

Jour 1 — Culture DevOps & Git collaboratif

DevOps pour le Développeur Web et Web Mobile

DWWM — ForEach Academy

Formateur : Fabrice Claeys

Programme de la journée

Horaire	Contenu
9h00-9h15	Accueil, tour de table, présentation de la semaine
9h15-10h00	1.1 Pourquoi le DevOps ? — histoire, culture, boucle, mesures
10h00-10h45	1.2 CI, CD... et CD — intégration, livraison, déploiement continu
10h45-11h00	<i>Pause</i>
11h00-11h30	1.3 Panorama des outils — une case par étape
11h30-12h30	TP 0 — Cartographie d'un pipeline
13h30-14h15	2.1 Les bases de Git — modèle mental, premiers commits (exercice 1.1)
14h15-15h30	2.2 Branches et merge · 2.3 Workflow d'équipe (+ pause)
15h30-16h00	2.4 Conflits
16h00-17h00	TP 1 — Projet collaboratif (suite demain) · récap et quiz

Objectifs de la journée

À la fin de la journée, vous saurez :

- **Expliquer** ce qu'est le DevOps et pourquoi les entreprises l'adoptent
- **Distinguer** intégration continue, livraison continue et déploiement continu
- **Cartographier** le pipeline d'une application web
- **Contribuer** à un dépôt partagé : branche → *Pull Request* → revue → merge
- **Résoudre** un conflit Git sans paniquer

“ Lien avec le titre pro DWWM : compétence « Préparer et exécuter le déploiement d'une application web ». Votre pipeline de la semaine pourra servir de preuve dans votre **dossier professionnel**. ”

Où en est-on ? La boucle DevOps



Aujourd'hui : comprendre la boucle entière, puis maîtriser les deux premières cases — **planifier** et **coder à plusieurs** avec Git.

Le fil rouge de la semaine : TaskFlow

Un gestionnaire de tâches **volontairement simple** : l'effort porte sur le DevOps, pas sur le code.

```
taskflow/  
├─ front/  # Vite + JavaScript (sans framework), servi par nginx  
├─ api/    # Node 20 + Express, port 3000, GET /health  
└─ (BDD)   # PostgreSQL 16 : une table "tasks"
```

Jour	Ce que vous ajoutez à TaskFlow
J1	Le pipeline cible (sur papier) + les règles de travail en équipe
J2	Un repo protégé, des images Docker pour l'API et le front
J3	Docker Compose + un pipeline d'intégration continue
J4	Le déploiement sur un serveur, en HTTPS, automatique et supervisé

1.1 Pourquoi le DevOps ?

Une histoire (trop) banale

“ Vendredi, 18h. Léa, développeuse, a fini sa fonctionnalité. « Ça marche sur ma machine ! » Elle envoie un zip à Karim, administrateur système.
Karim copie les fichiers sur le serveur par FTP. Le site affiche une page blanche.
Il manquait une version de PHP, un fichier de configuration et une étape de migration de la base de données.
Samedi, 2h du matin : retour à l'ancienne version. Lundi : réunion « qui a cassé la prod ? ». ”

- **FTP** (*File Transfer Protocol*) : envoi manuel de fichiers vers un serveur
- **Prod** (production) : l'environnement où sont les vrais utilisateurs

Questions : où sont les problèmes ? Est-ce la faute de Léa ? de Karim ?

Le « mur de la confusion »



Deux équipes, **deux objectifs opposés** : l'une veut changer, l'autre veut stabiliser. Résultat : livraisons rares, énormes, risquées... et des reproches.

Une analogie : la cuisine et la salle

Restaurant « mur de la confusion »

Les cuisiniers envoient les plats sans savoir s'ils arrivent chauds

Les serveurs reçoivent les reproches sans pouvoir agir

On change la carte une fois par an, tout d'un coup

Quand un plat rate, on cherche un coupable

Restaurant « DevOps »

Les cuisiniers voient la salle et les retours clients

Serveurs et cuisiniers partagent le même objectif : un client satisfait

On teste un nouveau plat par jour, en petite quantité

Quand un plat rate, on corrige la recette ensemble

“Le DevOps, c'est faire travailler **Dev** et **Ops** comme une seule équipe, avec un objectif commun : livrer **souvent et sans casse**.”

Petite histoire du DevOps

Année	Événement
2001	Manifeste Agile : livrer souvent, en petits incréments... mais seulement côté développement
2008	Patrick Debois (consultant belge) constate le mur Dev / Ops sur un projet de migration
2009	Conférence Flickr « <i>10+ deploys per day</i> » ; Debois organise les premiers DevOpsDays à Gand
2013	Roman <i>The Phoenix Project</i> (Gene Kim) : le DevOps raconté comme un thriller d'entreprise
2014+	Rapports annuels <i>State of DevOps</i> (équipe DORA), livre <i>Accelerate</i> (2018)

“ **DevOps** = contraction de *Development et Operations*. Le mot vient du nom de la conférence de 2009. ”

Ce qu'est (et n'est pas) le DevOps

C'est une **culture** + des **pratiques** + des **outils** pour livrer des logiciels rapidement et de façon fiable.

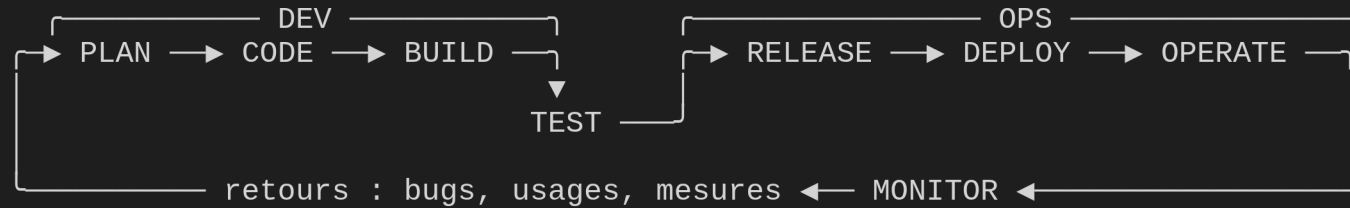
Le DevOps n'est pas...	... mais plutôt
Un outil (« on a acheté Jenkins, on est DevOps »)	Une façon de travailler, que les outils facilitent
Un poste isolé (« le DevOps, c'est Kevin »)	Une responsabilité partagée par toute l'équipe
Supprimer les Ops	Faire collaborer Dev et Ops dès le début
Aller vite au prix de la qualité	Aller vite grâce à la qualité (tests, automatisation)

CALMS : les 5 piliers

Lettre	Pilier	Concrètement
C	Culture	On collabore, on partage les objectifs, on ne cherche pas de coupable
A	Automatisation	Ce qui est répétitif (tests, build, déploiement) est fait par une machine
L	Lean (« sans gaspillage »)	Petits lots, on élimine les attentes et les tâches inutiles
M	Mesure	On mesure (temps de livraison, pannes) pour s'améliorer, pas pour punir
S	Partage (<i>Sharing</i>)	On documente, on partage les connaissances et les retours d'incident

“*Culture avant outils : une équipe qui se renvoie la faute restera lente, même avec les meilleurs outils du monde.*”

La boucle infinie du cycle de vie



- Le symbole ∞ (infini) : la boucle **ne s'arrête jamais**
- Chaque tour de boucle = **une petite amélioration** livrée aux utilisateurs
- Ce que l'on observe en production (*monitor*) nourrit la planification suivante (*plan*)

Les 8 étapes appliquées à TaskFlow

Étape	Question	Exemple TaskFlow
Plan	Que faut-il faire ?	Ticket « pouvoir supprimer une tâche »
Code	Comment l'écrire à plusieurs ?	Branche Git, <i>Pull Request</i> , revue
Build	Comment fabriquer l'application ?	<code>npm run build</code> , image Docker
Test	Est-ce que ça marche toujours ?	<code>npm test</code> lancé automatiquement
Release	Quelle version livre-t-on ?	Tag <code>v1.2.0</code> , image publiée
Deploy	Comment la mettre en ligne ?	Sur le serveur, sans geste manuel
Operate	Comment la faire tourner ?	Redémarrage auto, sauvegardes
Monitor	Comment savoir si tout va bien ?	<code>GET /health</code> , logs, alertes

Mesurer : les 4 indicateurs DORA

DORA (*DevOps Research and Assessment*) : équipe de recherche qui étudie des milliers d'entreprises chaque année.

Indicateur	Question simple	Mesure...
Fréquence de déploiement	À quelle fréquence livre-t-on en prod ?	la vitesse
Délai de mise en production	Combien de temps entre un commit et la prod ?	la vitesse
Taux d'échec des changements	Quel % de livraisons provoque une panne ?	la stabilité
Temps de restauration	Combien de temps pour réparer une panne ?	la stabilité

“Deux indicateurs de **vitesse**, deux de **stabilité** : les meilleures équipes sont bonnes **sur les quatre à la fois**.”

Les chiffres : élite vs faible

Indicateur	Équipes « élite »	Équipes « faibles »
Fréquence de déploiement	Plusieurs fois par jour	Une fois par mois... voire tous les 6 mois
Délai commit → prod	Moins d' un jour	Un à six mois
Taux d'échec	Environ 5 %	Souvent plus de 40 %
Temps de restauration	Moins d' une heure	Une semaine ou plus

Ordres de grandeur issus des rapports State of DevOps (DORA, 2019-2023).

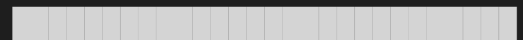
“

*Surprise : les équipes qui livrent **le plus souvent** sont aussi celles qui **cassent le moins**.*

”

Pourquoi « petit et souvent » est moins risqué

Livraison rare (tous les 3 mois)

 ← 300
changements d'un coup

Panne ? Lequel des 300 est fautif ?
→ des jours d'enquête

Livraisons fréquentes (chaque jour)

 ← 5 changements
à chaque fois

Panne ? C'est l'un des 5 derniers.
→ trouvé et corrigé en minutes

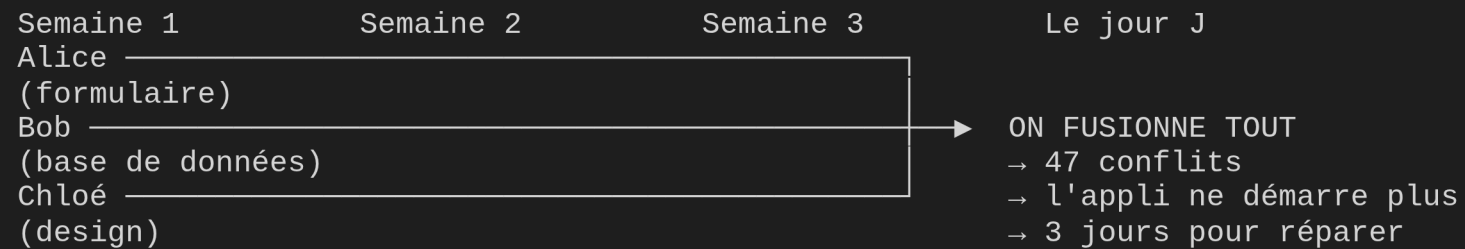
Analogie : déménager tout un appartement en un seul voyage, ou faire plusieurs petits trajets. Si un carton tombe, vous savez exactement lequel.

À retenir — 1.1

- Le DevOps est né du **mur** entre Dev (changer vite) et Ops (rester stable)
- C'est d'abord une **culture** (CALMS), les outils viennent ensuite
- Le cycle de vie d'une application est une **boucle infinie** en 8 étapes
- On mesure avec les 4 indicateurs **DORA** : 2 de vitesse, 2 de stabilité
- **Petit et souvent** = moins de risques, réparations plus rapides

1.2 CI, CD... et CD

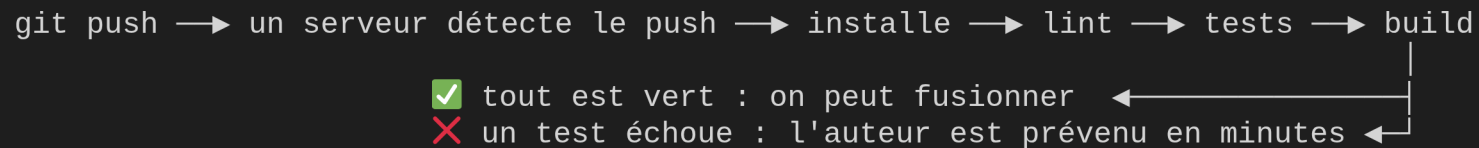
Le problème : l'intégration « big bang »



Chacun travaille **seul trop longtemps**. Plus on attend pour assembler (« intégrer ») le travail, plus l'assemblage est douloureux.

Intégration continue (CI)

CI = *Continuous Integration*. On **intègre** son travail dans la branche commune **souvent** (au moins une fois par jour), et **chaque** envoi est **vérifié automatiquement**.



Les 3 règles :

1. **Petits commits**, intégrés souvent
2. **Tests automatisés** : aucune vérification ne repose sur la mémoire d'un humain
3. **Build rouge = priorité** de toute l'équipe : on répare avant de continuer

La CI sur TaskFlow : ce que la machine exécute

Ce que vous tapiez à la main, la CI le fait à **chaque push** :

```
$ npm ci
added 312 packages in 6s
$ npm run lint
> eslint src tests
$ npm test
✓ tests/app.test.js (6 tests) 45ms
Test Files  1 passed (1)
Tests      6 passed (6)
```

- **Lint** : vérification automatique du style et des erreurs courantes (ESLint)
- **Tests** : vérifient que le code fait ce qu'il doit (Vitest)
- Si une seule commande échoue → le pipeline est **rouge** → la fusion est bloquée

Livraison continue (*Continuous Delivery*)

On va plus loin : à chaque changement validé, l'application est **prête à être déployée** (image construite, testée, rangée).

```
push → CI (tests) → build de l'image → image prête → [ BOUTON ] → prod
                                     |
                                     un humain décide QUAND
```

- Le déploiement est **une action simple** (un clic), pas une procédure de 3 pages
- L'humain choisit le moment : après une démo client, hors période de soldes...
- Abréviation : **CD**... comme la suivante, d'où la confusion !

Déploiement continu (*Continuous Deployment*)

Dernière marche : **aucun humain** entre le merge et la production.

push → CI (tests) → build de l'image → déploiement automatique → prod



si tous les tests passent, ça part en prod

- Exige une **grande confiance** dans les tests et la supervision
- Chaque merge sur `main` est visible par les utilisateurs quelques minutes plus tard
- C'est ce que vous ferez avec TaskFlow le **jour 4** : un merge → la prod change seule

Les trois côte à côte

	Intégration continue	Livraison continue	Déploiement continu
Sigle	CI	CD (<i>Delivery</i>)	CD (<i>Deployment</i>)
Ce qui est automatisé	Lint, tests, build	+ préparation de la version	+ mise en production
Humain avant la prod ?	—	Oui (un clic)	Non
Prérequis	Tests automatisés	CI + build reproductible	Livraison continue + confiance + supervision

Analogie de la boulangerie :

- **CI** = on goûte chaque fournée dès sa sortie du four
- **Livraison continue** = le pain est toujours prêt en vitrine, le vendeur le tend au client
- **Déploiement continu** = le pain part tout seul chez le client par drone 🚁

Les environnements : dev, test, prod

Environnement	À quoi il sert	Qui l'utilise	TaskFlow cette semaine
dev (local)	Écrire et essayer le code	Le développeur	Votre poste, <code>docker compose up</code>
test / <i>staging</i>	Vérifier dans des conditions proches de la prod	L'équipe, la CI, le client	Le pipeline GitHub Actions
prod	Servir les vrais utilisateurs	Tout le monde	Une stack Docker séparée sur votre poste (<code>taskflow-prod</code>), en HTTPS

- **VPS** (*Virtual Private Server*) : un serveur virtuel loué chez un hébergeur — ce qu'on utiliserait en vrai, vu en théorie au jour 4
- Règle d'or : les environnements doivent être **le plus semblables possible** — c'est ce que Docker va permettre

« You build it, you run it »

« Vous l'avez construit, vous le faites tourner. » — Werner Vogels, Amazon (2006)



Avant	Avec DevOps
Le dev livre et passe à autre chose	Le dev suit son code jusqu'en prod et regarde les logs
L'Ops découvre l'application le jour du déploiement	L'Ops participe dès la conception (serveur, sauvegardes, supervision)
Une panne = « c'est la faute à l'autre équipe »	Une panne = un problème de l'équipe , analysé sans blâme
Documentation absente ou périmée	Configuration écrite dans des fichiers versionnés (Dockerfile, workflows)

« En tant que développeur web, vous êtes **directement concerné** : le DevOps fait partie de votre métier. »

À retenir — 1.2

- **CI** : intégrer souvent + vérifier automatiquement à chaque push
- **Livraison continue** : toujours prêt à déployer, un humain appuie sur le bouton
- **Déploiement continu** : en prod sans humain, si tous les voyants sont verts
- Trois environnements : **dev** → **test** → **prod**, aussi semblables que possible
- Responsabilité **partagée** : « *you build it, you run it* »

1.3 Panorama des outils

Une case par étape de la boucle

Étape	Outils du marché (exemples)
Plan	GitHub Issues / Projects, Jira, Trello
Code	Git , GitHub, GitLab, Bitbucket
Build	npm, composer, Vite, Docker
Test	Jest, Vitest , PHPUnit, Playwright, ESLint
Release	Tags Git, GitHub Releases , registres d'images (Docker Hub, ghcr.io)
Pipeline CI/CD	GitHub Actions , GitLab CI, Jenkins
Deploy / Operate	VPS + Nginx , PaaS (Render, Railway...), cloud (AWS, Azure, GCP)
Monitor	Uptime Kuma , Grafana, Prometheus, Sentry

PaaS (*Platform as a Service*) : un hébergeur qui fait tourner votre application sans que vous gériez le serveur.

Ce que nous utiliserons cette semaine, et pourquoi

Besoin	Outil	Pourquoi ce choix
Versionner, collaborer	Git + GitHub	Outil du marché, <i>Pull Requests</i> , branches protégées
Environnement reproductible	Docker + Docker Compose	Règle le « ça marche sur ma machine »
CI/CD	GitHub Actions	Gratuit, intégré au dépôt
Registre d'images	ghcr.io (GitHub Container Registry)	Gratuit, aucun compte supplémentaire
Production	Stack Compose locale + Nginx HTTPS + runner GitHub Actions sur votre poste	Les vrais gestes de mise en prod sans payer de serveur
Supervision	<code>docker logs</code> , healthchecks, Uptime Kuma	Simple, suffisant pour une petite équipe

- **SSH** (*Secure Shell*) : connexion sécurisée à un serveur distant (vue en théorie au jour 4)

Le pipeline cible de TaskFlow (vendredi soir... sans stress)



“ Aujourd'hui : comprendre ce schéma. Jour 4 au soir : chaque flèche fonctionnera **pour de vrai**. ”

TP 0 — Cartographie d'un pipeline

`tp/tp0-cartographie-pipeline/`

TP 0 — Déroulé (1h)

Étape	Contenu	Durée
1	Lire l'étude de cas « La Fromagerie » : une petite équipe qui déploie en FTP le vendredi soir	10 min
2	Lister les problèmes, les classer par étape de la boucle DevOps	10 min
3	Lire un workflow GitHub Actions réel commenté (lire , pas écrire) et répondre aux questions	15 min
4	Dessiner le pipeline cible de TaskFlow (papier, Excalidraw ou mermaid)	15 min
5	Restitution en groupe	10 min

Énoncé complet : `tp/tp0-cartographie-pipeline/README.md`

TP 0 — Lire un workflow (sans l'écrire)

```
name: CI                                # nom affiché dans l'onglet Actions
on: [push, pull_request]                # QUAND : à chaque push et chaque PR
jobs:
  test:                                 # un "job" = une suite d'étapes
    runs-on: ubuntu-latest              # OÙ : une machine Linux fournie par GitHub
    steps:
      - uses: actions/checkout@v4        # récupérer le code
      - uses: actions/setup-node@v4      # installer Node
        with: { node-version: 20 }
      - run: npm ci                      # installer les dépendances
      - run: npm test                    # lancer les tests
```

Questions : quel est le **déclencheur** ? Que se passe-t-il si `npm test` échoue ? Quelle étape de la boucle manque ?

- **YAML** : format de fichier texte basé sur l'indentation (comme Python)

TP 0 — Livrable et critères

Votre schéma du pipeline cible de TaskFlow doit faire apparaître :

Élément	Exemple
Les étapes	lint, tests, build, publication, déploiement, supervision
Les outils	GitHub, GitHub Actions, Docker, Nginx...
Les déclencheurs	« à chaque PR », « à chaque merge sur main »
Qui fait quoi	humain (revue, merge) / machine (tout le reste)
Les contrôles	où le pipeline peut s'arrêter en rouge



Gardez ce schéma : il servira de base à `docs/ARCHITECTURE.md` et à votre présentation finale.



Après-midi — Git collaboratif

2.1 Les bases de Git

Avant de commencer : votre poste est prêt ?

Vérification	Commande	Résultat attendu
Git installé	<code>git --version</code>	<code>git version 2.x</code>
Identité configurée	<code>git config user.name && git config user.email</code>	vos nom et e-mail
Clé SSH reconnue par GitHub	<code>ssh -T git@github.com</code>	Hi <pseudo>! You've successfully authenticated...

- Une ligne en échec ? On la règle **maintenant**, avec `ressources/setup-poste.md` : sans clé SSH, impossible de faire un `git push` au TP 1
- **SSH** ici sert seulement à prouver à GitHub qui vous êtes, sans taper de mot de passe

Pourquoi Git ?

```
rapport.docx  
rapport_v2.docx  
rapport_v2_corrigé.docx  
rapport_final.docx  
rapport_final_VRAI.docx  
rapport_final_VRAI_relu_par_Karim.docx ← lequel est le bon ?
```

Git est un **système de gestion de versions** : il garde l'historique de **chaque** modification, avec **qui, quand et pourquoi**.

Analogie : un jeu vidéo avec des **points de sauvegarde**. Chaque commit est une sauvegarde à laquelle vous pouvez revenir.

Le modèle mental : 4 zones



- **Index** = le panier : vous choisissez ce qui part dans la prochaine sauvegarde
- **Commit** = la sauvegarde, identifiée par un **SHA** (empreinte unique, ex. `a1b2c3d`)

`git status` : où en suis-je ?

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    modified:   style.css

Untracked files:
  notes.txt
```

Section	Zone
<i>Changes to be committed</i>	Dans l' index (partira au prochain commit)
<i>Changes not staged</i>	Modifié, mais pas dans l'index
<i>Untracked files</i>	Fichier inconnu de Git

Ajouter, sauvegarder, envoyer

```
$ git add style.css
$ git commit -m "style: agrandit le titre de la page d'accueil"
[main 4c5d6e7] style: agrandit le titre de la page d'accueil
 2 files changed, 5 insertions(+), 2 deletions(-)
$ git push
To github.com:promo-dwwm/site-promo.git
 a1b2c3d..4c5d6e7  main -> main
```

- `git add .` ajoute **tout** : pratique, mais vérifiez avec `git status` avant
- `git push` envoie vos commits sur `origin` (le dépôt distant, en général GitHub)
- `a1b2c3d..4c5d6e7` : la branche distante est passée de l'ancien commit au nouveau

Lire l'historique : `git log --oneline --graph`

```
$ git log --oneline --graph --all
* 4c5d6e7 (HEAD -> main, origin/main) style: agrandit le titre
* a1b2c3d feat: ajoute la page d'accueil
* 9f8e7d1 chore: initialise le projet
```

Élément	Signification
4c5d6e7	Début du SHA du commit
HEAD -> main	HEAD = « vous êtes ici » ; vous êtes sur la branche main
origin/main	Où en est main sur GitHub (lors du dernier échange)
* et	Le graphe : utile dès qu'il y a plusieurs branches

“ Astuce : créez un alias → `git config --global alias.lg "log --oneline --graph --all"` puis `git lg` ”

Bien écrire ses messages : Conventional Commits

Format : `type: description à l'impératif, en minuscules`

Type	Quand l'utiliser	Exemple
<code>feat</code>	Nouvelle fonctionnalité	<code>feat: permet de supprimer une tâche</code>
<code>fix</code>	Correction de bug	<code>fix: corrige le compteur de tâches</code>
<code>docs</code>	Documentation	<code>docs: ajoute la procédure d'installation</code>
<code>style</code>	Mise en forme (sans changer la logique)	<code>style: indente le fichier main.js</code>
<code>refactor</code>	Réorganisation du code	<code>refactor: extrait la fonction formatDate</code>
<code>test</code>	Tests	<code>test: couvre la route DELETE /api/tasks</code>
<code>chore</code> / <code>ci</code>	Outillage / pipeline	<code>ci: lance les tests sur chaque PR</code>

“ Pourquoi ? Un historique lisible, et des outils capables de générer le *CHANGELOG* automatiquement (demain). ”

Bons et mauvais messages

✗ À éviter	✓ Préférer
modif	fix: affiche un message si le titre est vide
wip	feat: ajoute le formulaire de création (sans validation)
correction bug Karim	fix: empêche la double soumission du formulaire
aaaa	chore: met à jour Vite en 5.4
plein de trucs	Plusieurs commits, un sujet par commit

“ Règle : votre message doit permettre à un collègue de comprendre le commit **dans six mois**, sans ouvrir le code. ”

`.gitignore` : ce qui ne doit jamais être versionné

```
# Dépendances : se réinstallent avec npm ci
node_modules/
# Fichiers générés par le build
dist/
# Secrets et configuration locale
.env
# Fichiers de l'éditeur et du système
.vscode/
.DS_Store
```

Vérifier qu'un fichier est bien ignoré :

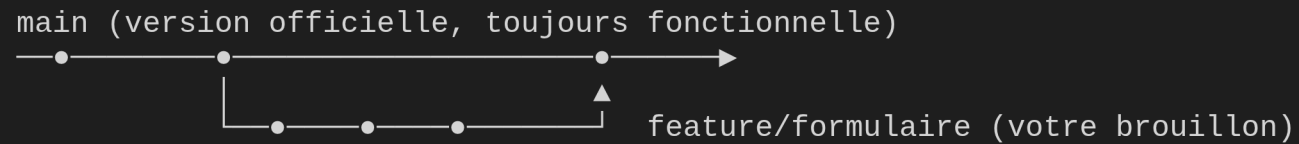
```
$ git check-ignore -v .env
.gitignore:6:.env      .env
```

“ Un `.env` poussé sur GitHub = un mot de passe **public**. Supprimer le fichier ensuite ne suffit pas : il reste dans l'historique. Il faut **changer le mot de passe**. ”

2.2 Branches et merge

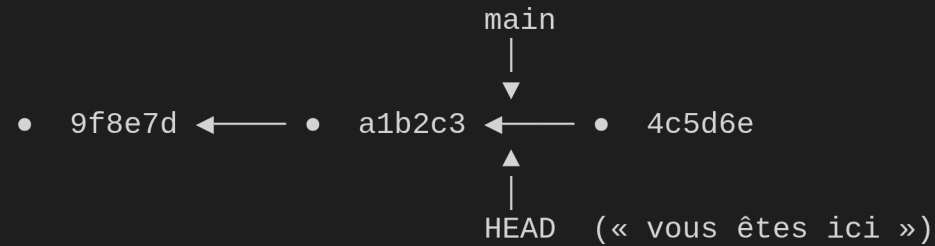
Pourquoi des branches ?

Analogie : une branche est un **brouillon parallèle**. Vous essayez une idée sur une copie, sans abîmer la version officielle. Si l'idée est bonne, vous la recopiez au propre (*merge*). Sinon, vous jetez le brouillon.



- Plusieurs personnes travaillent **en même temps** sans se marcher dessus
- `main` reste **déployable** à tout moment
- Chaque branche porte **un seul sujet** → revue plus simple

Une branche = un simple pointeur



- Une branche n'est **pas** une copie des fichiers : c'est une **étiquette** posée sur un commit
- À chaque nouveau commit, l'étiquette de la branche courante **avance** toute seule
- Créer une branche coûte presque rien : un fichier de 41 octets dans `.git/refs/heads/`

Créer une branche et y travailler

```
$ git switch -c feature/formulaire
Switched to a new branch 'feature/formulaire'
$ git branch
* feature/formulaire
  main
$ git commit -am "feat: ajoute le formulaire de création"
[feature/formulaire 7a8b9c0] feat: ajoute le formulaire de création
1 file changed, 12 insertions(+)
```

- `switch -c` = créer **et** se placer sur la branche (`-c` pour *create*)
- `git switch main` pour revenir sur `main` ; l'étoile `*` indique la branche courante
- Convention de nommage : `feature/...`, `fix/...`, `docs/...` + un nom court en minuscules

Merge « fast-forward » (avance rapide)

Cas : `main` n'a pas bougé depuis la création de la branche.



```
$ git switch main
$ git merge feature/formulaire
Updating 4c5d6e7..8d9e0f1
Fast-forward
 index.html | 12 ++++++++
```

Git se contente d'**avancer l'étiquette** `main` : aucun nouveau commit.

Merge avec commit de fusion

Cas : `main` a avancé pendant que vous travailliez (un collègue a fusionné).



```
$ git merge feature/formulaire
Merge made by the 'ort' strategy.
index.html | 12 ++++++++
```

Git crée un **commit de fusion** `M` qui a **deux parents**. L'historique garde la trace de la branche.

Fast-forward ou commit de fusion ?

	Fast-forward	Commit de fusion
Quand ?	<code>main</code> n'a pas bougé	<code>main</code> a avancé
Nouveau commit ?	Non	Oui (<code>Merge branch...</code>)
Historique	Linéaire	Montre la branche
Forcer un commit de fusion	<code>git merge --no-ff feature/x</code>	—

Ensuite, faites le ménage :

```
$ git branch -d feature/formulaire  
Deleted branch feature/formulaire (was 8d9e0f1).
```



*Sur GitHub, le bouton **Merge pull request** crée par défaut un commit de fusion.*



Partager une branche sur GitHub

```
$ git push -u origin feature/formulaire
Enumerating objects: 5, done.
Writing objects: 100% (3/3), 412 bytes | 412.00 KiB/s, done.
remote: Create a pull request for 'feature/formulaire' on GitHub by visiting:
remote:      https://github.com/promo-dwwm/site-promo/pull/new/feature/formulaire
* [new branch]      feature/formulaire -> feature/formulaire
branch 'feature/formulaire' set up to track 'origin/feature/formulaire'.
```

- `-u` (*upstream*) relie votre branche locale à la branche distante, **une seule fois**
- Ensuite, un simple `git push` suffit
- GitHub vous propose directement le lien pour ouvrir une *Pull Request*



Démo — Branches et merge

`demos/demo-git-branches/`

1. Créer deux branches depuis `main`, un commit sur chacune
2. Observer le graphe avec `git log --oneline --graph --all`
3. Merge **fast-forward** de la première
4. Merge **avec commit de fusion** de la seconde
5. Observer le commit `M` et ses deux parents : `git show --format=raw HEAD`



Suivez sur votre poste : les commandes sont dans le `README.md` de la démo.



2.3 Workflow d'équipe

GitHub Flow : simple et efficace

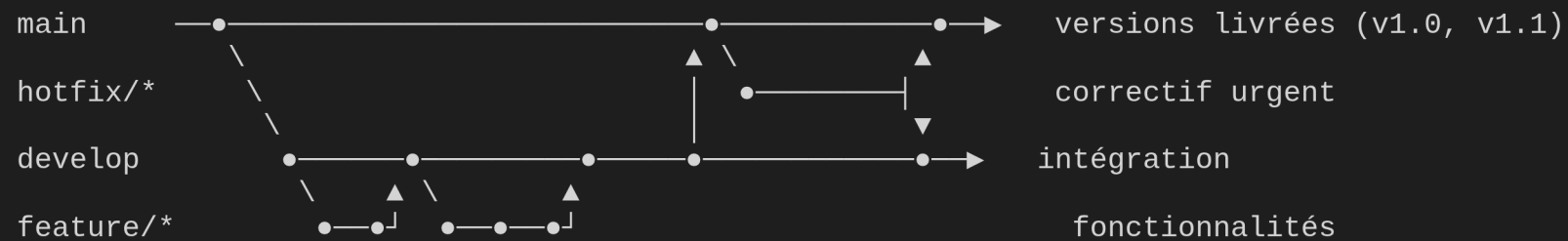
Pourquoi des règles communes ? Comme le code de la route : sans règles partagées, même de bons conducteurs se percutent.



1. On crée une branche **depuis** `main`
2. On commite, on pousse, on ouvre une **Pull Request**
3. Revue + tests automatiques verts → **merge** dans `main`
4. `main` est déployée (souvent automatiquement)

Git Flow (simplifié)

Pour les projets avec des **versions planifiées** (application mobile, logiciel livré au client).



- `main` : uniquement les versions livrées · `develop` : le travail en cours
- `feature/*` partent de `develop` · `hotfix/*` partent de `main` et reviennent dans les deux

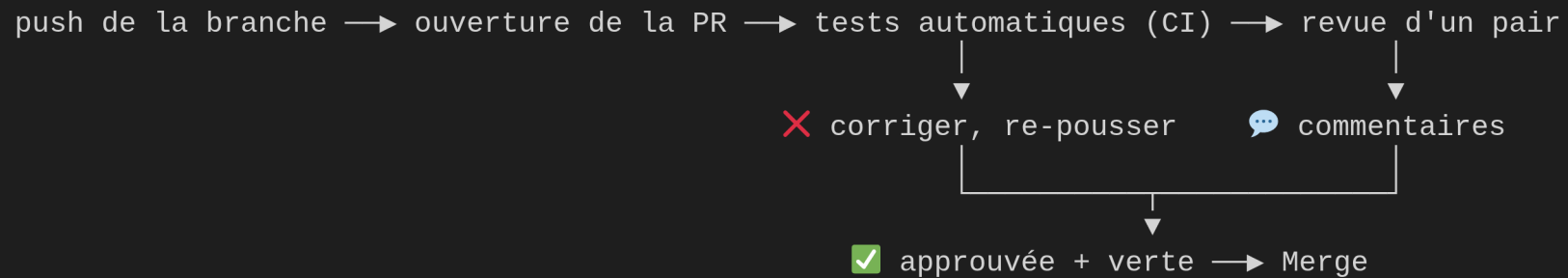
Lequel choisir ?

	GitHub Flow	Git Flow simplifié
Branches longues	<code>main</code> seule	<code>main</code> + <code>develop</code>
Idéal pour	Applications web déployées souvent	Versions planifiées, plusieurs versions maintenues
Complexité	Faible	Moyenne
Compatible déploiement continu	✅ Oui	Plus difficile

“ **Cette semaine : GitHub Flow.** `main` toujours déployable, une branche par sujet, tout passe par une Pull Request. C'est le choix le plus courant pour une application web. ”

La Pull Request (PR)

Une **PR** (*Pull Request*, « demande de fusion ») = « voici mes changements, pouvez-vous les relire avant de les fusionner dans `main` ? »



Une bonne PR : **un titre clair**, une description (**quoi** et **pourquoi**), une capture d'écran si visuel, **petite** (< 300 lignes).

La revue de code : quoi regarder ?

Question	Exemple de remarque
Est-ce que ça fait ce que la PR annonce ?	« La suppression marche, mais la tâche reste dans le compteur »
Est-ce lisible ?	« Le nom <code>x</code> pourrait devenir <code>taskTitle</code> »
Y a-t-il des tests ?	« Un test pour le titre vide serait utile »
Des secrets ou fichiers en trop ?	« Le <code>.env</code> est dans la PR, attention ! »
Est-ce que ça casse autre chose ?	« Le CSS modifié s'applique aussi à la page de connexion »

*La revue n'est pas un examen : c'est une **conversation** pour améliorer le code et **partager la connaissance**.*

Commenter avec bienveillance

✗ À éviter

« C'est faux. »

« Pourquoi tu as fait ça ?? »

« Code illisible. »

Ne rien dire de positif

✓ Préférer

« Je crois que si la liste est vide, on a une erreur ici. Qu'en penses-tu ? »

« Peux-tu m'expliquer ce choix ? Je ne suis pas sûr de comprendre. »

« Suggestion : découper cette fonction en deux pour la rendre plus lisible. »

« Bien vu pour la gestion d'erreur ! 👍 »

- Parlez du **code**, pas de la **personne**
- Distinguez **bloquant** et **suggestion** (préfixe `nit:` = détail facultatif)
- Sur GitHub : *Comment*, *Approve* ou *Request changes*

Protéger la branche `main`

Sans protection, n'importe qui peut pousser directement sur `main` ... et casser la prod.

GitHub → **Settings** → **Branches** → **Add branch protection rule** (`main`) :

- ☒ *Require a pull request before merging* (+ 1 approbation)
- ☒ *Require status checks to pass* (dès que la CI existera, jour 3)

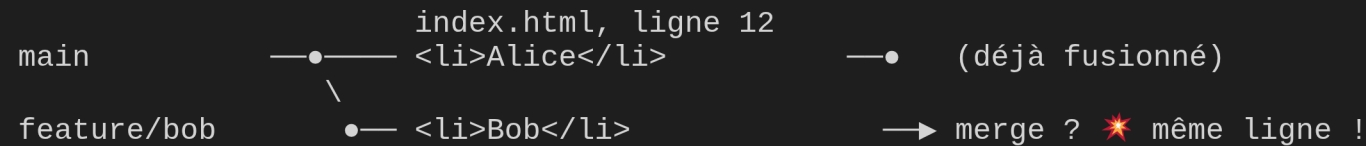
```
$ git push origin main
remote: error: GH006: Protected branch update failed for refs/heads/main.
remote: - Changes must be made through a pull request.
! [remote rejected] main -> main (protected branch hook declined)
```

“ Avec un compte GitHub gratuit, la protection de branche n'est disponible que sur les dépôts **publics**. ”

2.4 Conflits

Pourquoi un conflit ?

Deux personnes modifient **la même ligne** du même fichier, sur deux branches différentes. Git ne sait pas laquelle garder : il **vous demande** de choisir.



- Ce n'est **pas** une erreur ni un bug : c'est Git qui refuse de deviner à votre place
- Deux modifications sur des lignes **différentes** → Git fusionne tout seul
- Plus une branche vit longtemps, plus le risque de conflit augmente

Le message de Git

```
$ git merge main
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
$ git status
On branch feature/bob
You have unmerged paths.
  (fix conflicts and run "git commit")
Unmerged paths:
  both modified:   index.html
```

*Pas de panique : rien n'est perdu. Le fichier contient **les deux versions**, encadrées par des marqueurs.*

Lire les marqueurs de conflit

```
<ul>
<<<<<< HEAD
  <li>Bob</li>
=====
  <li>Alice</li>
>>>>>> main
</ul>
```

Marqueur	Signification
<<<<<< HEAD	Début de votre version (branche courante)
=====	Séparation entre les deux versions
>>>>>> main	Fin de la version qui arrive (ici depuis main)

Résoudre dans VS Code

VS Code affiche des boutons au-dessus du conflit :

Bouton	Effet
<i>Accept Current Change</i>	Garde votre version
<i>Accept Incoming Change</i>	Garde la version qui arrive
<i>Accept Both Changes</i>	Garde les deux (ici, c'est ce qu'on veut : deux <code></code>)

Ou éditez à la main. Résultat attendu, **sans aucun marqueur** :

```
<ul>
  <li>Alice</li>
  <li>Bob</li>
</ul>
```

Terminer (ou abandonner) la fusion

```
$ git add index.html  
$ git commit -m "merge: intègre main dans feature/bob"  
[feature/bob 2e3f4a5] merge: intègre main dans feature/bob  
$ git push
```

Vous êtes perdu ? Revenez à l'état d'avant la fusion :

```
$ git merge --abort
```

Éviter les conflits :

- Branches **courtes** (quelques heures à 2 jours), PR **petites**
- Récupérer `main` **souvent** dans sa branche
- Se parler : « je modifie le header cet après-midi »

Démo — Un conflit, de A à Z

`demos/demo-git-conflict/`

1. Deux branches modifient la même ligne de `index.html`
2. La première est fusionnée sans souci
3. La seconde déclenche un **conflit** : lecture du message et des marqueurs
4. Résolution dans VS Code (*Accept Both*), `git add`, `git commit`
5. Vérification : `git log --oneline --graph` montre le commit de fusion

“

À vous : `exercices/jour1/` contient un ***kata conflit*** à faire seul.

”

TP 1 — Projet collaboratif

`tp/tp1-git-collaboratif/`

TP 1 — Déroulé (1h aujourd'hui, suite demain)

Par groupes de 3-4 : un repo partagé, le site « **Promo DWWM** », une carte HTML par apprenant.

Étape	Action	Commande / geste
1	Un membre crée le repo, ajoute les autres en collaborateurs	GitHub → <i>Settings</i> → <i>Collaborators</i>
2	Chacun clone le repo	<code>git clone git@github.com:<groupe>/site-promo.git</code>
3	Chacun crée sa branche	<code>git switch -c feature/carte-<prenom></code>
4	Ajouter <code>cartes/<prenom>.html</code> , commiter, pousser	<code>git push -u origin feature/carte-<prenom></code>
5	Ouvrir une PR, demander une revue à un pair	Bouton <i>Compare & pull request</i>
6	Relire la PR d'un autre membre, commenter, approuver	Onglet <i>Files changed</i>
7	Merger les PR approuvées	<i>Merge pull request</i>

TP 1 — Le conflit volontaire

Tout le monde ajoute **sa ligne** dans la **même** liste de `index.html` :

```
<ul class="promo">
  <li><a href="cartes/alice.html">Alice</a></li>
  <!-- chacun ajoute sa ligne ICI -->
</ul>
```

- La première PR passe ; les suivantes affichent *This branch has conflicts*
- Résolvez **en local** : `git switch main && git pull`, `git switch feature/carte-<prenom>`, `git merge main`, résoudre, `git push`

Critères de réussite : chaque membre a une PR **revue et fusionnée**, au moins un conflit **résolu**, aucun marqueur `<<<<<<` dans `main`.

Si ça ne marche pas : `git status` d'abord, puis appelez le formateur. Exercices en solo : `exercices/jour1/`

Récap de la journée

Bloc	L'essentiel
1.1 Pourquoi le DevOps	Faire tomber le mur Dev / Ops ; culture CALMS ; boucle infinie ; DORA
1.2 CI / CD / CD	CI = vérifier chaque push ; livraison = prêt à déployer ; déploiement = sans humain
1.3 Outils	Une case par étape ; cette semaine : GitHub, Docker, Actions, Nginx, Uptime Kuma
2.1 Git	4 zones : travail → index → dépôt local → distant ; messages <i>Conventional Commits</i>
2.2 Branches	Une branche = un pointeur ; merge <i>fast-forward</i> ou avec commit de fusion
2.3 Équipe	GitHub Flow ; PR + revue bienveillante ; <code>main</code> protégée
2.4 Conflits	Même ligne modifiée ; lire les marqueurs, choisir, <code>add</code> + <code>commit</code>

Quiz — 5 questions

1. Citez les **5 piliers** de **CALMS**.
2. Quelle est la différence entre **livraison continue** et **déploiement continu** ?
3. Que fait `git add` exactement ?
4. Dans quel cas un merge est-il **fast-forward** ?
5. Dans un conflit, que représente la partie entre `<<<<<< HEAD` et `=====` ?



Réfléchissez 2 minutes, notez vos réponses, puis passez à la slide suivante.



Quiz — Réponses

1. **C**ulture, **A**utomatisation, **L**ean, **M**esure, partage (**S**haring)
2. Livraison continue : prêt à déployer, **un humain** déclenche. Déploiement continu : en prod **sans humain**
3. Il place les modifications dans l'**index** (le « panier ») pour le **prochain commit**
4. Quand la branche cible (`main`) **n'a pas avancé** depuis la création de la branche : Git avance juste l'étiquette
5. **Votre** version, celle de la branche courante (`HEAD`)

Glossaire du jour (1/2)

Terme	Définition
DevOps	<i>Development + Operations</i> : culture et pratiques pour livrer vite et de façon fiable
CALMS	Culture, Automatisation, Lean, Mesure, Partage (<i>Sharing</i>)
DORA	<i>DevOps Research and Assessment</i> : 4 indicateurs (fréquence, délai, taux d'échec, restauration)
CI	<i>Continuous Integration</i> : intégrer souvent, vérifier automatiquement chaque push
CD	<i>Continuous Delivery</i> (prêt à déployer) ou <i>Continuous Deployment</i> (déployé sans humain)
Pipeline	Enchaînement automatique d'étapes (lint, tests, build, déploiement)
Prod	Production : l'environnement des vrais utilisateurs
VPS / PaaS	Serveur virtuel loué / plateforme qui héberge l'application à votre place
FTP / SSH	Transfert de fichiers (ancien, non chiffré) / connexion sécurisée à un serveur
YAML	Format texte basé sur l'indentation, utilisé pour les workflows et Docker Compose

Glossaire du jour (2/2)

Terme	Définition
Commit	Point de sauvegarde de l'historique, identifié par un SHA (empreinte unique)
Index (<i>staging</i>)	Zone où l'on prépare le prochain commit (<code>git add</code>)
HEAD	Pointeur « vous êtes ici » : le commit / la branche courante
origin	Nom par défaut du dépôt distant (GitHub)
Branche	Étiquette mobile pointant sur un commit ; un brouillon parallèle
Merge	Fusion d'une branche dans une autre (<i>fast-forward</i> ou commit de fusion)
PR	<i>Pull Request</i> : demande de fusion, avec revue et tests automatiques
Revue de code	Relecture du code d'un pair avant fusion
Conflit	Même zone modifiée sur deux branches : Git demande de choisir
Lint	Vérification automatique du style et des erreurs courantes

Demain

Rebase, versions et tags — puis **Docker** : fini le « ça marche sur ma machine »

Avant demain : vérifiez que `docker run hello-world` fonctionne sur votre poste

Questions ?