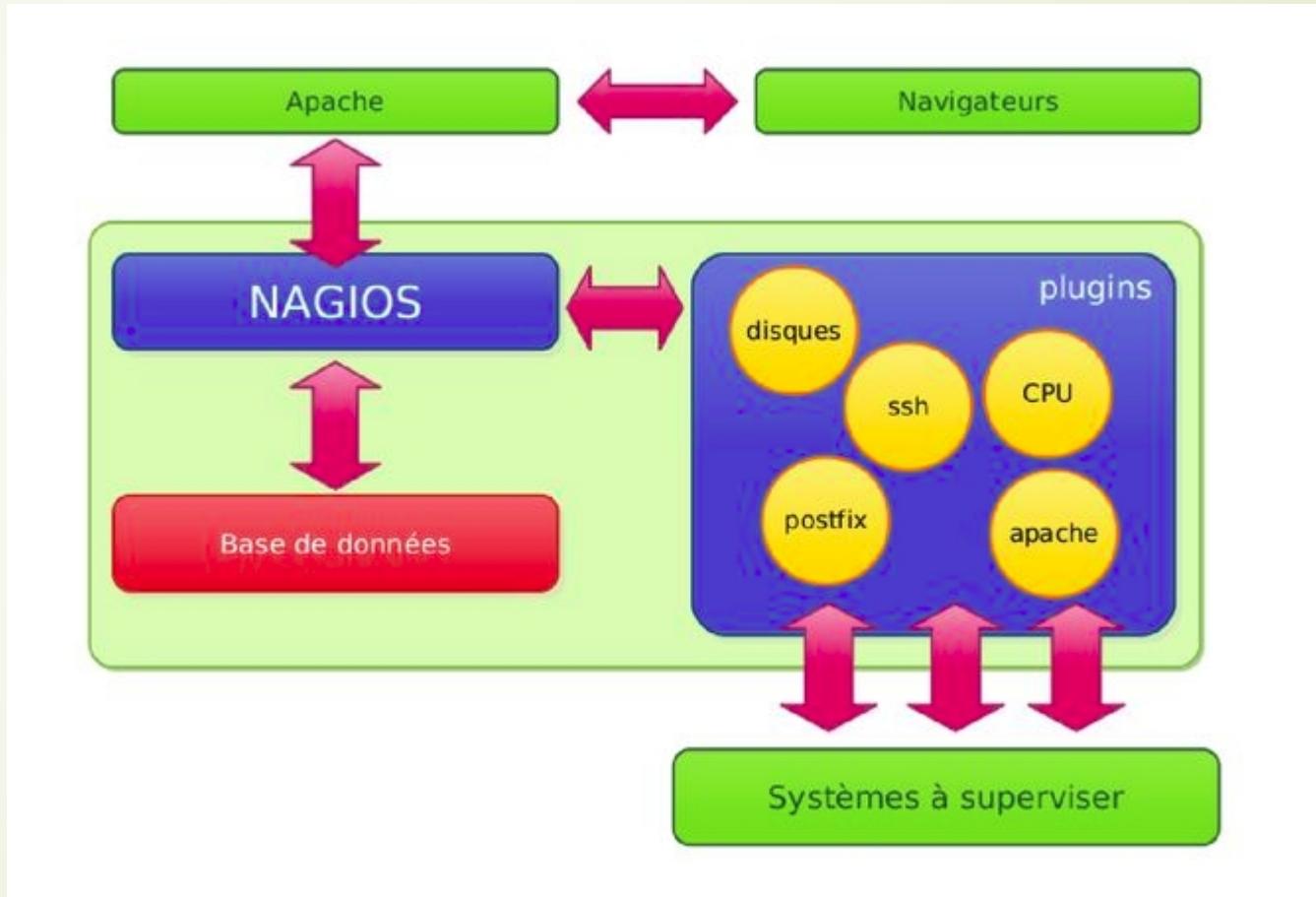
- Présentation
- Solution de monitoring client / serveur
- Composé de trois parties :
 - un ordonnanceur
 - des plugins
 - une interface web



Nagios

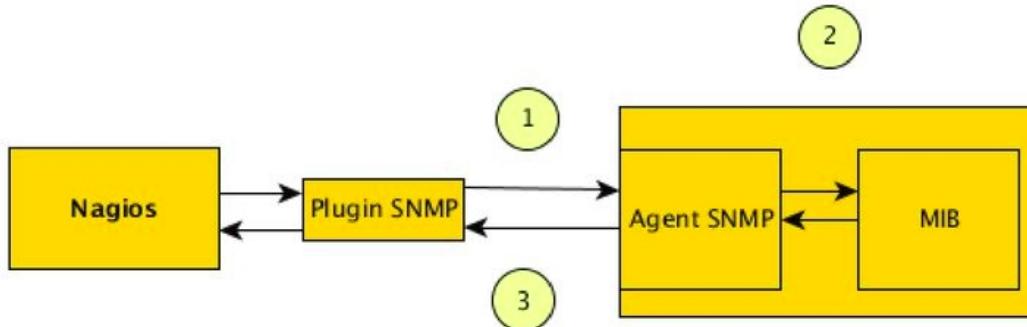
- Fonctionnement

- Utilisation de plugins pour récupérer les métriques
- Différents types d'interrogation
 - Active Check : SNMP /NRPE
 - Passive Check NSCA
- Active Check : Nagios initie la demande d'information
- Passve Check : le serveur est à l'écoute de scripts distants



Nagios

- Protocoles
- SNMP « Simple Network Management Protocol » : va lire les informations sur un matériel dans la MIB management information base, sur le port UDP 161



- 1. Requête SNMP
- 2. Récupération de l'objet désiré dans la MIB
- 3. Réponse SNMP

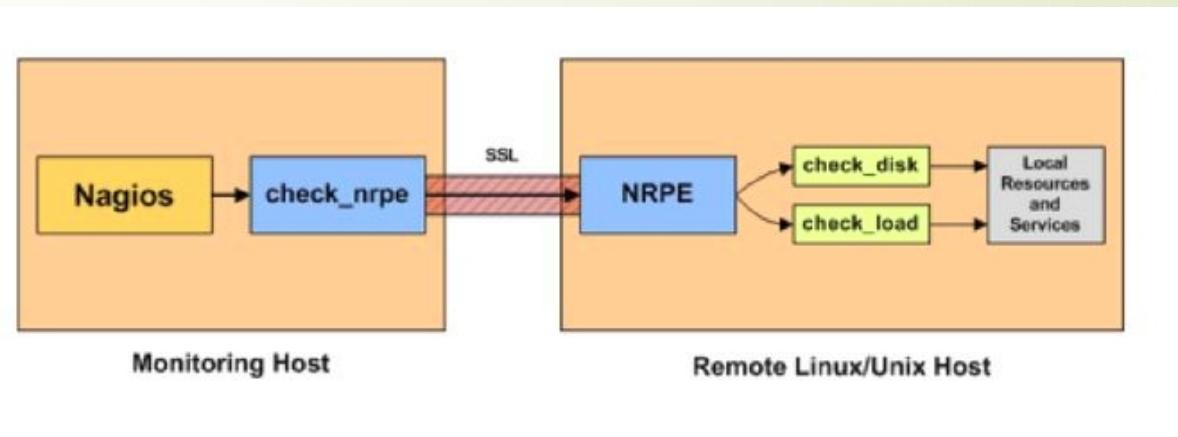
Nagios

- Protocoles

- Nagios Remote Plugin Executor : un plugin va exécuter un démon installé sur un serveur distant. Ce démon va exécuter un script de prélèvement de métrique et retourner les résultats aux plugins

- Valeurs de retour des plugins

0	=> OK
1	=> WARNING
2	=> CRITICAL
3	=> UNKNOWN



- NSCA : Vérification passive : Nagios ne fait qu'écouter en attente d'informations de clients