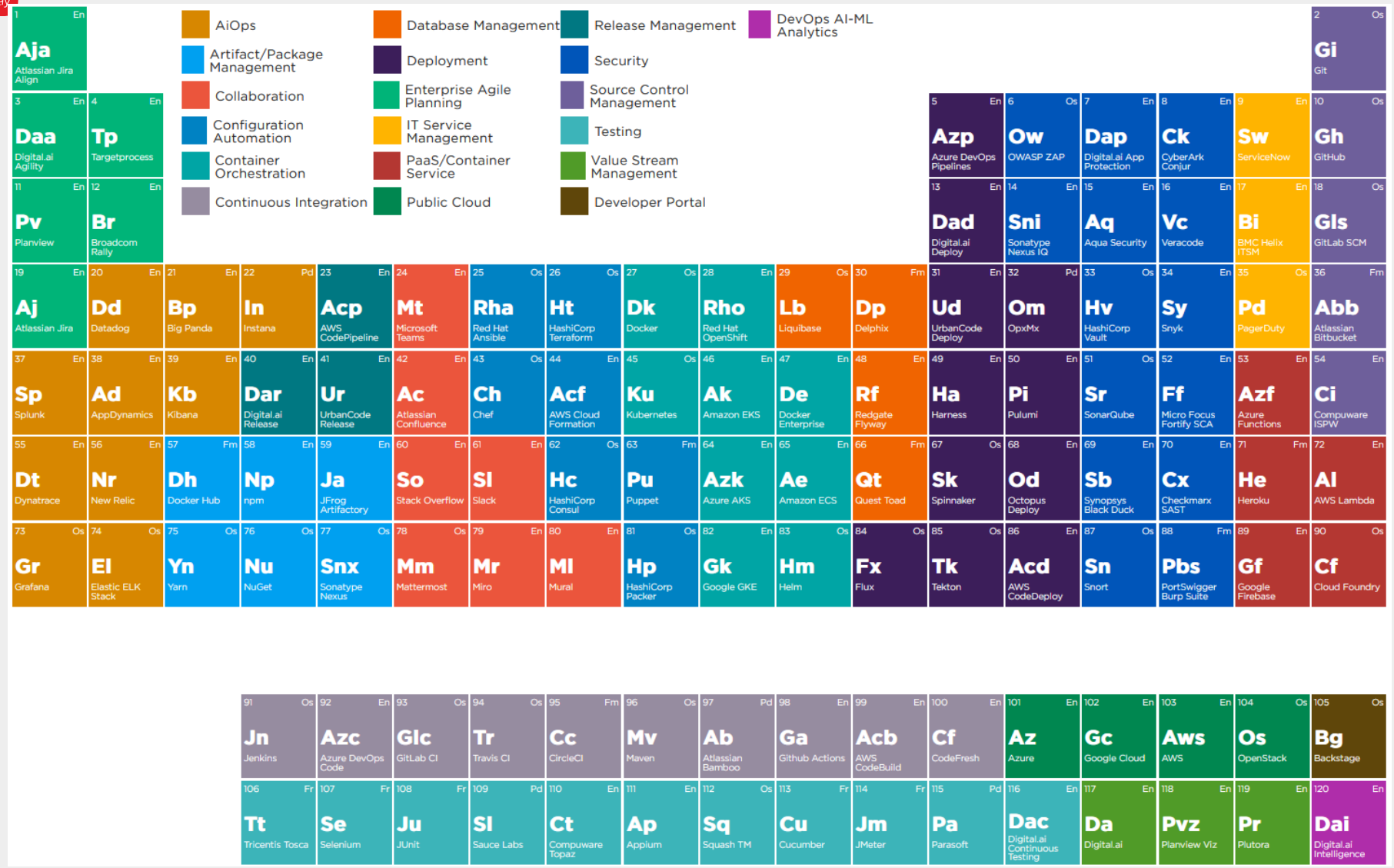
PROCESSUS DEVOPS

I. Introduction: chaine d'outils
II. Tests fonctionnels
III. Analyse Statique de code
IV. Déploiement automatique
V. Test d'acceptations
VI. La Production
VII. KPIs

I. Introduction: chaine d'outils



“ **https://digital.ai/fr/learn/devsecops-periodic-table/** ”

Terminologie importante

- **Artefact**

  Tout élément d'un projet de développement logiciel, y compris la documentation, les plans de test, les images, les fichiers de données et les modules exécutables

- **Interface de programmation d'application (API)**

  Ensemble de protocoles utilisés pour créer des applications pour un système d'exploitation spécifique ou comme interface entre des modules ou des applications

- **Microservices**

  Une architecture logicielle composée de modules plus petits qui interagissent via des API et peut être mise à jour sans affecter l'ensemble du système. Ceci est connu comme un couplage lâche

- **Virtualisation de système d'exploitation (OS)**

  Une méthode pour diviser un serveur en plusieurs partitions appelées "conteneurs" ou "environnements virtuels" afin d'empêcher les applications d'interférer les unes avec les autres

- **Conteneurs**

  Une façon de regrouper des logiciels dans des packages légers, autonomes et exécutables, incluant tout le nécessaire pour l'exécuter (code, exécution, outils système, bibliothèques système, paramètres) aux fins de développement, d'envoi et de déploiement.

- **Open source**

  Logiciel distribué avec son code source afin que les organisations d'utilisateurs finaux et les fournisseurs puissent le modifier à leurs propres fins

- **Machine learning**

  Analyse de données utilisant des algorithmes qui apprennent des données

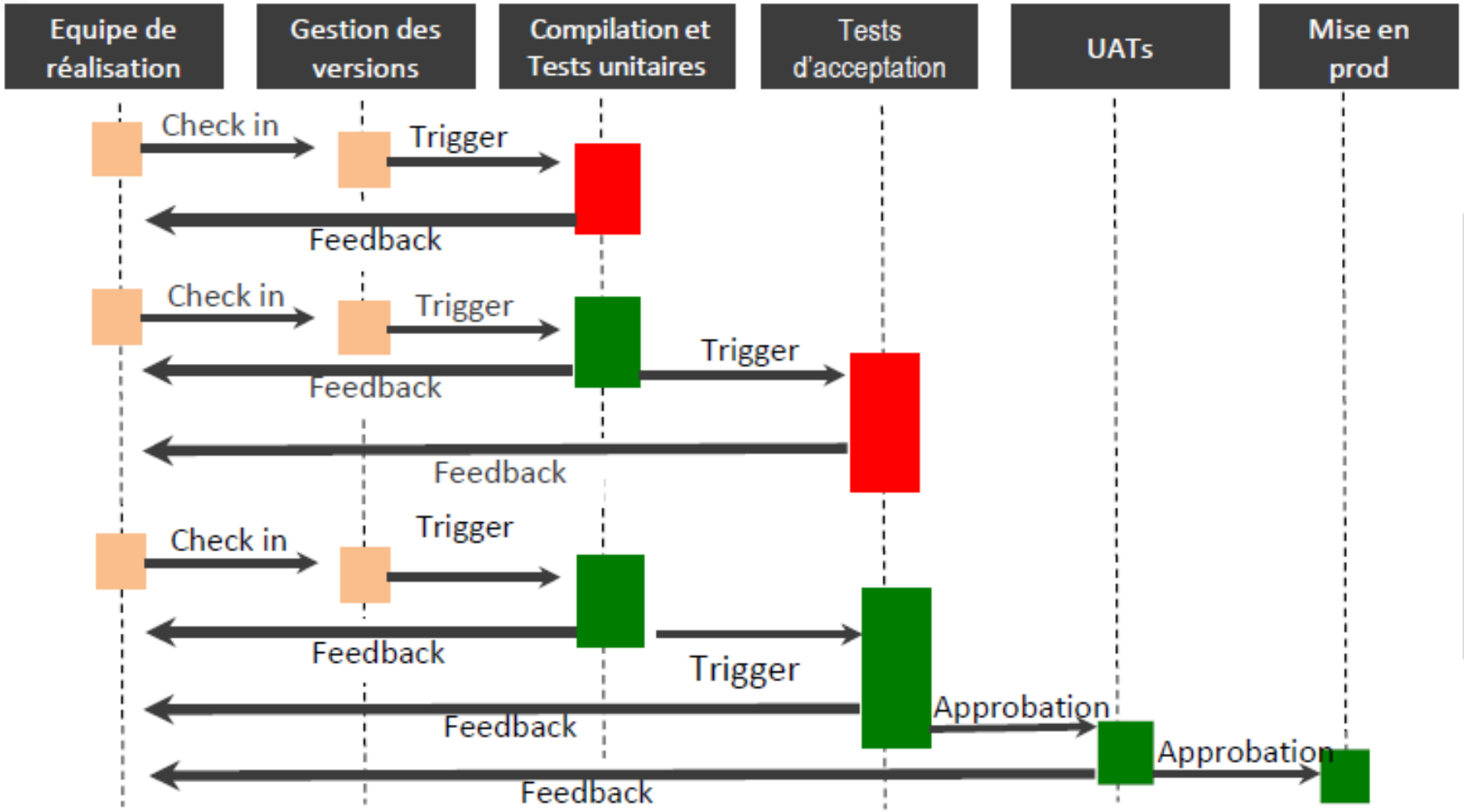
Pratiques d'automatisation DevOps

Une philosophie de toochain implique l'utilisation d'un ensemble intégré d'outils complémentaires spécifiques aux tâches pour automatiser les processus de livraison et de déploiement de bout en bout.

- Toolchain (par opposition à une solution à fournisseur unique)
- Outils partagés
- En libre service
- Prévoir l'architecture du logiciel de manière à permettre :
  - Automatisation des tests
  - Surveillance
- Infrastructure en tant que code
- Expérimentation

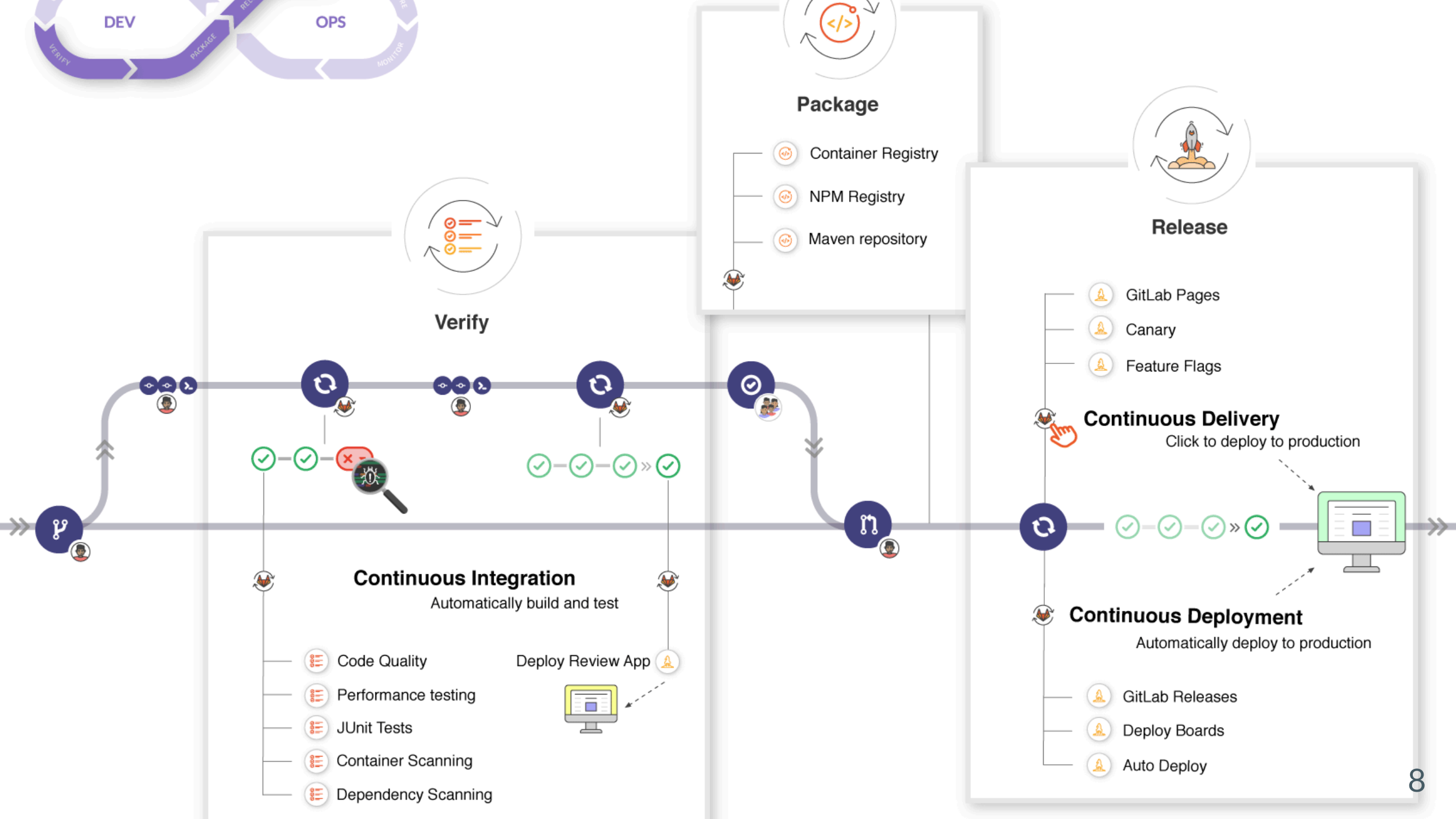
Eviter les outils qui renforcent les silos !

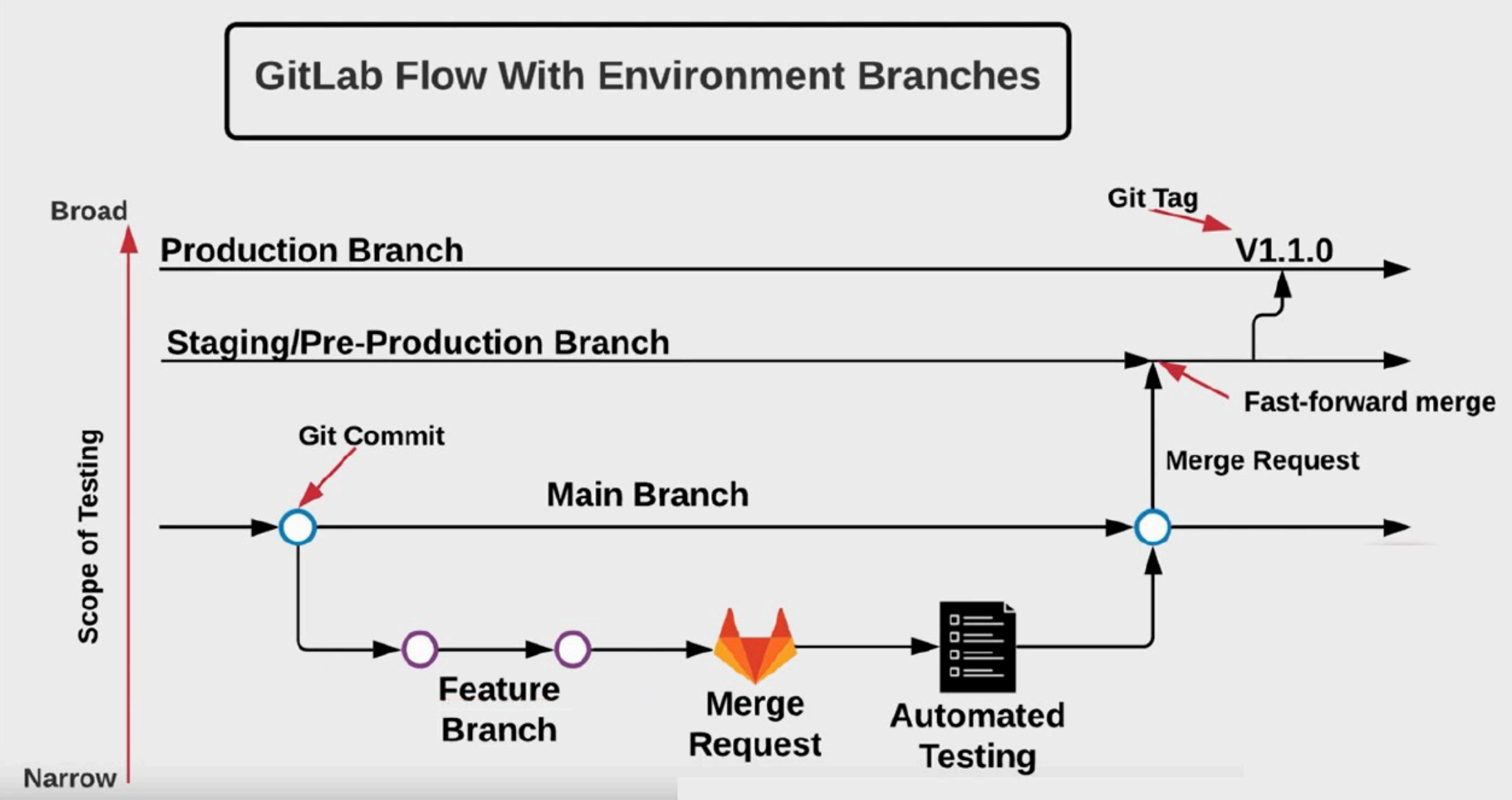
Le pipeline de déploiement



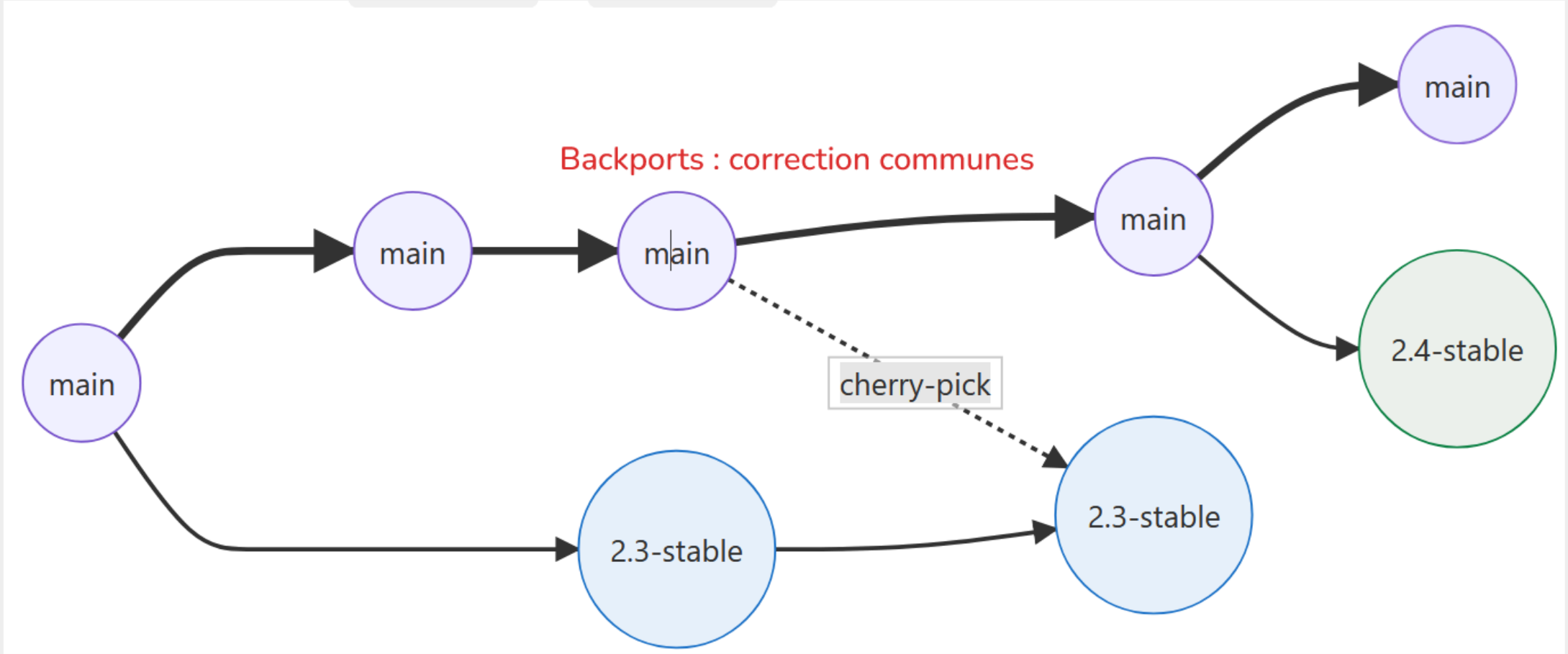
Le pipeline de déploiement est un processus automatisé permettant de gérer tous les changements, du check-in à la mise en production. Les toolchains couvrent des silos et automatisent le pipeline de déploiement.

Source : *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*


DEV
OPS
VERIFY
PACKAGE
MONITOR
Package
Container Registry
NPM Registry
Maven repository
Release
GitLab Pages
Canary
Feature Flags
Verify
Continuous Delivery
Click to deploy to production
Continuous Integration
Automatically build and test
Code Quality
Performance testing
JUnit Tests
Container Scanning
Dependency Scanning
Deploy Review App
Continuous Deployment
Automatically deploy to production
GitLab Releases
Deploy Boards
Auto Deploy

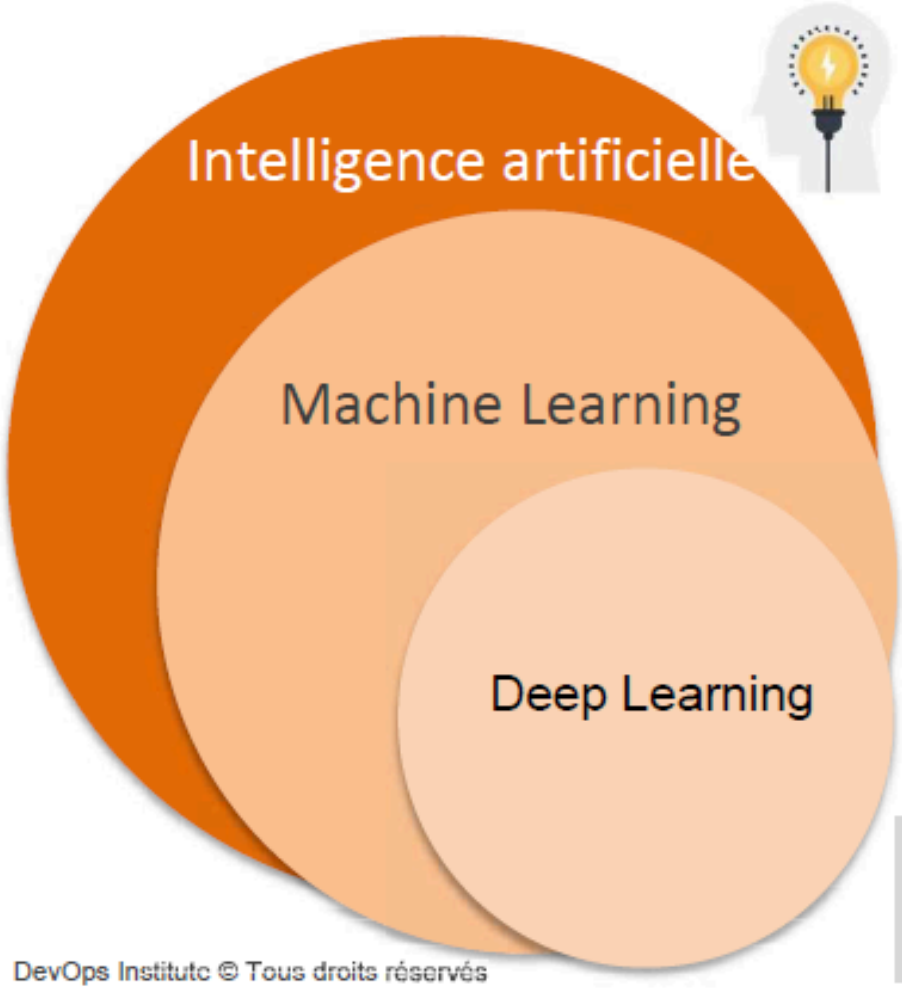

GitLab Flow With Environment Branches
Broad
Scope of Testing
Narrow
Git Tag
V1.1.0
Production Branch
Staging/Pre-Production Branch
Fast-forward merge
Merge Request
Git Commit
Main Branch
Feature Branch
Merge Request
Automated Testing

gitlab flow: trunk based avec branches de releases



IA & Machine Learning

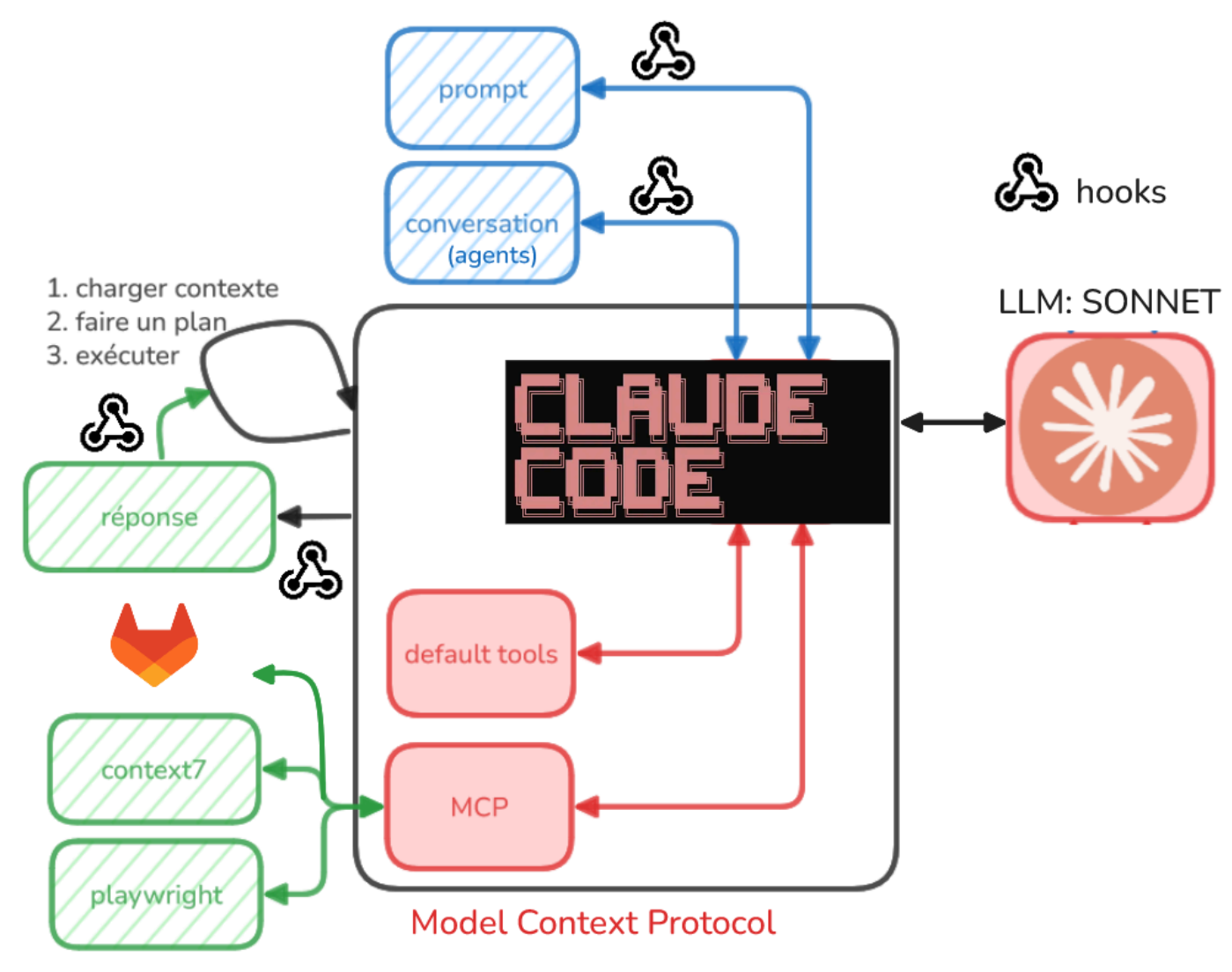
Donner aux ordinateurs la possibilité "d'apprendre" avec des données, sans être explicitement programmé.



- Les organisations collectent plus de données que jamais
- Il est difficile d'extraire pleinement la valeur de ces données.
- La science des données est une discipline de plus en plus populaire
- L'intelligence artificielle et l'apprentissage automatique permettent une analyse prédictive
- Peut trouver des tendances et des corrélations que les humains n'auraient pu trouver
- Augmente la contribution humaine
- Augmente la productivité
- Boucles de feedback automatisées

Définition : Analyse de données qui utilise des algorithmes pour apprendre des données.

exemple d'assistant IA: claude code



II. Tests fonctionnels

testCase

- fonction ou classe effectuant un test
- les **testCase** sont *découverts par un algorithme fonction* de
  - nom d'un dossier parent - *Ex: tests/*
  - nom du fichier - Ex: *test_*.py*
  - nom de classe ou de méthode/fonction ...
- l'alogrithme de découverte est configuré dans un fichier
- les **testCase** peuvent être aussi collectés par un objet **TestSuite**

procédure de test

- **A**rrange: créer le contexte de test
  - instanciations, connexions, descripteurs de fichier, ...
- **A**ct: exécuter le code à tester
- **A**ssert: **Δ** entre les valeurs *retournée et attendue*
- **C**leanup : libération de variables, fermeture de connexions ...

fixtures

- toute ressource et par prolongement une **fonction/méthode** qui retourne une **ressource nécessaire au test**
- les fixtures peuvent être **paramétrisées**
- les valeurs des paramètres peuvent être injectés depuis des *jeux de données* ou **Providers**

Mock ou MonkeyPatch

- objet qui **simule l'interface publique**
  - d'un objet *nécessaire au test* **mais pas objet du test**,
  - à la fois *lent ou compliqué à implémenter* dans le test
- les **valeurs de retour du Mock** sont programmées pour être
  - rapides et
  - attendues aux cas de test

assertions

- la grande majorités sont des évaluations booléennes dont le résultat signifie
  - true => **success** *"."*
  - false => **fail** *"F"*
- **xfail:** dans certains cas on peut inverser la logique
  - false => **success** *« eXpected to Fail !! »*
- le retour attendu peut être une **exception levée**

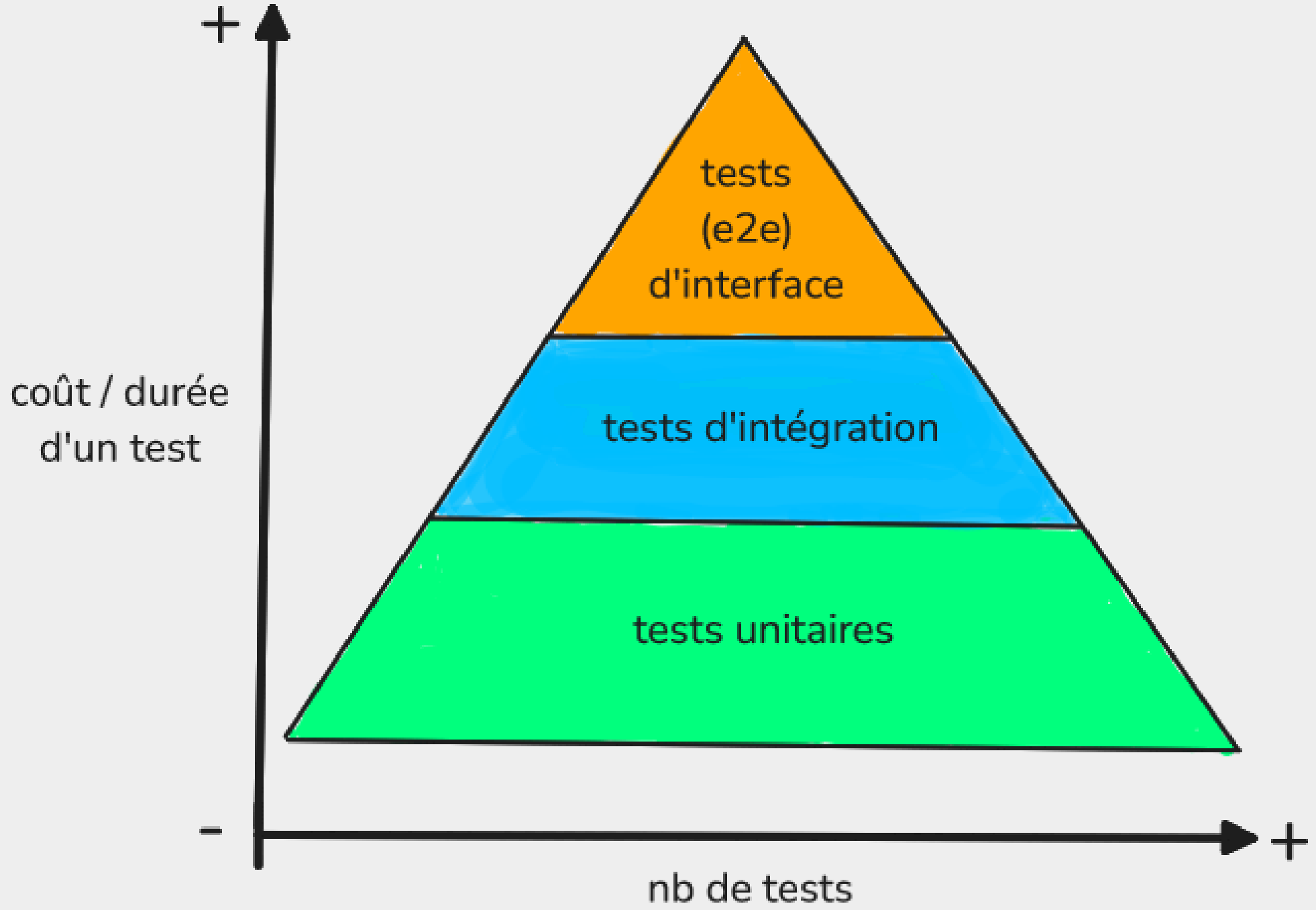
rapport de test

- par défaut, le processus de test retourne une *sortie en texte*
- on peut demander cette sortie dans différents formats
  - html - *visualisation*
  - **xml au format JUnit** - *interface avec un outil CICD*

couverture de code

> *ISTQB:* « Degré, exprimé en pourcentage, selon lequel un élément de couverture spécifié a été exécuté lors d'une suite de test »

- ratio: `(nb d'éléments testés / nb d'éléments testables) * 100`
- type d'éléments, i.e type de *couverture de code par*
  - les **méthodes**: méthode rencontrée *au - 1x par le test*
  - les **instructions**: ligne de code // => **simple et visualisable**
  - les **chemins**: flux de lignes de codes possibles - *trop complexe*


+
tests
(e2e)
d'interface
coût / durée
d'un test
tests d'intégration
tests unitaires
-
+
nb de tests

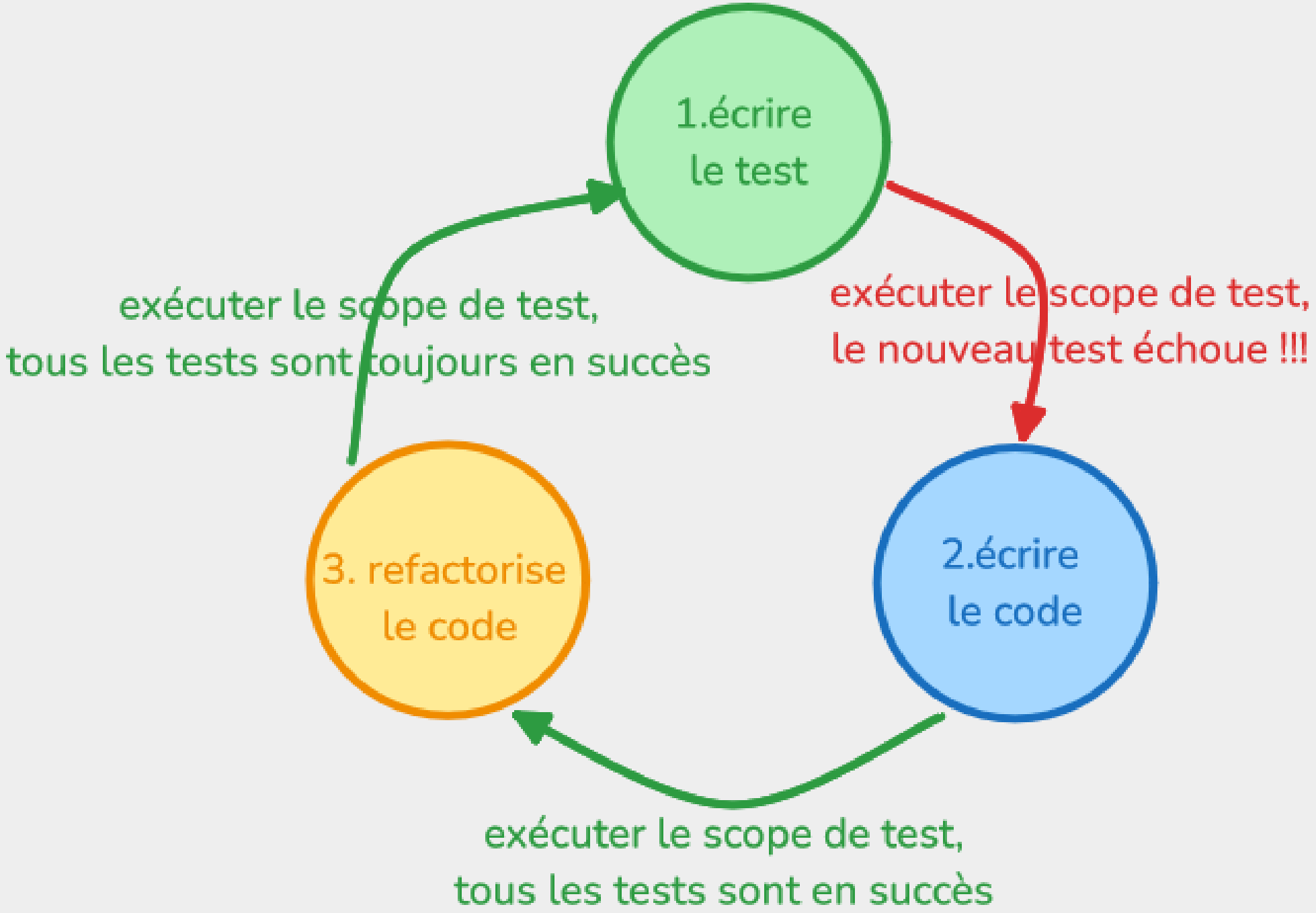
tests unitaires

- les **unités** sont les éléments de code les plus fins
  - en impératif/fonctionnel: **les fonctions**
  - en objet: **les méthodes**
- traditionnellement on écrit un *code à partir d'une spécification*
  => *on en déduit un test*
- **en TDD**, on écrit un **test à partir d'une spécification**
  => **on en déduit le code** de l'unité

> écrire les tests au plus tôt est un élément emlématique d'une démarche agile dite *« Shift Left »*

TDD

- ou « **T**ests **D**riven **D**evelopment »: développement piloté par les tests
- « *scope* » ou périmètre => ensemble de *tests existants*
- pour chaque *cas d'utilisation* de la fonctionnalité
  1. écrire le test + exécuter le scope => le **test échoue !**
  2. écrire le cas + exécuter le scope => le *test est en succès*
  3. réfactoriser le code + exécuter le scope => le *test est en succès*


1.écrire
le test
exécuter le scope de test,
le nouveau test échoue !!!
2.écrire
le code
exécuter le scope de test,
tous les tests sont en succès
3. refactorise
le code
exécuter le scope de test,
tous les tests sont toujours en succès

tests d'intégration

- tests des flux informatiques individuels hors exécution de l'application.
- ces flux sont décrits par des **spécifications fonctionnelles**
  - => *vision métier !*
  - écrites sous forme d'*« User Story »*
- séquencent les fonctionnalités unitaires déjà *testés en TDD*

> on veut donc tester l'assemblage des unités !!

BDD

- « **B**ehaviour **D**riven **D**ev. » Dev piloté par les comportements
- un comportement est une **spécification fonctionnelle** écrite sous forme d'*User Story US*

1. l'US est formalisée en utilisant le **langage Gherkin** dans un fichier *.feature*
   - une feature est composée de **scénarios** eux mêmes composés de **triplets Given | When | Then**
2. l'écriture du test reprend la composition de la feature
3. le code écrit et refactorisé *comme en TDD*

exemple de Gherkin

```
# withdraw.feature
Feature: un client retire une somme
   En tant que client
   Je veux retirer une somme de mon compte dans un distributeur
   même quand la banque est fermée

   Scenario Outline: le compte est solvable
      Given le compte a <balance> euros
      And le distributeur a au - <amount> euros
      When je demande <amount> euros
      Then j'obtients <amount> euros
      And mon solde est à <new-balance> euros
      Examples:
      | balance | amount | new-balance
      | 100     | 20     | 80
```

décomposition du test en « steps »

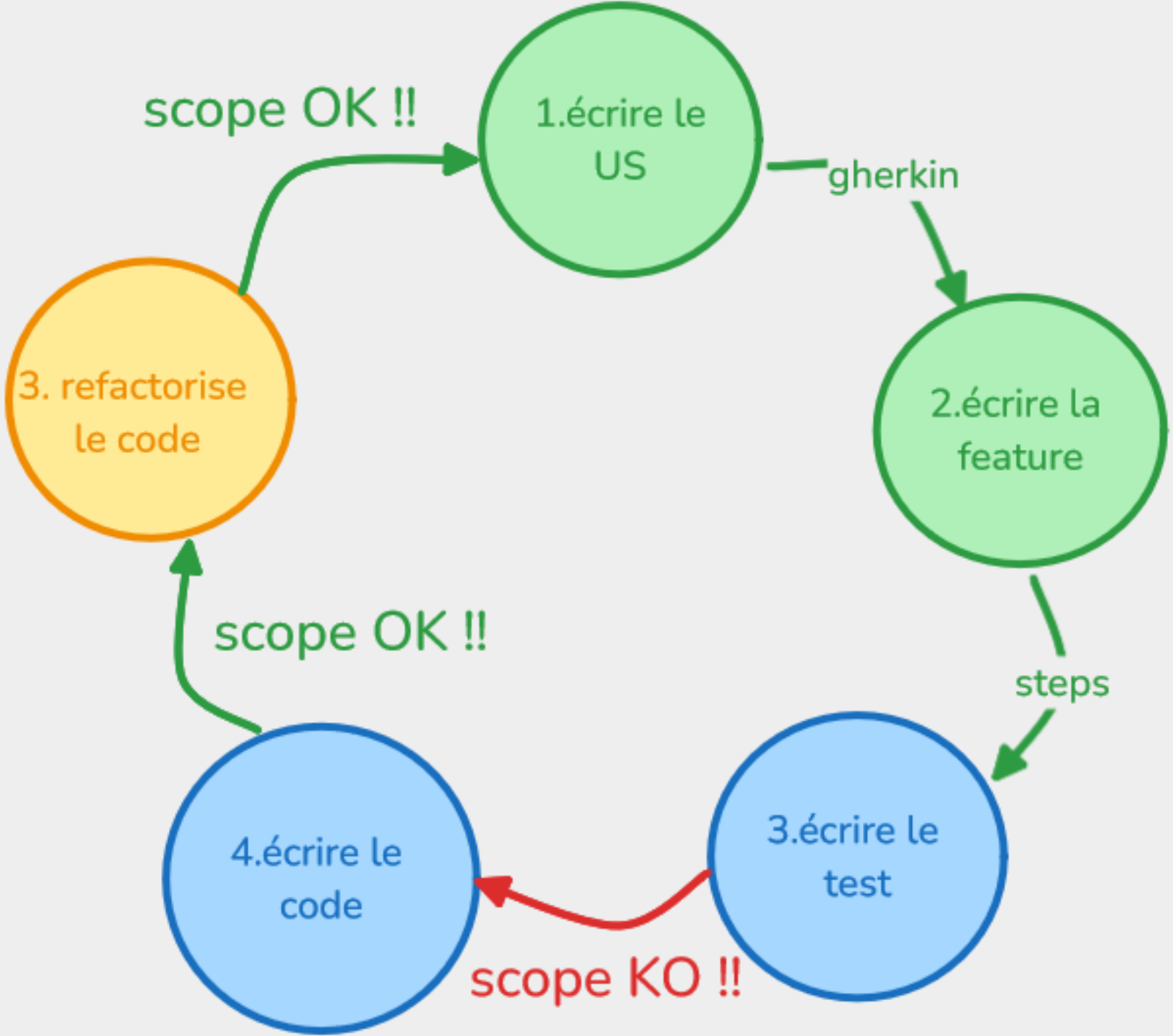
```
@scenario('withdraw.feature', 'le compte est solvable')
def test_withdraw(): pass

@given(parsers.parse("le compte a {balance:int} euros"))
def user_account(account, balance): account["balance"] = balance

@given(parsers.parse("le distrib a au - {amount:int} euros"))
def atm_balance(atm, amount): atm["qty"] = amount

@when("je demande ...")
def witdraw(account, atm, amount): account.withdraw(amount, atm)

@then("mon solde ...")
def assert_new_balance(account, new_balance):
    assert account.balance == new_balance
```

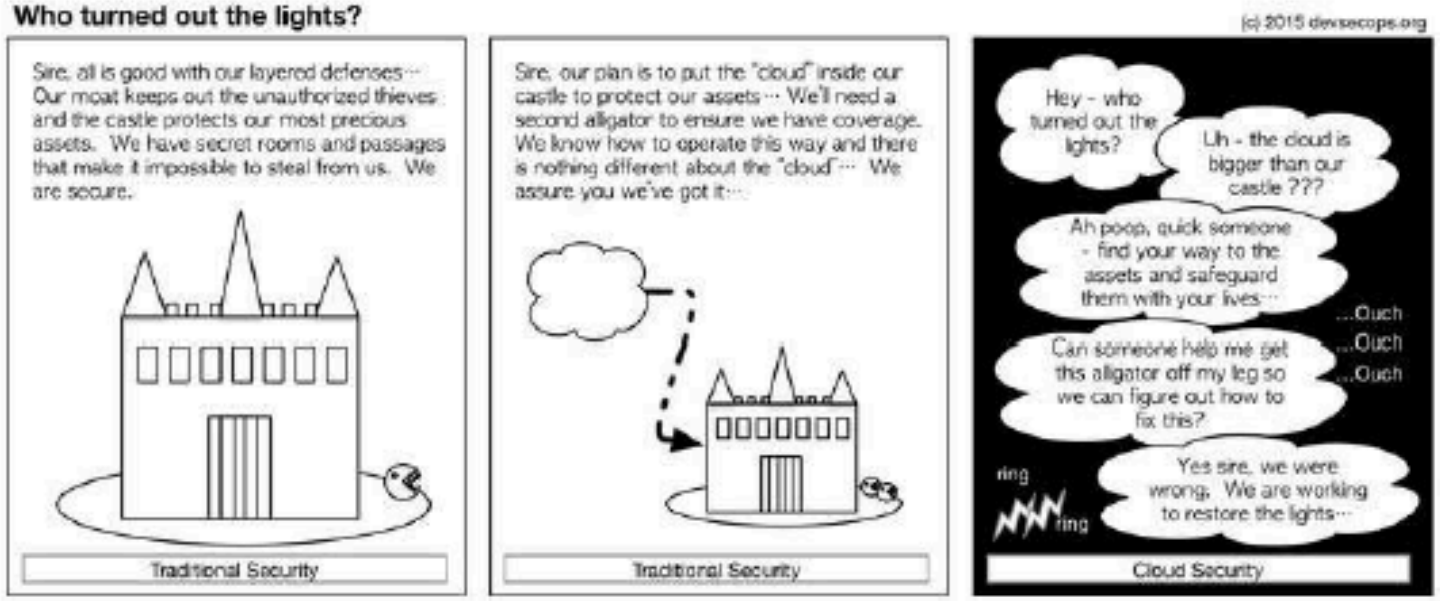

scope OK !!
1.écrire le US
gherkin
2.écrire la feature
steps
3.écrire le test
scope KO !!
4.écrire le code
scope OK !!
3. refactorise le code

III. Analyse Statique de code

DevSecOps

DevSecOps a pour but de s'appuyer sur le principe selon lequel « tout le monde est responsable de la sécurité » afin de répartir les prises de décision en matière de sécurité au niveau des personnes qui possèdent les compétences les plus adéquates.

www.devsecops.org



- Introduit la sécurité sous forme de code
- Adopte la stratégie de test « décalage vers la gauche »
- Utilise l'automatisation pour la résilience, les tests, la détection et l'audit
- Brise la contrainte de sécurité

conventions de code

- **règles** + ou - arbitraires relatives à un langage, une organisation
- **vars:** PascalCase, camelCase, snake_case, kebab-case, ALL_CAPS
- **aération:** saut de lignes, indentations, espaces, marge à droite
- documentation, commentaires, TODOs
- Règles *empiriques* : pas + que
  - 80 caractères par ligne
  - 25 lignes par méthode

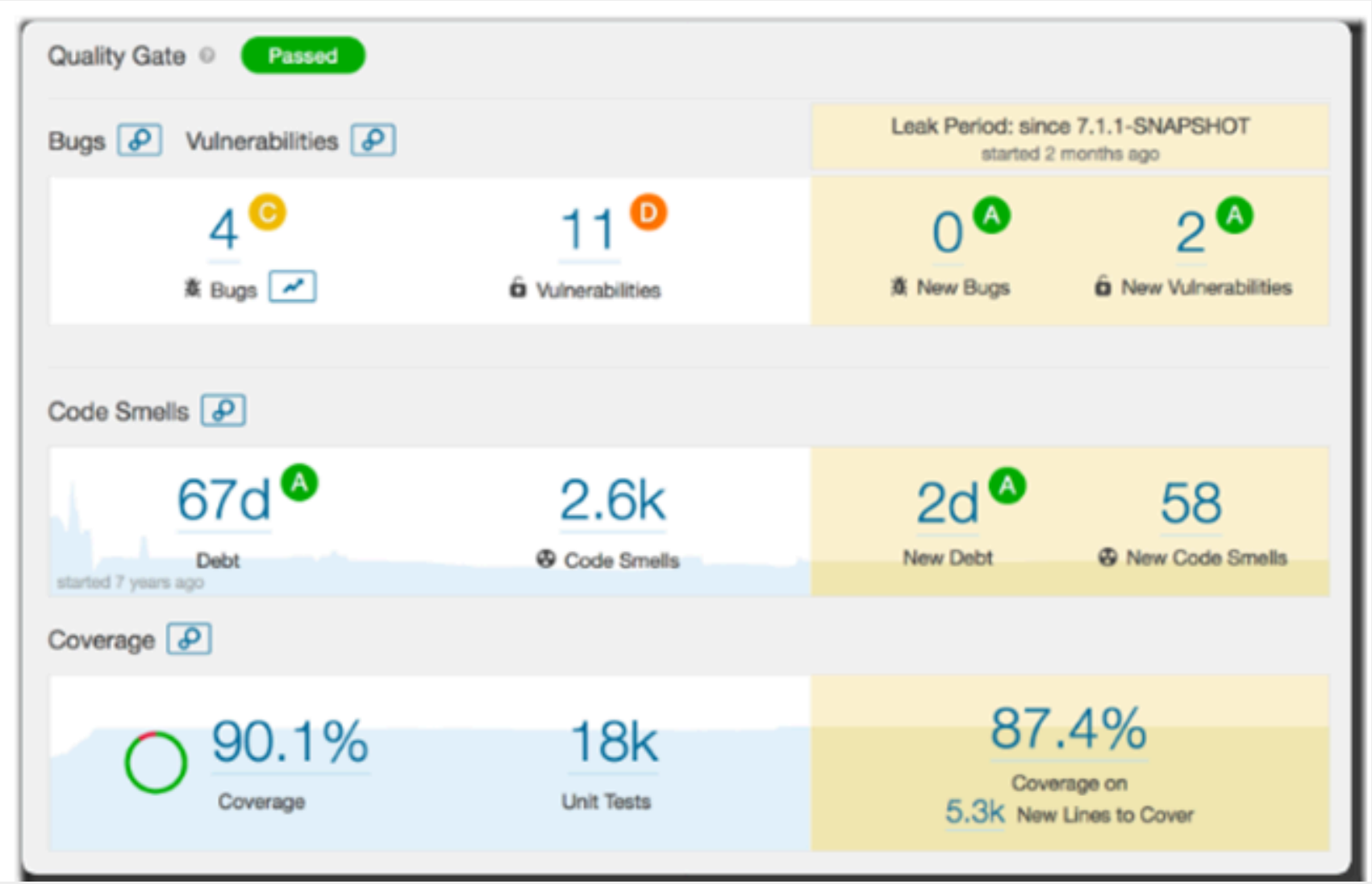
formatters

- outil capable de **reformater un code** selon les *conventions de code*
- conventions de code *configurables par un fichier*
- usage commun: exécuter un *formatter avant un commit git*
  - utilisation d'un git **hook pre-commit**

SAST

- ou « **S**tatic **A**pplication **S**ecurity **T**esting »
- analyse du code source de l'application pour trouver des:
  - *Bugs*: mauvais code susceptible de générer des bugs
  - *Code Smells*: idem mais pénalise la **maintenabilité**
  - *Security Hotspots*: idem susceptible de générer des failles
  - *Vulnérabilités*: **failles de sécurité** avérées

analyse de qualité



seuil de validation

- par défaut pour les **Merge Requests**

Conditions

Add Condition

Conditions on New Code

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A (Technical debt ratio is less than 5.0%)
Reliability Rating	is worse than	A (No bugs)
Security Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A (No vulnerabilities)

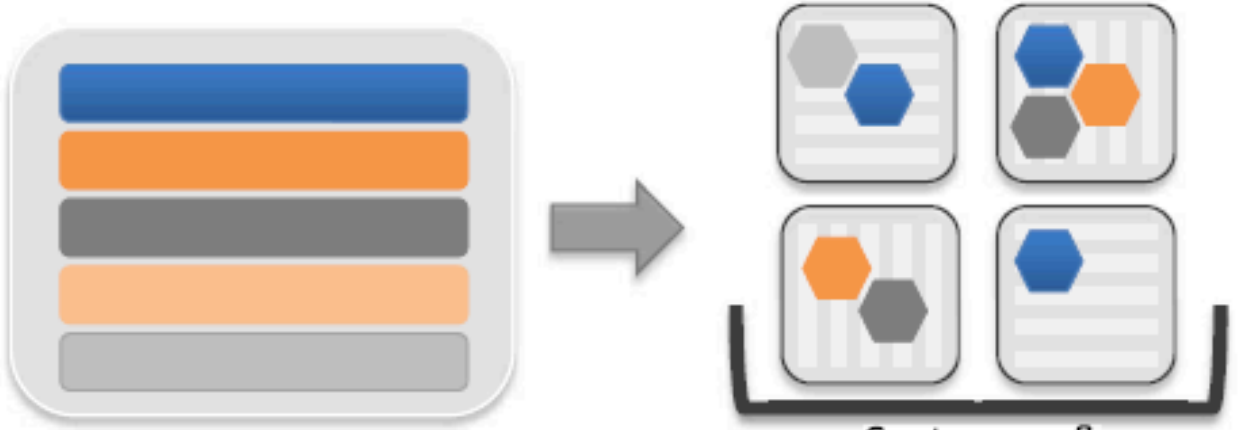
IV. Déploiement automatique

Cloud, conteneurs et microservices

Cloud computing : pratique consistant à utiliser des serveurs distants hébergés sur Internet, pour héberger des applications plutôt que des serveurs locaux dans un centre de données privé.

aws Azure Google Cloud Platform

Les équipes qui adoptent les caractéristiques essentielles du Cloud ont 24 fois plus de chances d'être des élites.



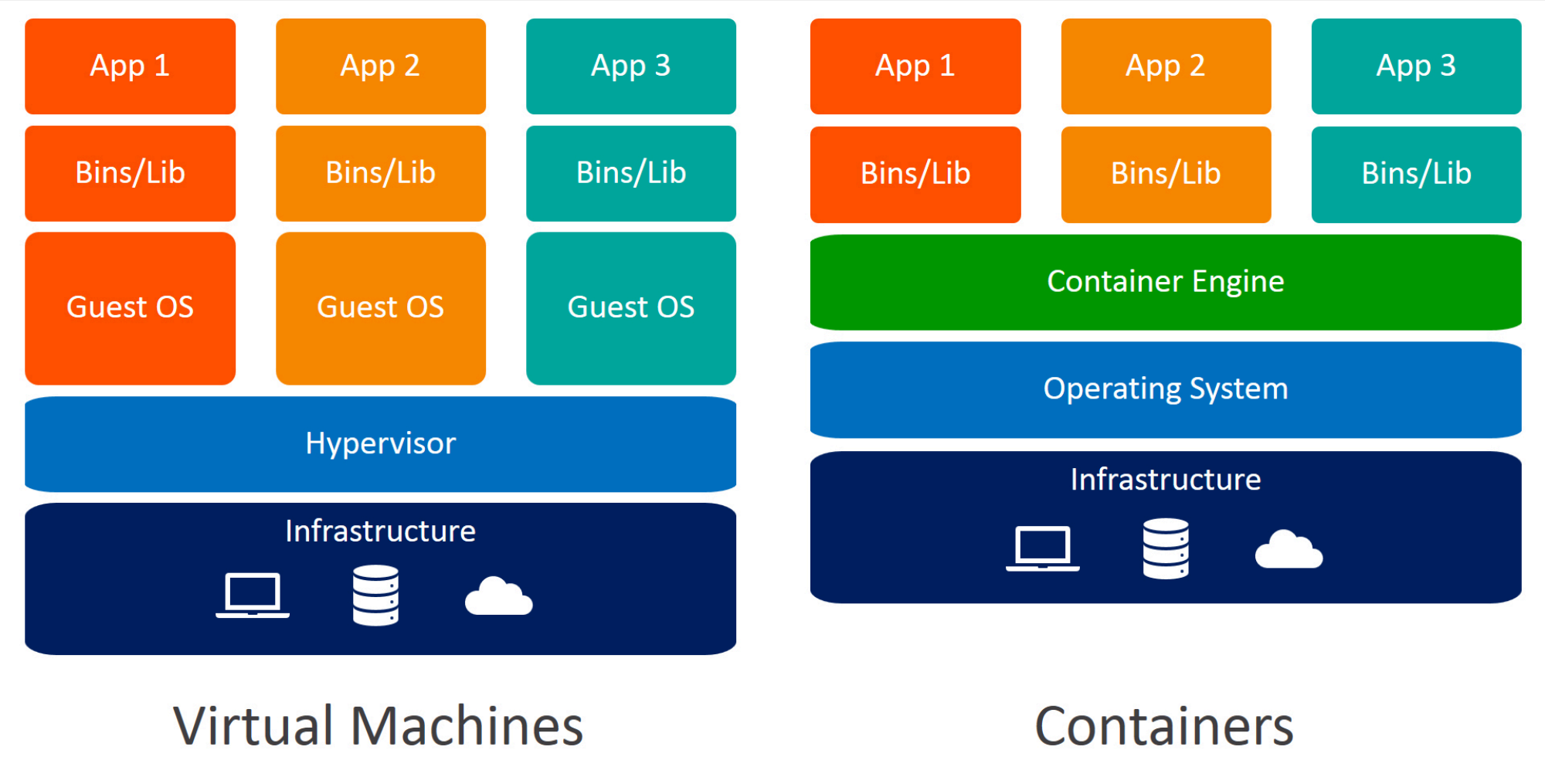
Monolithic

Conteneurs & Microservices

docker est un outil conçu pour faciliter la création, le déploiement et l'exécution d'applications à l'aide de conteneurs. Les conteneurs permettent à un développeur d'embarquer une application avec toutes les parties dont il a besoin, telles que des bibliothèques et autres dépendances, et de l'envoyer dans un seul package.

kubernetes est un système open source permettant de gérer des applications conteneurisées sur plusieurs hôtes, fournissant des mécanismes de base pour le déploiement, la maintenance et la mise à l'échelle des applications

unité d'exécution: VM vs conteneur



Infrastructure as Code (IaC)

> **Pilotage de l'état d'une infrastructure IT par des fichiers de configuration**

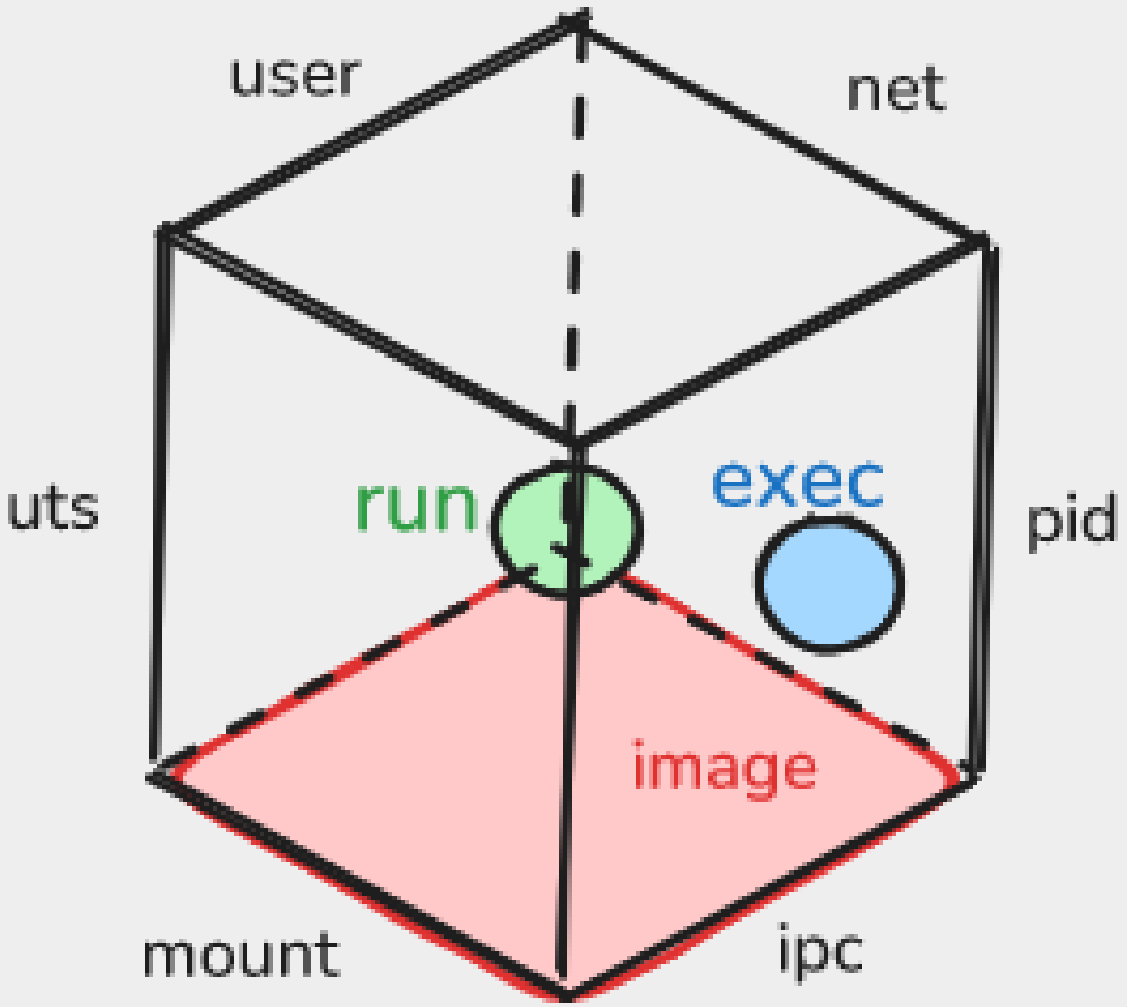
Langages de l'IaC

déclaratif: on décrit l'état final souhaité => **Besoin d'observabilité**

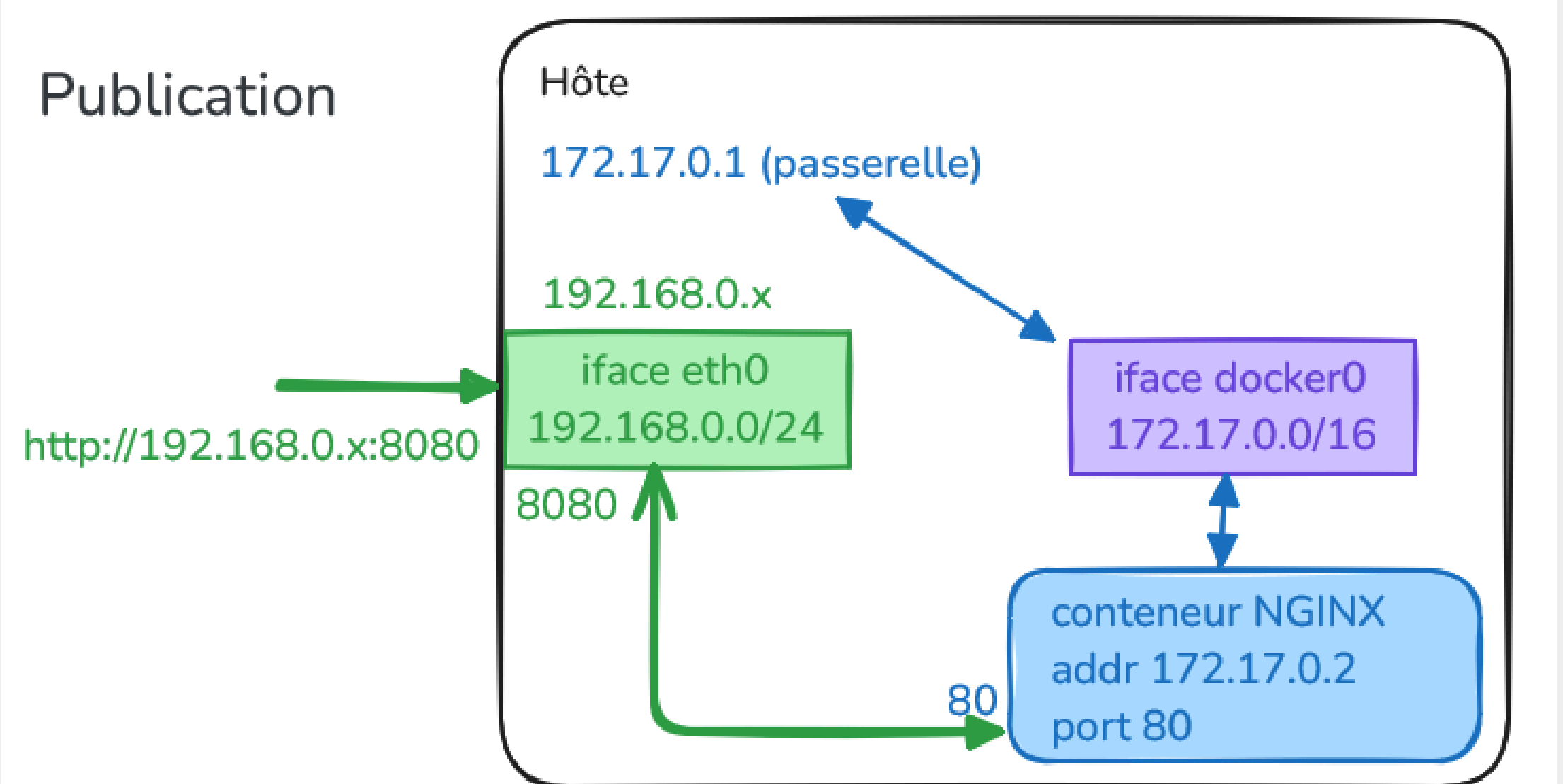
impératif: on décrit les étapes successives pour arriver à l'état final

conteneur docker

- processus isolé
- par des **namespaces** du noyau linux
- assis sur un *système de fichier* **"image"** délimité
- communique sur le **réseau**

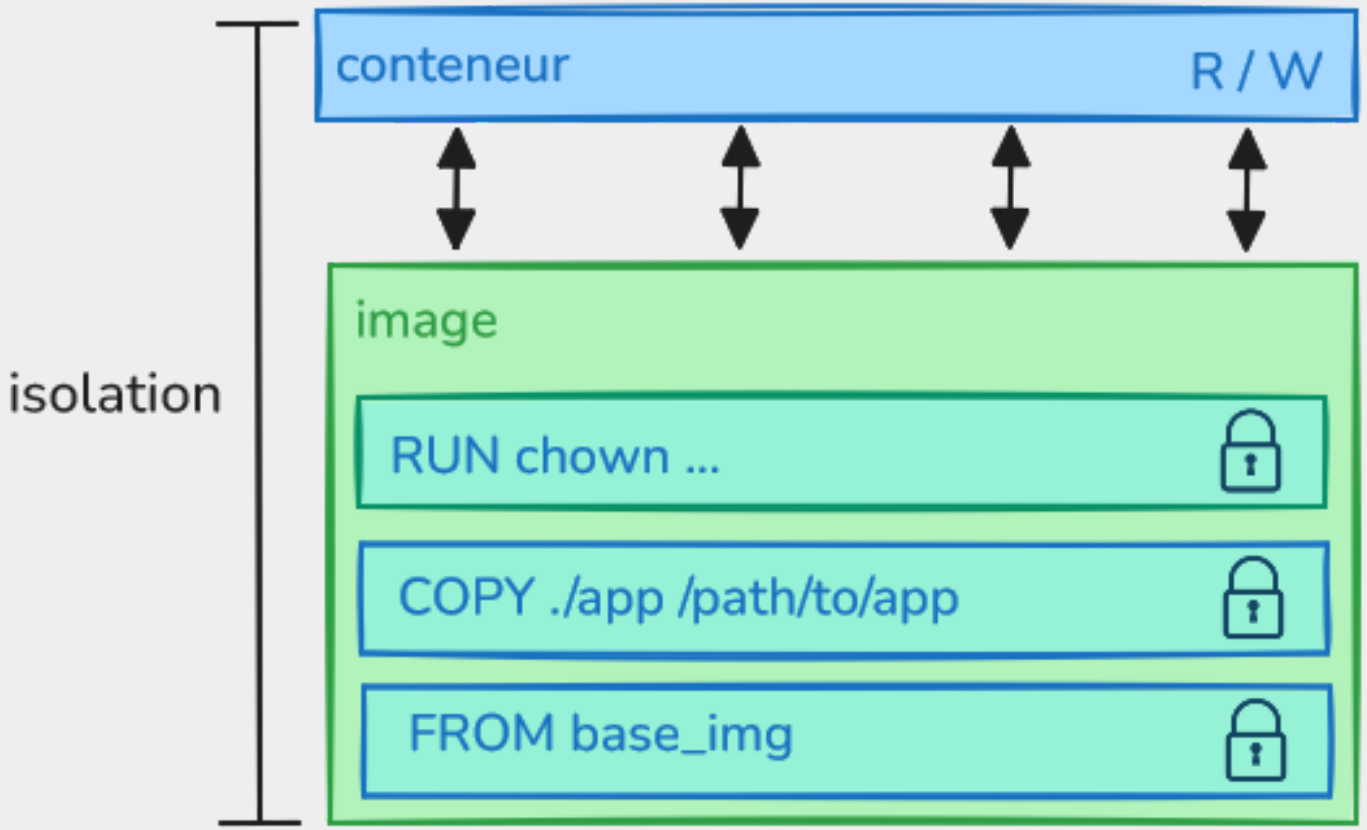


mise en réseau Docker

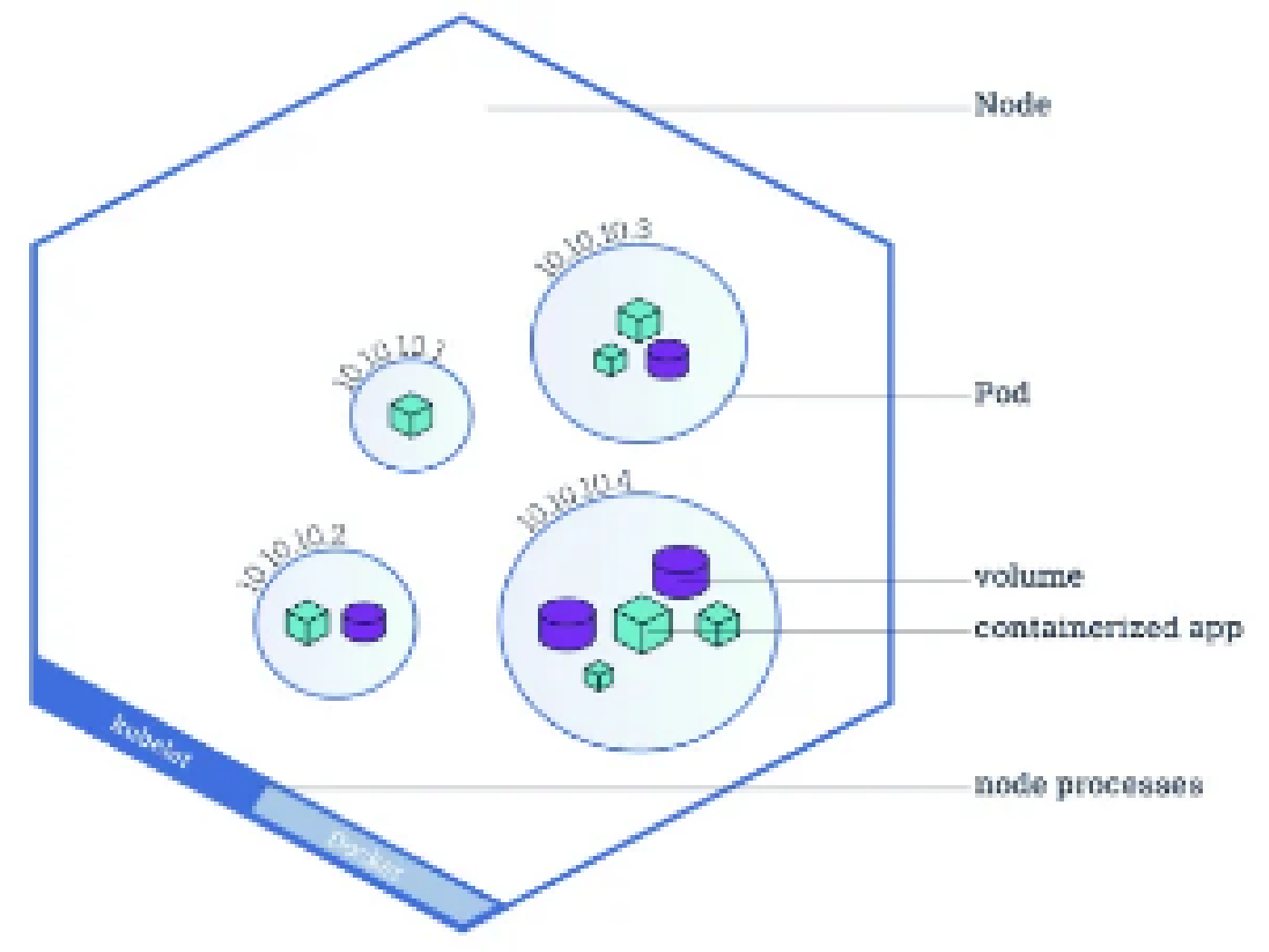


création d'une image: DOCKERFILE

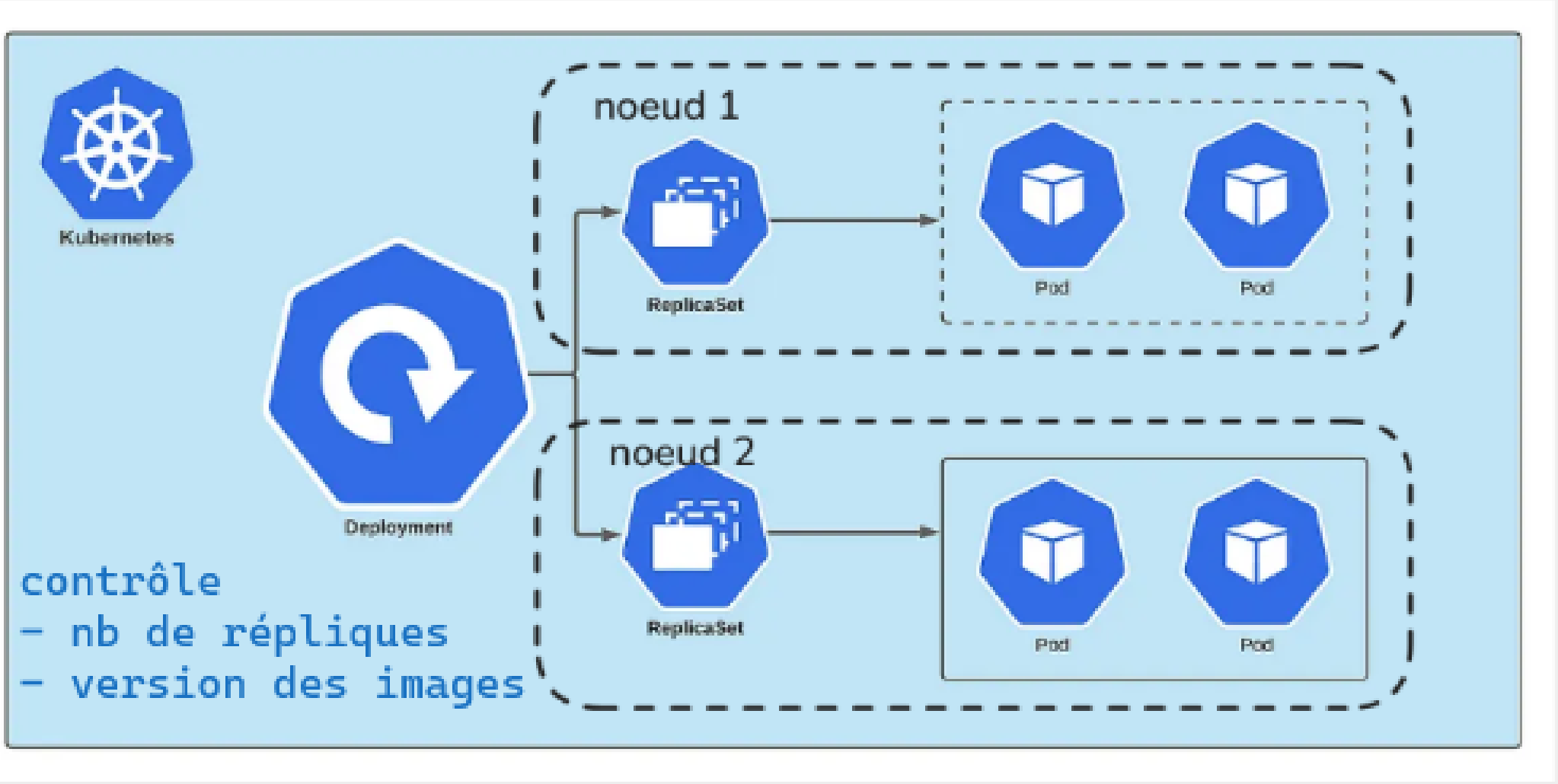
- instructions scriptées pour créer une image immutable
  => **INFRASTRUCTURE AS CODE**



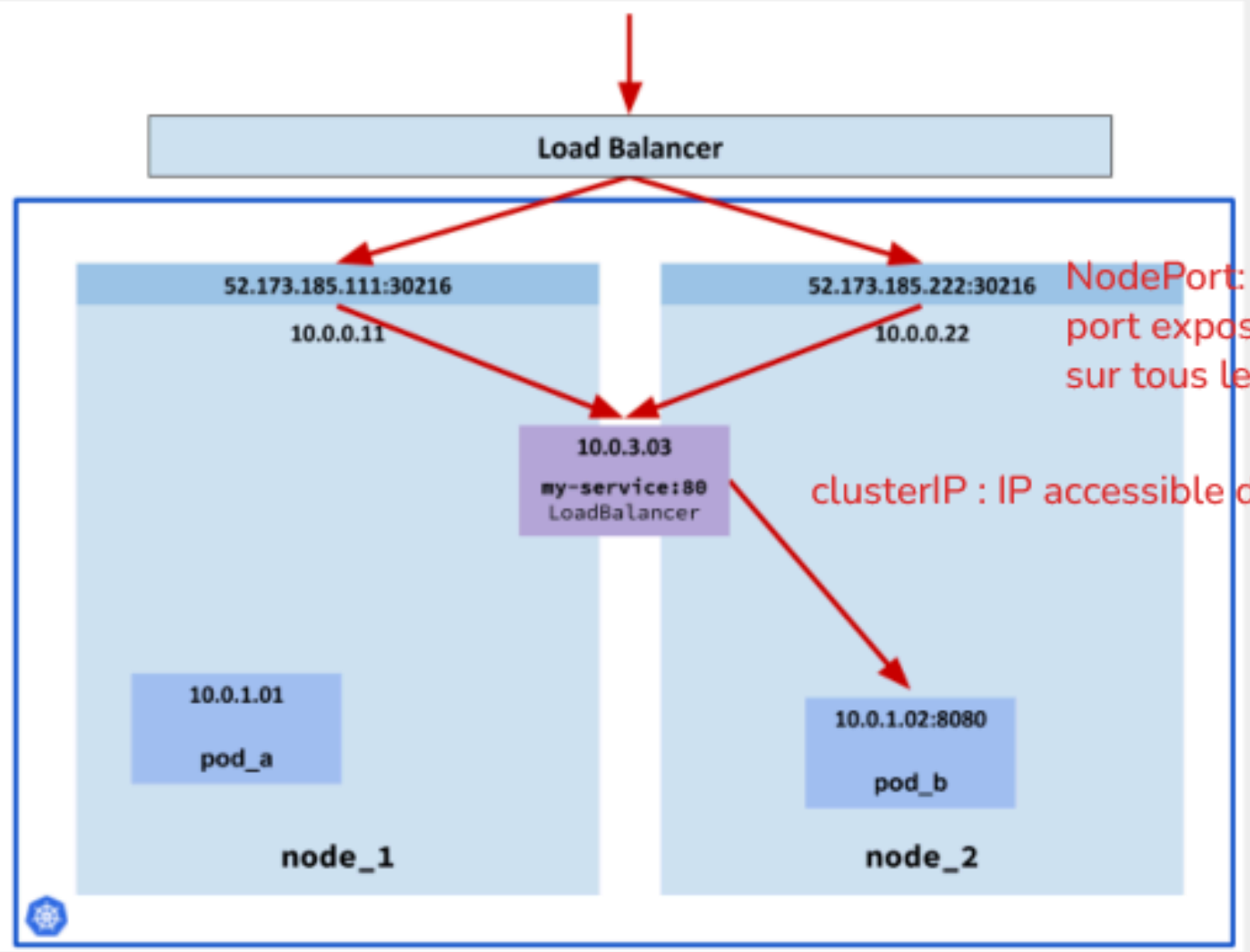
pod kubernetes



Déploiement kubernetes



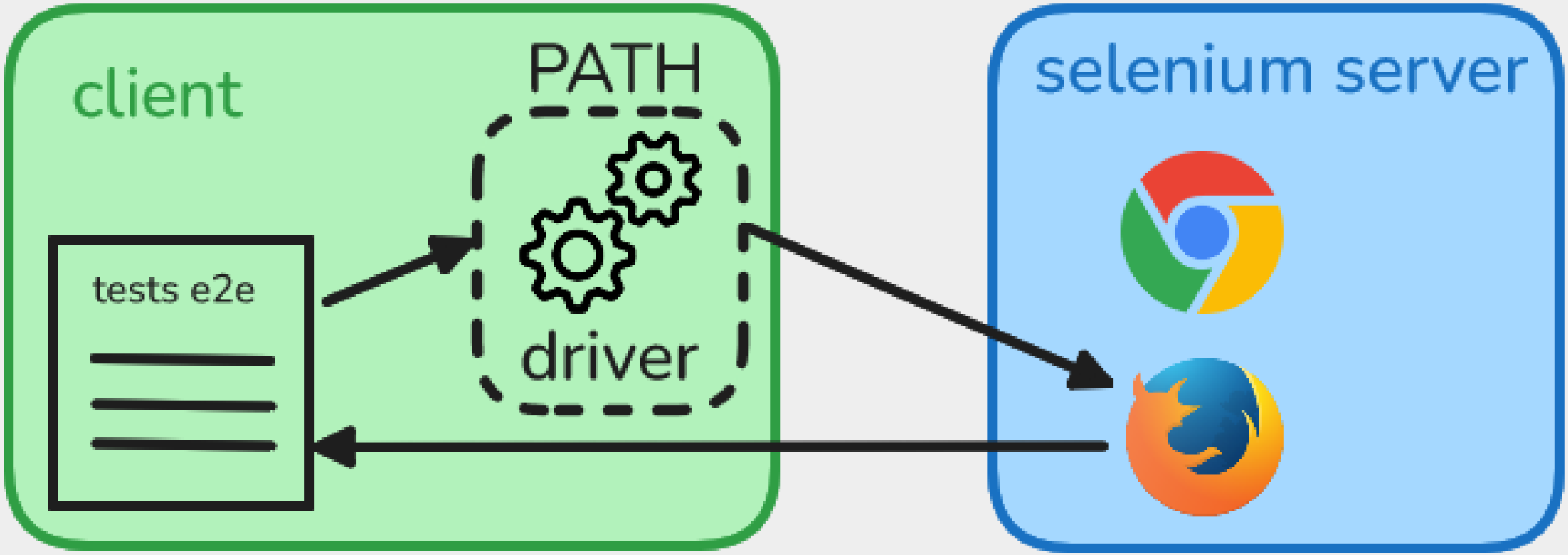
mise en réseau kubernetes



V. Test d'acceptations

tests d'interfaces

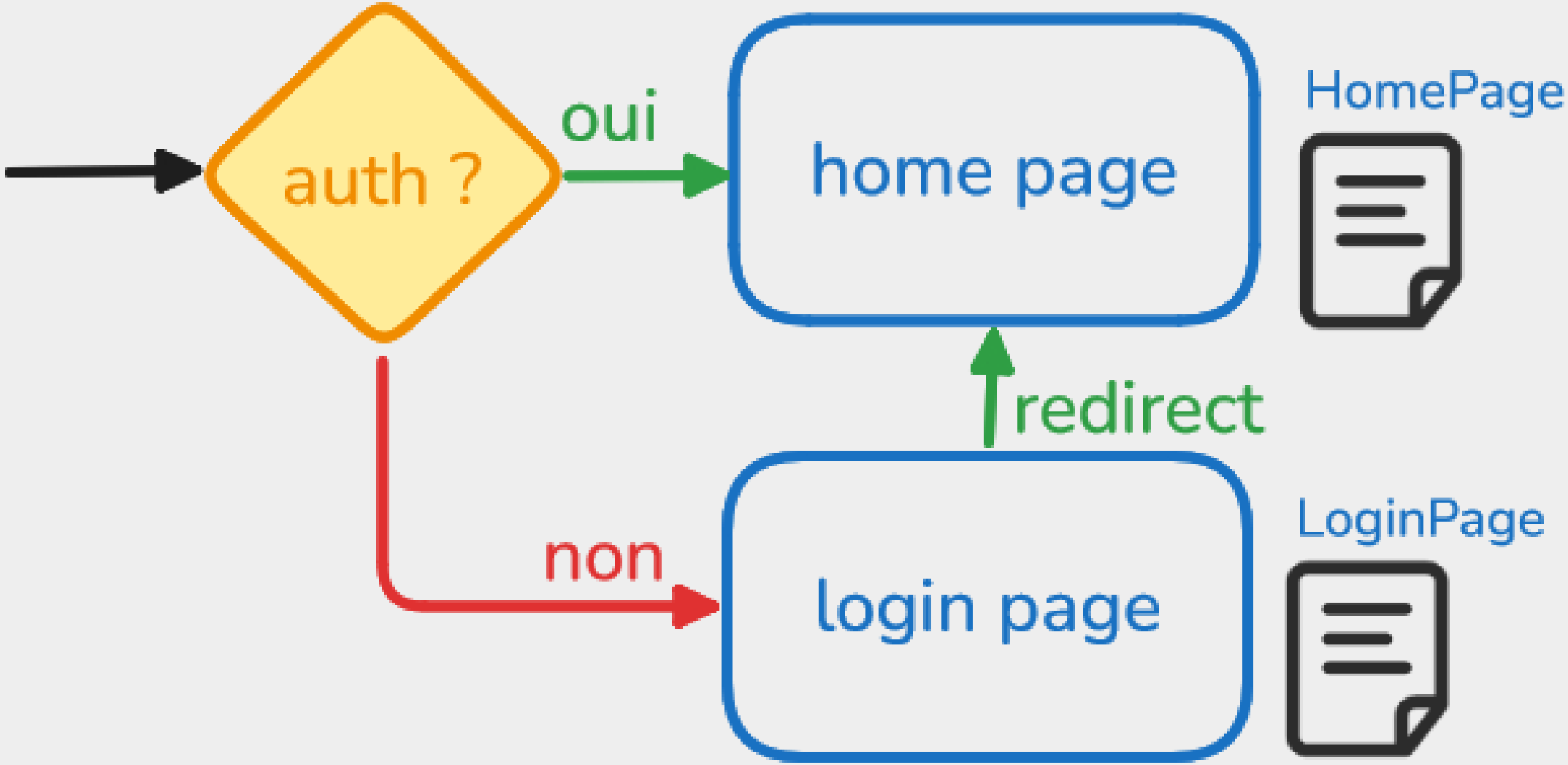
- ou tests E2E « **E**nd To **E**nd »
- consistent à *simuler la navigation* des utilisateurs sur l'application **en exécution**
- les tests doivent avoir accès à un ou plusieurs *moteurs de navigateurs*, au travers d'un **driver**
- plusieurs solutions comme **Selenium** présente une *solution client / serveur* dont le serveur contient des *moteurs configurables*


client
PATH
driver
tests e2e
selenium server

« Page Object Model »

POM est un *design pattern* utilisé par les tests e2e

- une navigation est une séquence d'accès à des **pages html** liées par des **liens http**
- un test e2e peut être structuré en **encapsulant** ces accès successifs dans des *classes représentant les pages*
- puisque une navigation est un *flux métier / utilisateur*, on peut utiliser le **formalisme bdd avec le POM**


auth ?
oui
home page
HomePage
redirect
non
login page
LoginPage

Tests de performances

- mesurent la **réponse** d'une application en la soumettant à différents **scenarios d'usage**
- outils: *JMeter, Gatling, K6, Locust, ...*

Type de Test	Objectif	Charge	Durée	Métrique Clé	Quand l'Utiliser
Test de Charge	charge fixe	ex: 1000 users	30m-2h	Tps de réponse, Débit	Avant release
Test de Stress	charge max	jusqu'au crash	15min-1h	cahrge max, Récupération	Planification capacité
Test de Pic	pics soudains de trafic	ex: x10 instantané	5-15min	Élasticité	Événements (Black Friday, lancement)

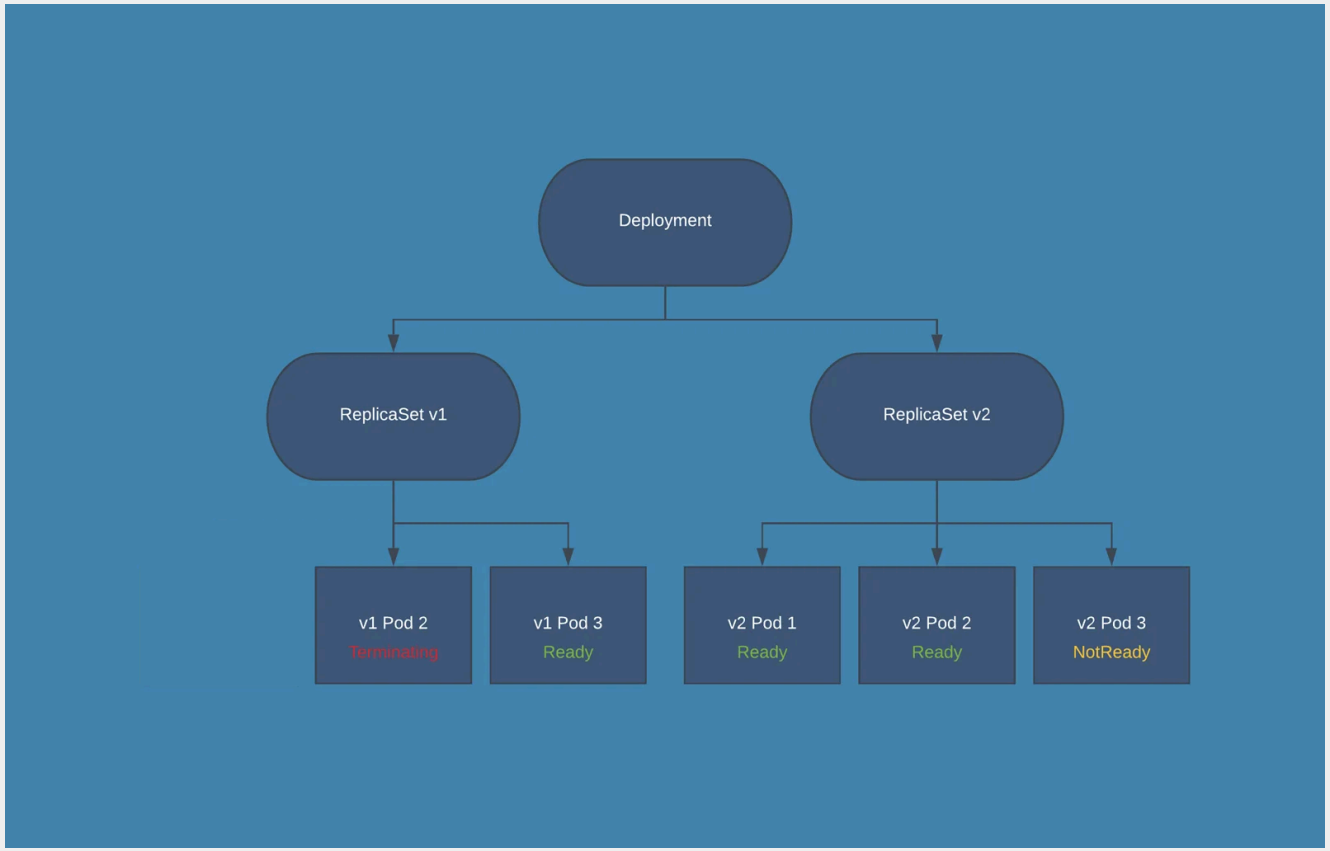
Type de Test	Objectif	Charge	Durée	Métrique Clé	Quand l'Utiliser
Test d' Endurance	Dégradations Fuites	modérée	heures, jours	Stabilité	avant prod.
Test de Volume	gros volumes	Var.	Var.	Performance DB, Temps requêtes	Migrat° de données
Test de Scalabilité	montée en répliques	paliers	1-3h	Scalabilité H/V	cloud µservices

Type de Test	Objectif	Charge	Durée	Métrique Clé	Quand l'Utiliser
Test de Capacité	capacité maximale	paliers	Moy.	Max. users	Dimensionner l'infra
Test de Concurrence	accès simul.	Haute concur.	Moy.	Deadlocks, Race conditions	Flux critiques (paiement)

VI. La Production

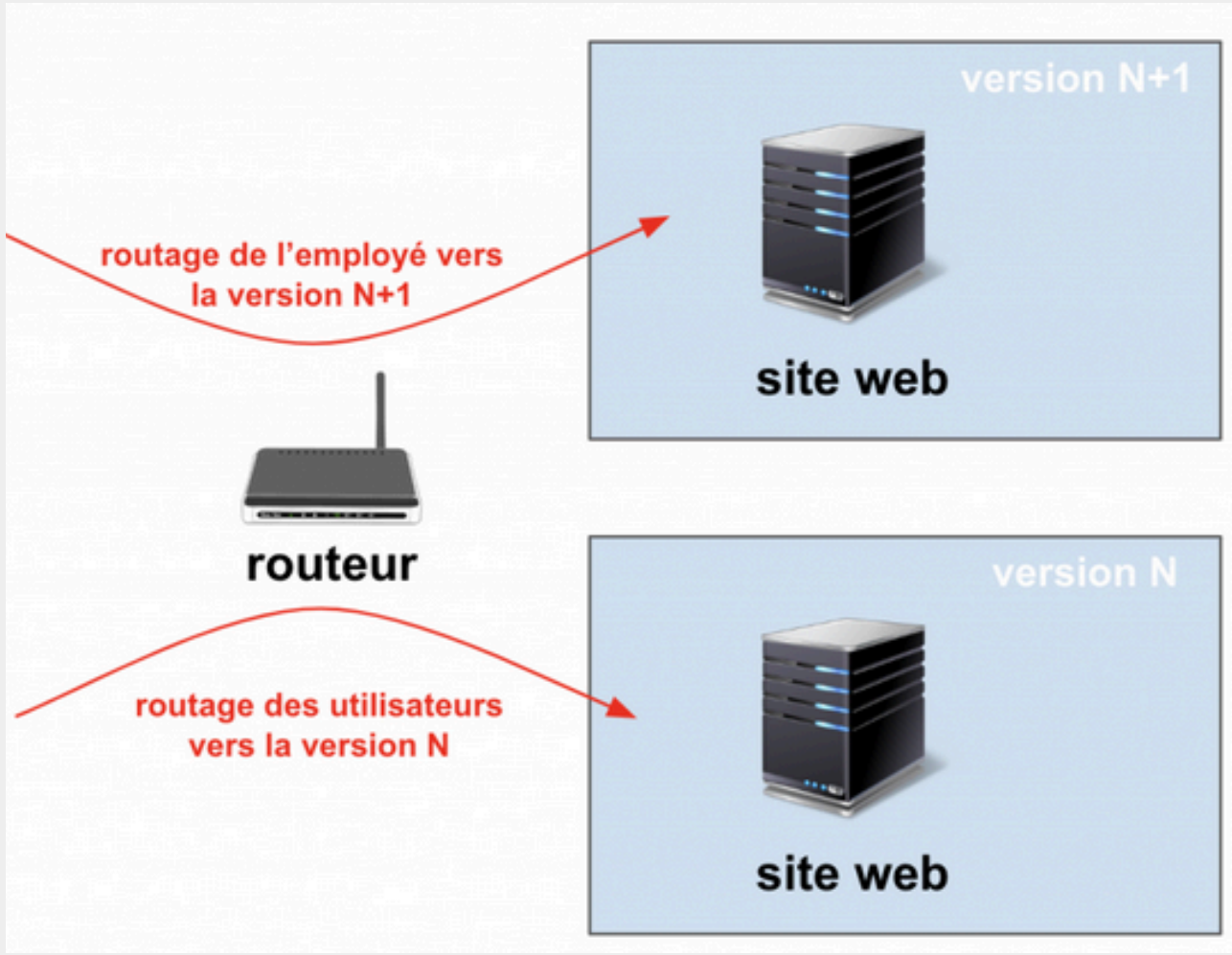
Déploiement bleu / vert

- permet de transférer progressivement le trafic sur une nouvelle de version sans interruption de service - *ex: Rolling Update k8s*

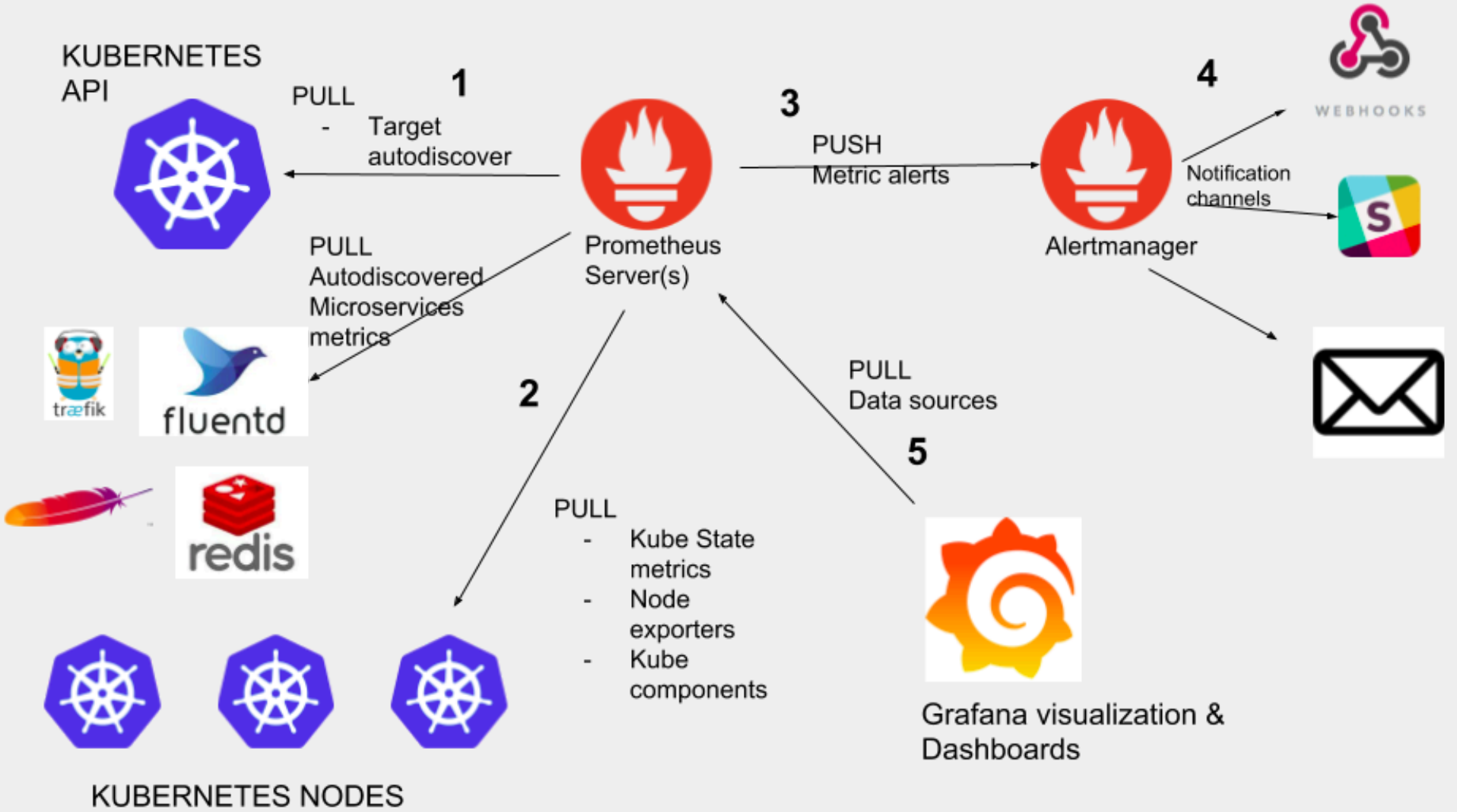


Déploiement Canari

- la nouvelle version est publiée auprès d'un *pool d'auditeurs sélectionnés*
- audit automatique possible en ajoutant de l'observabilité pour Rollback
  => **Cluster Immune System**



Monitoring : ex Promotheus



VII. KPIs

"Golden Metrics" de Google

Latence: temps de la réponse après une requête
=> demande de l'observabilité - *traces* - pour mesuer le parcours

Trafic: bande-passante, nb de sessions concurrentes

Erreurs: taux d'erreurs / intervalle de temps

Saturation: Tx d'occupation - *RAM, CPU, I/O disques*

Métriques de Flux LEAN

Cadence: rythme de la demande, porté sur la production **JIT**
=> soumis à la *saisonnalité* annuelle ou intra-quotidienne.

Cycle time: temps requis pour *fournir la valeur*.

Lead Time: temps entre la commande client et la livraison de valeur

Taux d'activité: % de temps de travail effectif sur une durée

VA Time: % de temps passé sur la valeur ajoutée

Métriques pour Scrum

Vélocité: qté moy. de travail fait pendant un sprint

=> mesurée en *points de récit* (barême arbitraire de compléxité des specs) ou en heures.

Avancement de sprint: - qté restante de travail restant en point de récit ou temps avant la fin du sprint.

=>affiché avec un *« Sprint Burntdown »*

Métriques ITIL

Mean Time Between Deploys: Durée moyenne entre 2 déploiements.

Mean Time Between Failures **MTBF** : Tps moy. de fonctionnement sans interruption => mesure la fiabilité et **disponibilité**

Mean Time to Detect Defects **MTTD** Tps moy. de remontée des bugs.

Mean Time to Repair **MTTR**: tps moy. requis pour la corriger un bug

Mean Time to Restore Service **MTRS**: Temps moyen entre une interruption de service et la reprise de service

MTRS = MTTD + MTTR + RS

Métriques logicielles

Nombre de méthodes par objet

Ratio lignes de codes/nombre de méthodes

% lignes dupliquées

Complexité cyclomatique d'une méthode : **pts de décision**

nb(if, case) + 1 (le chemin principal)

Seuils de Complexité Cyclomatique

Complexité	Évaluation	Action	Risque
1-10	✅ Simple	Aucune action	Faible
11-20	⚠️ Modérée	Envisager le refactoring	Moyen
21-50	🔴 Complexe	**Refactoriser**	Élevé
> 50	💀 Très complexe	**Refactoriser immédiatement**	Très élevé

> **Seuil recommandé : 10** (standard SonarQube, PMD, ESLint)

coefficient d'Instabilité

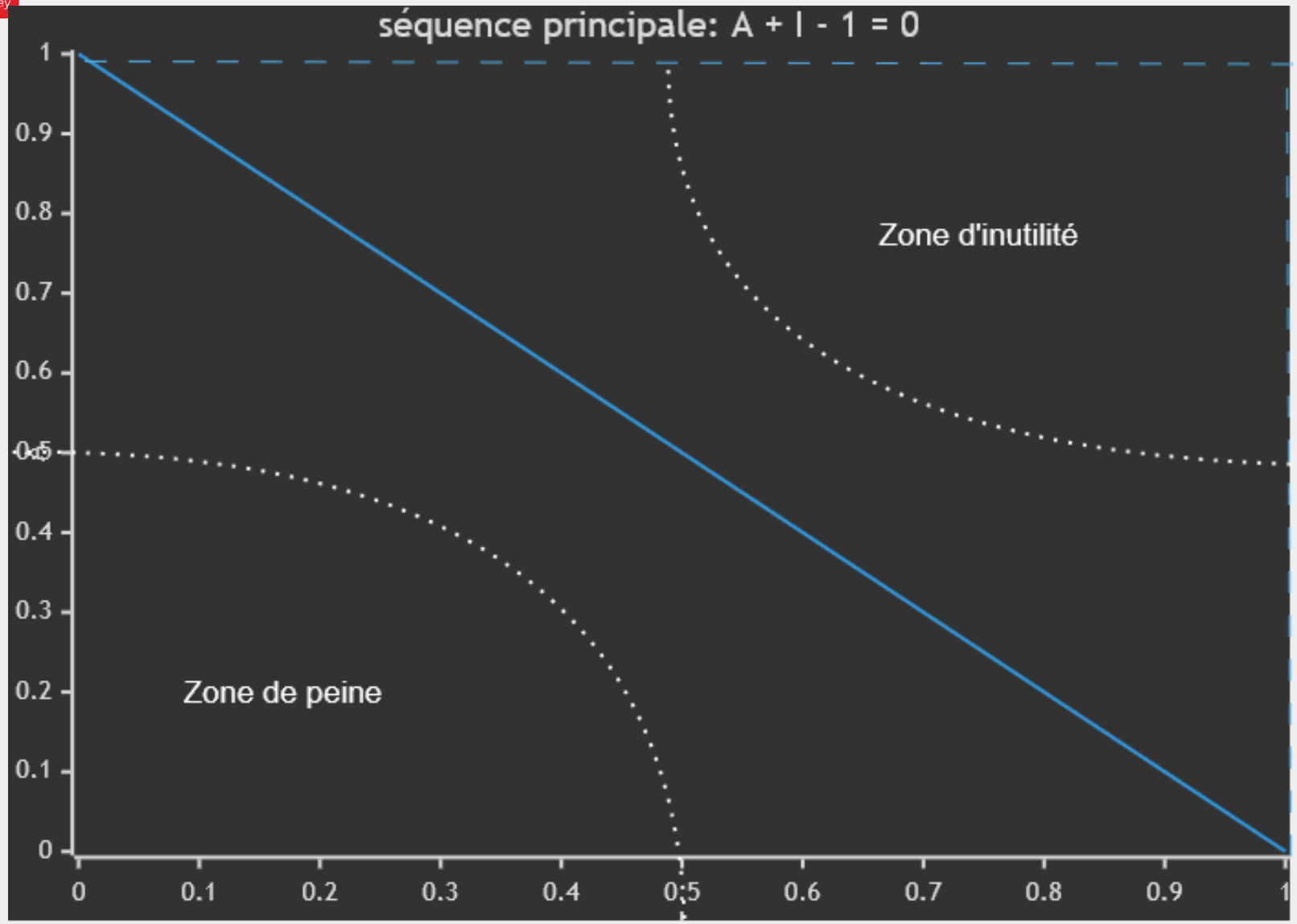
$$I = \frac{Ce}{Ca + Ce}$$

- Ce = 0 => I = 0 => stable (difficile à modifier)
- Ca = 0 => I = 1 => instable

coefficient d'Abstraction

$$A = \frac{Na}{Nc}$$

- *Na* = nb de classe abstraite ou interface
- *Nc* = nb de classes
- A = 1 => composant abstrait
- A = 0 => composant concret


séquence principale: A + I - 1 = 0
Zone d'inutilité
Zone de peine
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0;5
0.6
0.7
0.8
0.9

distance à la séquence principale

- pour chaque composant de la distribution statistique des composants du système on peut calculer
- la *distance à la séquence principale*:

$$D = \mid A + I - 1 \mid$$

- la *moyenne* et *l'écart type* **Z** de la distribution en A(I)

> “ les composants avec **D > Z sont considérés comme aberrants** ”

Métriques de tests

% Réussite des tests automatisés

Taux de couverture de code

Taux de couverture du **code critique** (valeur ajouté)

% Fuite des bugs: nb **bugs en prod.** / nb de bugs détectés

Maturité organisationnelle

modèle de maturité: outil d'audit du *niveau de maîtrise* de processus

rédaction d'une matrice *processus / niveau de maturité*
initial | géré | défini |mesuré | optimisé

Audit externe régulier pour mesurer l'évolution

exemple DevOps

	organisation	Travail d'équipe	CI	CD
optimisé	SAFE organization	Innovation Épanouissement	Métriques dataviz Réduction des contraintes	Monitoring & autoscale
mesuré	Agile pratiqué en communauté Business impliqué	Esprit de corps Transparence	Métriques prélevées Amélioration continue	Déploiments et orchestration Rollback
défini	Process de livraison régulière	Rapprochement des Devs & Ops Buts commun	Build et tests déclenchés depuis le SCM	Déploiement auto sur différents environnements
géré	Devs utilisent les principes Agile Ops séparés	Devs échangent connaissances Pratiques standard	Build et tests auto mais non reliés	Certains environnements sont provisionnés
initial	Devs & Ops en silos	Faible coordination et communication	Tests ad hoc Faible gestion du code source	Déploiement manuel