

# Orchestration de conteneurs et CI/CD



ESGI Lyon - M1-AL-ALT-S1 2023



QRCode to this presentation

- Présentation disponible à l'adresse: <https://dduportal.github.io/esgi-courses/fix/use-fontawesome-4-CDN>
- Version PDF de la présentation : Cliquez ici
- Contenu sous licence Creative Commons Attribution 4.0 International License
  - Une partie du contenu provient de <https://github.com/cicd-lectures/slides> (auteurs: Julien Levesy et moi-même)
- Code source de la présentation: <https://github.com/dduportal/esgi-courses>

# Comment utiliser cette présentation ?

- Pour naviguer, utilisez les flèches en bas à droite (ou celles de votre clavier)
  - Gauche/Droite: changer de chapitre
  - Haut/Bas: naviguer dans un chapitre
- Pour avoir une vue globale : utiliser la touche "o" (pour "**Overview**")

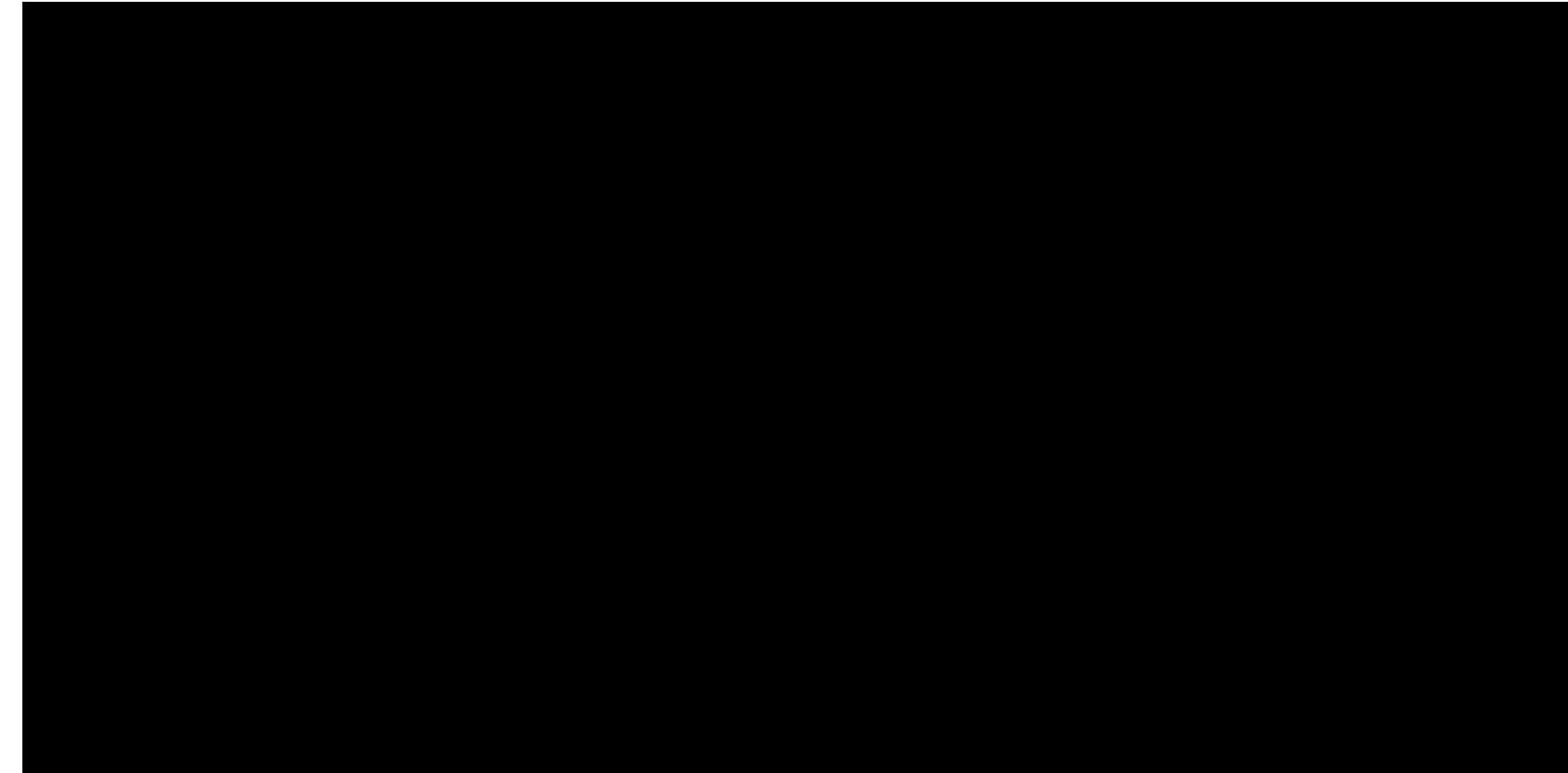
# Bonjour !

La suite: vers le bas ↓

# Damien DUPORTAL

- Staff Software Engineer chez CloudBees pour le projet Jenkins 
- Freelancer
- Me contacter :
  -  [damien.duportal @ gmail.com](mailto:damien.duportal@gmail.com)
  -  [dduportal](https://github.com/dduportal)
  -  [Damien Duportal](https://www.linkedin.com/in/damien-duportal/)

Et vous ?



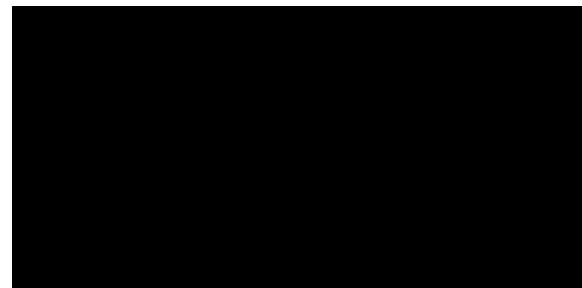
# A propos du cours

- Alternance de théorie et de pratique pour être le plus interactif possible
- Compatible Distanciel / Présentiel
- Contenu entièrement libre, open-source sous licence Creative Commons
  - N'hésitez pas ouvrir des Pull Request si vous voyez des améliorations ou problèmes: sur cette page (😉 wink wink)

# Calendrier

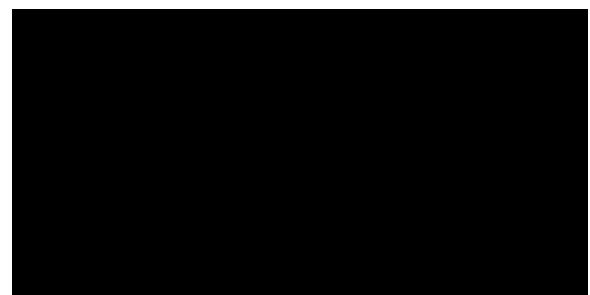
- Présentiel  Mercredi 22 novembre 2023 - 08h00 → 13h00
- Présentiel  Mercredi 20 décembre 2023 - 14h00 → 19h00
- Présentiel  Lundi 15 janvier 2024 - 09h45 → 17h15
- Présentiel  Mardi 16 janvier 2024 - 09h45 → 17h15

# Evaluation



- **Pourquoi ?** s'assurer que vous avez acquis un minimum de concepts
- **Quoi ?** Deux notes sur 20 (25% Contrôle Continu, 75% Projet)
- **Comment ?**
  - Contrôle Continu: Participation + attention pendant le cours, respect des échéances et des autres, éventuellement un QCM
  - Un projet GitHub (public) évalué selon une liste de critères *pré-déterminés*

# Plan



- Introduction
- Rappels (Ligne de commande, Git)
- Intégration Continue (CI)
- Rappels (Docker)
- Déploiement Continu (CD)
- Livraison Continue (CD aussi!)

La suite: vers la droite ➔

# Introduction



Vous avez dit "Continu" ?

Avant : le cycle en V



# What could go right?



what the client want

Copyright © <https://www.projectcartoon.com> under the Creative Commons Attribution 3.0 Unported License

Et dans la vraie vie c'est pire !



## How Projects Really Work

Copyright © <https://www.projectcartoon.com> under the Creative Commons Attribution 3.0 Unported License

# Problématique

- Travail basé sur:
  - Hypothèses initiales (capture du besoin)
  - Temps écoulé entre hypothèse et usage en production

⚠ Le besoin a forcément évolué dans ce laps de temps

⚠ Erreur de capture du besoin == coût de l'erreur décuplé avec le temps

# Agile

- Travailler par petites itérations **complètes**
  - Commencer petit
  - Confronter le logiciel au plus tôt aux utilisateurs.
  - Refaire des hypothèses basées sur ce que l'on à appris, et recommencer !
- "Embrasser" le changement : Votre logiciel va changer en **continu**



Source

# Manifeste Agile

- <https://agilemanifesto.org/iso/fr/manifesto.html>



Source

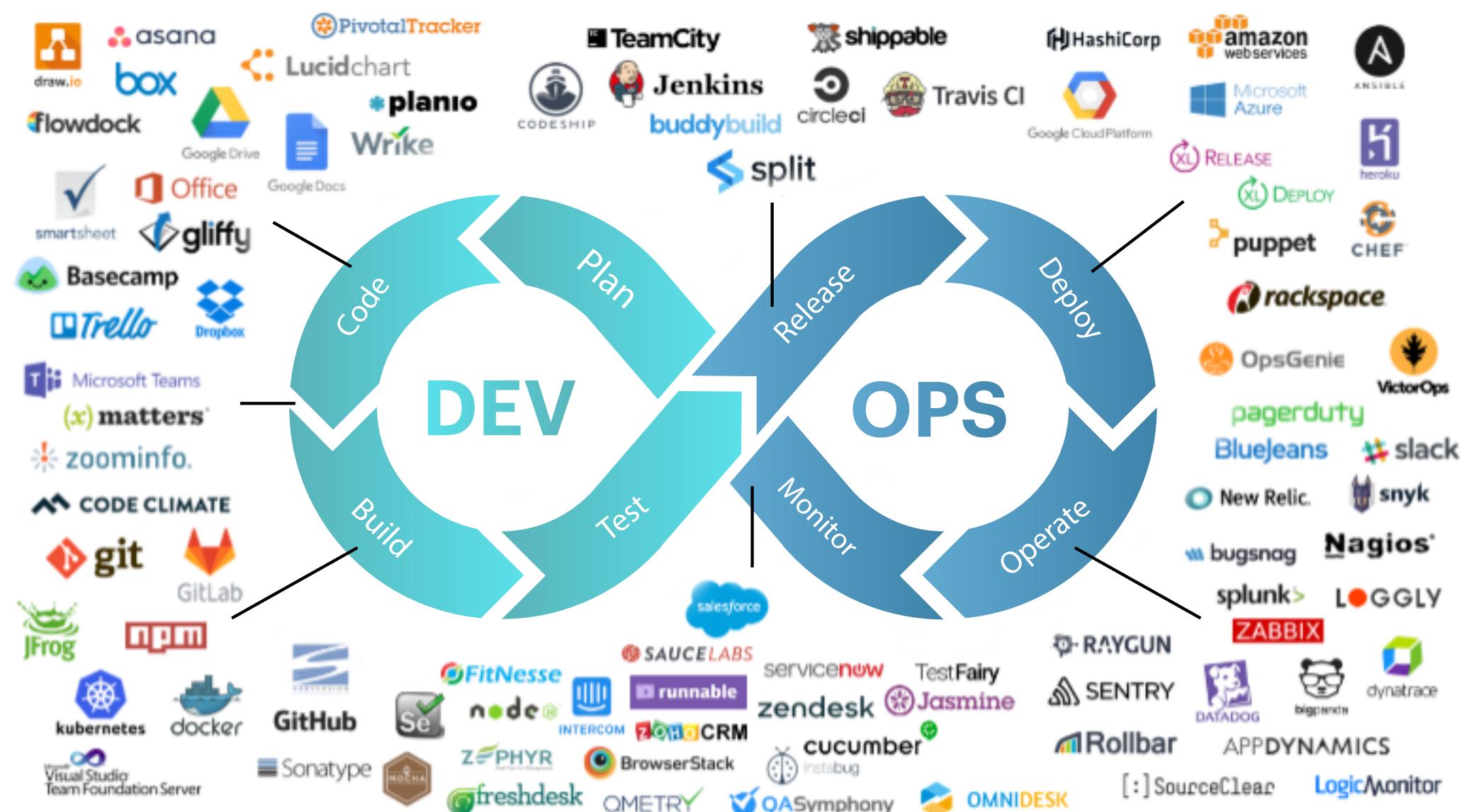
# La clé : gérer le changement!

- Le changement ne doit pas être un événement, ça doit être la **norme**
- Faire en sorte que changer quelque chose soit:
  - Facile
  - Rapide
  - Stable
  - Sûr

# Say Hello to DevOps



# Heureusement, vous avez des outils à disposition !



trollface

# Préparer votre environnement de travail

# Outils Nécessaires



- Un navigateur web récent (et décent)
- Un compte sur GitHub

# GitPod

GitPod.io : Environnement de développement dans le  "nuage"

- **But:** reproductible
- Puissance de calcul sur un serveur distant
- Éditeur de code VSCode dans le navigateur
- Gratuit pour 50h par mois ()

# Démarrer avec GitPod



- Rendez vous sur <https://gitpod.io>
- Authentifiez vous en utilisant votre compte GitHub:
  - Bouton "Login" en haut à droite
  - Puis choisissez le lien " Continue with GitHub"

△ Pour les "autorisations", passez sur la slide suivante

# Autorisations demandées par GitPod

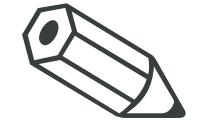


Lors de votre première connexion, GitPod va vous demander l'accès (à accepter) à votre email public configuré dans GitHub :

 gitpod github permissions

▲ Passez à la slide suivante avant d'aller plus loin

# Choix de l'Éditeur de Code

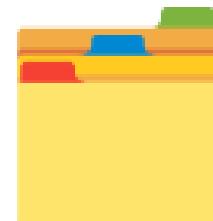


Choisissez l'éditeur "VSCode Browser" (la première tuile) :



⚠ Passez à la slide suivante avant d'aller plus loin

# Workspaces GitPod



- Vous arrivez sur la page listant les "workspaces" GitPod :
- Un workspace est une instance d'un environnement de travail virtuel (C'est un ordinateur distant)
- △ Faites attention à réutiliser le même workspace tout au long de ce cours△

 gitpod workspaces

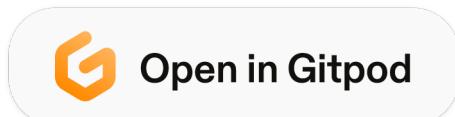
# Permissions GitPod <→ GitHub



- Pour les besoins de ce cours, vous devez autoriser GitPod à pouvoir effectuer certaines modifications dans vos dépôts GitHub
- Rendez-vous sur la page des intégrations avec GitPod
- Éditez les permissions de la ligne "GitHub" (les 3 petits points à droite) et sélectionnez uniquement :
  - user:email
  - public\_repo
  - workflow

# Démarrer l'environnement GitPod

Cliquez sur le bouton ci-dessous pour démarrer un environnement GitPod personnalisé:



Après quelques secondes (minutes?), vous avez accès à l'environnement:

- Gauche: navigateur de fichiers ("Workspace")
- Haut: éditeur de texte ("Get Started")
- Bas: Terminal interactif
- À droite en bas: plein de popups à ignorer (ou pas?)

Source disponible dans : <https://github.com/dduportal/esgi-gitpod>

# Checkpoint



- Vous devriez pouvoir taper la commande `whoami` dans le terminal de GitPod:
  - Retour attendu: `gitpod`
- Vous devriez pouvoir fermer le fichier "Get Started" ...
  - ... et ouvrir le fichier `.gitpod.yml`

On peut commencer !

# Ligne de commande

Guide de survie



# Problématique

- Communication Humain <→ Machine
- Base commune de TOUS les outils

# CLI

-  CLI == "Command Line Interface"
-  "Interface de Ligne de Commande"

# REPL

Pour les théoriciens et curieux :

-  REPL == "Read–eval–print loop"

[https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print\\_loop](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop)

# Anatomie d'une commande

```
ls --color=always -l /bin
```

 Copy

- Séparateur : l'espace
- Premier élément (`ls`) : c'est la commande
- Les éléments commençant par un tiret – sont des "options" et/ou drapeaux ("flags")
  - "Option" == "Optionnel"
- Les autres éléments sont des arguments (`/bin`)
  - Nécessaire (par opposition)

# Manuel des commandes

- Afficher le manuel de <commande> :

```
# Commande 'man' avec comme argument le nom de ladite commande
```

Copy

- Navigation avec les flèches haut et bas
  - Tapez / puis une chaîne de texte pour chercher
  - Touche n pour sauter d'occurrence en occurrence
- Touche q pour quitter



Essayez avec ls, chercher le mot color

- 💡 La majorité des commandes fournit également une option (--help), un flag (-h) ou un argument (help)
- Google c'est pratique aussi hein !

# Raccourcis

Dans un terminal Unix/Linux/WSL :

- CTRL + C : Annuler le process ou prompt en cours
- CTRL + L : Nettoyer le terminal
- CTRL + A : Positionner le curseur au début de la ligne
- CTRL + E : Positionner le curseur à la fin de la ligne
- CTRL + R : Rechercher dans l'historique de commandes

🎓 Essayez-les !

# Commandes de base 1/2

- `pwd` : Afficher le répertoire courant
  - 🎓 Option `-P` ?
- `ls` : Lister le contenu du répertoire courant
  - 🎓 Options `-a` et `-l` ?
- `cd` : Changer de répertoire
  - 🎓 Sans argument : que se passe t'il ?
- `cat` : Afficher le contenu d'un fichier
  - 🎓 Essayez avec plusieurs arguments
- `mkdir` : créer un répertoire
  - 🎓 Option `-p` ?

# Commandes de base 2/2

- echo : Afficher un (des) message(s)
- rm : Supprimer un fichier ou dossier
- touch : Créer un fichier
- grep : Chercher un motif de texte

# Arborescence de fichiers 1/2

- Le système de fichier a une structure d'arbre
  - La racine du disque dur c'est / :  ls -l /
  - Le séparateur c'est également / :  ls -l /usr/bin
- Deux types de chemins :
  - Absolu (depuis la racine): Commence par / (Ex. /usr/bin)
  - Sinon c'est relatif (e.g. depuis le dossier courant) (Ex . /bin ou local/bin/)



Source

# Arborescence de fichiers 2/2

- Le dossier "courant" c'est . :  ls -l ./bin # Dans le dossier /usr
- Le dossier "parent" c'est .. :  ls -l ../ # Dans le dossier /usr
- ~ (tilde) c'est un raccourci vers le dossier de l'utilisateur courant :  ls -l ~
- Sensible à la casse (majuscules/minuscules) et aux espaces : 

```
ls -l /bin
ls -l /Bin
mkdir ~/\"Accent tué"
ls -d ~/Accent\ tué
```

 Copy

# Un langage (?)

- Variables interpolées avec le caractère "dollar" \$ :

```
echo $MA_VARIABLE
echo "$MA_VARIABLE"
echo ${MA_VARIABLE}

# Recommendation
echo "${MA_VARIABLE}"

MA_VARIABLE="Salut tout le monde"

echo "${MA_VARIABLE}"
```

 Copy

- Sous commandes avec \$ (<command>) :

```
echo ">> Contenu de /tmp :\n$(ls /tmp)"
```

 Copy

- Des if, des for et plein d'autres trucs (<https://tldp.org/LDP/abs/html/>)

# Codes de sortie

- Chaque exécution de commande renvoie un code de retour (🇬🇧 "exit code")
  - Nombre entier entre 0 et 255 (en **POSIX**)
- Code accessible dans la variable **éphémère** \$? :

```
ls /tmp
echo $?

ls /do_not_exist
echo $?

# Une seconde fois. Que se passe-t'il ?
echo $?
```

Copy

# Entrée, sortie standard et d'erreur



```
ls -l /tmp
echo "Hello" > /tmp/hello.txt
ls -l /tmp
ls -l /tmp >/dev/null
ls -l /tmp 1>/dev/null

ls -l /do_not_exist
ls -l /do_not_exist 1>/dev/null
ls -l /do_not_exist 2>/dev/null

ls -l /tmp /do_not_exist
ls -l /tmp /do_not_exist 1>/dev/null 2>&1
```

 Copy

# Pipelines

- Le caractère "pipe" | permet de chaîner des commandes
  - Le "stdout" de la première commande est branchée sur le "stdin" de la seconde
- Exemple : Afficher les fichiers/dossiers contenant le lettre d dans le dossier /bin :

```
ls -l /bin  
ls -l /bin | grep "d" --color=auto
```

 Copy

# Exécution 1/2

- Les commandes sont des fichiers binaires exécutables sur le système :

```
command -v cat # équivalent de "which cat"  
ls -l "$(command -v cat)"
```

 Copy

- La variable d'environnement \$PATH liste les dossiers dans lesquels chercher les binaires
  -  Utiliser cette variable quand une commande fraîchement installée n'est pas trouvée

# Exécution 2/2

- Exécution de scripts :
  - Soit appel direct avec l'interpréteur : sh ~/monscript.txt
  - Soit droit d'exécution avec un "shebang" (e.g. #!/bin/bash)

```
$ chmod +x ./monscript.sh
$ head -n1 ./monscript.sh
#!/bin/bash
$ ./monscript.sh
# Exécution
```

Copy

# Git

Guide de survie



# Problématique

- Support de communication
  - Humain → Machine
  - Humain <→ Humain
- Besoins de traçabilité, de définition explicite et de gestion de conflits
  - Collaboration requise pour chaque changement (revue, responsabilités)

# Tracer le changement dans le code

avec un **VCS** :  Version Control System

également connu sous le nom de SCM ( Source Code Management)

# Pourquoi un VCS ?

- Pour conserver une trace de **tous** les changements dans un historique
  - "Source unique de vérité" ( *Single Source of Truth*)
- Pour **collaborer** efficacement sur un même référentiel

# Concepts des VCS

 scm local

 scm distributed

Source : <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

# Quel VCS utiliser ?



Nous allons utiliser **Git**

# Git

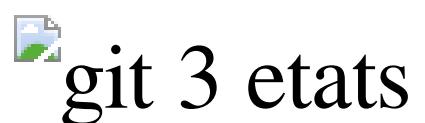
*Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.*

<https://git-scm.com/>



# Les 3 états avec Git

- L'historique ("Version Database") : dossier `.git`
- Dossier de votre projet ("Working Directory") - Commande
- La zone d'index ("Staging Area")



git 3 etats

Source : [https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Rudiments-de-Git#les\\_trois%C3%A9tats](https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Rudiments-de-Git#les_trois%C3%A9tats)



# Exercice avec Git - 1.1

- Dans le terminal de votre environnement GitPod:

- Créez un dossier vide nommé `projet-vcs-1` dans le répertoire `/workspace`, puis positionnez-vous dans ce dossier

```
mkdir -p /workspace/projet-vcs-1/  
cd /workspace/projet-vcs-1/
```

Copy

- Est-ce qu'il y a un dossier `.git/` ?
  - Essayez la commande `git status` ?
- Initialisez le dépôt git avec `git init`
  - Est-ce qu'il y a un dossier `.git/` ?
  - Essayez la commande `git status` ?

# ✓ Solution de l'exercice avec Git - 1.1

```
mkdir -p /workspace/projet-vcs-1/
cd /workspace/projet-vcs-1/
ls -la # Pas de dossier .git
git status # Erreur "fatal: not a git repository"
git init ./
ls -la # On a un dossier .git
git status # Succès avec un message "On branch main No commits yet"
```

Copy



## Exercice avec Git - 1.2

- Créez un fichier README.md dedans avec un titre et vos nom et prénoms
  - Essayez la commande git status ?
- Ajoutez le fichier à la zone d'indexation à l'aide de la commande git add (...)
  - Essayez la commande git status ?
- Créez un commit qui ajoute le fichier README.md avec un message, à l'aide de la commande git commit -m <message>
  - Essayez la commande git status ?

# ✓ Solution de l'exercice avec Git - 1.2

```
echo "# Read Me\n\nObi Wan" > ./README.md
git status # Message "Untracked file"

git add ./README.md
git status # Message "Changes to be committed"
git commit -m "Ajout du README au projet"
git status # Message "nothing to commit, working tree clean"
```

Copy

# Terminologie de Git - Diff et changeset

**diff:** un ensemble de lignes "changées" sur un fichier donné



**changeset:** un ensemble de "diff" (donc peut couvrir plusieurs fichiers)



# Terminologie de Git - Commit

**commit:** un changeset qui possède un (commit) parent, associé à un message



commit

**"HEAD":** C'est le dernier commit dans l'historique



scm basics legend



scm basics history



## Exercice avec Git - 2

- Affichez la liste des commits
- Affichez le changeset associé à un commit
- Modifiez du contenu dans README .md et affichez le diff
- Annulez ce changement sur README .md

# ✓ Solution de l'exercice avec Git - 2

```
git log  
  
git show # Show the "HEAD" commit  
echo "# Read Me\n\nObi Wan Kenobi" > ./README.md  
  
git diff  
git status  
  
git checkout -- README.md  
git status
```

Copy

# Terminologie de Git - Branche

- Abstraction d'une version "isolée" du code
- Concrètement, une **branche** est un alias pointant vers un "commit"





# Exercice avec Git - 3

- Créez une branche nommée `feature/html`
- Ajoutez un nouveau commit contenant un nouveau fichier `index.html` sur cette branche
- Affichez le graphe correspondant à cette branche avec `git log --graph`

# ✓ Solution de l'exercice avec Git - 3

```
git switch --create feature/html
# Ou alors: git branch feature/html && git switch feature/html
echo '<h1>Hello</h1>' > ./index.html
git add ./index.html && git commit --message="Ajout d'une page HTML par défaut" # -m / --message

git log
git log --graph
git lg # cat ~/.gitconfig => regardez la section section [alias], cette commande est déjà définie!
```

Copy



# Terminologie de Git - Merge

- On intègre une branche dans une autre en effectuant un **merge**
  - Un nouveau commit est créé, fruit de la combinaison de 2 autres commits





# Exercice avec Git - 4

- Mergez la branche `feature/html` dans la branche principale
  - $\Delta$  Pensez à utiliser l'option `--no-ff`
- Affichez le graphe correspondant à cette branche avec `git log --graph`

## ✓ Solution de l'exercice avec Git - 4

```
git switch main
git merge --no-ff feature/html # Enregistrer puis fermer le fichier 'MERGE_MSG' qui a été ouvert
git log --graph

# git lg
```

 Copy

# Feature Branch Flow

- **Une seule branche par fonctionnalité**

 scm feature branch workflow

# Exemple d'usages de VCS

- "Infrastructure as Code" :
  - Besoins de traçabilité, de définition explicite et de gestion de conflits
  - Collaboration requise pour chaque changement (revue, responsabilités)
- Code Civil:
  - <https://github.com/steeve/france.code-civil>
  - <https://github.com/steeve/france.code-civil/pull/40>
  - <https://github.com/steeve/france.code-civil/commit/b805ecf05a86162d149d3d182e04074ecf72c066>

# Checkpoint



- git est un des (plus populaires) système de contrôle de versions
  - Cet outil vous permet:
    - D'avoir un historique auditable de votre code source
    - De collaborer efficacement sur le code source (conflit git == "PARLEZ-VOUS")
- ⇒ C'est une ligne de commande (trop?) complète qui nécessite de pratiquer

# Intégration Continue (CI)

*Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove.*

— Martin Fowler

# Pourquoi la CI ?

**But :** Déetecter les fautes au plus tôt pour en limiter le coût



Source : <http://cartoontester.blogspot.be/2010/01/big-bugs.html>

# Qu'est ce que l'Intégration Continue ?

**Objectif :** que l'intégration de code soit un *non-événement*

- Construire et intégrer le code **en continu**
- Le code est intégré **souvent** (au moins quotidiennement)
- Chaque intégration est validée par une exécution **automatisée**

# Et concrètement ?



fail fast continuous integration

- Un•e développeu•se•r ajoute du code/branche/PR :
  - une requête HTTP est envoyée au système de "CI"
- Le système de CI compile et teste le code
- On ferme la boucle : Le résultat est renvoyé au développeu•se•r•s

# Quelques moteurs de CI connus

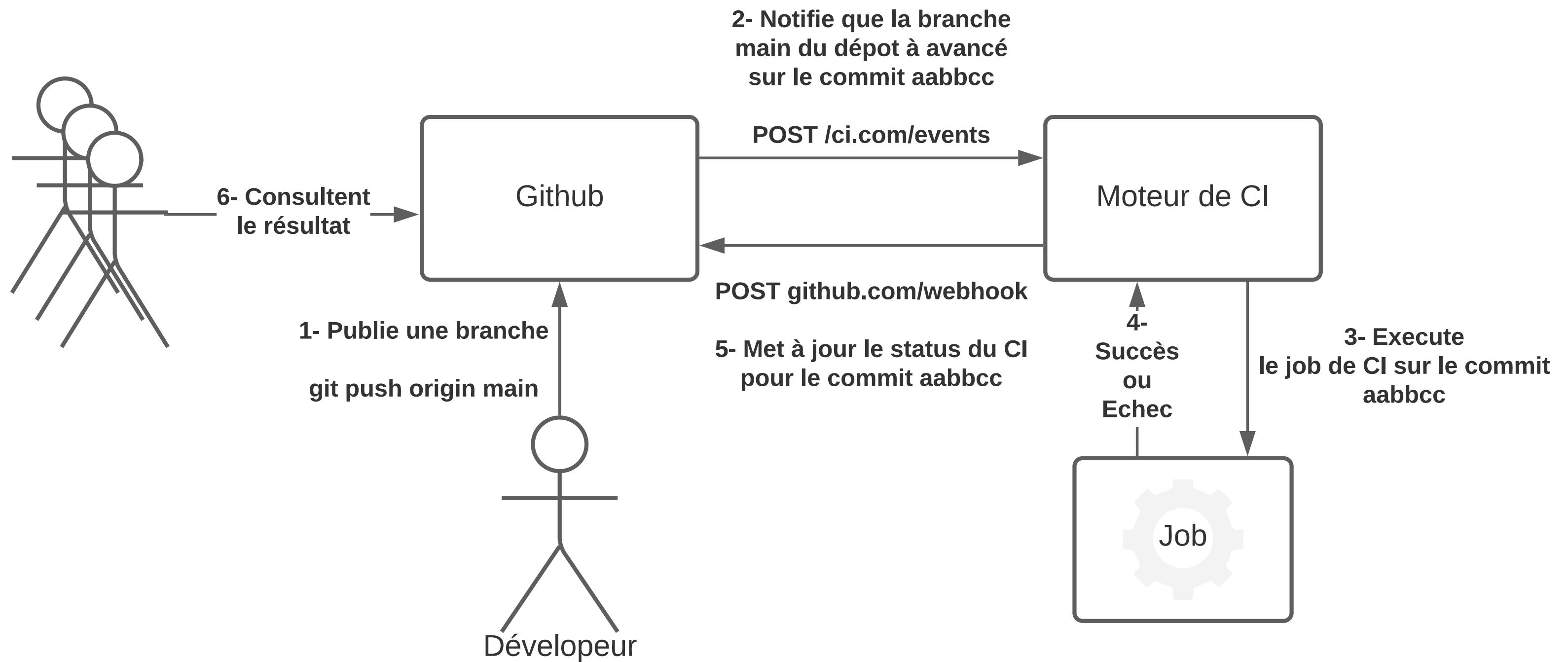
- A héberger soi-même : Jenkins, GitLab, Drone CI, CDS...
- Hébergés en ligne : Travis CI, Semaphore CI, Circle CI, Codefresh, GitHub Actions

# GitHub Actions

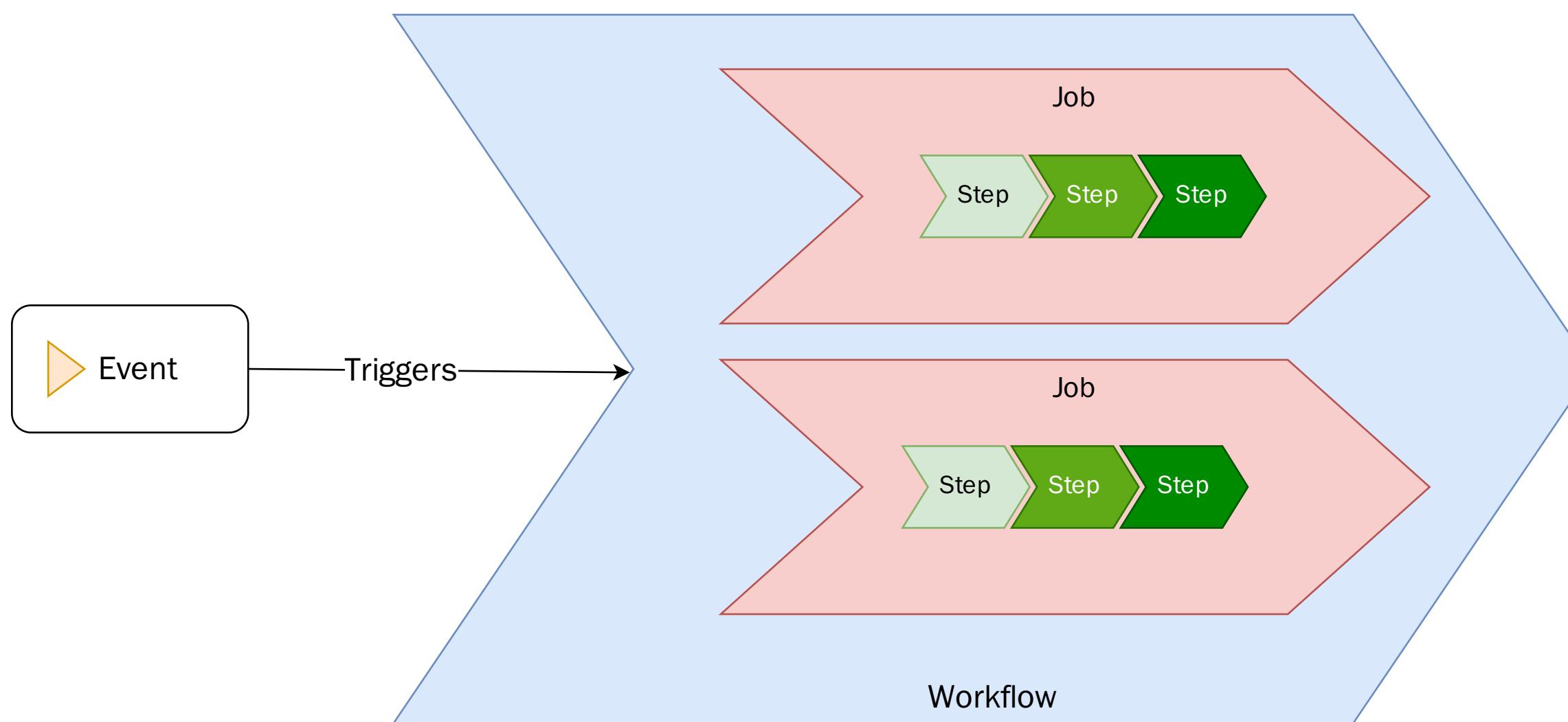
GitHub Actions est un moteur de CI/CD intégré à GitHub

- ✓ : Très facile à mettre en place, gratuit et intégré complètement
- ✗ : Utilisable uniquement avec GitHub, et DANS la plateforme GitHub

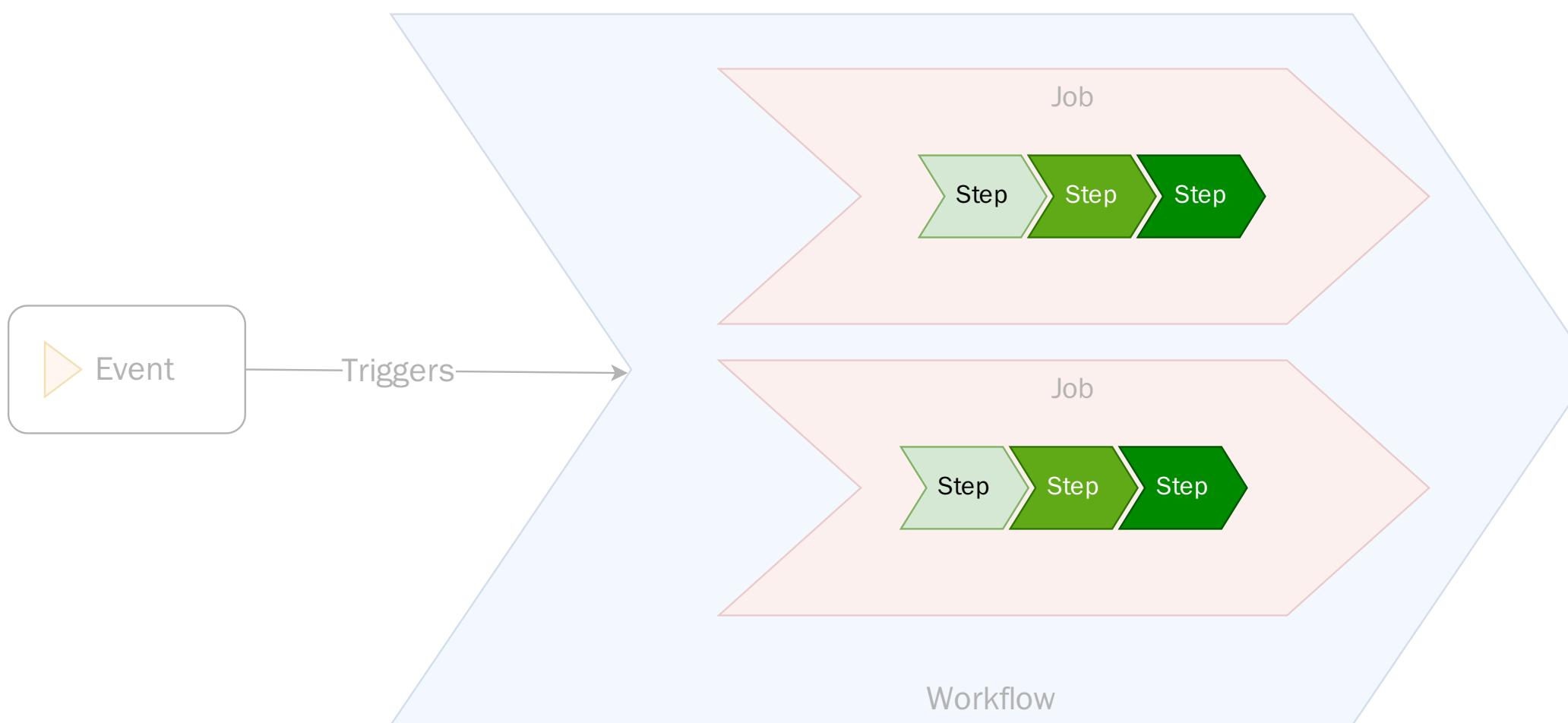
# Anatomie de déclenchement de GitHub Actions



# Concepts de GitHub Actions



# Concepts de GitHub Actions - Step 1/2



# Concepts de GitHub Actions - Step 2/2

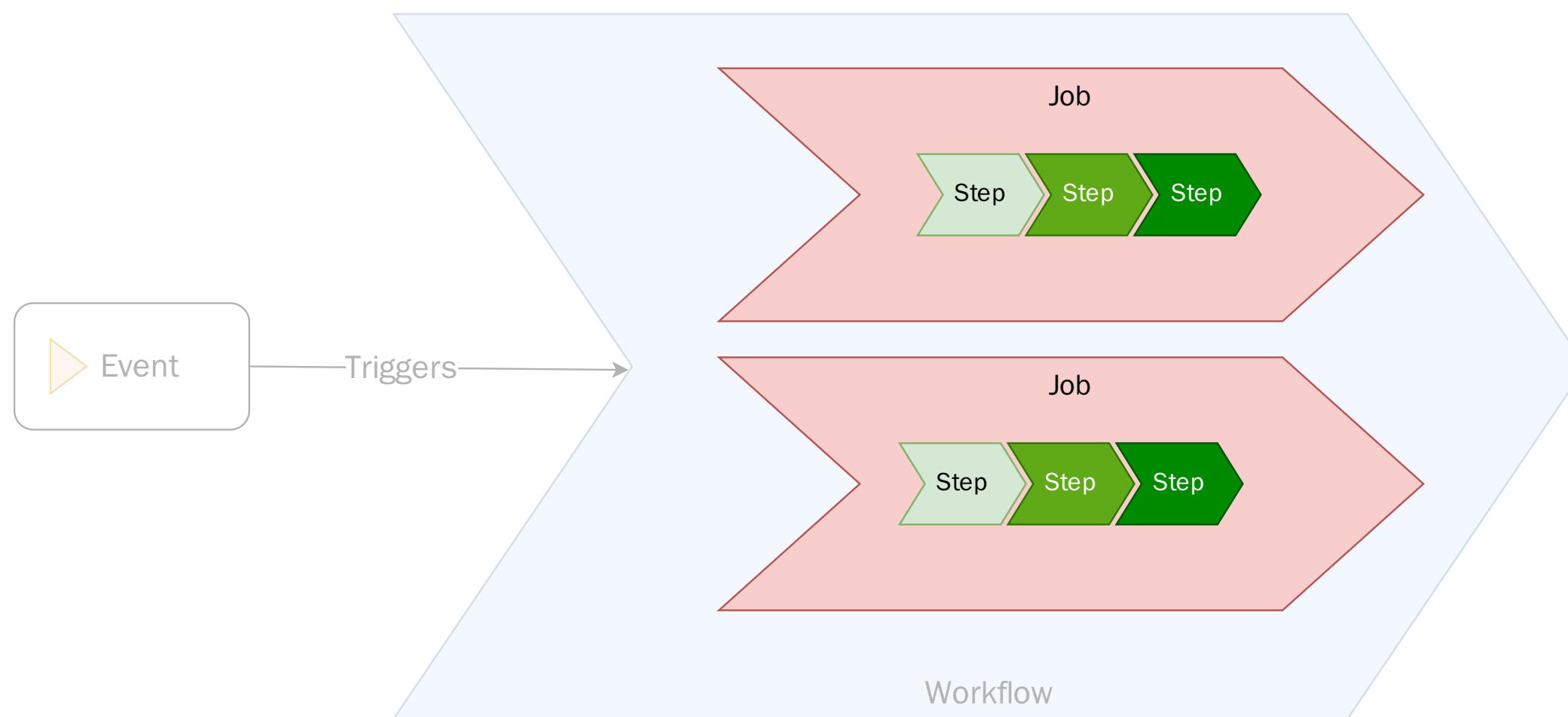
Une **Step** (étape) est une tâche individuelle à faire effectuer par le CI :

- Par défaut c'est une commande à exécuter - mot clef `run`
- Ou une "action" (quel est le nom du produit déjà ?) - mot clef `uses`
  - Réutilisables et partageables

```
steps: # Liste de steps
  # Exemple de step 1 (commande)
  - name: Say Hello
    run: echo "Hello ESGI"
  # Exemple de step 2 (une action)
  - name: 'Login to DockerHub'
    uses: docker/login-action@v2 # https://github.com/marketplace/actions/docker-login
    with:
      username: ${{ secrets.DOCKERHUB_USERNAME }}
      password: ${{ secrets.DOCKERHUB_TOKEN }}
```

 Copy

# Concepts de GitHub Actions - Job 1/2



# Concepts de GitHub Actions - Job 2/2

Un **Job** est un groupe logique de tâches :

- Enchaînement *séquentiel* de tâches
- Regroupement logique : "qui a un sens"
  - Exemple : "compiler puis tester le résultat de la compilation"

```
jobs: # Map de jobs
  build: # 1er job, identifié comme 'build'
    name: 'Build Slides'
    runs-on: ubuntu-22.04 # cf. prochaine slide "Concepts de GitHub Actions - Runner"
    steps: # Collection de steps du job
      - name: 'Build the JAR'
        run: mvn package
      - name: 'Run Tests on the JAR file'
        run: mvn verify
  deploy: # 2nd job, identifié comme 'deploy'
    # ...
```

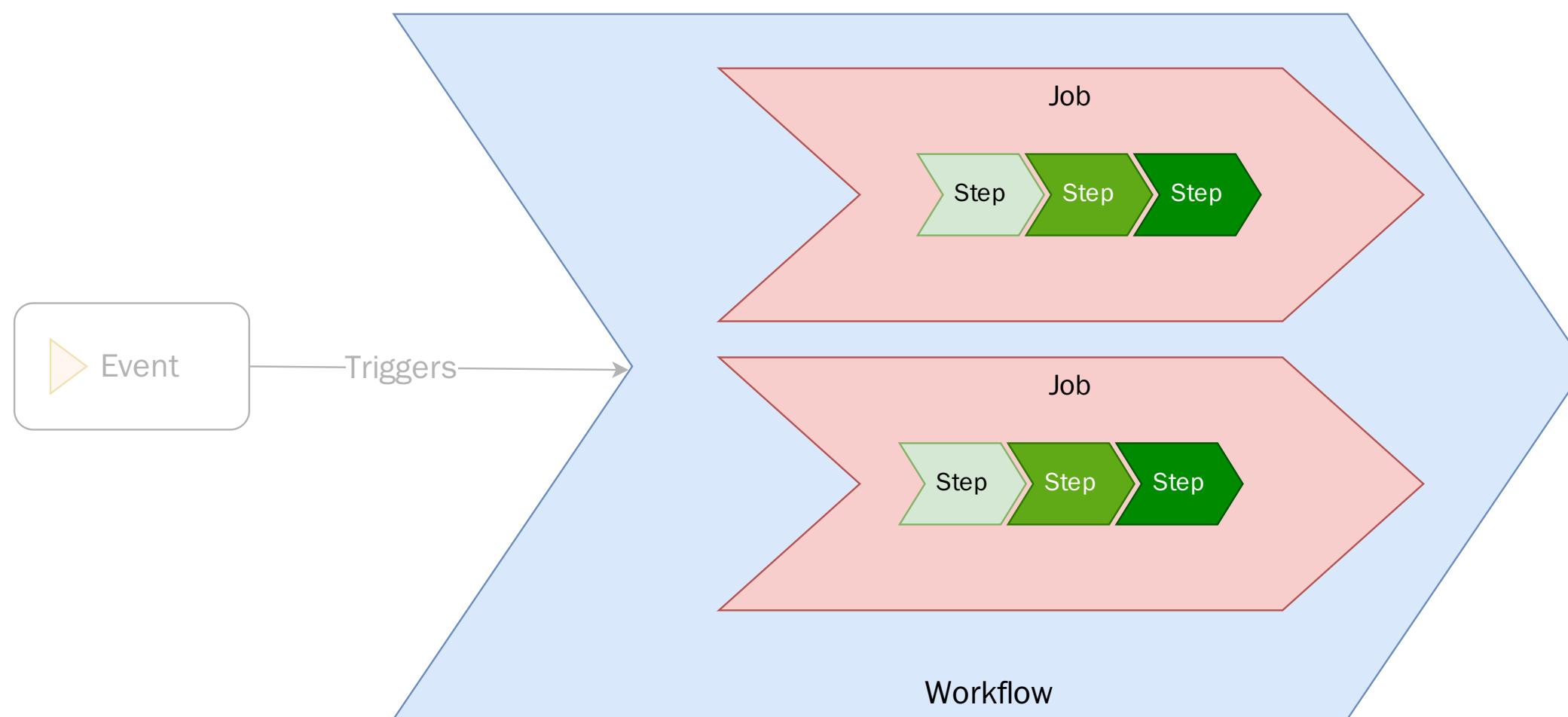
 Copy

# Concepts de GitHub Actions - Runner

Un **Runner** est un serveur distant sur lequel s'exécute un job.

- Mot clef `runs-on` dans la définition d'un job
- Défaut : machine virtuelle Ubuntu dans le cloud utilisé par GitHub
- D'autres types sont disponibles (macOS, Windows, etc.)
- Possibilité de fournir son propre serveur

# Concepts de GitHub Actions - Workflow 1/2



# Concepts de GitHub Actions - Workflow 2/2

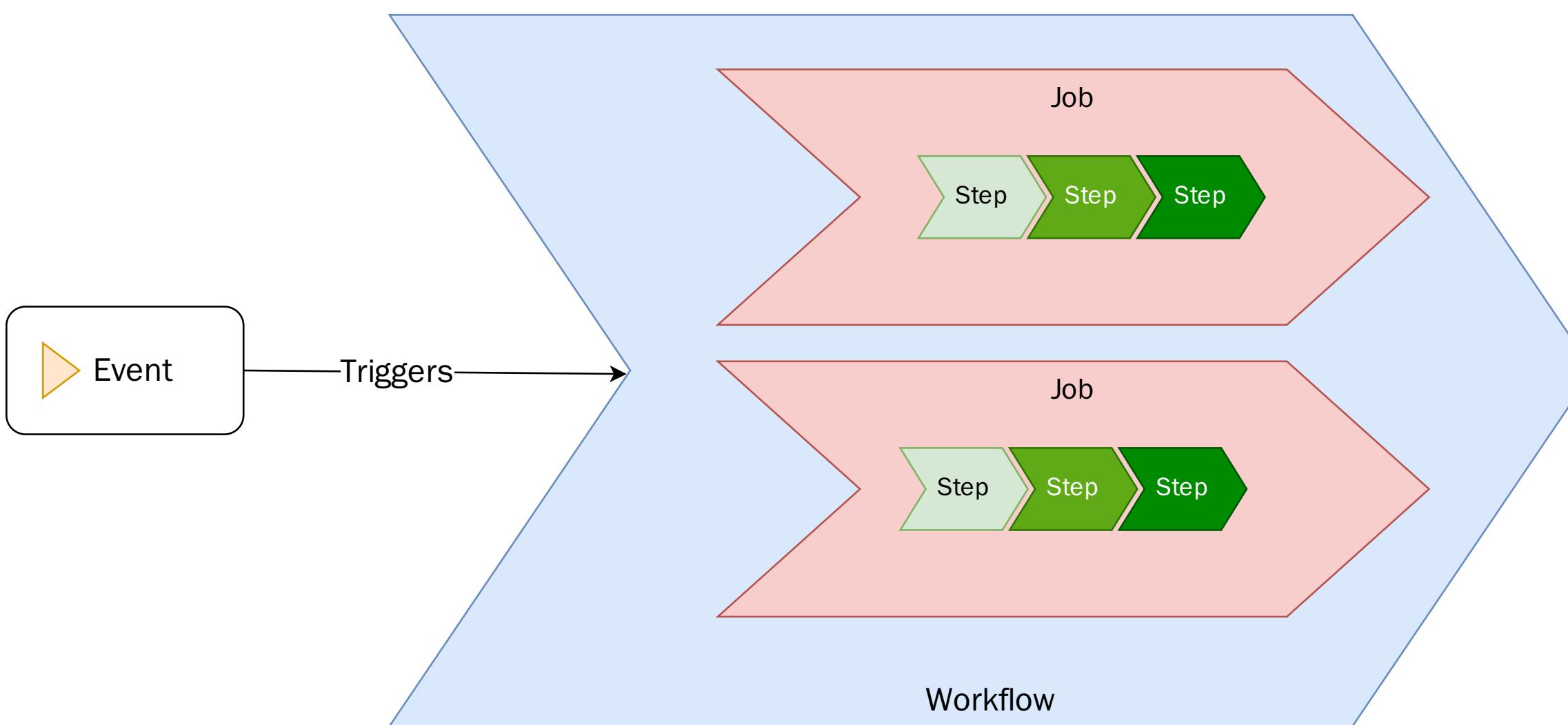
Un **Workflow** est une procédure automatisée composée de plusieurs jobs, décrite par un fichier YAML.

- On parle de "Workflow/Pipeline as Code"
- Chemin : .github/workflows/<nom du workflow>.yml
- On peut avoir *plusieurs* fichiers donc *plusieurs* workflows

```
.github/workflows
├── ci-cd.yaml
├── bump-dependency.yml
└── nightly-tests.yaml
```

 Copy

# Concepts de GitHub Actions - Évènement 1/2



# Concepts de GitHub Actions - Évènement 2/2

Un **évenement** du projet GitHub (push, merge, nouvelle issue, etc. ) déclenche l'exécution du workflow

- Plein de type d'évènements : push, issue, alarme régulière, favori, fork, etc.
  - Exemple : "Nouveau commit poussé", "chaque dimanche à 07:00", "une issue a été ouverte" ...
- Un workflow spécifie le(s) évènement(s) qui déclenche(nt) son exécution
  - Exemple : "exécuter le workflow lorsque un nouveau commit est poussé ou chaque jour à 05:00 par défaut"

# Concepts de GitHub Actions : Exemple Complet

Workflow File :

```
name: Node.js CI
on: # Évènements déclencheurs
  - push:
    branch: main # Lorsqu'un nouveau commit est poussé sur la branche "main"
  - schedule:
    cron: */15 * * * * # Toutes les 15 minutes
jobs:
  test-linux:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3
      - run: npm install
      - run: npm test
  test-mac:
    runs-on: macos-12
    steps:
      - uses: actions/checkout@v3
      - run: npm install
      - run: npm test
```

Copy

# Essayons GitHub Actions

- **But :** nous allons créer notre premier workflow dans GitHub Actions
- N'hésitez pas à utiliser la documentation de GitHub Actions:
  - Accueil
  - Quickstart
  - Référence



# Exercice: Créez un dépôt (git dans) GitHub

- En étant authentifié dans GitHub,
- Créez un nouveau dépôt nommé esgi-ci-cd
  - Pas de "template" (modèle)
  - Visibilité publique
  - Initialisation avec un fichier README.md



# Exercice: Récupérez le dépôt dans Gitpod

- Obtenez l'URL (HTTPS) du dépôt GitHub fraîchement créé
  - Depuis la page du dépôt, cliquez sur le bouton vert intitulé "**Code**"
- Dans Gitpod :
  - Depuis un terminal, positionnez-vous dans le dossier `/workspace`,
  - Clonez le dépôt avec la commande

```
git clone https://github.com/xxx/{student_gh_repository_name}
```

Copy

- Si le dépôt n'apparaît pas dans l'"Explorer" à gauche :

```
code -a /workspace/{student_gh_repository_name}
```

Copy



# Exercice: Exemple simple avec GitHub Actions

- Dans le dépôt `esgi-ci-cd`, sur la branch `main`,
  - Créez le fichier `.github/workflows/bonjour.yml` avec le contenu suivant :

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - run: echo "Bonjour 🙋"
```

[Copy](#)

- Commitez puis poussez le fichier sur GitHub:

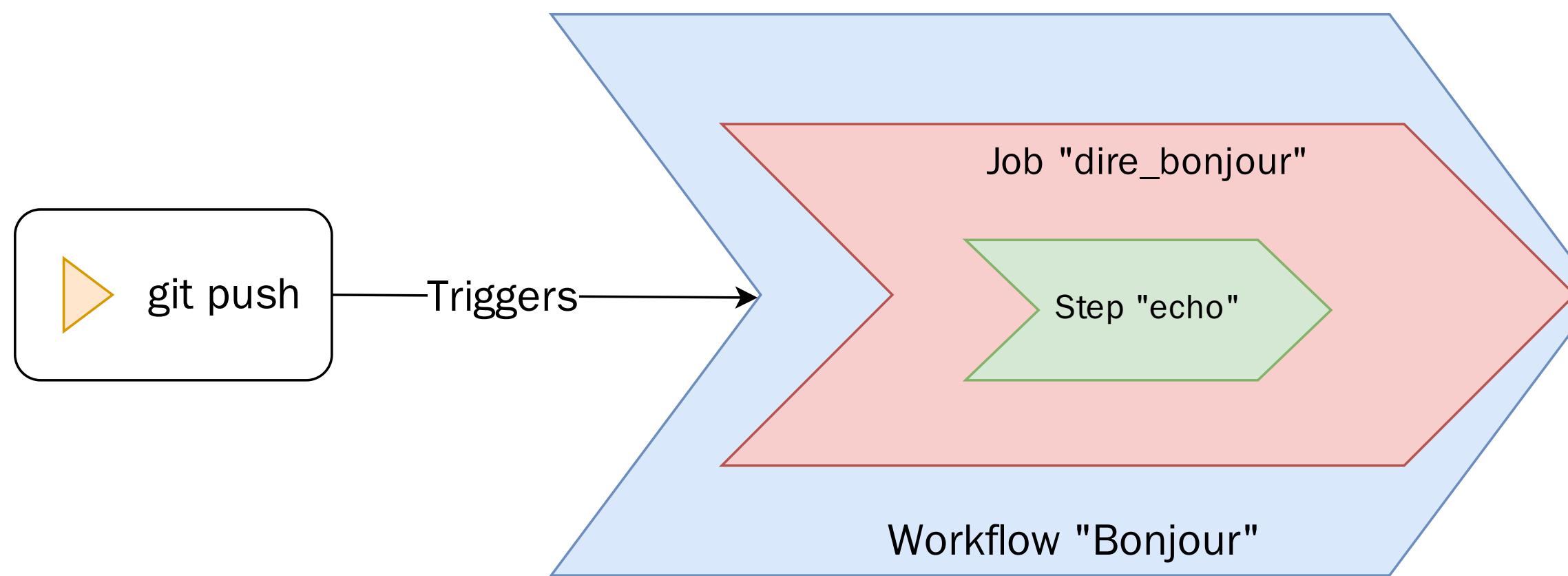
```
git add .github/workflows/bonjour.yml
git commit -m 'Create GitHub workflow Bonjour'
git push origin main
```

[Copy](#)



# Exercice: Exemple simple avec GitHub Actions : Récapète

- Revenez sur la page GitHub de votre projet et naviguez dans l'onglet "Actions" :
  - Voyez-vous un workflow ? Et un Job ? Et le message affiché par la commande echo ?



# Exemple GitHub Actions : Checkout

- Supposons que l'on souhaite utiliser le code du dépôt...
  - Essayez: modifiez le fichier `bonjour.yml` pour afficher le contenu de `README.md`:

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - run: ls -l # Liste les fichiers du répertoire courant
      - run: cat README.md # Affiche le contenu du fichier `README.md` à la base du dépôt
```

Copy

- Est-ce que l'étape “cat README.md” se passe bien ? (SPOILER: non ✗ )



# Exercice GitHub Actions : Checkout

- **But :** On souhaite récupérer ("checkout") le code du dépôt dans le job
- C'est à vous d'essayer de *réparer* le job :
  - L'étape "cat README.md" doit être conservée et doit fonctionner
  - Utilisez l'action "checkout" (Documentation) du marketplace GitHub Action
  - Vous pouvez vous inspirer du Quickstart de GitHub Actions

# ✓ Solution GitHub Actions : Checkout

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: ls -l # Liste les fichiers du répertoire courant
      - run: cat README.md # Affiche le contenu du fichier `README.md` à la base du dépôt
```

Copy

# Exemple : Environnement d'exécution

- Notre workflow doit s'assurer que "la vache" 🐄 doit nous lire 💬 le contenu du fichier README.md
  - WAT 😳 ?
- Essayez la commande `cat README.md | cowsay` dans GitPod
  - Modifiez l'étape "cat README.md" du workflow pour faire la même chose dans GitHub Actions
  - SPOILER: ✘ (la commande `cowsay` n'est pas disponible dans le runner GitHub Actions)



# Exercice : Personnalisation dans le workflow

- **But :** exécuter la commande `cat README.md | cowsay` dans le workflow comme dans GitPod
- C'est à vous de mettre à jour le workflow pour personnaliser l'environnement :
  - Cherchez comment installer `cowsay` dans le runner GitHub (`runs-on...`)

# ✓ Solution : Personnalisation dans le workflow

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: |
          sudo apt-get update
          sudo apt-get install --yes cowsay
      - run: cat README.md | cowsay
```

Copy

# Checkpoint



- L'intégration Continue est un ensemble de pratiques pour s'assurer que le code est **continuellement** "vérifié"
- GitHub Actions est un des nombreux systèmes permettant de faire de l'intégration continue

⇒ 🤔 Problème : comment gérer les différences entre l'environnement d'IC et les machines de travail  
?

# CI: Application d'exemple

# Application "Say Hello"

- Web application écrite en NodeJS
- Fonctionnalité : réponds "Hello World" à une requête HTTP GET sur /

```
/*eslint semi: ["warning", "always"]*/  
  
const http = require('http');  
  
const hostname = '127.0.0.1';  
const port = 3000;  
  
const server = http.createServer((req, res) => {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello World\n')  
});  
  
server.listen(port, hostname, () => {  
  console.log(`Server running at http://${hostname}:${port}/`);  
});
```

Copy



# Exercice : Essayons cette application dans GitPod

- Créez un fichier `index.js` à la racine de votre dépôt avec le contenu de la slide précédente
- Ouvrez 2 terminaux
  - Dans le premier terminal, exécutez l'application :

```
node ./index.js # CTRL-C to stop
```

Copy

- Dans le second terminal, faites-lui dire bonjour :

```
curl --verbose http://127.0.0.1:3000
```

Copy



# Cycle de vie / Pipeline

- C'est tout ? Bien sûr que non ! Il y a plein d'étapes (avant et après)
- Commençons par un cycle de vie NodeJS classique :
  - Récupération des dépendances
  - Lint (Analyse statique)
  - Test (peu importe le type)
- 💡 NPM à la rescousse



# Exercice : Mise en place d'un cycle de vie

1/2

- Créez un fichier package.json à la racine de votre dépôt:

```
{  
  "name": "sayhello",  
  "version": "1.0.0",  
  "description": "A web application which says hello",  
  "main": "index.js",  
  "scripts": {  
    "lint": "jshint index.js",  
    "start": "node index.js"  
  },  
  "author": "",  
  "license": "ISC",  
  "engines": {  
    "node": "19"  
  },  
  "devDependencies": {  
    "jshint": "^2.13.6"  
  },  
  "jshintConfig": {  
    "esversion": 6  
  }  
}
```

Copy



# Exercice : Mise en place d'un cycle de vie

## 2/2

- Essayez de lancer l'application dans GitPod :

```
# Récupérer les dépendances - Prérequis systématique
npm install

# Exécuter le script "launch-app"
npm run start # CTRL-C to stop

# Second terminal:
curl --verbose http://127.0.0.1:3000
```

 Copy



# Exercice : Lint de l'application dans GitPod

```
npm run lint  
# Indique une erreur  
# index.js: line 7, col 27, Missing semicolon.
```

Copy



# Checkpoint

On a donc un pipeline à 2 étapes :

- Récupération des dépendances
- Lint (qui indique une erreur).



# Exercice : Mise en place du CI Node/NPM

- **But :** exécuter les mêmes étapes que précédemment dans GitHub Actions
- Modifiez votre workflow pour :
  - Ajouter node\_modules/ dans un fichier .gitignore à la racine
  - Enlever les commandes "cowsay" de l'exercice précédent
  - Exécuter les commandes npm install puis npm run lint dans 2 étapes distinctes

✖ Résultat attendu : la même erreur

# ✓ Solution : Mise en place du CI Node/NPM

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: npm install
      - run: npm run lint
```

Copy



# Environnement d'exécution du CI

**Problème :** On souhaite avoir les mêmes outils dans notre CI ainsi que dans nos environnements de développement

- Environnements d'exécutions différents :
  - Système d'exploitation ? (macOS, Windows, Ubuntu Linux, Arch Linux, etc.)
  - Architecture du processeur ? (Intel, AMD, ARM, PowerPC, Risc-V, etc.)
  - SDKs installés (quelle version de NodeJS ? et de NPM ? etc.)



Que dis l'étape "npm install" du workflow ?



# Exercice : CI avec le tooling NodeJS

- **But :** Utilisez l'action GitHub <https://github.com/actions/setup-node> pour avoir la même version de NodeJS + NPM que dans GitPod
- C'est à vous de mettre à jour le workflow pour que l'étape npm install ne signale plus d'incompatibilité de version

# ✓ Solution : CI avec le tooling NodeJS

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - uses: actions/setup-node@v3
        with:
          node-version: 19
      - run: npm install
      - run: npm run lint
```

Copy



## Exercice : Corriger le CI

- **But:** Le workflow est toujours en échec. Faites le nécessaire pour corriger les erreurs

# Checkpoint

- GitHub Action fournit des actions pour installer et configurer nos environnements de CI
- Quand le CI est rouge, on le corrige en priorité !

# Plus de Git !

aka. Mettre son code en sécurité



# Une petite histoire

Votre dépôt est actuellement sur votre ordinateur.

- Que se passe t'il si :
  - Votre disque dur tombe en panne ?
  - On vous vole votre ordinateur ?
  - Vous échappez votre tasse de thé / café sur votre ordinateur ?
  - Une météorite tombe sur votre bureau et fracasse votre ordinateur ?

Testé, pas approuvé.

# Comment éviter ça ?

- Répliquer votre dépôt sur une ou plusieurs machines !
- Git est pensé pour gérer ce type de problème

# Gestion de version décentralisée

- Chaque utilisateur maintient une version du dépôt *local* qu'il peut changer à souhait
- Indépendant du commit, ils peuvent "pousser" une version sur un dépôt **distant**
- Un dépôt *local* peut avoir plusieurs dépôts **distant**s.

# Centralisé vs Décentralisé



Source [Geek for Geeks](#)

Cela rends la manipulation un peu plus complexe, allons-y pas à pas :-)



# Consulter l'historique de commits

- Dans votre workspace GitPod,
- positionnez-vous dans le dépôt esgi-ci-cd,
- modifiez le fichier README .md, puis commitez,
- et enfin, affichez l'historique de commits.

# ✓ Consulter l'historique de commits

```
cd /workspace/esgi-devops-2013
echo "A new line" >> ./README.md
git add ./README.md
git commit -m "add a new line to the documentation"

# Liste tous les commits présent sur la branche main.
git log --graph
```

Copy

# Associer un dépôt distant (1/2)

Git permet de manipuler des "remotes"

- Image "distante" (sur un autre ordinateur) de votre dépôt local.
- Permet de publier et de rapatrier des branches.
- Le serveur maintient sa propre arborescence de commits, tout comme votre dépôt local.
- Un dépôt peut posséder N remotes.

# Associer un dépôt distant (2/2)

```
# Liste les remotes associés à votre dépôt
git remote -v
# Vous devriez voir l'URL de votre dépôt en tant que 'origin'

## La commande 'git clone' a effectué l'action ci-dessous pour vous :
# git remote add origin https://<URL de votre dépôt>
```

 Copy

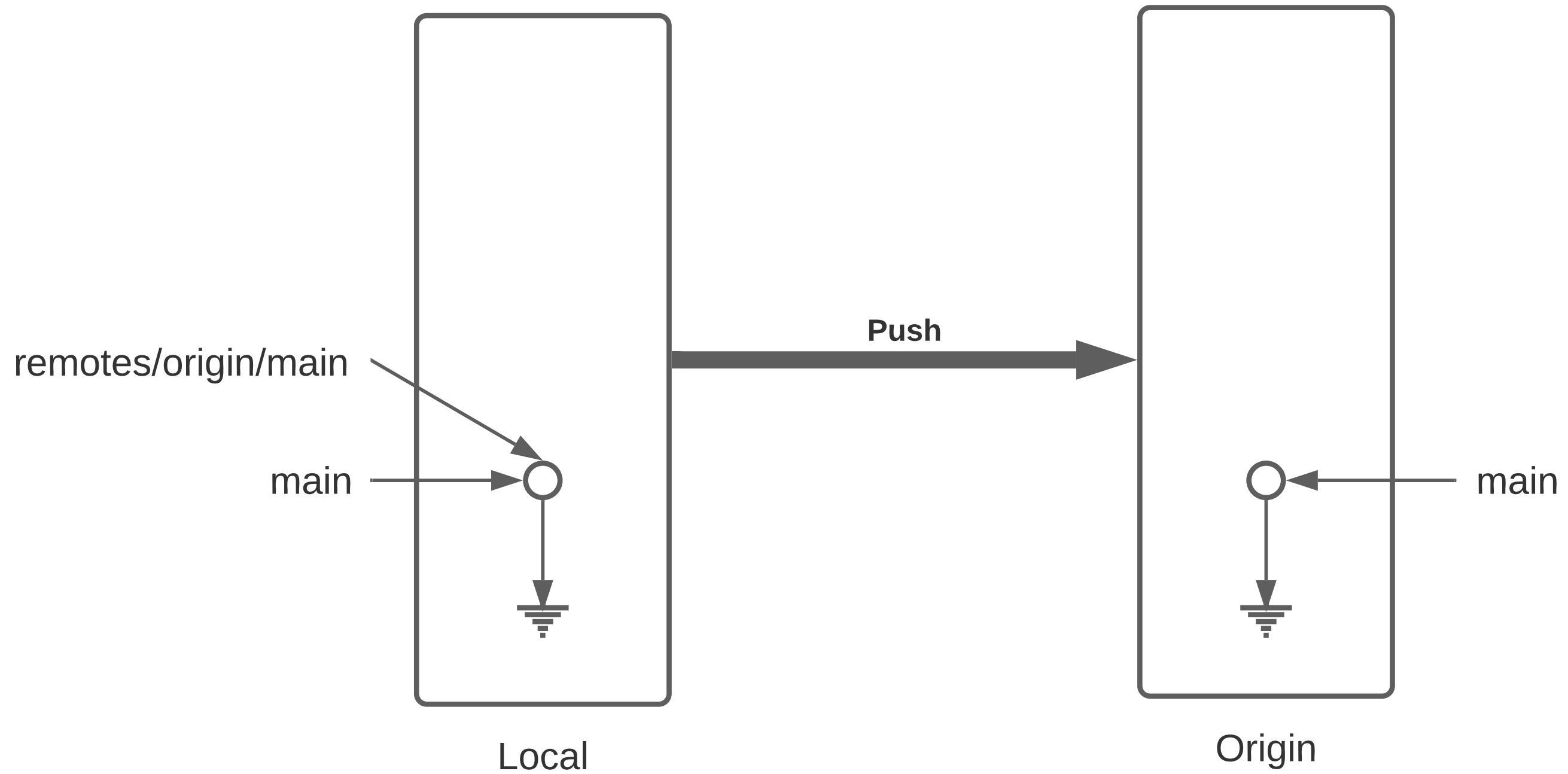
# Publier une branche dans sur dépôt distant

Maintenant qu'on a un dépôt, il faut publier notre code dessus !

```
# git push <remote> <votre_branche_courante>
git push origin main
```

 Copy

# Que s'est il passé ?

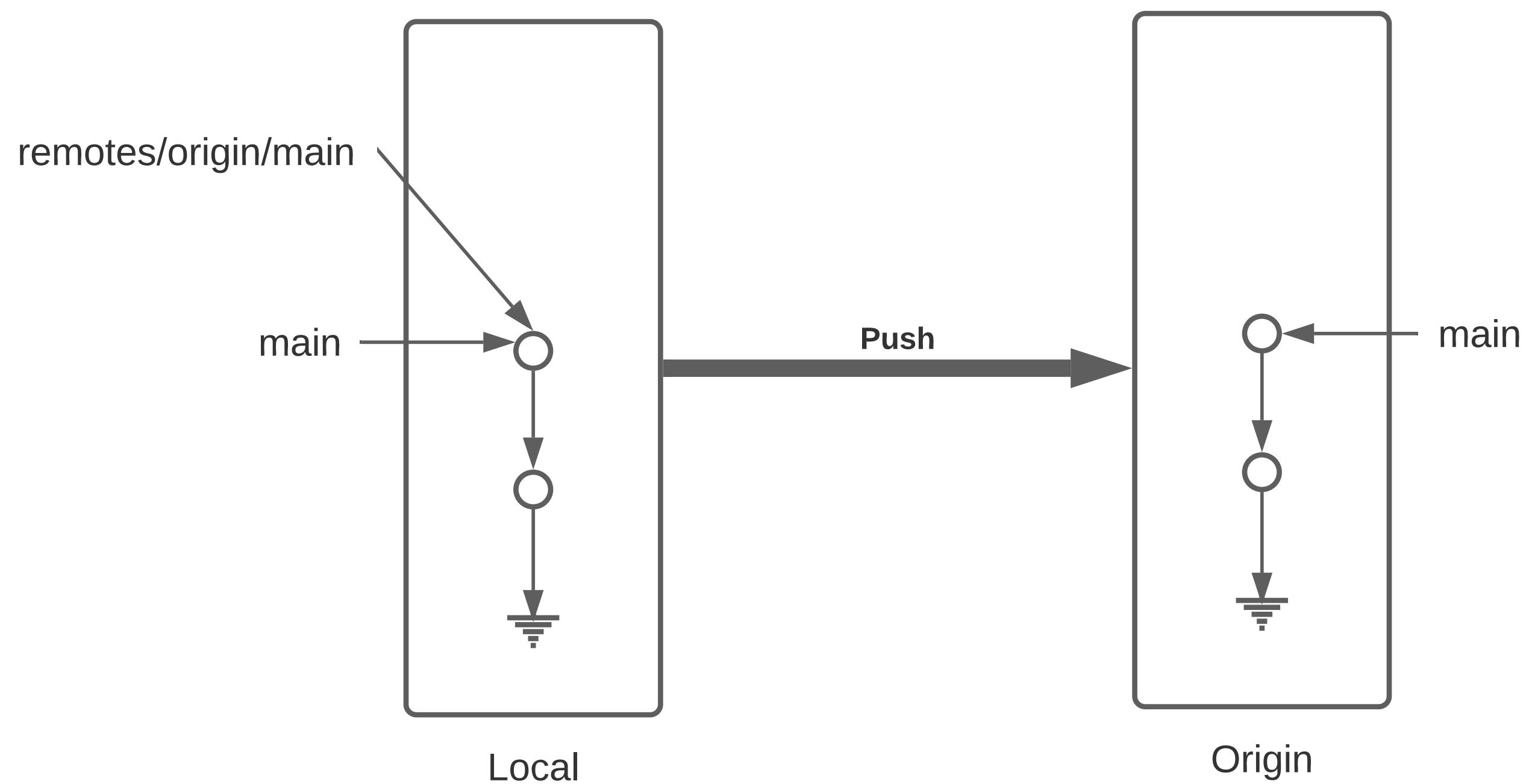


- git a envoyé la branche main sur le remote origin
- ... qui à accepté le changement et mis à jour sa propre branche main.
- git a créé localement une branche distante origin/main qui suis l'état de main sur le remote.
- Vous pouvez constater que la page github de votre dépôt affiche le code source

# Refaisons un commit !

```
git commit --allow-empty -m "Yet another commit"  
git push origin main
```

 Copy



# Branche distante

Dans votre dépôt local, une branche "distante" est automatiquement maintenue par git

C'est une image du dernier état connu de la branche sur le remote.

Pour mettre a jour les branches distantes depuis le remote il faut utiliser :

```
git fetch <nom_du_remote>
```

```
# Lister toutes les branches y compris les branches distances
git branch -a

# Notez qu'est listé remotes/origin/main

# Mets a jour les branches distantes du remote origin
git fetch origin

# Rien ne se passe, votre dépôt est tout neuf, changeons ça!
```

Copy



# Créez un commit depuis GitHub directement

- Cliquez sur le bouton éditer en haut à droite du "README"
- Changez le contenu de votre README
- Dans la section "Commit changes"
  - Ajoutez un titre de commit et une description
  - Cochez "Commit directly to the main branch"
  - Validez

GitHub crée directement un commit sur la branche main sur le dépôt distant

# ✓ Rapatrier les changements distants

```
# Mets à jour les branches distantes du dépôt origin
git fetch origin

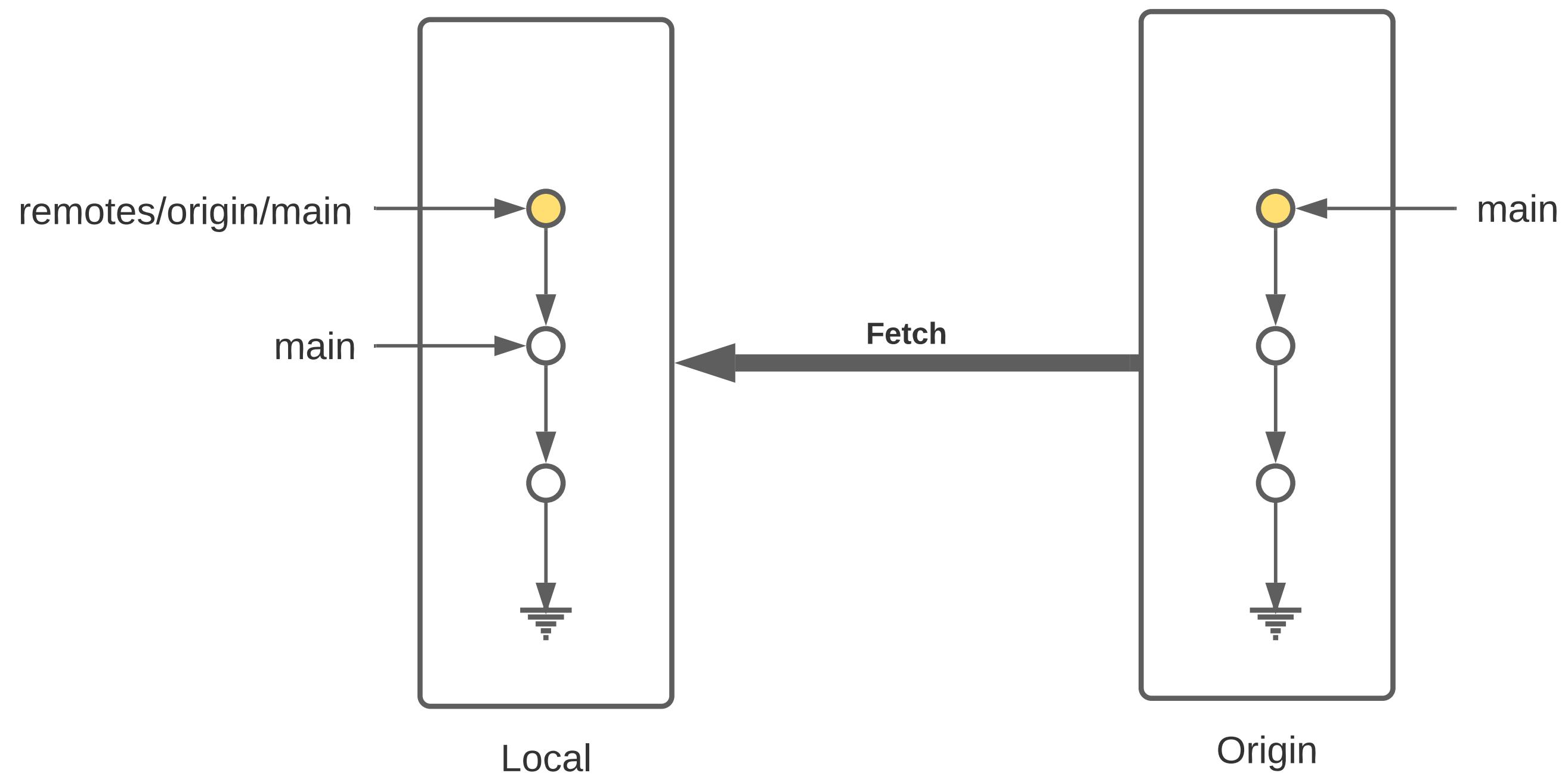
# La branche distante main a avancé sur le remote origin
# => La branche remotes/origin/main est donc mise à jour

# Ouvrez votre README
code ./README.md

# Mystère, le fichier README ne contient pas vos derniers changements?
git log

# Votre nouveau commit n'est pas présent, AHA !
```

 Copy



# Branche Distante VS Branche Locale

Le changement à été rapatrié, cependant il n'est pas encore présent sur votre branche main locale

```
# Merge la branch distante dans la branche locale.  
git merge origin/main
```

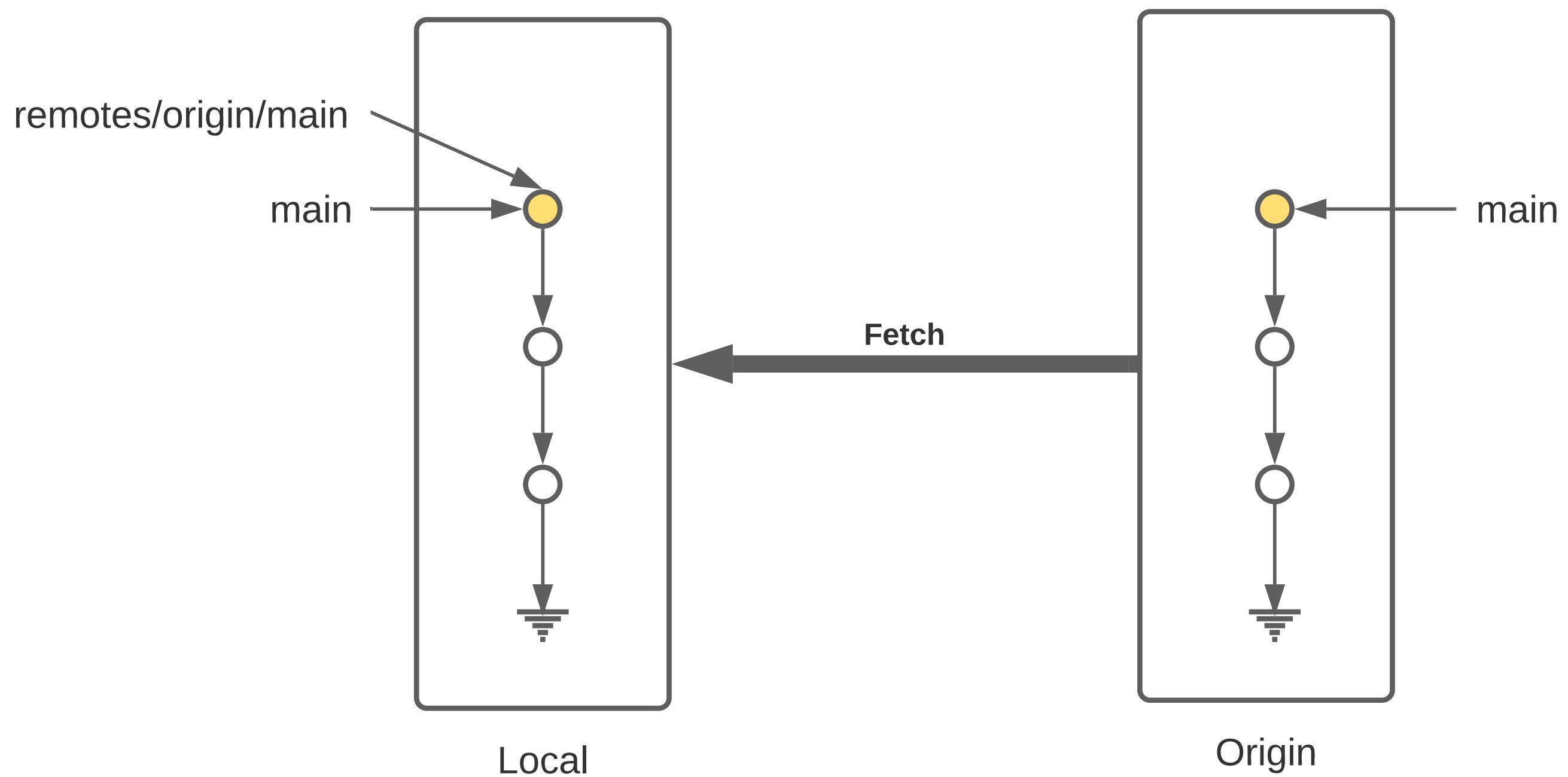
Copy

Vu que votre branche main n'a pas divergé (== partage le même historique) de la branche distante,  
git merge effectue automatiquement un "fast forward".

```
Updating 1919673..b712a8e
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
```

Copy

Cela signifie qu'il fait "avancer" la branche main sur le même commit que la branche  
origin/main



```
# Liste l'historique de commit  
git log  
  
# Votre nouveau commit est présent sur la branche main !  
# Juste au dessus de votre commit initial !
```

Copy

Et vous devriez voir votre changement dans le fichier README.md



# Git(Hub|Lab|teal...)

- Un dépôt distant peut être hébergé par n'importe quel serveur sans besoin autre qu'un accès SSH ou HTTPS.
- Une multitude de services facilitent et enrichissent encore git: (GitHub, Gitlab, Gitea, Bitbucket...)

# Checkpoint

⇒ git + Git(Hub|Lab|teal...) = superpowers ! 

- GUI de navigation dans le code
- Plateforme de gestion et suivi d'issues
- Plateforme de revue de code
- Intégration aux moteurs de CI/CD
- And so much more...

# Git à plusieurs

# Limites de travailler seul

- Capacité finie de travail
- Victime de propres biais
- On ne sait pas tout



# Travailler en équipe ? Une si bonne idée ?

- ... Mais il faut communiquer ?
- ... Mais tout le monde n'a pas les mêmes compétences ?
- ... Mais tout le monde y code pas pareil ?

**Collaborer c'est pas évident, mais il existe des outils et des méthodes pour vous aider.**

Cela reste des outils, ça ne résoud pas tout non plus.

# Git multijoueur

- Git permet de collaborer assez aisément
- Chaque développeur crée et publie des commits...
- ... et rapatrie ceux de ses camarades !
- C'est un outil très flexible... chacun peut faire ce qu'il lui semble bon !

... et (souvent) ça finit comme ça !

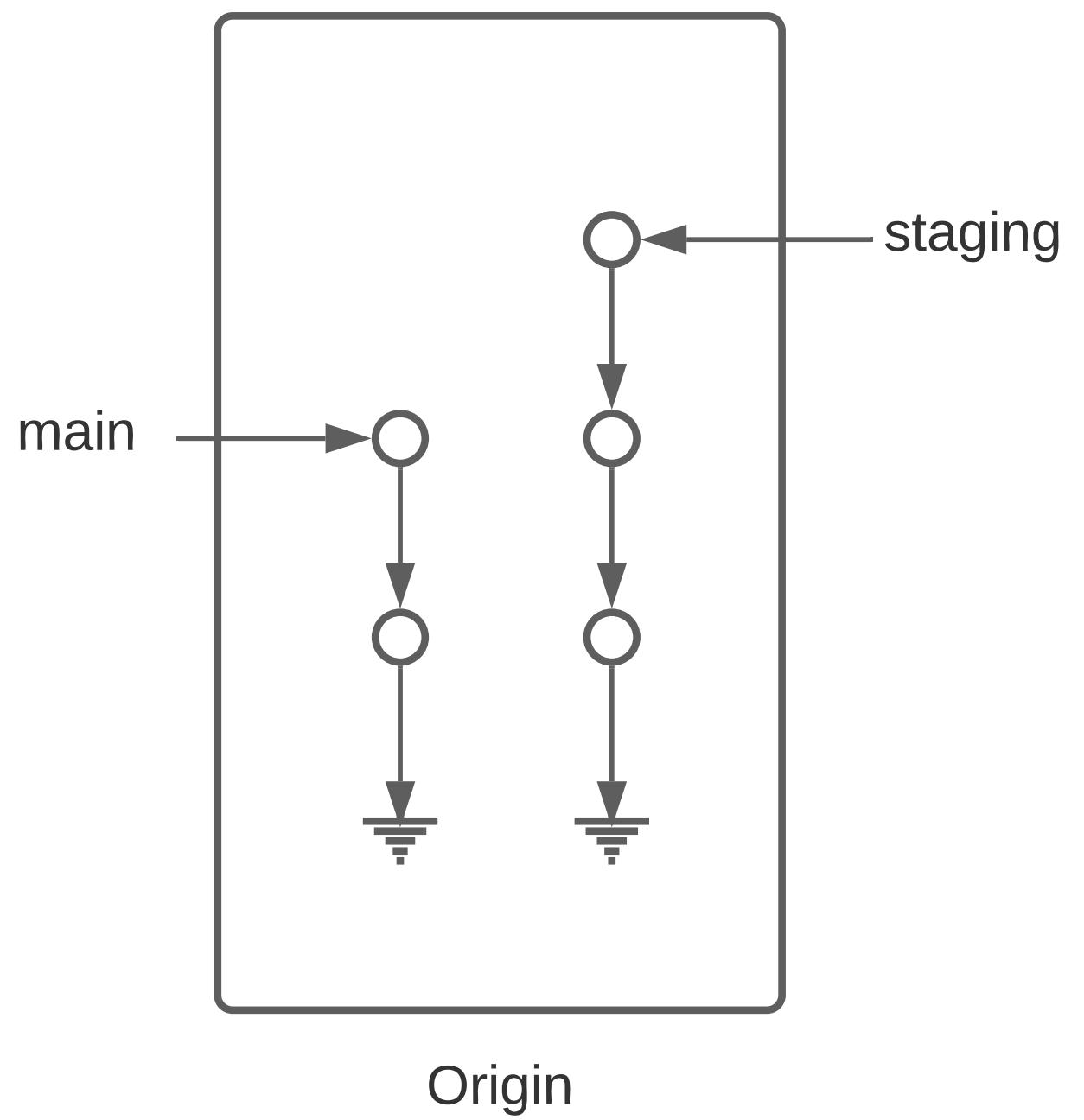


# Un Exemple de Git Flow

(Attachez vous aux idées générales... les détails varient d'un projet à l'autre!)

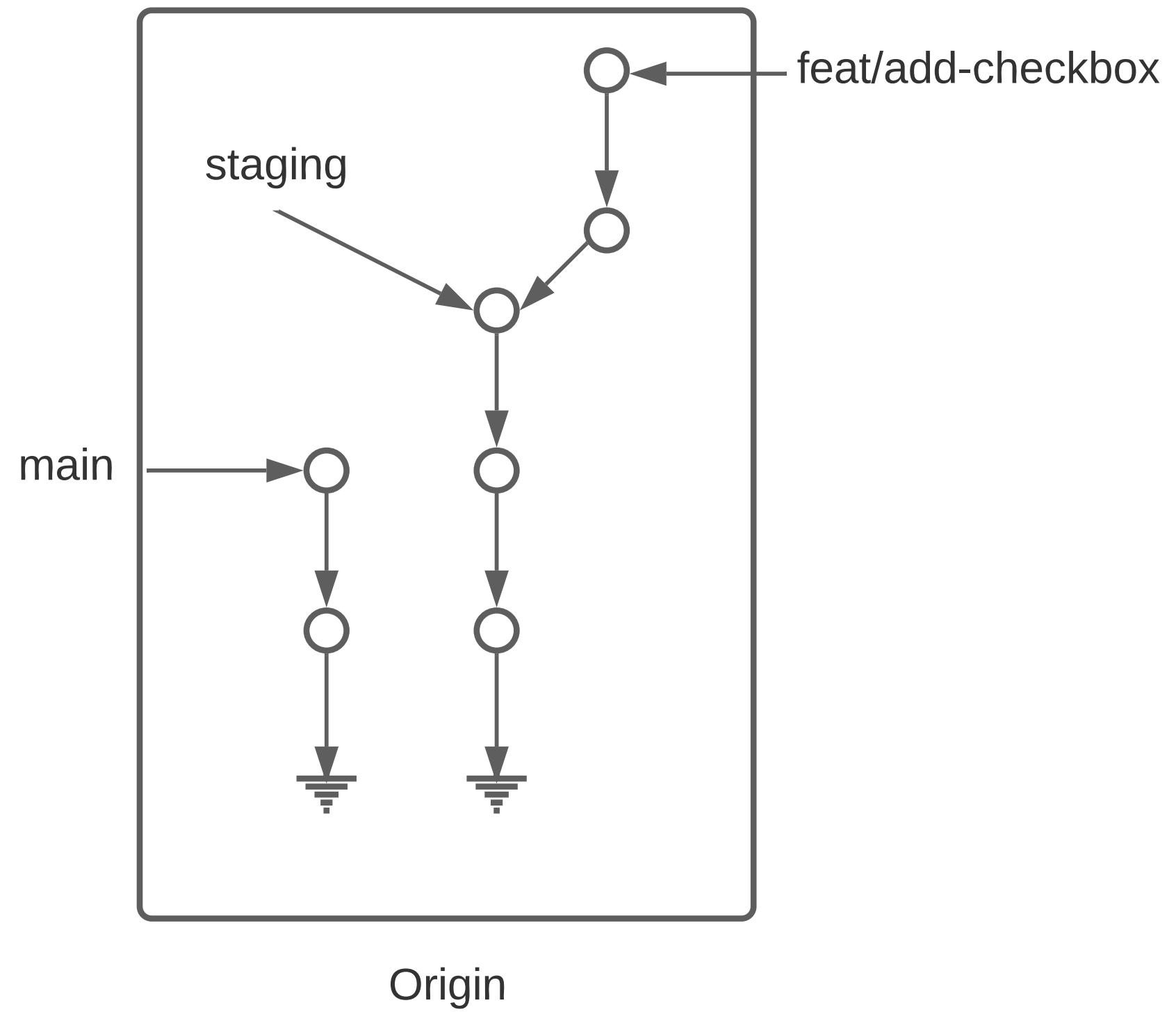
# Gestion des branches

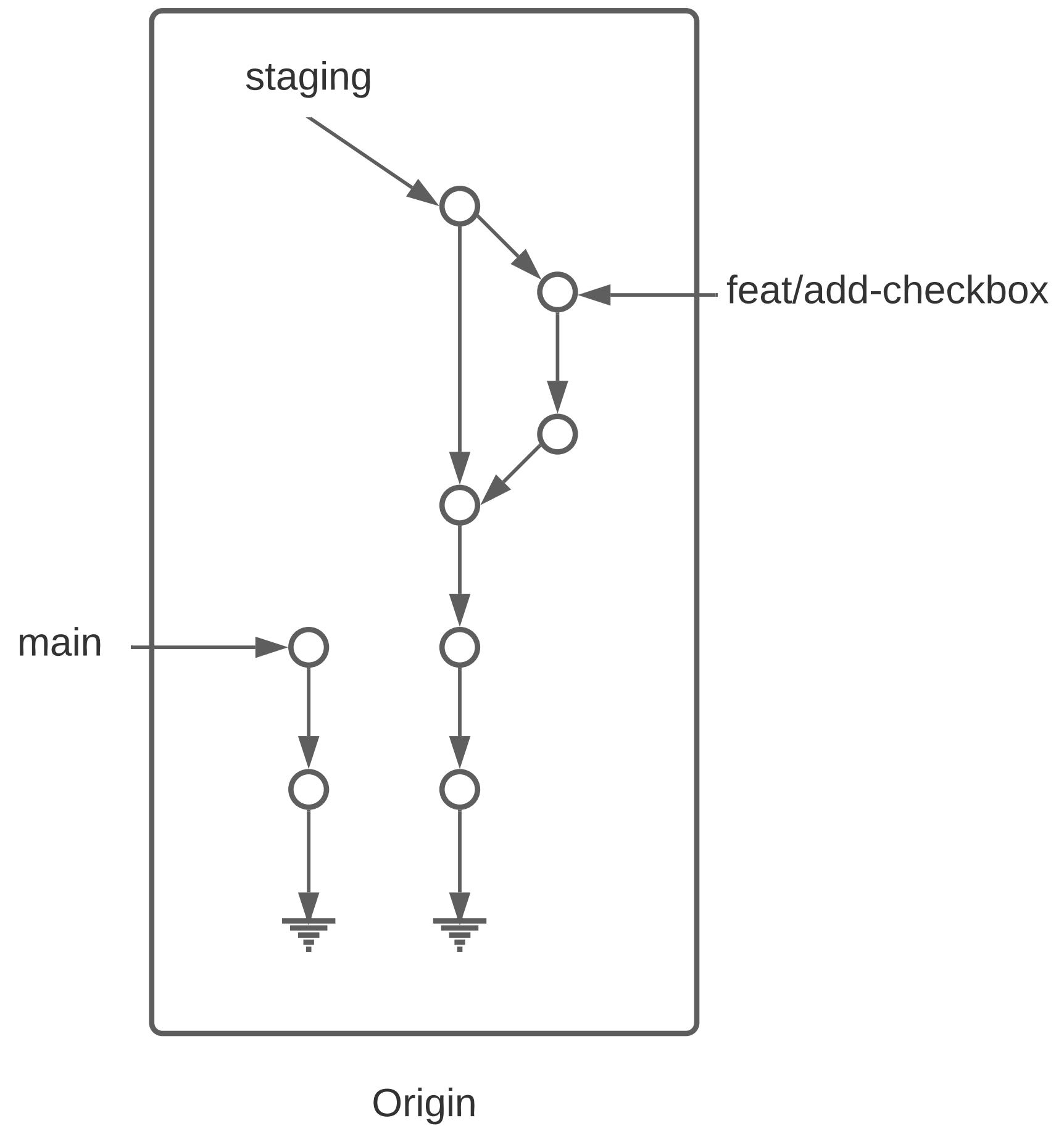
- Les "versions" du logiciel sont maintenues sur des branches principales (main, staging)
- Ces branches reflètent l'état du logiciel
  - **main**: version actuelle en production
  - **staging**: prochaine version



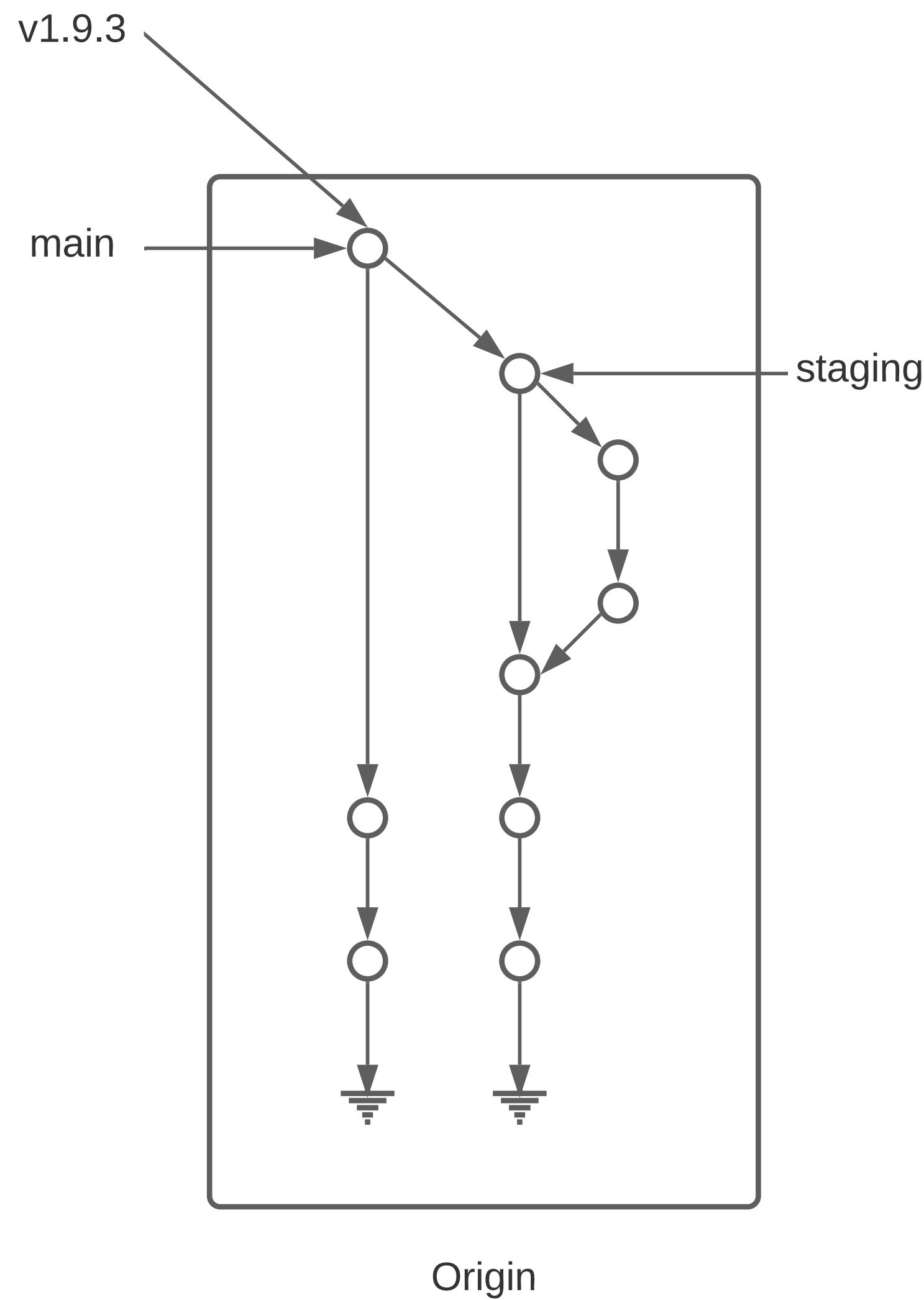
# Gestion des branches

- Chaque groupe de travail (développeur, binôme...)
  - Crée une branche de travail à partir de la branche staging
  - Une branche de travail correspond à **une chose à la fois**
  - Pousse des commits dessus qui implémentent le changement





Quand le travail est fini, la branche de travail est "mergée" dans staging



# Gestion des remotes

Où vivent ces branches ?

# Plusieurs modèles possibles

- Un remote pour les gouverner tous !
- Chacun son propre remote (et les commits seront bien gardés)
- ... whatever floats your boat!

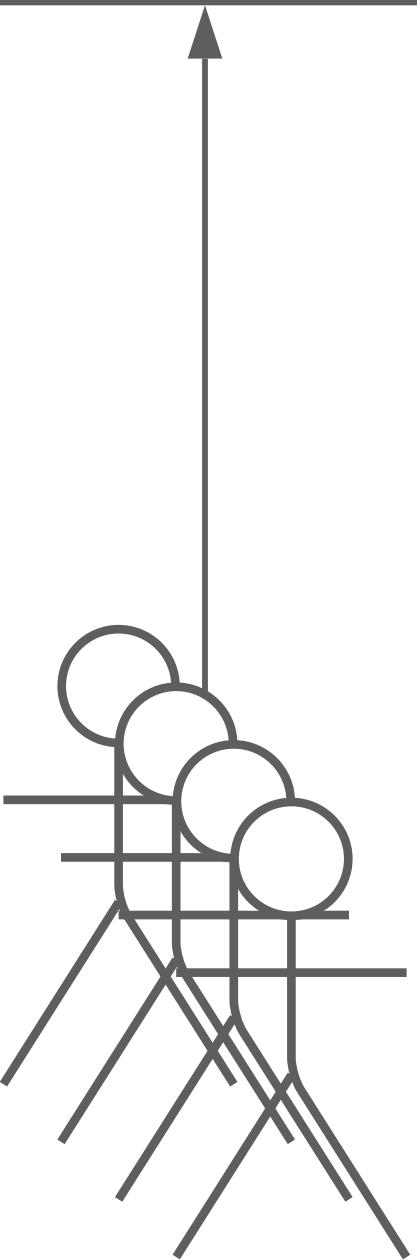
# Un remote pour les gouverner tous

Tous les développeurs envoient leur commits et branches sur le même remote

- Simple à gérer ...
- ... mais nécessite que tous les contributeurs aient accès au dépôt
  - Adapté à l'entreprise, peu adapté au monde de l'open source

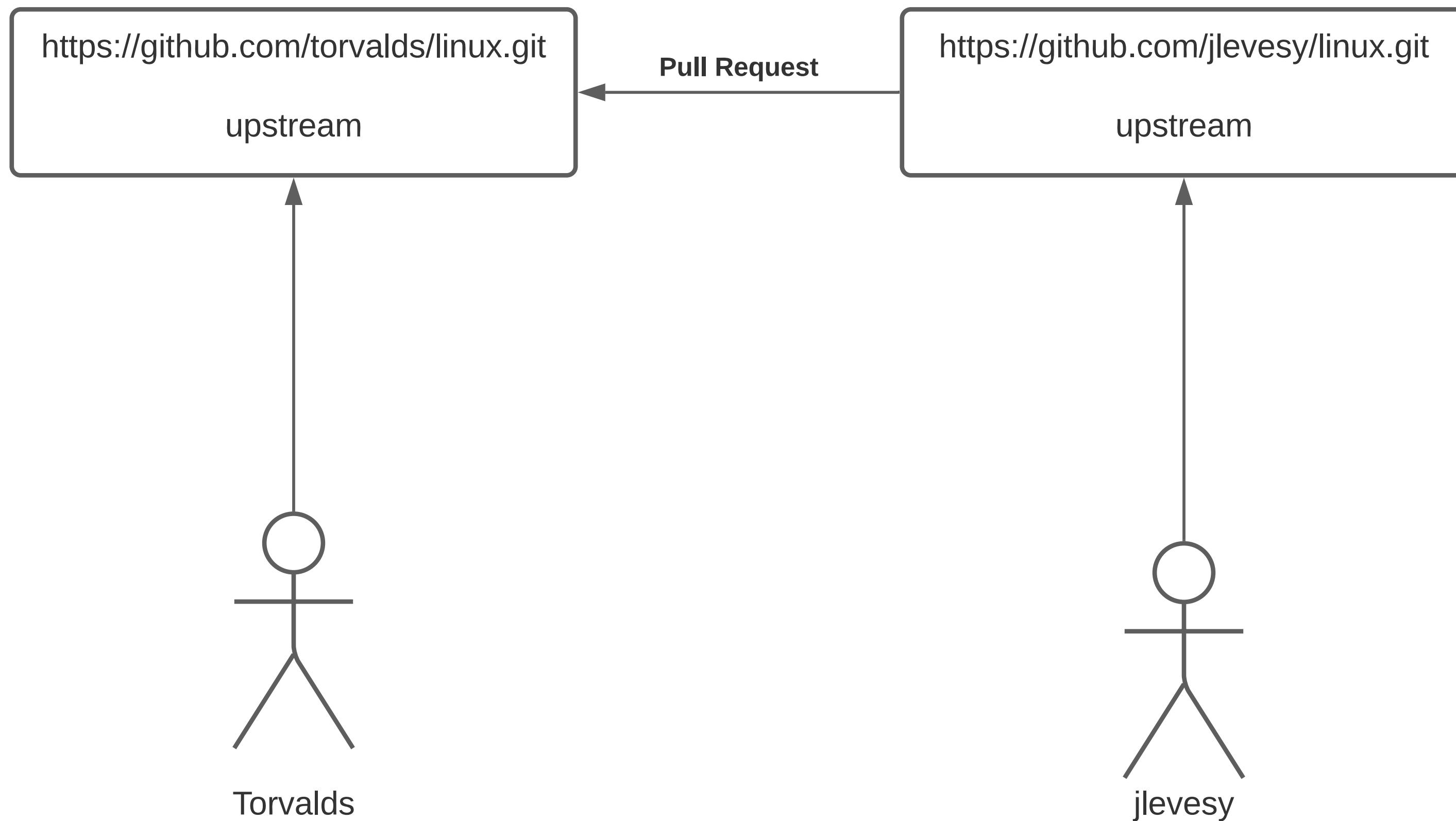
<https://github.com/torvalds/linux.git>

upstream



# Chacun son propre remote

- La motivation: le contrôle d'accès
  - Tout le monde peut lire le dépôt principal. Personne ne peut écrire dessus.
  - Tout le monde peut dupliquer le dépôt public et écrire sur sa copie.
  - Toute modification du dépôt principal passe par une procédure de revue.
  - Si la revue est validée, alors la branche est "mergée" dans la branche cible
- C'est le modèle poussé par GitHub !



# Forks ! Forks everywhere !

Dans la terminologie GitHub:

- Un fork est un remote copié d'un dépôt principal
  - C'est là où les contributeurs poussent leur branche de travail.
- Les branches de version (main, staging...) vivent sur le dépôt principal
- La procédure de ramener un changement d'un fork à un dépôt principal s'appelle la Pull Request (PR)

# Docker

"La Base"

# Pourquoi ?



🤔 Quel est le problème ?



matrixfromhell

Problème de temps **exponentiel**

# Déjà vu ?

L'IT n'est pas la seule industrie à résoudre des problèmes...

 also a matrix from hell

# Solution: Le conteneur intermodal

"Separation of Concerns"



blue shipping container

# Comment ça marche ?

"Virtualisation Légère"

 container vs vm

# Conteneur != VM

"Separation of concerns": 1 "tâche" par conteneur



# VMs && Conteneurs

Non exclusifs mutuellement



cont vm not excl

# Comment ça marche ?



# Exercice : Votre premier conteneur

C'est à vous (ouf) !

- Retournez dans Gitpod
- Dans un terminal, tapez la commande suivante :

```
docker container run hello-world  
# Équivalent de l'ancienne commande 'docker run'
```

Copy



# Anatomie

- Un service "Docker Engine" tourne en tâche de fond et publie une API REST
- La commande `docker run ...` a envoyé une requête POST au service
- Le service a téléchargé une **Image** Docker depuis le registre **DockerHub**,
- Puis a exécuté un **conteneur** basé sur cette image



# Exercice : Où est mon conteneur ?

C'est à vous !

```
docker container ls --help  
# ...  
docker container ls  
# ...  
docker container ls --all
```

Copy

⇒ 🤔 comment comprenez vous les résultats des 2 dernières commandes ?

# ✓ Solution : Où est mon conteneur ?

Le conteneur est toujours présent dans le "Docker Engine" même en étant arrêté

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES	Copy
109a9cdd3ec8	hello-world	"/hello"	33 seconds ago	Exited (0) 17 seconds ago		festive_faraday	

- Un conteneur == une commande "conteneurisée"
  - cf. colonne "**COMMAND**"
- Quand la commande s'arrête : le conteneur s'arrête
  - cf. code de sortie dans la colonne "**STATUS**"



# Exercice : Cycle de vie d'un conteneur simple

- Lancez un nouveau conteneur nommé bonjour
  - 💡 docker container run --help ou Documentation en ligne
- Affichez les "logs" du conteneur (==traces d'exécution écrites sur le stdout + stderr de la commande conteneurisée)
  - 💡 docker container logs --help ou Documentation en ligne
- Lancez le conteneur avec la commande docker container start
  - Regardez le résultat dans les logs
- Supprimez le container avec la commande docker container rm

# ✓ Solution : Cycle de vie d'un conteneur simple

```
docker container run --name=bonjour hello-world
# Affiche le texte habituel

docker container logs bonjour
# Affiche le même texte : pratique si on a fermé le terminal

docker container start bonjour
# N'affiche pas le texte mais l'identifiant unique du conteneur 'bonjour'

docker container logs bonjour
# Le texte est affiché 2 fois !

docker container ls --all
# Le conteneur est présent
docker container rm bonjour
docker container ls --all
# Le conteneur n'est plus là : il a été supprimé ainsi que ses logs

docker container logs bonjour
# Error: No such container: bonjour
```

Copy



# Que contient "hello-world" ?

- C'est une "image" de conteneur, c'est à dire un modèle (template) représentant une application auto-suffisante.
  - On peut voir ça comme un "paquetage" autonome
- C'est un système de fichier complet:
  - Il y a au moins une racine /
  - Ne contient que ce qui est censé être nécessaire (dépendances, librairies, binaires, etc.)

# Docker Hub

- <https://hub.docker.com/> : C'est le registre d'images "par défaut"
  - Exemple : Image officielle de conteneur "Ubuntu"
- 🎓 Cherchez l'image hello-world pour en voir la page de documentation
  - 💡 pas besoin de créer de compte pour ça
- Il existe d'autre "registres" en fonction des besoins (GitHub GHCR, Google GCR, etc.)



# Exercice : conteneur interactif

- Quel distribution Linux est utilisée dans le terminal Gitpod ?
  - 💡 Regardez le fichier `/etc/os-release`
- Exécutez un conteneur interactif basé sur `alpine:3.17` (une distribution Linux ultra-légère) et regardez le contenu du fichier au même emplacement
  - 💡 `docker container run --help`
  - 💡 Demandez un `tty` à Docker
  - 💡 Activez le mode interactif
- Exécutez la même commande dans un conteneur basé sur la même image mais en **NON** interactif
  - 💡 Comment surcharger la commande par défaut ?

# ✓ Solution : conteneur interactif

```
$ cat /etc/os-release
# ... Ubuntu ...

$ docker container run --tty --interactive alpine:3.17
/ # cat /etc/os-release
# ... Alpine ...
# Notez que le "prompt" du terminal est différent DANS le conteneur
/ # exit
$ docker container ls --all

$ docker container run alpine:3.17 cat /etc/os-release
# ... Alpine ...
```

Copy



# Exercice : conteneur en tâche de fond

- Exécutez un conteneur, basé sur l'image nginx en tâche de fond ("Background"), nommé webserver-1
  - 💡 On parle de processus "détaché" (ou bien "démonisé")
  - ⚠ Pensez bien à docker container ls
- Regardez le contenu du fichier /etc/os-release dans ce conteneur
  - 💡 docker container exec
- Essayez d'arrêter, démarrer puis redémarrer le conteneur
  - ⚠ Pensez bien à docker container ls à chaque fois
  - 💡 stop, start, restart

# ✓ Solution : conteneur en tâche de fond

```
docker container run --detach --name=webserver-1 nginx
# <ID du conteneur>

docker container ls
docker container ls --all

docker container exec webserver-1 cat /etc/os-release
# ... Debian ...

docker container stop webserver-1
docker container ls
docker container ls --all

docker container start webserver-1
docker container ls
docker container ls --all

docker container start webserver-1
docker container ls
```

Copy

# Checkpoint



- Docker essaie de résoudre le problème de l'empaquetage le plus "portable" possible
    - On n'en a pas encore vu les effets, ça arrive !
  - Vous avez vu qu'un conteneur permet d'exécuter une commande dans un environnement "préparé"
    - Catalogue d'images Docker par défaut : Le Docker Hub
  - Vous avez vu qu'on peut exécuter des conteneurs selon 3 modes :
    - "One shot"
    - Interactif
    - En tâche de fond
- ⇒ 🤔 Mais comment ces images sont-elles fabriquées ? Quelle confiance leur accorder ?

# Docker Images



# Pourquoi des images ?

- Un **conteneur** est toujours exécuté depuis une **image**.
- Une **image de conteneur** (ou "Image Docker") est un modèle ("template") d'application auto-suffisant.

⇒ Permet de fournir un livrable portable (ou presque).

 Application Auto-Suffisante ?

 docker app self sufficient

# C'est quoi le principe ?

 dockerfile flow



# Pourquoi fabriquer sa propre image ?

Problème :

```
cat /etc/os-release
# ...
git --version
# ...

# Même version de Linux que dans GitPod
docker container run --rm ubuntu:22.04 git --version
# docker: Error response from daemon: failed to create shim task: OCI runtime create failed: runc create failed: unable t

# En interactif ?
docker container run --rm --tty --interactive ubuntu:22.04 bash
root@e72555a4f95a:/# git --version
# bash: git: command not found
```

Copy



# Fabriquer sa première image

- **But :** fabriquer une image Docker qui contient git
- Dans votre workspace Gitpod, créez un dossier nommé docker-git/
- Dans ce dossier, créer un fichier Dockerfile avec le contenu ci-dessous :

```
FROM ubuntu:22.04
RUN apt-get update && apt-get install --yes --no-install-recommends git
```

Copy

- Fabriquez votre image avec la commande

```
docker image build --tag=docker-git /workspace/docker-git
```

Copy

- Testez l'image fraîchement fabriquée

- docker image ls

# ✓ Fabriquer sa première image

```
cat <<EOF >Dockerfile
FROM ubuntu:22.04
RUN apt-get update && apt-get install --yes --no-install-recommends git
EOF

docker image build --tag=docker-git ./

docker image ls | grep docker-git

# Doit fonctionner
docker container run --rm docker-git:latest git --version
```

Copy

# Conventions de nommage des images

```
[REGISTRY/] [NAMESPACE/] NAME [:TAG | @DIGEST]
```

Copy

- Pas de Registre ? Défaut: registry.docker.com
- Pas de Namespace ? Défaut: library
- Pas de tag ? Valeur par défaut: latest
  - $\triangle$  Friends don't let friends use latest
- Digest: signature unique basée sur le contenu

# Conventions de nommage : Exemples

- ubuntu:22.04 ⇒ [registry.docker.com/library/ubuntu:22.04](https://registry.docker.com/library/ubuntu:22.04)
- dduportal/docker-asciidoc ⇒  
[registry.docker.com/dduportal/docker-asciidoc:latest](https://registry.docker.com/dduportal/docker-asciidoc:latest)
- ghcr.io/dduportal/docker-asciidoc:1.3.2@sha256:xxxx



# Utilisons les tags

- Rappel : Friends don't let friends use latest
- Il est temps de "taguer" votre première image !

```
docker image tag docker-git:latest docker-git:1.0.0
```

Copy

- Testez le fonctionnement avec le nouveau tag
- Comparez les 2 images dans la sortie de docker image ls

# ✓ Utilisons les tags

```
docker image tag docker-git:latest docker-git:1.0.0

# 2 lignes
docker image ls | grep docker-git
# 1 ligne
docker image ls | grep docker-git | grep latest
# 1 ligne
docker image ls | grep docker-git | grep '1.0.0'

# Doit fonctionner
docker container run --rm docker-git:1.0.0 git --version
```

Copy



# Mettre à jour votre image (1.1.0)

- Mettez à jour votre image en version 1.1.0 avec les changements suivants :
  - Ajoutez un `LABEL` dont la clef est `description` (et la valeur de votre choix)
  - Configurez git pour utiliser une branche `main` par défaut au lieu de `master` (commande  
`git config --global init.defaultBranch main`)
- Indices :
  - 💡 Commande `docker image inspect <image name>`
  - 💡 Commande `git config --get init.defaultBranch` (dans le conteneur)
  - 💡 Ajoutez des lignes **à la fin** du Dockerfile
  - 💡 Documentation de référence des Dockerfile

# ✓ Mettre à jour votre image (1.1.0)

```
# Dockerfile
FROM ubuntu:22.04
RUN apt-get update && apt-get install --yes --no-install-recommends git
LABEL description="Une image contenant git préconfiguré"
RUN git config --global init.defaultBranch main

docker image build -t docker-git:1.1.0 ./docker-git/
# [+] Building 3.1s (7/7) FINISHED
# => [internal] load build definition from Dockerfile
# => => transferring dockerfile: 235B
# => [internal] load .dockerignore
# => => transferring context: 2B
# => [internal] load metadata for docker.io/library/ubuntu:22.04
# => [1/3] FROM docker.io/library/ubuntu:22.04
# => CACHED [2/3] RUN apt-get update && apt-get install --yes --no-install-recommends git
# => [3/3] RUN git config --global init.defaultBranch main
# => exporting to image
# => => exporting layers
# => => writing image sha256:66b3733ff8dbd0cd8968e885fbf8a87c06a4d6ceca8eecb2d0a5ad40145bca1c
# => => naming to docker.io/library/docker-git:1.1.0

docker container run --rm docker-git:1.0.0 git config --get init.defaultBranch
docker container run --rm docker-git:1.1.0 git config --get init.defaultBranch
# main
```

Copy

Copy

# Cache d'images & Layers

```
=> CACHED [2/3] RUN apt-get update && apt-get install --yes --no-install-recommends git
```

 Copy

 En fait, Docker n'a PAS exécuté cette commande la seconde fois ⇒ ça va beaucoup plus vite !



 Essayez de voir les layers avec (dans Gitpod) `gitpod` `gitpod` `dive <image>:<tag>`



# Cache d'images & Layers

- **But :** manipuler le cache d'images
- Commencez par vérifier que le cache est utilisé : relancez la dernière commande `docker image build` (plusieurs fois s'il le faut)
- Invalidez le cache en ajoutant le paquet APT `make` à installer en même temps que `git`
  - `^ Tag 1.2.0`
- Vérifiez que le cache est bien présent de nouveau

# ✓ Cache d'images & Layers

```
# Build one time
docker image build -t docker-git:1.1.0 ./docker-git/
# Second time is fully cached
docker image build -t docker-git:1.1.0 ./docker-git/

cat Dockerfile
# FROM ubuntu:22.04
# RUN apt-get update && apt-get install --yes --no-install-recommends git make
# LABEL description="Une image contenant git préconfiguré"
# RUN git config --global init.defaultBranch main

# Build one time
docker image build -t docker-git:1.2.0 ./docker-git/
# Second time is fully cached
docker image build -t docker-git:1.2.0 ./docker-git/

## Vérification
# Renvoie une erreur
docker run --rm docker-git:1.1.0 make --version
# Doit fonctionner
docker run --rm docker-git:1.2.0 make --version
```

Copy

# Checkpoint



- Une image Docker fournit un environnement de système de fichier auto-suffisant (application, dépendances, binaries, etc.) comme modèle de base d'un conteneur
- Les images Docker ont une convention de nommage permettant d'identifier les images très précisément
- On peut spécifier une recette de fabrication d'image à l'aide d'un `Dockerfile` et de la commande `docker image build`

⇒ 🤔 et si on utilisait Docker pour nous aider dans l'intégration continue ?

# CI avec Docker Images

# Personnaliser l'environnement GitHub Actions

Plusieurs solutions existent, chacune avec ses avantages / inconvénients :

- Personnaliser l'environnement dans votre workflow: ( $\Delta$  sensible aux mises à jour,  $\checkmark$  facile à mettre en place)
  -  C'est ce qu'on à fait dans nos workflows précédemment
- Spécifier un environnement préfabriqué pour le workflow ( $\Delta$  complexe,  $\checkmark$  portable)
  - C'est ce qu'on va essayer avec Docker 



# Exercice : Environnement préfabriqué simple

- **But** : exécuter le workflow dans un environnement le plus proche possible du développement
  - En utilisant le même environnement que GitPod (**même version** de node et npm)
- C'est à vous de mettre à jour le workflow pour exécuter les étapes dans la même image Docker que GitPod :
  - Image utilisée dans GitPod
  - Utilisation d'un container comme runner GitHub Actions
  - Contraintes d'exécution de container dans GitHub Actions (--user=root)

# ✓ Solution : Environnement préfabriqué simple

```
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    container:
      image: ghcr.io/dduportal/esgi-gitpod
      options: --user=root
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: npm install
      - run: npm run lint
```

Copy

- Quel est l'impact en terme de temps d'exécution du changement précédent ?
- **Problème :** Le temps entre une modification et le retour est crucial



wait here



# Exercice : Environnement préfabriqué local

- **But :** Utiliser un Dockerfile pour fabriquer une image à utiliser
- 🤔 GitHub Actions ne permet pas de spécifier un Dockerfile
- C'est à vous. Mettez à jour votre workflow pour :
  - Fabriquer une image Docker contenant npm et node, même versions que dans GitPod et qui installe les dépendances
    - 💡 Pensez à copier le code source !
  - Exécuter la commande `npm run lint` dans un conteneur basé sur cette image

# ✓ Exercice : Environnement préfabriqué local

```
# Dockerfile
FROM node:19
WORKDIR /app
COPY ./package.json /app/
COPY ./package-lock.json /app/
RUN npm ci
COPY ./index.js /app/

# bonjour.yml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    env:
      IMAGE_NAME: "mynode:19"
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: docker image build --tag="${IMAGE_NAME}" ./
      - run: docker container run "${IMAGE_NAME}" npm run lint
```

 Copy

 Copy



## Réfléchissons ensemble

- Impact sur le temps de build ?
- Quelles limites voyez-vous ?

# Checkpoint



- On peut exécuter des tâches de CI dans des images de container
- Un curseur est à positionner entre des images préfabriquées qui peuvent être lourdes ou fabriquer soit-même sa propre image

# Docker : Volumes



# Quel est le problème ?

- Faut-il fabriquer sa propre image à chaque fois ?
  - Ref. copie du README pour utiliser `cowsay`
- Comment partager des fichiers entre conteneurs ?
- Et entre hôte et conteneur ?

# Problème des Layers

- ⏳ Performances : les layers ont de très mauvais performance I/O (disque)
- 💾 Si vous supprimez un conteneur, ses layers disparaissent
- ✗ Pas de partage de layer possible entre conteneurs

# Docker Volumes

- Ce sont des dossier stockés sur l'hôte,
- gérés par le Docker Engine,
- Pouvant être partagés,
- Et survivant aux conteneurs

⇒ Parfait 



# Votre premier volume

- Vérifiez que le dossier /app n'existe pas au préalable :

```
docker container run --rm alpine ls -la /app
```

[Copy](#)

- Essayez maintenant avec un volume :

```
docker volume create app-data
docker container run --rm --volume=app-data:/app alpine ls -la /app
docker volume ls
```

[Copy](#)

- Essayez de créer un fichier dedans :

```
docker container run --rm --volume=app-data:/app alpine touch /app/
```

[Copy](#)

```
docker container run --rm --volume=app-data:/app alpine ls -la /app
```

# Partager avec l'hôte

- Concept : partager un dossier/fichier de l'hôte
  - Contrainte : chemin absolu
- ⚠ à n'utiliser que pour des commandes "one shot"
  - Entorse à la portabilité (file owner, permission, contrainte du chemin)
- 🎓 Dans GitPod, essayez la commande suivante :

```
docker container run --rm --volume /workspace:/app alpine ls -la /app
```

 Copy



# CI : Partager le dépôt dans le conteneur

- Revenons à l'image Docker dans GitHub Actions
- Enlevez la ligne `COPY ./ /app` du workflow
- Utilisez un partage avec l'hôte pour le fichier source `index.js`
- 💡 `--workdir=\"$ (pwd) "`

# ✓ CI : Partager le dépôt dans le conteneur

```
# Dockerfile
FROM node:19
WORKDIR /app
COPY package.json /app/
COPY package-lock.json /app/
RUN npm ci
# COPY ./index.js /app/

# bonjour.yml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    env:
      IMAGE_NAME: "mynode:19"
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: docker image build --tag="${IMAGE_NAME}" .
    # Uncomment to validate your work (should fail the next step)
    # - run: echo 'Hello Error' > ./index.js
      - run: docker container run --volume="$(pwd)":$(pwd) --workdir="$(pwd)" "${IMAGE_NAME}" npm run lint
```

Copy

Copy



# Sécurité des conteneurs

- **Buts :**
  - Limiter l'impact d'un processus malicieux
  - S'assurer des meilleures performances de production
  - Obtenir une liste exhaustive des dossier utilisés par l'application
- Essayez d'exécuter un conteneur `webserver-2`, basé sur l'image `nginx`, avec l'option `--read-only` activée
  - 💡 Pensez à regarder les logs si le conteneur est en erreur
  - 💡 Déclarez un volume pour chaque dossier/fichier nécessitant une écriture

# ✓ Sécurité des conteneurs

```
docker container run --detach --name=webserver-2 --read-only \  
--volume=/var/cache/nginx --volume=/var/run nginx
```

Copy

# Checkpoint



- Docker sépare le cycle de vie des conteneurs de celui des données
  - Utile pour les mises à jours d'images
- Les données des conteneurs peuvent être partagées en utilisant des volumes
- Bonne pratique : utiliser les options de volumes de Docker permet de mieux communiquer les caractéristiques et contraintes de l'application

⇒ DevOps The Right Way 

# Docker : Réseau



# Quel est le problème ?

- Comment accéder aux serveurs webs dans des conteneurs ?

# Réseau dans Docker

- Docker utilise des réseaux virtuels privés
  - Un peu comme votre box internet
- Votre point d'entrée : la commande `docker network`
  - 🎓 Affichons la liste des réseaux par défaut :

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
9affa4f4faa9	bridge	bridge	local
d454eb8ccca5a	host	host	local
80a73237a778	none	null	local

Copy

# Réseau Bridge default



docker bridge 1



# Comment accéder au serveur web en tâche de fond ?

- Assurez-vous que le conteneur `webserver-1` est toujours en fonctionnement
  - Sinon: 🎓 Exercice : conteneur en tâche de fond

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ee5b70fa72c3	nginx	"/docker-entrypoint...."	3 seconds ago	Up 2 seconds	80/tcp	webserver-1

Copy

⇒ Super, le port 80 (TCP) est annoncé (on parle d'"exposé"). Mais où ?



# Un client HTTP en ligne de commande

En utilisant la ligne de commande `curl`, affichez en ligne de commande :

- Le contenu (HTML) de la page `http://google.com`
- Les en-tête HTTP (requête et réponse) associés à la page `http://google.com`
- Le contenu de la page vers laquelle `http://google.com` redirige



`man curl / https://curl.se/docs/manpage.html`

# ✓ Un client HTTP en ligne de commande

```
curl http://google.com
# <HTML> ...

curl --verbose --output /dev/null http://google.com
#< HTTP/1.1 301 Moved Permanently
#< Location: http://www.google.com/
# ...

curl --location http://google.com
# <!doctype html> ...
```

 Copy



## Accéder au serveur web via le réseau privé

- **But :** Affichez la page du serveur web qui tourne dans le conteneur `webserver-1`
- Obtenez l'adresse IP privée du conteneur avec la commande `docker container inspect`
- Utilisez la command `curl`

# ✓ Accéder au serveur web via le réseau privé

```
docker container inspect webserver-1
# ...
# docker inspect webserver-1 | grep IPAddress

curl http://172.17.0.x:80 # x peut changer
# ...
<title>Welcome to nginx!</title>
# ...

docker container stop webserver-1

curl http://172.17.0.x:80
# Erreur
```

Copy

# Réseau privé / public

- Un réseau privé doit le rester !
- Solution : publier le(s) port(s) sur les interface réseau publiques



# Accéder au serveur web via un port publié

- **But :** Créez un nouveau conteneur `webserver-public` accessible publiquement
- Utilisez le port 8080 publique
- 💡 Flag `--publish` pour `docker container run`
- 💡 GitPod va vous proposer un popup : choisissez "Open Browser"

# ✓ Accéder au serveur web via un port publié

```
docker container run --detach --name=webserver-public --publish 8080:80 nginx  
1c5020a818887b1eb4b14b7e75f490db97fac4172c96cc918df63c8c2a0fbaff
```

 Copy

```
curl http://localhost:8080  
# ...
```

# Autres Réseaux



docker networks all



## Autre Réseaux

- Comparons les interfaces réseaux de la VM GitPod et de 3 conteneurs basés sur alpine:3.17 :
  - Un conteneur dans le réseau default
  - Un conteneur avec --network=host
  - Un conteneur avec --network=none
- 💡 Commande ip addr pour afficher les interfaces réseaux sous Linux

# ✓ Autre Réseaux

```
$ ip addr
1: lo: # ...
    inet 127.0.0.1/8 scope host lo
# ...
4: docker0: # ...
# ...

$ docker container run --rm alpine:3.17 ip addr
1: lo: # ...
    inet 127.0.0.1/8 scope host lo
# ...
    inet 172.17.0.x/16  # ...
# ...

$ docker container run --rm --network=host alpine:3.17 ip addr
1: lo: # ...
    inet 127.0.0.1/8 scope host lo
# ...
4: docker0: # ...
# ...

gitpod /workspace $ docker container run --rm --network=none alpine:3.17 ip addr
1: lo: # ...
    inet 127.0.0.1/8 scope host lo
```

Copy

# Réseaux personnalisés

- Vous pouvez créer vos propres réseaux isolés les un des autres
- Le fonctionnement reste le même (IP privées, ports à publier)
- Avantage: Pour chaque réseau "bridge" (hors default, Docker fournit un serveur DNS automatique !)



# Réseaux personnalisés 1/2

- Créez un réseau nommé esgi-1 avec la commande `docker network create`
- Exécuter un conteneur avec les propriétés suivantes :
  - Nom : `webserver-private`
  - Type : détaché (tâche de fond)
  - Dans le réseau esgi-1 ( `--network=`)
  - Image : `nginx`



## Réseaux personnalisés 2/2

- Exécutez un second conteneur interactif
  - Interactif
  - --entrypoint=bash
  - Réseau attaché : esgi-1
  - Image : nginx
- Essayez la commande curl <http://webserver-private>

# ✓ Réseaux personnalisés

```
docker network create ls
docker network create esgi-1
docker network create ls

docker container run --detach --network=esgi-1 --name=webserver-private nginx

docker container run --rm --tty --interactive --network=esgi-1 --entrypoint=bash nginx
root@ac99e0beb95d:/# curl --verbose http://webserver-private
*   Trying 172.18.0.2:80...
* Connected to webserver-private (172.18.0.2) port 80 (#0)
> GET / HTTP/1.1
> Host: webserver-private
> User-Agent: curl/7.74.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
# ...
```

Copy

# Checkpoint



- Docker gère les réseaux automatiquement pour vous
- Un conteneur peut exposer un service sur son réseau privé et également le publier
- Les réseaux personnalisés fournissent un serveur de nom de domaines intégré 

# GitHub Forks



## Exercice: Créez un fork

- Nous allons vous faire forker vos dépôts respectifs
- Trouvez vous un binôme dans le groupe et échangez vos URL de dépôts
- Depuis la page du dépôt de votre binôme, cliquez en haut à droite sur le bouton **Fork**.





À vous de jouer : dans le dépôt de votre binôme, vous allez ajouter un étape de "lint"



# Exercice: Contribuez au projet de votre binôme (1/5)

Première étape: on clone le fork dans son environnement de développement

```
cd /workspace/  
  
# Clonez votre fork  
git clone <url_de_votre_fork>  
  
# Se placer dans le fork  
cd <nom_du_fork>  
  
# Créez votre feature branch  
git switch --create feat/hadolint
```

Copy



# Exercice: Contribuez au projet de votre binôme (2/5)

Maintenant voici la liste des choses à faire:

- Ajoutez une étape de "lint" dans le workflow, avant de fabriquer l'image
  - La ligne de commande `hadolint` est pré-installée dans votre GitPod
  - GitHub action hadolint
- Corrigez /ignorez les erreurs du lint afin d'avoir un build qui passe



# Exercice: Contribuez au projet de votre binôme (3/5)

Pour tester votre changement :

```
hadolint ./Dockerfile
```

Copy



## Exercice: Contribuez au projet de votre binôme (4/5)

Une fois que vous êtes satisfait de votre changement il vous faut maintenant créer un commit et pousser votre nouvelle branche sur votre fork.



# Exercice: Contribuez au projet de votre binôme (5/5)

Dernière étape: ouvrir une pull request!

- Rendez vous sur la page de votre projet
- Sélectionnez votre branche dans le menu déroulant "branches" en haut a gauche.
- Cliquez ensuite sur le bouton ouvrir une pull request
- Remplissez le contenu de votre PR (titre, description, labels) et validez.



# La procédure de Pull Request

**Objectif** : Valider les changements d'un contributeur

- Technique : est-ce que ça marche ? est-ce maintenable ?
- Fonctionnel : est-ce que le code fait ce que l'on veux ?
- Humain : Propager la connaissance par la revue de code.
- Méthode : Tracer les changements.

# Revue de code ?

- Validation par un ou plusieurs pairs (technique et non technique) des changements
- Relecture depuis GitHub (ou depuis le poste du développeur)
- Chaque relecteur émet des commentaires // suggestions de changement
- Quand un relecteur est satisfait d'un changement, il l'approuve

- La revue de code est un **exercice difficile et potentiellement frustrant** pour les deux parties.
  - Comme sur Twitter, on est bien à l'abri derrière son écran ;=)
- En tant que contributeur, **soyez respectueux** de vos relecteurs : votre changement peut être refusé et c'est quelque chose de normal.
- En tant que relecteur, **soyez respectueux** du travail effectué, même si celui-ci comporte des erreurs ou ne correspond pas à vos attentes.



Astuce: Proposez des solutions plutôt que simplement pointer les problèmes.



# Exercice: Relisez votre PR reçue !

- Vous devriez avoir reçu une PR de votre binôme :-)
- Relisez le changement de la PR
- Effectuez quelques commentaires (bonus: utilisez la suggestion de changements), si c'est nécessaire
- Si elle vous convient, approuvez la!
- En revanche ne la "mergez" pas, car il manque quelque chose...

# Validation automatisée

**Objectif:** Valider que le changement n'introduit pas de régressions dans le projet

- A chaque fois qu'un nouveau commit est créé dans une PR, une succession de validations ("checks") sont déclenchés par GitHub
- Effectue des vérifications automatisées sur un commit de merge entre votre branche cible et la branche de PR

# Quelques exemples

- Analyse syntaxique du code (lint), pour détecter les erreurs potentielles ou les violations du guide de style
- Compilation du projet
- Exécution des tests automatisés du projet
- Déploiement du projet dans un environnement de test...

Ces "checks" peuvent être exécutés par votre moteur de CI ou des outils externes.



# Exercice: Déclencher un Workflow de CI sur une PR

- Votre PR n'a pas déclenché le workflow de CI de votre binôme 🤔
- Il faut modifier votre workflow pour qu'il se déclenche aussi sur une PR
- "Une chose à la fois" : faites une PR dédiée au déclenchement, puis mettez à jour la PR initiale
- La documentation se trouve par ici

# Checkpoint

**Règle d'or:** Si le CI est rouge, on ne merge pas la pull request !

Même si le linter "ilécon", même si on a la flemme et "sépanou" qui avons cassé le CI.

# Versions

# Pourquoi faire des versions ?

- Un changement visible d'un logiciel peut nécessiter une adaptation de ses utilisateurs
- Un humain ça s'adapte, mais un logiciel il faut l'adapter!
- Cela permet de contrôler le problème de la compatibilité entre deux logiciels.

# Une petite histoire

Le logiciel que vous développez utilise des données d'une API d'un site de vente.

```
// Corps de la réponse d'une requête GET https://supersite.com/api/item
[
  {
    "identifier": 1343,
    // ...
  }
]
```

 Copy

Voici comment est représenté un item vendu dans votre code.

```
public class Item {
  // Identifiant de l'item représenté sous forme d'entier.
  private int identifier;
  // ...
}
```

 Copy

Le site décide tout d'un coup de changer le format de l'identifiant de son objet en chaîne de caractères.

```
// Corps de la réponse d'une requête GET https://supersite.com/api/item
[
  {
    "identifier": "lolilol13843",
    // ...
  }
]
```

 Copy

Que se passe t'il du côté de votre application ?

com.fasterxml.jackson.databind.JsonMappingException

# Que s'est-il passé ?

- Votre application ne s'attendait pas à un identifiant sous forme de chaîne de caractères !
- Le fournisseur de l'API a "changé le contrat" de son API d'une façon non rétrocompatible avec votre l'existant.
  - Cela s'appelle un  **Breaking Change**

# Comment éviter cela ?

- Laisser aux utilisateurs une marge de manœuvre pour "accepter" votre changement.
  - Donner une garantie de maintien des contrats existants.
  - Informer vos utilisateurs d'un changement non rétrocompatible.
  - Anticiper les changements non rétrocompatibles à l'aide de stratégies (dépréciation).

# Bonjour versions !

- Une version cristallise un contrat respecté par votre application.
- C'est un jalon dans l'histoire de votre logiciel

# Quoi versionner ?

Le problème de la compatibilité existe dès qu'une dépendance entre deux bouts de code existe.

- Une API
- Une librairie
- Un langage de programmation
- Le noyau linux

# Version sémantique

La norme est l'utilisation du format vX.Y.Z (Majeur.Mineur.Patch)



(source betterprograming)

Un changement **ne changeant pas le périmètre fonctionnel** incrémente le numéro de version **patch**.

Un changement changeant le périmètre fonctionnel de façon **rétrocompatible** incrémente le numéro de version **mineure**.

Un changement changeant le périmètre fonctionnel de façon **non rétrocompatible** incrémente le numéro de version **majeure**.

## En résumé

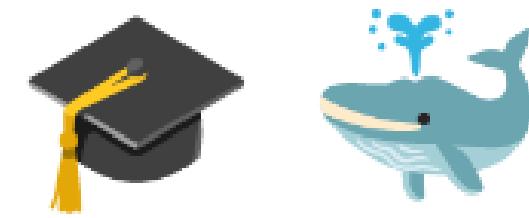
- Changer de version mineure ne devrait avoir aucun d'impact sur votre code.
- Changer de version majeure peut nécessiter des adaptations.

# Concrètement avec une API

- Offrir à l'utilisateur un moyen d'indiquer la version de l'API à laquelle il souhaite parler
  - Via un préfixe dans le chemin de la requête:
    - `https://monsupersite.com/api/v2.3/item`
  - Via un en-tête HTTP:
    - `Accept-version: v2.3`

# Version VS Git

- Un identifiant de commit est de granularité trop faible pour un l'utilisateur externe.
- Utilisation de **tags** git pour définir des versions.
- Un **tag** git est une référence sur un commit.



# Docker Hub

Si vous n'avez pas déjà un compte sur le DockerHub, créez-en un maintenant (nécessite une validation)



# "Taguez" et déployez la version 1.0.0

- Depuis GitPod, créez un tag git local 1.0.0

- 💡 git tag 1.0.0 -a -m "Première release 1.0.0, mode manuel"

- Fabriquez l'image Docker avec le tag (Docker) 1.0.0

- 💡 docker image build --tag=XXXX/sayhello:1.0.0 #... avec XXXX étant votre username DockerHub

- Déployez l'image sur le DockerHub

- 💡 docker login, docker image push

- Déployez le tag sur GitHub

- 💡 git push origin <tagname>

# ✓ "Taguez" et déployez la version 1.0.0

```
DOCKER_USERNAME=dduportal
TAG=1.0.0

git tag "${TAG}" -a -m "Première release ${TAG}, mode manuel"

docker image build --tag="${DOCKER_USERNAME}"/sayhello:"${TAG}" ./
docker login --username="${DOCKER_USERNAME}"
# ...

docker image push "${DOCKER_USERNAME}"/sayhello:"${TAG}"

git push origin "${TAG}"

# Vérifiez DockerHub et GitHub après ça
```

 Copy

# Checkpoint

- La notion de "version" est un outil de communication aux consommateurs de nos produits logiciels
- Le "semantic versioning" est une des façons les plus usitées pour gérer les politiques de version
- Nous avons déployé manuellement notre première image Docker, avec synchronisation code source  
↔ image Docker

⇒ 😞 C'était fort manuel. Et si on regardait à automatiser tout ça ?

# "Continuous Everything"

# Livraison Continue

## Continuous Delivery (CD)

# Pourquoi la Livraison Continue ?

- Diminuer les risque liés au déploiement
- Permettre de récolter des retours utilisateurs plus souvent
- Rendre l'avancement visible par **tous**

*How long would it take to your organization to deploy a change that involves just one single line of code?*

— Mary and Tom Poppendieck

# Qu'est ce que la Livraison Continue ?

- Suite logique de l'intégration continue:
  - Chaque changement est **potentiellement** déployable en production
  - Le déploiement peut donc être effectué à **tout** moment

*Your team prioritizes keeping the software **deployable** over working on new features*

— Martin Fowler

La livraison continue est l'exercice de **mettre à disposition automatiquement** le produit logiciel pour qu'il soit prêt à être déployé à tout moment.

# Livraison Continue avec GitHub Actions

# Pré-requis: exécution conditionnelle des jobs

- Il est possible d'exécuter conditionnellement un job ou un step à l'aide du mot clé `if` (documentation de `if`)

```
jobs:  
  release:  
    # Lance le job release uniquement si la branche est main.  
    if: contains('refs/heads/main', github.ref)  
  steps:  
    # ...
```

 Copy



# Secret GitHub / DockerHub Token

- Créez un personal access token dans le DockerHub avec les droits "Read,Write"
  - ne partagez pas ce token, ne le mettez PAS dans GitPod, ne l'enregistrez pas sur votre disque dur
  - Documentation "Manage access tokens"
- Insérez le token DockerHub comme secret dans votre dépôt GitHub
  - Creating encrypted secrets for a repository



# Livraison Continue sur le DockerHub

- **But :** Automatiser le déploiement de l'image dans le DockerHub lorsqu'un tag est poussé
- Changez votre workflow de CI de façon à ce que, sur un push de tag, les tâches suivantes soient effectuées :
  - Comme avant: Lint, Build, Test de l'image (en affichant le README)
  - SI c'est un tag, alors il faut pousser (et éventuellement reconstruire avec le bon nom) l'image sur le DockerHub
- 💡 Utilisez les GitHub Action suivantes :
  - docker-login
  - build-and-push-docker-images

# ✓ Livraison Continue sur le DockerHub

```
# ...
steps:
  # ... Lint,
  # ... Build
  # ... Test
- name: Login to Docker Hub
  uses: docker/login-action@v2
  if: startsWith(github.ref, 'refs/tags/')
  with:
    username: xxxxx
    password: ${{ secrets.DOCKERHUB_TOKEN }}
- name: Push if on `main`
  uses: docker/build-push-action@v4
  if: startsWith(github.ref, 'refs/tags/')
  with:
    push: true
    context: ./
    tags: xxxxx/sayhello:${{ github.ref_name }}
```

Copy



# Déploiement Continu

 Continuous Deployment / "CD"

# Qu'est ce que le Déploiement Continu ?

- Version "avancée" de la livraison continue:
  - Chaque changement **est** déployé en production, de manière **automatique**

# Continuous Delivery versus Deployment



continuous depl vs delivery

Source : <http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment>

# Bénéfices du Déploiement Continu

- Rend triviales les procédures de mise en production et de rollback
  - Encourage à mettre en production le plus souvent possible
  - Encourage à faire des mises en production incrémentales
- Limite les risques d'erreur lors de la mise en production
- Fonctionne de 1 à 1000 serveurs et plus encore...

# Qu'est ce que "La production" ?

- Un (ou plusieurs) ordinateur ou votre / vos applications sont exécutées
- Ce sont là où vos utilisateurs "utilisent" votre code
  - Que ce soit un serveur web pour une application web
  - Ou un téléphone pour une application mobile
- Certaines plateformes sont plus ou moins outillées pour la mise en production automatique



# Déploiement Continu sur le DockerHub

- **But :** Déployer votre image `sayhello` continuellement sur le DockerHub
- Changez votre workflow de CI de façon à ce que, sur un push sur la branch `main`, les tâches suivantes soient effectuées :
  - Comme avant: Lint, Build, Test de l'image (en affichant le README)
  - SI c'est la branche `main`, alors il faut pousser l'image avec le tag `latest` sur le DockerHub
  - Conservez le cas avec les tags

# ✓ Déploiement Continu sur le DockerHub

```
# ...
steps:
  # ... Lint,
  # ... Build
  # ... Test
  - name: Login to Docker Hub
    uses: docker/login-action@v2
    if: contains('refs/heads/main', github.ref)
    with:
      username: xxxxx
      password: ${{ secrets.DOCKERHUB_TOKEN }}
  - name: Push if on `main`
    uses: docker/build-push-action@v4
    if: contains('refs/heads/main', github.ref)
    with:
      push: true
      context: ./
      tags: xxxxx/sayhello:latest
# ... Deploy if tagTag
```

Copy



# Checkpoint

- La livraison continue et le déploiement continu étendent les concepts du CI
- Les 2 sont automatisées, mais un être humain est nécessaire comme déclencheur pour la 1ère
- Le choix dépend des risques et de la "production"
- On a vu comment automatiser le déploiement dans GitHub Actions
  - Conditions dans le workflow
  - Gestion de secrets

# Bibliographie

# Ligne de commande

- <https://tldp.org>
- <https://en.wikipedia.org/wiki/POSIX>
- [https://en.wikipedia.org/wiki/Read%20%93eval%20%93print\\_loop](https://en.wikipedia.org/wiki/Read%20%93eval%20%93print_loop)
- <https://linuxhandbook.com/linux-directory-structure/>

# Git / VCS

- <https://docs.github.com>
- <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>
- <http://martinfowler.com/bliki/VersionControlTools.html>
- <http://martinfowler.com/bliki/FeatureBranch.html>
- <https://about.gitlab.com/2014/09/29/gitlab-flow/>
- <https://www.atlassian.com/git/tutorials/comparing-workflows>
- <http://nvie.com/posts/a-successful-git-branching-model/>

# Intégration Continue

- <http://martinfowler.com/articles/continuousIntegration.html>
- <http://martinfowler.com/bliki/ContinuousDelivery.html>
- <https://jaxenter.com/implementing-continuous-delivery-117916.html>
- <https://technologyconversations.com/2014/04/29/continuous-delivery-introduction-to-concepts-and-tools/>
- <http://blog.arungupta.me/continuous-integration-delivery-deployment-maturity-model>
- <http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment>

# Docker

- <https://dduportal.github.io/cours/cnam-docker-2018>
- <https://kodekloud.com/blog/docker-for-beginners/>
- <https://www.slideshare.net/dotCloud/why-docker>
- <https://docs.docker.com/engine/reference/builder/>
- <https://www.r-bloggers.com/2021/05/best-practices-for-r-with-docker/>
- <https://github.com/wagoodman/dive>
- <https://docs.docker.com/engine/tutorials/networkingcontainers/>
- <https://towardsdatascience.com/docker-networking-919461b7f498>

# Livraison/Déploiement Continu

- <http://martinfowler.com/articles/continuousIntegration.html>
- <http://martinfowler.com/bliki/ContinuousDelivery.html>
- <https://jaxenter.com/implementing-continuous-delivery-117916.html>
- <https://technologyconversations.com/2014/04/29/continuous-delivery-introduction-to-concepts-and-tools/>
- <http://blog.arungupta.me/continuous-integration-delivery-deployment-maturity-model>
- <http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment>

# Autre

- <https://cicd-lectures.github.io/slides/>
- <https://dduportal.github.io/cours/>

# Merci !

✉️ [damien.duportal @ gmail.com](mailto:damien.duportal@gmail.com)

Slides: <https://dduportal.github.io/esgi-courses/fix/use-fontawesome-4-CDN>

 QRCode to this presentation

Source on  : <https://github.com/dduportal/esgi-courses>