

Contents

Étape 4 : déployer TaskFlow en « production »	1
Prérequis	1
Ce que vous allez construire	2
Partie A. Déploiement manuel (1h15)	2
Étape A.0 : ajouter les fichiers de production au dépôt (15 min)	2
Étape A.1 : créer le dossier de déploiement (5 min)	4
Étape A.2 : écrire le .env de production (10 min)	4
Étape A.3 : se connecter au registre (5 min, seulement si vos paquets sont privés)	5
Étape A.4 : générer le certificat auto-signé (5 min)	5
Étape A.5 : lancer la production (10 min)	6
Étape A.6 : vérifier HTTPS, la redirection et l'isolation (10 min)	7
Étape A.7 : sauvegarder, puis tester la restauration (15 min)	8
Étape A.8 : adapter docs/DEPLOIEMENT.md (5 min)	9
Je valide avec le formateur (partie A)	9
Partie B. Déploiement automatique (1h)	9
Étape B.1 : démarrer le runner (15 min)	9
Étape B.2 : créer l'environnement production (5 min)	11
Étape B.3 : ajouter le job deploy à release.yml (15 min)	11
Étape B.4 : la preuve, la production change seule (15 min)	14
Étape B.5 : rollback vers la version précédente (10 min)	14
Étape B.6 (bonus) : livraison continue avec approbation	15
Je valide avec le formateur (partie B)	15
Questions de réflexion	16

Étape 4 : déployer TaskFlow en « production »

Quand : jour 4 | **Durée** : partie A ~1h15 (matin), partie B ~1h (après-midi) **Objectif** : partie A, mettre TaskFlow en production à la main sur votre poste, derrière un reverse proxy Nginx en HTTPS, avec une sauvegarde **restaurée** ; partie B, **automatiser** exactement ces gestes : un merge sur main redéploie seul, un smoke test valide, et un rollback se fait en un clic.

On n'automatise que ce qu'on sait faire à la main : faites la partie A en entier avant la partie B.

Prérequis

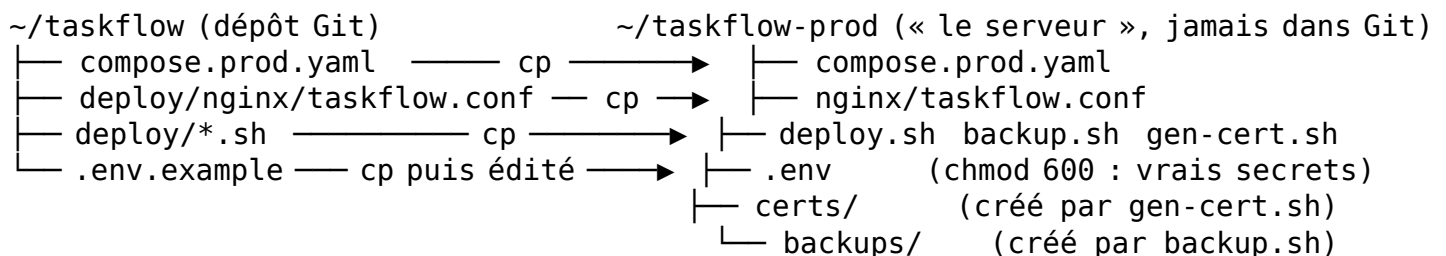
- L'étape 3 est terminée : le run **Release** est vert et les images taskflow-api et taskflow-front sont sur ghcr.io.
- Si vos paquets sont **privés** : le PAT read:packages créé à l'étape 3.8 (sinon, recréez-en un de la même façon).
- openssl installé (openssl version).
- Les ports 80 et 443 libres : ss -ltn | grep -E ':(80|443)\b' ne doit rien afficher (macOS : lsof -iTCP:80 -sTCP:LISTEN). S'ils sont pris (ou si votre Docker refuse les ports inférieurs à 1024), vous utiliserez **8081** et **8443** (voir A.2). N'utilisez pas 8080 : c'est le port de votre stack de développement.

- Slides du jour 4, blocs 5.1 à 5.3 et 4.4.

Vocabulaire

Terme	Signification
Reverse proxy	Un serveur placé devant l'application : point d'entrée unique, il gère HTTPS et transmet les requêtes
HTTPS / TLS	HTTP chiffré ; TLS (<i>Transport Layer Security</i>) est le protocole de chiffrement, qui s'appuie sur un certificat
Certificat auto-signé	Un certificat que l'on signe soi-même : le chiffrement fonctionne, mais le navigateur ne peut pas vérifier l'identité du site, d'où un avertissement
Dossier de déploiement (DEPLOY_DIR)	~/taskflow-prod : il joue le rôle du serveur ; c'est le seul endroit où vivent .env, certs/ et backups/
Smoke test (test de fumée)	Une vérification rapide après une mise en route : « est-ce que ça répond ? »
Rollback	Revenir à la version précédente
Runner self-hosted	Un runner GitHub Actions qui tourne chez vous et va chercher ses jobs chez GitHub

Ce que vous allez construire



Navigateur —https→ proxy (Nginx, 80→301, 443) → front:80 → api:3000 → db:5432 (v
stack Compose « taskflow-prod » ; seul « proxy » publie des ports

Partie A. Déploiement manuel (1h15)

Étape A.0 : ajouter les fichiers de production au dépôt (15 min)

Ces fichiers sont fournis dans la solution : vous les **ajoutez**, vous les **lisez**, puis vous les relisez en PR.

```

export FORMATION=~/formation-dwwm-devops # adaptez : votre clone du dépôt de l
cd ~/taskflow
git switch main && git pull
git switch -c feature/prod
  
```

```
cp "$FORMATION/tp/fil-rouge-taskflow/solution/compose.prod.yaml" .
cp "$FORMATION/tp/fil-rouge-taskflow/solution/.env.example" .env.example
cp -r "$FORMATION/tp/fil-rouge-taskflow/solution/deploy" .
mkdir -p docs
cp "$FORMATION/tp/fil-rouge-taskflow/solution/docs/DEPLOIEMENT.md" docs/
```

Ajoutez à la fin du .gitignore (certificats, clés privées et sauvegardes ne vont **jamais** dans Git) :

```
# Dossier de déploiement ("prod") : certificats, clés privées et sauvegardes
certs/
backups/
*.pem
*.key
*.crt
```

Lisez les fichiers et répondez (à l'oral avec votre binôme, ou par écrit) :

Fichier	Trouvez...
compose.prod.yaml	Le nom du projet (ligne name:). Quel service est le seul à avoir ports: ? D'où viennent les images (image:) ? Que se passe-t-il si POSTGRES_PASSWORD manque (:?) ?
.env.example	Les variables à remplir. Pourquoi GHCR_OWNER doit-il être en minuscules ? Où le mot de passe apparaît-il deux fois ?
deploy/nginx/taskflow.conf	Le bloc qui renvoie le code 301. Le chemin du certificat. Vers quel service proxy_pass envoie-t-il ?
deploy/deploy.sh	Les quatre commandes compose (pull, up, exec, ps). Où est noté le tag déployé ?
deploy/backup.sh	Où vont les sauvegardes ? Combien de jours sont conservés ? Quelle est la commande de restauration (en-tête) ?

Puis :

```
git add compose.prod.yaml .env.example deploy docs/DEPLOIEMENT.md .gitignore
git commit -m "build: ajoute la configuration de production (compose, proxy Nginx, sauvegardes)"
git push -u origin feature/prod
```

Ouvrez la PR, attendez la CI verte, faites-la relire par votre binôme si possible, fusionnez, puis :

```
git switch main && git pull
```

Résultat attendu : main contient compose.prod.yaml, deploy/ (avec nginx/taskflow.conf et trois scripts) et docs/DEPLOIEMENT.md. Le merge déclenche aussi un run **Release** : de nouvelles images sont publiées.

Si ça ne marche pas - cp: cannot stat : la variable FORMATION ne pointe pas vers votre clone du dépôt de la formation (ls "\$FORMATION"). - Votre ancien .env.example (étape 2) est remplacé : c'est voulu. Le nouveau contient aussi POSTGRES_PASSWORD, et compose.yaml (développement) ignore les autres variables.

Pourquoi ? Ces fichiers **sont** du code : versionnés, relus, et c'est le pipeline qui les copiera sur le serveur cet après-midi. Seuls les **secrets** (.env) et les **données** (certs/, backups/, le volume) restent hors de Git.

Étape A.1 : créer le dossier de déploiement (5 min)

```
mkdir -p ~/taskflow-prod/nginx
cd ~/taskflow-prod
cp ~/taskflow/compose.prod.yaml .
cp ~/taskflow/deploy/nginx/taskflow.conf nginx/
cp ~/taskflow/deploy/*.sh .
chmod +x *.sh
ls -l . nginx
```

Résultat attendu :

```
.:
-rwxr-xr-x  backup.sh
-rw-r--r--  compose.prod.yaml
-rwxr-xr-x  deploy.sh
-rwxr-xr-x  gen-cert.sh
drwxr-xr-x  nginx

nginx:
-rw-r--r--  taskflow.conf
```

Pourquoi ? Sur un vrai serveur, on ferait la même chose avec scp (copie par SSH, *Secure Shell*) dans /srv/taskflow. Seul l'accès change ; les fichiers et les commandes sont identiques.

Étape A.2 : écrire le .env de production (10 min)

```
cd ~/taskflow-prod
cp ~/taskflow/.env.example .env
chmod 600 .env
PASS=$(openssl rand -hex 24)
sed -i "s/changez-moi/$PASS/g" .env # macOS : sed -i '' "s/changez-moi/$PASS/g" .env
nano .env
```

Dans nano, modifiez seulement :

- GHCR_OWNER= votre pseudo GitHub **en minuscules** (par exemple alice-dwwm) ;
- HTTP_PORT et HTTPS_PORT : gardez 80 et 443, ou mettez 8081 et 8443 si ces ports sont pris.

Enregistrez (Ctrl+O, Entrée) et quittez (Ctrl+X). Vérifiez :

```
ls -l .env
grep -E '^(POSTGRES_PASSWORD|DATABASE_URL|GHCR_OWNER|HTTPS_PORT)= ' .env
docker compose -f compose.prod.yaml config --quiet && echo "Configuration valide"
```

Résultat attendu :

```
-rw----- 1 vous vous 1234 ... .env
POSTGRES_PASSWORD=3f9c...e1 (48 caractères)
DATABASE_URL=postgres://taskflow:3f9c...e1@db:5432/taskflow
GHCR_OWNER=alice-dwwm
HTTPS_PORT=443
Configuration valide
```

Le **même** mot de passe apparaît dans POSTGRES_PASSWORD et dans DATABASE_URL.

Si ça ne marche pas - required variable GHCR_OWNER is missing a value (ou POSTGRES_PASSWORD manquant dans .env) : la variable est absente ou mal nommée. Comparez avec ~/taskflow/.env.example. - Le mot de passe est encore changez-moi : la commande sed n'a pas fonctionné (sous macOS, utilisez la variante).

Pourquoi ? chmod 600 : seul votre utilisateur peut lire ce fichier (-rw-----). openssl rand -hex 24 produit un mot de passe long et aléatoire, fait uniquement de chiffres et de lettres a à f : aucun caractère spécial qui casserait l'URL de DATABASE_URL. Le .env de production n'existe **qu'ici** : ni dans Git, ni dans une image.

Étape A.3 : se connecter au registre (5 min, seulement si vos paquets sont privés)

Si vous avez rendu vos paquets publics à l'étape 3.8, passez à A.4. Sinon :

```
read -rs GHCR_PAT # collez votre PAT read:packages, puis Entrée
echo "$GHCR_PAT" | docker login ghcr.io -u <vous> --password-stdin
```

Résultat attendu : Login Succeeded.

Si ça ne marche pas - unauthorized: authentication required : le jeton est expiré, mal collé, ou n'a pas le droit read:packages. Recréez-en un (étape 3.8, option B).

Étape A.4 : générer le certificat auto-signé (5 min)

```
cd ~/taskflow-prod
./gen-cert.sh
ls -l certs/
openssl x509 -in certs/taskflow.crt -noout -subject -enddate
```

Résultat attendu :

```
Certificat créé dans certs/ (valable 365 jours, pour taskflow.localhost)
-rw-r--r-- 1 vous vous 1302 ... taskflow.crt
-rw-r--r-- 1 vous vous 1704 ... taskflow.key
```

```
subject=CN=taskflow.localhost
notAfter=... (dans un an)
```

Si ça ne marche pas - openssl n'est pas installé : `sudo apt install openssl` (Ubuntu, WSL) ou `brew install openssl` (macOS).

Pourquoi ? Le proxy a besoin d'un certificat et de sa clé privée pour parler HTTPS. Sur un vrai serveur avec un nom de domaine, **Let's Encrypt** (autorité gratuite, via l'outil certbot) fournit un certificat reconnu par tous les navigateurs. Ici, `taskflow.localhost` n'est pas un vrai domaine : on signe nous-mêmes.

Étape A.5 : lancer la production (10 min)

```
cd ~/taskflow-prod
./deploy.sh latest
```

Résultat attendu (après le téléchargement des images) :

```
==> Déploiement de la version : latest (dans /home/vous/taskflow-prod)
```

```
==> Téléchargement des images (pull)
```

```
...
```

```
==> Redémarrage des services
```

```
...
```

```
==> Rechargement du proxy
```

```
==> État des services
```

NAME	IMAGE	...	STATUS
taskflow-prod-api-1	ghcr.io/alice-dwwm/taskflow-api:latest	...	Up 25 seconds
taskflow-prod-db-1	postgres:16-alpine	...	Up 36 seconds
taskflow-prod-front-1	ghcr.io/alice-dwwm/taskflow-front:latest	...	Up 13 seconds
taskflow-prod-proxy-1	nginx:1.27-alpine	...	Up 2 seconds

```
==> Nettoyage des images inutilisées
```

```
...
```

```
==> Déploiement terminé : latest
```

`deploy.sh` ne fait rien de magique : c'est `docker compose -f compose.prod.yaml pull`, puis `up -d`, puis un rechargement de Nginx. Vous pouvez aussi lancer la stack directement avec `docker compose -f compose.prod.yaml up -d`.

Si ça ne marche pas - `denied` ou `unauthorized` au `pull` : paquets privés et pas de `docker login` (A.3). - `manifest unknown` : `GHCR_OWNER` contient des majuscules ou une faute ; comparez avec le nom affiché dans *Packages*. - `Bind for 0.0.0.0:80 failed: port is already allocated` (ou `permission denied` sur le port 80) : mettez `HTTP_PORT=8081` et `HTTPS_PORT=8443` dans `.env`, puis relancez `./deploy.sh latest`. - `dependency failed to start: container taskflow-prod-api-1 is unhealthy` : `docker compose -f compose.prod.yaml logs api`. Souvent : `DATABASE_URL` ne contient pas le même mot de passe que `POSTGRES_PASSWORD`. - proxy redémarre en boucle : `docker compose -f compose.prod.yaml logs proxy` ; le plus souvent, `certs/` est vide (relancez `./generate-cert.sh`).

Pourquoi ? Pas de `build` : en production : on **télécharge** les images construites

et testées par le pipeline. Le nom de projet taskflow-prod sépare cette stack de celle de développement : conteneurs, réseau et volume différents, sur la même machine.

Étape A.6 : vérifier HTTPS, la redirection et l'isolation (10 min)

Dans le terminal (-k accepte le certificat auto-signé ; --resolve associe le nom à votre machine) :

```
curl -sk --resolve taskflow.localhost:443:127.0.0.1 https://taskflow.localhost/health
curl -sI -H "Host: taskflow.localhost" http://localhost/ | grep -iE '^(HTTP|location)'
docker compose -f compose.prod.yaml ps --format 'table {{.Service}}\t{{.Status}}\t{{.Ports}}'
docker compose -f compose.prod.yaml exec proxy nginx -t
```

(Avec les ports 8081 et 8443 : --resolve taskflow.localhost:8443:127.0.0.1 https://taskflow.localhost:8443/health et http://localhost:8081/.)

Résultat attendu :

```
{"status":"ok","db":"up","version":"4e7a1c2","env":"production"}
HTTP/1.1 301 Moved Permanently
Location: https://taskflow.localhost/
SERVICE    STATUS                                PORTS
api          Up 3 minutes (healthy)               3000/tcp
db           Up 3 minutes (healthy)               5432/tcp
front        Up 3 minutes (healthy)               80/tcp
proxy        Up 2 minutes (healthy)               0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

La version est le SHA court du dernier commit publié par **Release** : c'est la preuve que vous faites tourner l'image du pipeline. Seul proxy a des ports avec une flèche -> : les autres ne sont pas joignables depuis l'extérieur.

Ouvrez maintenant <https://taskflow.localhost> (ou <https://taskflow.localhost:8443>) dans le navigateur. Il affiche un **avertissement de sécurité** (« Votre connexion n'est pas privée », « Attention : risque probable de sécurité ») : c'est **normal** avec un certificat auto-signé. Cliquez sur *Paramètres avancés* puis *Continuer vers taskflow.localhost* (ou *Accepter le risque et poursuivre*). TaskFlow s'affiche, avec un cadenas barré : la connexion **est** chiffrée, mais le navigateur ne peut pas vérifier qui a signé le certificat. Le pied de page affiche version <sha> (production).

Ouvrez aussi <http://taskflow.localhost> : le navigateur est redirigé vers <https://>.

Si ça ne marche pas - 502 Bad Gateway : front ou api n'est pas encore prêt ou a planté. `docker compose -f compose.prod.yaml ps`, puis `logs front` et `logs api`. - Le navigateur ne trouve pas taskflow.localhost : essayez <https://localhost> ; certains navigateurs ou réseaux d'entreprise ne résolvent pas *.localhost. - Avec HTTPS_PORT=8443, la redirection depuis `http://` vous envoie sur le port 443 : c'est une limite connue du lab (expliquée dans `taskflow.conf`). Ouvrez directement l'adresse en `https://...:8443`. - Sous Windows, navigateur Windows et Docker dans WSL2 (le Linux intégré à Windows) : localhost est normalement transmis à WSL2 ; sinon, utilisez l'adresse IP de WSL (`hostname -I`

dans Ubuntu).

Pourquoi ? Le reverse proxy est le **seul** point d'entrée : il chiffre (HTTPS), redirige HTTP vers HTTPS, et transmet au front. nginx -t vérifie la syntaxe **avant** de recharger : une faute de frappe ne coupe jamais le site.

Étape A.7 : sauvegarder, puis tester la restauration (15 min)

Créez **trois tâches** dans le navigateur. Puis :

```
cd ~/taskflow-prod
./backup.sh
ls -lh backups/
```

Résultat attendu :

```
==> Sauvegarde vers /home/vous/taskflow-prod/backups/taskflow-20261022-103512.sql.gz
==> Rotation : suppression des sauvegardes de plus de 7 jours
==> Terminé : 1 sauvegarde(s) conservée(s)
-rw-r--r-- 1 vous vous 1,1K ... taskflow-20261022-103512.sql.gz
```

Regardez ce qu'il y a dedans :

```
gunzip -c backups/taskflow-*.sql.gz | grep -E 'CREATE TABLE|COPY public.tasks' -A 4
```

Simulez une catastrophe : supprimez une tâche dans le navigateur (bouton *Supprimer*), puis comptez :

```
docker compose -f compose.prod.yaml exec db psql -U taskflow -d taskflow -c "SELECT j
```

Restaurez (remplacez le nom du fichier par celui affiché par `ls backups/`) et chronométrez :

```
time (gunzip -c backups/taskflow-20261022-103512.sql.gz | \
  docker compose -f compose.prod.yaml exec -T db psql -U taskflow -d taskflow)
docker compose -f compose.prod.yaml exec db psql -U taskflow -d taskflow -c "SELECT j
```

Résultat attendu : la restauration affiche une série de DROP TABLE, CREATE TABLE, COPY 3... puis la requête montre de nouveau **les trois tâches**. Rechargez le navigateur : la tâche supprimée est revenue. Notez la durée affichée par time.

Si ça ne marche pas - the input device is not a TTY : il manque -T après exec (indispensable quand on envoie des données par |). - Erreur : la sauvegarde est vide : la base ne répondait pas ; vérifiez `docker compose -f compose.prod.yaml ps`. - No such file or directory sur `backups/taskflow-...` : copiez le nom exact avec `ls backups/`.

Pourquoi ? Une sauvegarde qui n'a jamais été restaurée **n'existe pas** : on ne sait pas si elle fonctionne. La durée notée, c'est le temps d'indisponibilité en cas de vrai problème. backup.sh garde 7 jours (rotation) ; en production, on le planifie chaque nuit (cron, voir l'en-tête du script).

Étape A.8 : adapter docs/DEPLOIEMENT.md (5 min)

Relisez ~/taskflow/docs/DEPLOIEMENT.md (copié en A.0). Corrigez ce qui diffère chez vous : le chemin REPO= (votre dépôt est ~/taskflow), vos ports si vous utilisez 8081 et 8443, la durée de restauration mesurée en A.7. Puis :

```
cd ~/taskflow
git switch -c docs/deploiement
git commit -am "docs: adapte la procédure de déploiement à mon poste"
git push -u origin docs/deploiement
```

Ouvrez la PR, puis fusionnez-la quand la CI est verte.

Je valide avec le formateur (partie A)

- ☐ docker compose -f compose.prod.yml ps : les quatre services sont healthy, seul proxy publie des ports.
- ☐ https://taskflow.localhost affiche TaskFlow ; je sais expliquer l'avertissement du navigateur.
- ☐ http:// renvoie un 301 vers https://.
- ☐ /health répond "status":"ok" avec le SHA court publié par le pipeline et "env":"production".
- ☐ .env est en -rw----- et n'existe que dans ~/taskflow-prod.
- ☐ Une sauvegarde a été **restaurée** : la tâche supprimée est revenue.

Questions de réflexion :

1. Citez les trois catégories « code / configuration / données » et rangez-y chaque élément de ~/taskflow-prod.
 2. Pourquoi seul proxy publie-t-il des ports ? Que risquerait-on si db publiait 5432 ?
 3. Que faudrait-il changer pour un vrai serveur avec le domaine taskflow.example.fr ? (Indice : certificat, DNS, scp.)
-

Partie B. Déploiement automatique (1h)

Ce matin, vous avez : copié des fichiers, fait un docker login, lancé ./deploy.sh, vérifié /health. On confie maintenant ces gestes au pipeline.

```
merge sur main → release.yml └─ job build (ubuntu-latest, chez GitHub) : images → ghcr
                                   └─ job deploy ([self-hosted, lab], sur VOTRE poste) : needs: build
                                       copie → docker login → ./deploy.sh <sha> → smoke test /health
```

Les runners ubuntu-latest sont chez GitHub : ils ne peuvent pas atteindre votre poste. Le runner **self-hosted** tourne sur votre poste et **va chercher** ses jobs chez GitHub (connexion sortante) : rien à ouvrir sur votre machine.

Étape B.1 : démarrer le runner (15 min)

1. Sur GitHub, dans **votre** dépôt taskflow : **Settings > Actions > Runners > New self-hosted runner**, choisissez *Linux*. Dans la section *Configure*, repérez ./config.sh -

-url ... --token AAAA... et copiez **seulement** la valeur après --token (valable environ une heure). Ne suivez pas les instructions de téléchargement.

2. Configurez le runner du lab :

```
cd "$FORMATION/ressources/lab/runner"  
cp .env.example .env  
echo "$HOME/taskflow-prod"           # le chemin ABSOLU à mettre dans DEPLOY_DIR  
nano .env
```

Renseignez REPO_URL=https://github.com/<vous>/taskflow, RUNNER_TOKEN= (le jeton copié) et DEPLOY_DIR= (le chemin affiché par echo, sans ~ ni \$HOME).

3. Démarrez-le :

```
docker compose up -d --build  
docker compose logs -f runner
```

Résultat attendu à la fin des logs (quittez avec Ctrl+C, le runner continue de tourner) :

```
✓ Runner successfully added  
✓ Runner connection is good  
...  
Listening for Jobs
```

Sur GitHub, **Settings** > **Actions** > **Runners** : lab-runner apparaît avec les étiquettes self-hosted et lab et une pastille verte **Idle** (connecté, en attente de travail).

Vérifiez que le runner voit le Docker de votre poste et le dossier de déploiement :

```
docker compose exec runner docker ps --format '{{.Names}}'  
docker compose exec runner sh -c 'ls "$DEPLOY_DIR"'
```

Résultat attendu : la liste contient taskflow-prod-proxy-1, taskflow-prod-api-1... (les conteneurs **de votre poste**), puis le contenu de ~/taskflow-prod (compose.prod.yaml, deploy.sh, nginx, certs...).

Si ça ne marche pas (voir aussi le tableau de [../ressources/lab/runner/README.md](#))

- Invalid configuration provided for token ou erreur 404 : jeton expiré ou mal copié, ou REPO_URL faux. Nouveau jeton, correction du .env, puis docker compose up -d --force-recreate.
- Renseignez DEPLOY_DIR dans .env : la variable est vide.
- ls affiche un dossier vide : DEPLOY_DIR n'est pas exactement le chemin réel (echo \$HOME/taskflow-prod).
- permission denied ... docker.sock : Docker n'est pas démarré, ou votre installation n'expose pas /var/run/docker.sock (Docker Desktop l'expose ; sinon, appelez le formateur).

Pourquoi le même chemin ? Quand le runner lance docker compose, c'est le **Docker de votre poste** qui lit les chemins des volumes (./nginx/taskflow.conf, ./certs). Ils doivent donc exister au même endroit dans le conteneur et sur le poste.

Sécurité : le socket Docker, ce sont les clés de la maison. Le runner monte /var/run/docker.sock : il peut lancer n'importe quel conteneur, y compris un qui lit tout votre disque. C'est l'équivalent des droits root sur votre poste, et **quiconque peut faire exécuter un workflow sur ce runner contrôle votre machine**. - Acceptable ici : lab personnel, runner rattaché à **votre seul** dépôt,

job deploy déclenché uniquement par un push sur main (protégée) ou par vous (bouton *Run workflow*). - Votre dépôt est public : vérifiez **Settings > Actions > General**, section sur les workflows des PR venant de *forks* (copies du dépôt par d'autres personnes) : exigez une approbation pour **tous** les contributeurs extérieurs. - **Jamais** de runner self-hosted partagé, ni exécutant les PR d'inconnus. En entreprise : runners éphémères et isolés. - À la fin de la formation : docker compose down dans ressources/lab/runner, puis supprimez le runner dans **Settings > Actions > Runners**.

Étape B.2 : créer l'environnement production (5 min)

1. **Settings > Environments > New environment > nom : production > Configure environment**. Ne cochez rien pour l'instant (pas d'approbation obligatoire).
2. **Seulement si** votre HTTPS_PORT n'est pas 443 : **Settings > Secrets and variables > Actions > onglet Variables > New repository variable** : nom HTTPS_PORT, valeur 8443.

Résultat attendu : production apparaît dans la liste des environnements (et HTTPS_PORT dans les variables si besoin).

Pourquoi ? L'environnement garde l'historique des déploiements (onglet *Deployments*) et permet d'ajouter plus tard une approbation humaine. Aucun **secret** à créer : le GITHUB_TOKEN du run suffit pour docker login, et le .env de production ne quitte jamais votre poste.

Étape B.3 : ajouter le job deploy à release.yml (15 min)

```
cd ~/taskflow
git switch main && git pull
git switch -c feature/deploy
code .github/workflows/release.yml
```

Trois modifications :

1. Le déclencheur manuel reçoit un champ tag (pour le rollback). Remplacez la ligne workflow_dispatch: par :

```
# Déclenchement MANUEL depuis l'interface GitHub, avec un tag optionnel (rollback)
workflow_dispatch:
  inputs:
    tag:
      description: "Tag d'image à déployer (rollback). Vide = construire et déployer"
      required: false
      default: ""
      type: string
```

2. Le job build est sauté lors d'un rollback. Ajoutez sous name: Construire et publier les images :

```
# Sauté quand on lance un rollback avec un tag : les images existent déjà.
if: ${ inputs.tag == '' }
```

3. Ajoutez le job deploy à la fin du fichier (même indentation que build:) :

```
# ----- Job 2 : déployer la "prod" -----
deploy:
  name: Déployer la prod
  needs: build
  # Déployer si build a réussi OU a été volontairement sauté (rollback).
  if: ${ always() && (needs.build.result == 'success' || needs.build.result == 'sk

  # NOTRE runner, sur le poste ; il fournit la variable DEPLOY_DIR (~/.taskflow-prod
  runs-on: [self-hosted, lab]

  # Historique des déploiements, approbation éventuelle
  environment: production

  permissions:
    contents: read
    packages: read    # pour que « docker pull » puisse lire les images privées

  env:
    # Tag à déployer : celui saisi à la main (rollback), sinon celui qui vient d'êt
    DEPLOY_TAG: ${ inputs.tag != '' && inputs.tag || needs.build.outputs.tag }
    # Port HTTPS de la prod : variable de dépôt, 443 par défaut
    HTTPS_PORT: ${ vars.HTTPS_PORT || '443' }

  steps:
    # Sécurité : le tag sera utilisé dans des commandes shell ; on n'accepte que de
    - name: Valider le tag et le dossier de déploiement
      run: |
        if [[ ! "$DEPLOY_TAG" =~ ^[A-Za-z0-9._-]+$ ]]; then
          echo "Tag invalide : '$DEPLOY_TAG'" >&2
          exit 1
        fi
        if [ -z "${DEPLOY_DIR:-}" ] || [ ! -d "$DEPLOY_DIR" ]; then
          echo "DEPLOY_DIR absent ou introuvable : le runner du lab est-il bien con
          exit 1
        fi
        echo "Version à déployer : $DEPLOY_TAG, dans $DEPLOY_DIR"

    - uses: actions/checkout@v4

    # On ne copie JAMAIS .env ni certs/ : ils n'existent que dans le dossier de dép
    - name: Copier les fichiers de déploiement
      run: |
        mkdir -p "$DEPLOY_DIR/nginx"
        cp compose.prod.yaml "$DEPLOY_DIR/"
        cp deploy/nginx/taskflow.conf "$DEPLOY_DIR/nginx/"
        cp deploy/*.sh "$DEPLOY_DIR/"
        chmod +x "$DEPLOY_DIR"/*.sh
```

```

# Le jeton passe par une variable puis l'entrée standard : jamais en clair dans
- name: Connexion à ghcr.io
  env:
    GHCR_TOKEN: ${ secrets.GITHUB_TOKEN }
    GHCR_USER: ${ github.actor }
  run: printf '%s' "$GHCR_TOKEN" | docker login ghcr.io -u "$GHCR_USER" --passw

- name: Lancer deploy.sh
  run: |
    cd "$DEPLOY_DIR"
    ./deploy.sh "$DEPLOY_TAG"

# Smoke test : l'application répond-elle vraiment, en HTTPS ?
# -f : échec si HTTP >= 400 ; -k : certificat auto-signé accepté ; 10 essais es
# host.docker.internal = la machine hôte, vue depuis le conteneur du runner.
- name: Smoke test
  run: |
    curl -fsk --retry 10 --retry-all-errors --retry-delay 3 \
      "https://host.docker.internal:${HTTPS_PORT}/health"
    echo
    echo "Déploiement validé : $DEPLOY_TAG"

```

Raccourci si vous êtes en retard : le fichier complet est dans la solution. cp "\$FORMATION/tp/fil-rouge-taskflow/solution/.github/workflows/release.yml" .github/workflows/release.yml, puis lisez ce qui a changé avec git diff.

Vérifiez votre fichier complet : il doit être identique (aux commentaires près) à [solution/.github/workflows/release.yml](#). Puis :

```

git add .github/workflows/release.yml
git commit -m "ci: déploie automatiquement la prod après chaque merge"
git push -u origin feature/deploy

```

Ouvrez la PR, attendez la CI verte, **fusionnez**. Allez tout de suite dans l'onglet **Actions**.

Résultat attendu : le run **Release** enchaîne Construire et publier les images puis Déployer la prod. Pendant le second job, le runner passe à **Active** dans **Settings** > **Actions** > **Runners**. Le step *Smoke test* affiche :

```

{"status":"ok","db":"up","version":"9b0c1d2","env":"production"}
Déploiement validé : 9b0c1d2

```

Sur votre poste :

```

cat ~/taskflow-prod/.deployed_tag

```

affiche ce même SHA court. Sur la page d'accueil du dépôt, la rubrique *Deployments* indique production.

Si ça ne marche pas - Le job deploy reste *Queued* (« Waiting for a runner to pick up this job ») : runner arrêté ou *Offline* (docker compose ps et docker compose logs runner dans ressources/lab/runner), ou runs-on ne correspond pas aux étiquettes ([self-hosted, lab]). - DEPLOY_DIR absent ou introuvable

: variable vide ou faux chemin dans le .env du runner. - denied au pull dans le job : le paquet est privé et n'autorise pas ce dépôt. Page du paquet > **Package settings** > *Manage Actions access* > ajoutez le dépôt taskflow avec le rôle *Read*. - Smoke test: curl: (7) Failed to connect to host.docker.internal port 443 : votre proxy écoute sur un autre port ; créez la variable HTTPS_PORT (B.2). Vérifiez aussi que extra_hosts est présent dans le compose.yaml du runner. - Après un déploiement automatique, ./deploy.sh à la main affiche Permission denied : le runner écrit les fichiers en tant que root. sudo chown -R "\$USER" ~/taskflow-prod règle le problème.

Pourquoi ? Chaque geste de la partie A a son step : copie, docker login (avec le jeton du run au lieu de votre PAT), ./deploy.sh <sha>. Le smoke test remplace votre coup d'œil dans le navigateur : un déploiement « réussi » qui ne répond pas est un **échec**, et le job devient rouge.

Étape B.4 : la preuve, la production change seule (15 min)

```
cd ~/taskflow
git switch main && git pull
git switch -c feature/titre
```

Dans front/index.html, changez le titre visible et l'onglet, par exemple :

```
<title>TaskFlow de Prénom</title>
```

```
<h1>TaskFlow de Prénom</h1>
```

```
cd front && npm run format:check && cd ..
git commit -am "feat: personnalise le titre de TaskFlow"
git push -u origin feature/titre
```

Ouvrez la PR, attendez la CI verte, fusionnez. **Ne touchez plus à rien** : regardez l'onglet **Actions**, puis rechargez <https://taskflow.localhost> une fois le run **Release** vert.

Résultat attendu : le nouveau titre s'affiche, et le pied de page indique la nouvelle version (le SHA court du commit de merge), comme /health :

```
curl -sk --resolve taskflow.localhost:443:127.0.0.1 https://taskflow.localhost/health
```

Personne n'a tapé de commande sur le « serveur » : c'est du **déploiement continu**.

Si ça ne marche pas - L'ancien titre reste affiché : videz le cache (Ctrl+Shift+R). Vérifiez cat ~/taskflow-prod/.deployed_tag. - format:check échoue : npm run format dans front/, puis committez.

Étape B.5 : rollback vers la version précédente (10 min)

Retrouvez le SHA court de la version **précédente** (celle de l'étape B.3) : dans l'onglet **Actions**, ouvrez le run **Release** précédent, step *Smoke test* (« Déploiement validé : ... »), ou dans **Packages** > taskflow-front, liste des versions.

Lancez le rollback, au choix :

- interface : **Actions > Release > Run workflow** > champ *tag* : le SHA précédent > **Run workflow** ;
- ligne de commande : `gh workflow run release.yml -f tag=<sha-précédent>`.

Résultat attendu : dans le run, le job Construire et publier les images est **sauté** (gris), Déployer la prod s'exécute, et le smoke test affiche Déploiement validé : <sha-précédent>. Rechargez la page : l'**ancien titre** revient, et /health affiche l'ancienne version.

Pour revenir à la dernière version : relancez **Run workflow** avec le SHA le plus récent (ou laissez le champ vide pour reconstruire et déployer main).

Si ça ne marche pas - Tag invalide : copiez seulement les 7 caractères, sans espace. - manifest unknown dans deploy.sh : ce tag n'existe pas sur ghcr.io (vérifiez dans **Packages**).

Pourquoi ? Revenir en arrière **sans rien reconstruire**, en une minute, parce que chaque version est une image identifiée par son SHA. En urgence, la même chose à la main : `cd ~/taskflow-prod && ./deploy.sh <sha>`. Attention : le rollback revient sur le **code**, pas sur les **données** ; si une version a abîmé la base, il faut aussi restaurer une sauvegarde.

Étape B.6 (bonus) : livraison continue avec approbation

Settings > Environments > production > ☒ *Required reviewers* > ajoutez-vous (ou votre binôme) > **Save protection rules**. Fusionnez une petite PR : le job deploy affiche *Waiting for review* et attend qu'une personne clique sur **Review deployments > Approve and deploy**.

Pourquoi ? Avec une approbation humaine avant la production, on passe du **déploiement continu** à la **livraison continue** : tout est prêt automatiquement, mais un humain décide du moment.

Sur un vrai serveur (montré en slides, pas pratiqué) : - le job deploy tourne sur ubuntu-latest, chez GitHub, et **se connecte en SSH** au serveur avec une clé privée stockée dans un secret (SSH_PRIVATE_KEY) ; la clé publique est dans `~/.ssh/authorized_keys` d'un utilisateur dédié (deploy, non-root) ; - les fichiers sont copiés avec scp, puis `ssh deploy@serveur "cd /srv/taskflow && ./deploy.sh <sha>"` ; - Nginx est souvent installé directement sur le serveur, avec un certificat **Let's Encrypt** (certbot) renouvelé automatiquement ; le smoke test devient `curl -f https://votre-domaine/health, sans -k` ; - le serveur doit être protégé : connexion par clé uniquement, pare-feu (ufw), mises à jour.

Une esquisse complète du job est dans la section F de docs/DEPLOIEMENT.md. Alternative « pull » : un outil comme Watchtower, sur le serveur, surveille le registre et met à jour les conteneurs tout seul.

Je valide avec le formateur (partie B)

- ☐ Le runner lab-runner est **Idle** dans **Settings > Actions > Runners**, avec les étiquettes self-hosted et lab.
- ☐ Un merge sur main a redéployé la production **sans aucune commande** de ma part ; le smoke test est vert.

- ☐ Le nouveau titre est en ligne et `/health` affiche le SHA court du dernier merge.
- ☐ J'ai fait un rollback par *Run workflow* avec un tag précédent, et l'ancien titre est revenu.
- ☐ Je sais expliquer pourquoi monter le socket Docker est dangereux, et dans quelles conditions c'est acceptable.

Questions de réflexion

1. Comparez la partie A et la partie B : quels gestes ont disparu ? Lesquels restent manuels (et pourquoi) ?
2. Pourquoi le job `deploy` ne copie-t-il jamais `.env` ni `certs/` ? Où vivraient-ils sur un vrai serveur ?
3. Que se passe-t-il si le smoke test échoue ? La production est-elle revenue à l'ancienne version toute seule ? Que pourriez-vous ajouter au workflow pour cela ?
4. Le job `build` tourne chez GitHub, le job `deploy` chez vous. Pourquoi ne pas tout faire sur le runner self-hosted ?
5. Avec un dépôt public et ce runner, quel est le scénario d'attaque ? Quels réglages le rendent impossible ?