Jour 4 — Déploiement, automatisation, supervision

DevOps — Titre professionnel DWWM

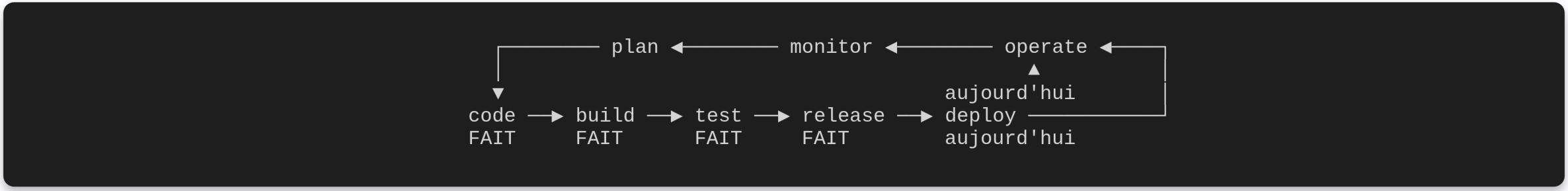
ForEach Academy

Formateur : Fabrice Claeys

Programme de la journée

Horaire	Contenu
9h00-9h10	Accueil, où en est-on ?
9h10-9h40	**Bloc 5.1 — Le serveur** (théorie + démo `demos/demo-ssh-serveur/`)
9h40-10h25	**Bloc 5.2 — Configuration et reverse proxy**
10h25-10h35	Pause
10h35-11h00	**Bloc 5.3 — Variables d'environnement, HTTPS, sauvegarde**
11h00-12h15	**TP fil rouge — Étape 4a** : déploiement manuel
13h15-13h45	**Bloc 4.4 — Déploiement continu**
13h45-14h40	**TP fil rouge — Étape 4b** : déploiement automatique
14h40-15h15	**Blocs 6.1 et 6.2 — Supervision et bonnes pratiques**
15h15-15h40	**TP fil rouge — Étape 5** : analyse d'un incident
15h40-16h00	Pause, récap, quiz, glossaire
16h00-17h00	**Évaluation** : QCM (15 min) puis mises en situation

Où en est-on sur la boucle DevOps ?



- **Fait** : code versionné, images Docker, CI verte, images publiées sur ghcr.io
- **Ce matin** : déployer TaskFlow **à la main** dans une « production », derrière Nginx en HTTPS
- **Cet après-midi** : **automatiser** ce déploiement, puis **surveiller** et **diagnostiquer**

> *Fil conducteur DevOps : **on n'automatise que ce qu'on sait faire à la main.** Chaque geste de ce matin deviendra un step du workflow cet après-midi.*

Bloc 5.1 — Le serveur

« Mais concrètement, l'application tourne où ? »

Où héberger une application ?

	VPS (serveur privé virtuel)	PaaS (plateforme en tant que service)	Cloud (fournisseur d'infrastructure)
Exemples	OVHcloud, Scaleway, Hetzner	Render, Railway, Fly.io, Heroku	AWS, Azure, Google Cloud
Vous gérez	Le système, Docker, Nginx, les mises à jour	Seulement le code (ou l'image)	Ce que vous choisissez : tout ou presque rien
Prix indicatif	5 à 10 € / mois	Gratuit à quelques dizaines d'€	À l'usage, vite complexe
Avantage	Contrôle total, gestes universels	Rapide : `git push` → en ligne	Très grande échelle, services managés
Limite	Vous êtes l'administrateur	Moins de contrôle, dépendance au fournisseur	Courbe d'apprentissage, facture

> *Le VPS est le meilleur terrain d'apprentissage : tout ce qu'un PaaS fait **pour vous**, vous le faites **vous-même** — et vous comprenez ce qui se passe.*

SSH par clé : le cadenas et la clé

SSH (*Secure Shell*) : protocole pour ouvrir un terminal **chiffré** sur une machine distante.

L'analogie

- La **clé publique** est un **cadenas** : vous en déposez un exemplaire sur le serveur. Tout le monde peut le voir, personne ne peut l'ouvrir sans la clé
- La **clé privée** est **la clé** : elle ne quitte **jamais** votre poste
- À la connexion, le serveur vérifie que vous possédez la clé qui ouvre son cadenas

Pourquoi pas un mot de passe ?

- Un mot de passe se **devine** : des robots en essaient des milliers par heure sur tout serveur public
- Une clé ed25519 est **impossible à deviner**
- On désactive donc la connexion par mot de passe (`PasswordAuthentication no`)

Créer et utiliser une paire de clés

```
ssh-keygen -t ed25519 -C "moi@poste"        # crée ~/.ssh/id_ed25519 (privée) et .pub (publique)
ssh-copy-id deploy@taskflow.example.com     # dépose la clé publique dans ~/.ssh/authorized_keys
ssh deploy@taskflow.example.com             # connexion, sans mot de passe
```

```
# ~/.ssh/config : un raccourci par serveur
Host taskflow
    HostName taskflow.example.com          # adresse du serveur
    User deploy                            # utilisateur distant
    IdentityFile ~/.ssh/id_ed25519         # quelle clé privée utiliser
```

- Ensuite : `ssh taskflow` suffit. Copier un fichier : `scp compose.prod.yaml taskflow:/srv/taskflow/`
- La clé privée : `chmod 600`, jamais dans Git, jamais envoyée par messagerie

Sécuriser le serveur : les trois réflexes

Réflexe	Pourquoi	Comment
Utilisateur non-root	`root` peut tout casser ; une erreur ou une intrusion aurait tous les droits	`adduser deploy`, puis `sudo` pour les commandes d'administration
Connexion par clé uniquement	Bloquer les robots qui devinent les mots de passe	`/etc/ssh/sshd_config` : `PermitRootLogin no`, `PasswordAuthentication no`
Pare-feu	N'ouvrir que les portes utiles	`ufw allow OpenSSH`, `ufw allow 80,443/tcp`, `ufw enable`

```
sudo apt update && sudo apt upgrade -y      # mises à jour de sécurité, à faire régulièrement
sudo apt install -y docker.io docker-compose-v2
sudo usermod -aG docker deploy              # deploy peut utiliser Docker sans sudo
```

> *`sudo` (superuser do) : exécuter **une** commande avec les droits administrateur, en la tapant consciemment.*

Démo formateur — `demos/demo-ssh-serveur/`

1. Générer une paire de clés, regarder les deux fichiers (`cat ~/.ssh/id_ed25519.pub`)
2. Se connecter **par mot de passe**, déposer la clé publique avec `ssh-copy-id`
3. Se reconnecter : plus de mot de passe demandé
4. Désactiver `PasswordAuthentication`, tenter une connexion sans clé → **refusée**
5. Copier `compose.prod.yaml` avec `scp`, lancer `docker compose` à distance
6. Activer le pare-feu, vérifier avec `sudo ufw status`

> *Vous regardez, vous notez les commandes : c'est exactement ce que vous ferez sur votre premier VPS (en stage, en entreprise ou pour votre portfolio).*

Dans le TP : votre poste joue le rôle du serveur

Geste	Sur un vrai serveur	Dans notre TP
Accéder au serveur	`ssh deploy@taskflow.example.com`	Ouvrir un terminal sur votre poste
Dossier de l'application	`/srv/taskflow/`	`~/taskflow-prod/`
Copier les fichiers	`scp`	`cp`
Configuration secrète	`.env` en `chmod 600`, hors Git	**Identique**
Lancer l'application	`docker compose -f compose.prod.yaml up -d`	**Identique** (stack `taskflow-prod`)
Reverse proxy, HTTPS	Nginx + Let's Encrypt	Conteneur Nginx + certificat auto-signé
Déploiement automatique	Job GitHub + `ssh`	Runner sur votre poste (cet après-midi)

> *Seul l'**accès** change. Les fichiers, les commandes et les réflexes sont **les mêmes** qu'en production réelle.*

Bloc 5.2 — Code, configuration, données : trois choses distinctes

	Où ça vit	Dans Git ?	Exemple TaskFlow
Code	Dans l'**image** publiée sur ghcr.io	Oui (le source)	`taskflow-api:a1b2c3d`
Configuration	Fichiers de déploiement + `.env` du serveur	Les fichiers oui, le `.env` **jamais**	`compose.prod.yaml`, `nginx/`, `.env`
Données	Volume Docker, sauvegardes	**Jamais**	`pgdata`, `backups/`

```
~/taskflow-prod/              ← « le serveur »
├── compose.prod.yaml         copié depuis le dépôt
├── nginx/taskflow.conf       copié depuis deploy/nginx/
├── deploy.sh  backup.sh  gen-cert.sh   copiés depuis deploy/
├── .env                      créé à la main ICI SEULEMENT (chmod 600)
├── certs/                    certificat + clé privée, générés ici
└── backups/                  sauvegardes de la base
```

`compose.prod.yaml` (1/3) — la base

```yaml
name: taskflow-prod                        # nom de la stack, distincte de celle de dev
services:
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: taskflow
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:?manquant dans .env}   # erreur claire si absent
      POSTGRES_DB: taskflow
    volumes:
      - pgdata:/var/lib/postgresql/data
    restart: unless-stopped                # relance après un crash ou un redémarrage du poste
    # healthcheck : identique à compose.yaml (pg_isready)
```

- `name: taskflow-prod` : conteneurs, réseau et volume **séparés** de votre environnement de dev
- `${VAR:?message}` : en dev on tolérait une valeur par défaut ; en production, on **refuse** de démarrer sans mot de passe

`compose.prod.yaml` (2/3) — API et front : des images, pas de build

```yaml
  api:
    image: ghcr.io/${GHCR_OWNER}/taskflow-api:${IMAGE_TAG:-latest}   # image publiée par le CI
    env_file: .env                          # DATABASE_URL, APP_ENV=production...
    depends_on:
      db: { condition: service_healthy }
    restart: unless-stopped

  front:
    image: ghcr.io/${GHCR_OWNER}/taskflow-front:${IMAGE_TAG:-latest}
    depends_on:
      api: { condition: service_healthy }
    restart: unless-stopped                 # pas de « ports: » : invisible depuis l'extérieur
```

- **Aucun** `build:` : la production exécute **l'image testée par le CI**, rien n'est recompilé
- `IMAGE_TAG` choisit la version : `a1b2c3d` (un commit précis) ou `latest`

`compose.prod.yaml` (3/3) — le proxy, seule porte d'entrée

```yaml
  proxy:
    image: nginx:1.27-alpine
    ports:
      - "${HTTP_PORT:-80}:80"             # le SEUL service qui publie des ports
      - "${HTTPS_PORT:-443}:443"
    volumes:
      - ./nginx/taskflow.conf:/etc/nginx/conf.d/default.conf:ro   # notre configuration
      - ./certs:/etc/nginx/certs:ro       # certificat et clé (ro = lecture seule)
    depends_on: [front]
    restart: unless-stopped
volumes:
  pgdata:
```

> *Différence avec le dev : en dev, le front publiait `8080`. En prod, **personne** sauf `proxy` n'est joignable de l'extérieur.*

Le reverse proxy : l'accueil de l'immeuble

L'analogie

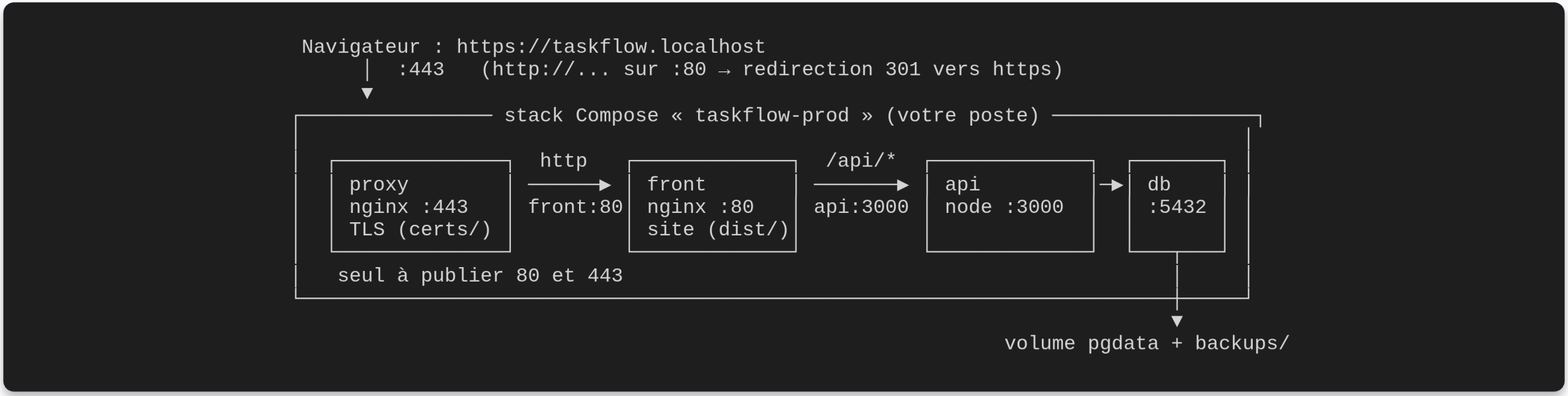
- Le visiteur ne connaît qu'**une adresse** : celle de l'immeuble
- L'**accueil** (le reverse proxy) vérifie, puis l'oriente vers le bon bureau
- Les bureaux (front, api, db) ne donnent **pas** sur la rue
- Comme un **standard téléphonique** : un seul numéro, plusieurs postes internes

Pourquoi un reverse proxy ?

- **Un seul point d'entrée** : un seul service exposé, les autres cachés
- **HTTPS (TLS)** géré à un seul endroit, pas dans chaque application
- **Plusieurs applications** sur une même machine (`app1.fr` , `app2.fr`)
- Bonus : journaux d'accès, compression, limitation de débit

> *Reverse (« inverse ») : un proxy classique représente le **client** ; celui-ci représente le **serveur**.*

Architecture de la production TaskFlow



- Deux Nginx, deux rôles : `proxy` (entrée, HTTPS) et `front` (sert le site, relaie `/api`)

`deploy/nginx/taskflow.conf` (1/2) — HTTP → HTTPS

```
server {
    listen 80;                                  # HTTP, non chiffré
    server_name taskflow.localhost localhost;   # noms de domaine concernés
    location / {                                # pour tous les chemins…
        return 301 https://$host$request_uri;   # « allez plutôt là » : même adresse en https
    }
}
```

- `server { }` = un site virtuel ; `location` = règles pour un chemin d'URL (adresse web) ; `301` = redirection **permanente**
- La solution ajoute `location = /nginx-health` (répond `ok`) pour le healthcheck du conteneur `proxy`
- Vérifier : `curl -I http://taskflow.localhost` → `301 Moved Permanently` + `Location: https://...`

```
docker compose -f compose.prod.yaml exec proxy nginx -t          # TOUJOURS tester la syntaxe
docker compose -f compose.prod.yaml exec proxy nginx -s reload   # puis recharger, sans coupure
```

> *Un `;` oublié + un redémarrage = site coupé. Sur un serveur avec Nginx installé par `apt` : `sudo nginx -t && sudo systemctl reload nginx`.*

`deploy/nginx/taskflow.conf` (2/2) — HTTPS → front

```
server {
    listen 443 ssl;                                    # HTTPS
    server_name taskflow.localhost localhost;
    ssl_certificate     /etc/nginx/certs/taskflow.crt;  # certificat (public)
    ssl_certificate_key /etc/nginx/certs/taskflow.key;  # clé privée (secrète)
    location / {
        proxy_pass http://front:80;                    # nom du service Compose
        proxy_set_header Host              $http_host;  # nom (+ port) demandé
        proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

- `front:80` : comme `db:5432` hier, le nom du service est résolu par le réseau Compose
- Les chemins `/etc/nginx/certs/...` sont ceux **dans le conteneur** (montés depuis `./certs`)

Les en-têtes `X-Forwarded-*` : pourquoi ?

Sans eux, l'application ne voit qu'**un seul client** : le proxy.

```
Client 203.0.113.7 ──https──▶ proxy ──http──▶ front ──▶ api
                                                l'api voit : IP = proxy, protocole = http  (faux !)
```

En-tête	Ce qu'il transmet	Utile pour
`Host`	Le nom de domaine demandé (`$http_host` garde aussi le port)	Générer des liens corrects
`X-Forwarded-For`	L'adresse IP réelle du client (+ les proxys traversés)	Journaux, limitation par IP
`X-Forwarded-Proto`	Le protocole d'origine (`https`)	Savoir que la connexion était sécurisée
`X-Real-IP`	L'IP du client, seule	Variante simple de `X-Forwarded-For`

> *IP (Internet Protocol) : l'adresse d'une machine sur le réseau.*

Le `.env` du serveur

```
# ~/taskflow-prod/.env — créé à la main, JAMAIS dans Git
POSTGRES_PASSWORD=8f3c...e91a                  # généré : openssl rand -hex 24
DATABASE_URL=postgres://taskflow:8f3c...e91a@db:5432/taskflow
APP_ENV=production
GHCR_OWNER=votre-compte                        # en minuscules
IMAGE_TAG=latest
HTTP_PORT=80
HTTPS_PORT=443
```

```
chmod 600 .env          # lecture/écriture pour vous seul (6), rien pour les autres (00)
```

- Partir de `.env.example` (versionné, **sans** vraies valeurs), le compléter **sur le serveur**
- Le nom des variables doit être **exact** : une faute de frappe = une variable ignorée

Télécharger des images privées : `docker login ghcr.io`

```
echo "$GHCR_PAT" | docker login ghcr.io -u votre-compte --password-stdin
# Login Succeeded
```

- Vos packages ghcr.io sont **privés** par défaut : la production doit s'authentifier
- Jeton personnel (PAT, *Personal Access Token*) : *GitHub → Settings → Developer settings → Personal access tokens (classic)* → droit `read:packages` uniquement
- `--password-stdin` : le jeton passe par l'entrée standard, il n'apparaît pas dans l'historique du terminal
- Cet après-midi, le job de déploiement se connectera avec le `GITHUB_TOKEN` du run

> *Moindre privilège : un jeton qui ne sert qu'à **lire** des images ne doit pas pouvoir **écrire** dans vos dépôts.*

HTTPS en 5 minutes

HTTPS = HTTP (*HyperText Transfer Protocol*, le protocole du web) + **TLS** (*Transport Layer Security*).

TLS garantit…	Sans TLS
Confidentialité : les échanges sont chiffrés	Mot de passe lisible sur le Wi-Fi du café
Intégrité : personne ne modifie la page en route	Publicité ou script injecté au passage
Authenticité : vous parlez bien à `taskflow.fr`	Faux site impossible à détecter

- Le serveur présente un **certificat** : « je suis `taskflow.fr`, voici ma clé publique »
- Ce certificat est **signé** par une **autorité de certification** (CA, *Certificate Authority*) que le navigateur connaît déjà
- Le cadenas du navigateur = « certificat valide, signé par une autorité de confiance, pour ce nom »

Let's Encrypt (réel) vs auto-signé (notre TP)

	Let's Encrypt (vrai serveur)	Auto-signé (notre TP)
Qui signe ?	Une autorité reconnue, gratuite	Vous-même
Prérequis	Un nom de domaine qui pointe vers le serveur, port 80 ouvert	Rien
Navigateur	Cadenas, aucun avertissement	Avertissement « connexion non sécurisée »
Durée	90 jours, **renouvelé automatiquement**	Celle que vous choisissez
Commande	`sudo certbot --nginx -d taskflow.fr`	`./gen-cert.sh`

- `certbot` (outil de Let's Encrypt) obtient le certificat, modifie la configuration Nginx et programme le renouvellement
- Avec l'auto-signé, le chiffrement est **identique** ; seule l'**authenticité** n'est pas garantie par un tiers
- L'avertissement du navigateur est **normal** ici : *Avancé → Continuer vers taskflow.localhost*

Générer le certificat auto-signé : `deploy/gen-cert.sh`

```
mkdir -p certs
openssl req -x509 -nodes -newkey rsa:2048 -days 365 \
  -keyout certs/taskflow.key \
  -out certs/taskflow.crt \
  -subj "/CN=taskflow.localhost" \
  -addext "subjectAltName=DNS:taskflow.localhost,DNS:localhost,IP:127.0.0.1,DNS:host.docker.internal"
chmod 644 certs/taskflow.key certs/taskflow.crt   # lisibles par le conteneur Nginx
```

Option	Signification
`-x509`	Produire directement un certificat (pas une demande de signature)
`-nodes`	Clé privée non protégée par mot de passe (Nginx doit la lire seul)
`-newkey rsa:2048`	Créer une nouvelle clé RSA (algorithme de chiffrement) de 2048 bits
`-subj` / `-addext`	Les noms couverts : `taskflow.localhost`, mais aussi `host.docker.internal` (smoke test)

> *`*.localhost` pointe toujours vers votre propre machine : pas besoin de modifier le DNS. Le dossier `certs/` n'existe **que** dans `~/taskflow-prod/`, jamais dans Git.*

`deploy/backup.sh` (extrait commenté)

```bash
#!/usr/bin/env bash
set -euo pipefail                                  # s'arrêter à la première erreur
cd "$(dirname "$0")"                               # se placer dans ~/taskflow-prod
mkdir -p backups
FILE="backups/taskflow-$(date +%Y%m%d-%H%M%S).sql.gz"
docker compose -f compose.prod.yaml exec -T db \
  pg_dump -U taskflow -d taskflow --clean --if-exists | gzip > "$FILE"
[ -s "$FILE" ] || { echo "Sauvegarde vide !" >&2; exit 1; }      # refuser un fichier vide
find backups -name 'taskflow-*.sql.gz' -mtime +7 -delete         # rotation : 7 jours
echo "Sauvegarde OK : $FILE"
```

- `pg_dump` exporte la base en fichier SQL (les ordres pour la recréer) ; `exec -T` = sans terminal (indispensable en script) ; **rotation** = garder 7 jours
- Le volume ne protège pas d'un `down -v`, d'un disque qui lâche ou d'une migration ratée. Sur un vrai serveur : copier les sauvegardes **ailleurs** et planifier avec `cron` (planificateur Unix) : `0 3 * * * ~/taskflow-prod/backup.sh`

Tester la restauration

```
cd ~/taskflow-prod
./backup.sh                                          # 1. sauvegarder
docker compose -f compose.prod.yaml exec db \
  psql -U taskflow -d taskflow -c "DELETE FROM tasks;"   # 2. simuler une catastrophe
gunzip -c backups/taskflow-<date>.sql.gz | \
  docker compose -f compose.prod.yaml exec -T db psql -U taskflow -d taskflow   # 3. restaurer
docker compose -f compose.prod.yaml exec db \
  psql -U taskflow -d taskflow -c "SELECT count(*) FROM tasks;"   # 4. vérifier
```

> ***Une sauvegarde qui n'a jamais été restaurée n'existe pas.*** *Faites-le maintenant, au calme, et notez la durée : c'est le temps d'indisponibilité en cas de vrai problème.*

TP fil rouge — Étape 4a : déploiement manuel (1h15)

Énoncé : `tp/fil-rouge-taskflow/etape4-deploiement.md` (partie A)

```
mkdir -p ~/taskflow-prod && cd ~/taskflow-prod
cp ~/taskflow/compose.prod.yaml . && cp -r ~/taskflow/deploy/nginx .
cp ~/taskflow/deploy/*.sh . && cp ~/taskflow/.env.example .env
nano .env && chmod 600 .env                # vraies valeurs, ici seulement
./gen-cert.sh                              # certificat auto-signé dans certs/
echo "$GHCR_PAT" | docker login ghcr.io -u votre-compte --password-stdin
./deploy.sh latest                         # pull + up -d (détail cet après-midi)
```

Validation : `docker compose -f compose.prod.yaml ps` tout `healthy` · `https://taskflow.localhost` affiche TaskFlow · `http://` redirige vers `https://` · `/health` répond `"status":"ok"` · une sauvegarde **restaurée**

Étape 4a — Si ça ne marche pas

Symptôme	Cause probable	Correction
`bind: address already in use` sur 80 ou 443	Un autre programme (ou la stack de dev) occupe le port	`HTTP_PORT=8081`, `HTTPS_PORT=8443` dans `.env`
`denied` ou `unauthorized` au `pull`	Pas connecté à ghcr.io, ou jeton sans `read:packages`	Refaire `docker login ghcr.io`
`manifest unknown`	Tag inexistant ou `GHCR_OWNER` avec majuscules	Vérifier *Packages* sur GitHub
`POSTGRES_PASSWORD manquant dans .env`	`.env` absent ou variable mal nommée	Comparer avec `.env.example`
`proxy` redémarre en boucle	Erreur dans `taskflow.conf` ou certificat absent	`logs proxy`, relancer `./gen-cert.sh`
502 Bad Gateway	`front` ou `api` pas prêt ou en erreur	`ps`, puis `logs front` et `logs api`
Avertissement de sécurité du navigateur	Certificat auto-signé	Normal : l'accepter

Bloc 4.4 — Ce qu'on a fait ce matin devient un workflow

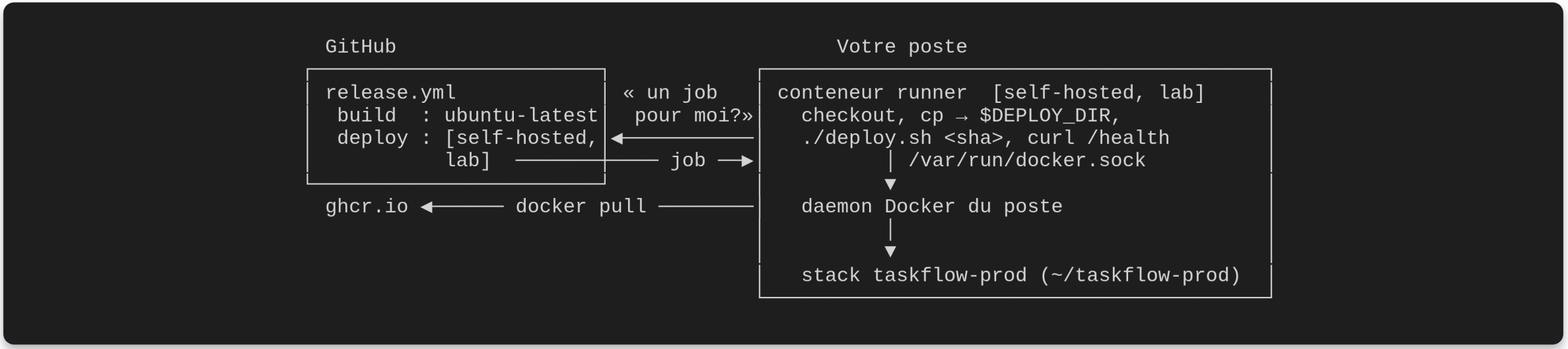
Geste manuel (étape 4a)	Step du job `deploy`
Copier `compose.prod.yaml`, `nginx/`, scripts	`cp` vers `$DEPLOY_DIR` (jamais `.env` ni `certs/`)
`docker login ghcr.io` avec un PAT	`docker login ghcr.io` avec le `GITHUB_TOKEN` du run
`./deploy.sh <tag>`	`./deploy.sh` avec le **SHA court du commit** mergé
Ouvrir `/health` dans le navigateur	**Smoke test** : `curl -f .../health` (échec = job rouge)

```
PR verte ──▶ merge ──▶ release.yml ┬─ job build  (ubuntu-latest) : images → ghcr.io
                                   └─ job deploy ([self-hosted, lab]) : needs: build
                                        copie → deploy.sh <sha> → smoke test /health
```

> *Smoke test (test de fumée) : vérification rapide après une mise en route — « est-ce que ça fume ? ».*

Pourquoi un runner auto-hébergé ici ?

- Les runners `ubuntu-latest` sont **chez GitHub** : ils ne peuvent pas entrer sur votre poste
- Un runner **self-hosted** tourne **sur votre poste** et **va chercher** les jobs chez GitHub (connexion sortante) : rien à ouvrir



Démarrer le runner — `ressources/lab/runner/`

```
cd ressources/lab/runner
cp .env.example .env            # REPO_URL, RUNNER_TOKEN, DEPLOY_DIR=/home/<vous>/taskflow-prod
docker compose up -d --build
docker compose logs -f          # attendre « Listening for Jobs »
```

- `RUNNER_TOKEN` : *Settings → Actions → Runners → New self-hosted runner* (valable environ 1 h)
- Le runner apparaît dans cette page avec le statut **Idle** (en attente) et les labels `self-hosted`, `lab`
- `DEPLOY_DIR` : chemin **absolu**, monté **au même chemin** dans le runner — car c'est le daemon Docker du poste qui résout les chemins des volumes de `compose.prod.yaml`
- Le runner monte le **socket Docker** (`/var/run/docker.sock`) : ses commandes `docker` pilotent le Docker de votre poste

Sécurité : le socket Docker, c'est les clés de la maison

- Accéder au socket Docker = pouvoir lancer **n'importe quel conteneur**, y compris un qui monte tout le disque : c'est l'équivalent des **droits root sur le poste**
- Donc : quiconque peut faire exécuter un workflow sur ce runner **contrôle votre machine**

Acceptable	Jamais
Lab personnel, dépôt **à vous**, runner rattaché à **ce seul dépôt**	Dépôt public **sans** approbation des workflows de *forks*
Job `deploy` déclenché uniquement par un **push sur** `main` (protégée)	Runner exécutant des PR venant de *forks*
Arrêter le runner après la formation (`docker compose down`)	Runner partagé entre plusieurs personnes ou projets

> *Votre dépôt est public : activez Settings > Actions > « Require approval for all outside collaborators ». En entreprise : dépôt privé, runners **éphémères** (détruits après chaque job), isolés, sans socket Docker de l'hôte.*

Secrets et environment `production`

- **Aucun secret à créer** pour notre déploiement : `GITHUB_TOKEN` suffit pour `docker login ghcr.io`
- Le `.env` de production reste **sur le poste**, dans `$DEPLOY_DIR` : il ne transite jamais par GitHub

Environment GitHub : *Settings → Environments → New environment →* `production`

Option	Effet
Historique	Onglet *Deployments* : qui a déployé quoi, quand
Required reviewers	Le job attend une approbation humaine → **livraison continue**
Sans reviewer	Déploiement automatique dès le merge → **déploiement continu**
Secrets d'environment	Visibles **seulement** par les jobs qui déclarent `environment: production`

`deploy/deploy.sh` (extrait commenté)

```
set -euo pipefail                                   # s'arrêter à la première erreur
TAG="${1:?Usage: ./deploy.sh <tag>}"                # tag obligatoire, ex. a1b2c3d
cd "${DEPLOY_DIR:-$(dirname "$0")}"                 # le dossier « serveur »
compose() { IMAGE_TAG="$TAG" docker compose -f compose.prod.yaml "$@"; }   # raccourci
compose pull                                        # télécharger les images de ce tag
compose up -d --remove-orphans                      # recréer ce qui a changé
compose exec -T proxy nginx -s reload               # relire la config Nginx copiée
compose ps                                          # afficher l'état des services
echo "$TAG" > .deployed_tag                         # mémoriser la version en ligne
docker image prune -f                               # supprimer les anciennes images
```

- `IMAGE_TAG` est lu par `compose.prod.yaml` : c'est **lui** qui choisit la version déployée
- `up -d` ne recrée **que** les conteneurs dont l'image ou la config a changé ; la base et son volume restent en place
- Extrait simplifié de `deploy/deploy.sh` (la solution réessaie le `reload` le temps que `proxy` démarre)

`release.yml` — le job `deploy` (1/2)

```yaml
deploy:
  needs: build                              # après le build et la publication des images
  if: ${{ always() && (needs.build.result == 'success' || needs.build.result == 'skipped') }}
  runs-on: [self-hosted, lab]               # notre runner, sur le poste
  environment: production                   # historique, approbation éventuelle
  permissions: { contents: read, packages: read }
  env:
    DEPLOY_TAG: ${{ inputs.tag != '' && inputs.tag || needs.build.outputs.tag }}
    HTTPS_PORT: ${{ vars.HTTPS_PORT || '443' }}
```

- `DEPLOY_TAG` : le SHA court calculé par `build`… ou le tag saisi à la main lors d'un **rollback**
- `if: always() && ...` : déployer aussi quand `build` a été **sauté** volontairement (rollback, voir plus loin)
- `$DEPLOY_DIR` n'est pas ici : c'est une variable **du conteneur runner** (définie dans son `.env`)

`release.yml` — le job `deploy` (2/2) : les steps

```yaml
steps:
  - uses: actions/checkout@v4          # récupère compose.prod.yaml et deploy/
  - name: Copier les fichiers de déploiement (jamais .env ni certs/)
    run: |
      mkdir -p "$DEPLOY_DIR/nginx"
      cp compose.prod.yaml deploy/*.sh "$DEPLOY_DIR/"
      cp deploy/nginx/taskflow.conf "$DEPLOY_DIR/nginx/"
  - name: Connexion à ghcr.io
    env: { GHCR_TOKEN: "${{ secrets.GITHUB_TOKEN }}", GHCR_USER: "${{ github.actor }}" }
    run: printf '%s' "$GHCR_TOKEN" | docker login ghcr.io -u "$GHCR_USER" --password-stdin
  - run: cd "$DEPLOY_DIR" && ./deploy.sh "$DEPLOY_TAG"
```

- Le jeton passe par une variable d'environnement puis l'entrée standard : jamais en clair dans la commande
- La solution valide aussi le format du tag avant de l'utiliser dans une commande (sécurité)

`release.yml` — le smoke test

```yaml
- name: Smoke test
  run: |
    curl -fsk --retry 10 --retry-all-errors --retry-delay 3 \
      "https://host.docker.internal:${HTTPS_PORT}/health"
```

Option	Rôle
`-f`	Échouer si le code HTTP est ≥ 400 → le job devient **rouge**
`-s` / `-k`	Silencieux / accepter le certificat auto-signé
`--retry 10 --retry-all-errors --retry-delay 3`	Réessayer 10 fois, toutes les 3 s : les conteneurs mettent quelques secondes à être prêts
`host.docker.internal`	Depuis le conteneur runner, l'adresse **du poste**, là où `proxy` publie le port 443

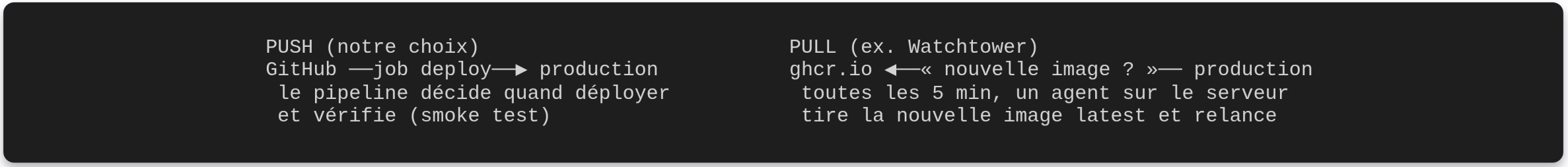
> *Un déploiement « réussi » qui ne répond pas n'est **pas** réussi : c'est le smoke test qui en décide.*

Sur un vrai serveur : la même chose par SSH (montré, pas pratiqué)

```yaml
deploy:
  needs: build
  runs-on: ubuntu-latest                # runner GitHub : il joint un serveur PUBLIC
  environment: production
  steps:
    - name: Déployer par SSH
      env:
        SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}   # clé privée stockée en secret
      run: |
        umask 077 && echo "$SSH_PRIVATE_KEY" > key       # fichier lisible par nous seuls
        ssh -i key -o StrictHostKeyChecking=accept-new deploy@taskflow.example.com \
          "cd /srv/taskflow && ./deploy.sh ${{ needs.build.outputs.tag }}"
```

- La clé **publique** correspondante est dans `~/.ssh/authorized_keys` de l'utilisateur `deploy` du serveur
- Le smoke test devient `curl -f https://taskflow.example.com/health` — certificat Let's Encrypt, donc sans `-k`

Push ou pull ?



	Push (runner, SSH)	Pull (Watchtower)
Qui déclenche ?	Le pipeline, après le build	Un agent qui surveille le registre
Visibilité	Tout dans l'onglet Actions, job rouge si échec	Dans les logs du serveur seulement
Accès entrant	Le pipeline doit joindre le serveur	**Aucun** : le serveur sort vers le registre
Version	Tag précis (SHA)	Souvent `latest` : moins de contrôle

> *Watchtower : conteneur qui surveille les images des autres conteneurs et les met à jour. Simple, mais on perd le contrôle fin et le smoke test dans le pipeline.*

Rollback : revenir à la version précédente

Une version cassée est en ligne : on redéploie **un ancien tag**, sans rien reconstruire.

```yaml
on:
  push: { branches: [main] }
  workflow_dispatch:                      # bouton « Run workflow »
    inputs:
      tag: { description: "Tag à redéployer (vide = build normal)", default: "" }
# build  : if: ${{ inputs.tag == '' }}     → sauté lors d'un rollback
# deploy : if: ${{ always() && (needs.build.result == 'success' || needs.build.result == 'skipped') }}
#          DEPLOY_TAG: ${{ inputs.tag != '' && inputs.tag || needs.build.outputs.tag }}
```

```bash
gh workflow run release.yml -f tag=a1b2c3d     # ou : onglet Actions → Release → Run workflow
cd ~/taskflow-prod && ./deploy.sh a1b2c3d      # en urgence, à la main
```

> *Le rollback revient sur le **code**, pas sur les **données** : si la version cassée a abîmé la base, il faut aussi restaurer une sauvegarde.*

TP fil rouge — Étape 4b : déploiement automatique (55 min)

Énoncé : `tp/fil-rouge-taskflow/etape4-deploiement.md` (partie B)

1. Démarrer le runner (`ressources/lab/runner/`) → **Idle** dans *Settings* → *Actions* → *Runners*
2. Créer l'environment `production`
3. Ajouter le job `deploy` à `release.yml` (`needs: build`, `runs-on: [self-hosted, lab]`) avec copie, `deploy.sh`, smoke test
4. **La preuve** : modifier le titre de la page dans `front/index.html` → PR → CI verte → merge → regarder la production **changer seule** ; `/health` affiche le nouveau SHA
5. **Rollback** : *Run workflow* avec le tag précédent → l'ancien titre revient
6. Bonus : ajouter un reviewer obligatoire sur `production` et observer l'attente d'approbation

Si le job reste *Queued* : runner arrêté ou labels différents (`docker compose logs` du runner).

S6 — Supervision et bonnes pratiques

« Ce sont les utilisateurs qui nous ont prévenus… »

Question à laquelle répondre en une minute	Outil
Est-ce que ça marche ?	Healthchecks, `/health`, Uptime Kuma
Que s'est-il passé ?	Les logs (journaux)
Depuis quand, combien ?	Métriques, tableaux de bord (pour aller plus loin)

Les logs avec Docker Compose

```
cd ~/taskflow-prod
docker compose -f compose.prod.yaml logs api                # tout l'historique de l'api
docker compose -f compose.prod.yaml logs -f api             # suivre en direct (Ctrl+C)
docker compose -f compose.prod.yaml logs --since 15m api    # les 15 dernières minutes
docker compose -f compose.prod.yaml logs --tail 50 proxy    # les 50 dernières lignes
docker compose -f compose.prod.yaml logs -f                 # tous les services, mélangés
```

- Une application conteneurisée écrit ses logs sur la **sortie standard** (*stdout*), Docker les collecte
- Ne **jamais** écrire les logs dans un fichier à l'intérieur du conteneur : ils disparaîtraient avec lui

Logs JSON et niveaux

```
{"time":"2026-10-22T14:03:12.481Z","level":"info","method":"GET","path":"/api/tasks","status":200,"ms":4.2}
{"time":"2026-10-22T14:03:15.007Z","level":"error","message":"relation \"tasks\" does not exist"}
```

Niveau	Quand	Action
`debug`	Détails pour le développement	Désactivé en production
`info`	Fonctionnement normal (une requête servie)	Aucune
`warn`	Anormal mais pas bloquant (requête lente)	À surveiller
`error` / `fatal`	Une opération a échoué / l'application s'arrête	**Agir**

```
docker compose -f compose.prod.yaml logs --no-log-prefix api | grep '"level":"error"'
docker compose -f compose.prod.yaml logs --no-log-prefix api | jq 'select(.status >= 500)'
```

JSON (*JavaScript Object Notation*) : une ligne = un objet, facile à filtrer par programme (`jq`).

Healthchecks et `restart` : qui fait quoi ?

Mécanisme	Ce qu'il fait	Ce qu'il ne fait pas
`restart: unless-stopped`	Relance un conteneur dont le processus **s'est arrêté** (crash, redémarrage du poste)	Réparer une mauvaise configuration : il relance en boucle (`Restarting`)
`healthcheck`	Marque le conteneur `healthy` / `unhealthy`	**Redémarrer** un conteneur `unhealthy` (Docker Compose ne le fait pas)
`depends_on: service_healthy`	Attend qu'une dépendance soit prête **au démarrage**	Surveiller ensuite
Smoke test du job `deploy`	Fait **échouer le déploiement** (job rouge) si `/health` ne répond pas	Surveiller **après** le déploiement

- `GET /health` de TaskFlow : `200 {"status":"ok","db":"up",...}` ou `503` si la base est injoignable
- Il manque un élément : **quelqu'un qui regarde en permanence et prévient** → Uptime Kuma

Uptime Kuma — Démo `demos/demo-uptime-kuma/`

```
cd demos/demo-uptime-kuma
docker compose up -d                # Uptime Kuma (port 3001) + une « cible » qui imite /health
```

1. Ouvrir `http://localhost:3001`, créer le compte administrateur
2. *Add New Monitor* → type **HTTP(s) - Keyword**, intervalle 60 s, mot-clé `ok`
3. Surveiller la cible de démo, puis votre production : `https://host.docker.internal/health` avec *Ignore TLS/SSL error* (certificat auto-signé)
4. `docker compose stop cible` → la sonde passe au **rouge** ; on montre les **notifications** (e-mail, Discord, Telegram…)
5. `docker compose start cible` → **vert**, avec la durée de l'indisponibilité et le pourcentage de disponibilité

> *Pour aller plus loin : **Prometheus** (collecte de métriques chiffrées) + **Grafana** (tableaux de bord et alertes).*

Gérer les erreurs proprement

Code HTTP	Signification	Exemple TaskFlow
`400`	Requête invalide (faute du client)	`POST /api/tasks` avec un titre vide
`404`	Ressource introuvable	`DELETE /api/tasks/999`
`500`	Erreur interne (bug côté serveur)	Exception non prévue dans une route
`502`	Le proxy n'obtient pas de réponse valide du service derrière	`api` arrêtée ou en boucle de redémarrage
`503`	Service temporairement indisponible	`/health` quand la base ne répond pas

```
app.use((err, _req, res, _next) => {                          // gestionnaire d'erreurs Express
  log({ level: 'error', message: err.message });              // le détail : dans les logs
  return res.status(500).json({ error: 'Erreur interne du serveur' });   // l'utilisateur : message neutre
});
```

> *Ne **jamais** renvoyer la stack trace (pile d'appels) au client : elle révèle chemins, versions et bibliothèques à un attaquant.*

Méthode de diagnostic : du symptôme vers la cause

```
1. Symptôme          Que voit l'utilisateur ? code HTTP, message, page blanche
       |             curl -sk https://taskflow.localhost/health
       ▼
2. Proxy Nginx       docker compose -f compose.prod.yaml logs --tail 50 proxy
       |             502 = « j'ai transmis, mais derrière, ça ne répond pas »
       ▼
3. État des services docker compose -f compose.prod.yaml ps
       |             qui est unhealthy ? Restarting ? Exited ?
       ▼
4. Logs du suspect   docker compose -f compose.prod.yaml logs --since 15m api
       |             chercher "level":"error" ou "fatal"
       ▼
5. Cause racine      configuration (.env) ? code (dernier déploiement) ? base ? disque ?
                     → corriger, redéployer, vérifier /health et ps, écrire le post-mortem
```

> *Une hypothèse à la fois, une vérification à la fois. Noter l'heure et ce qu'on a regardé : ce sera la chronologie du post-mortem.*

Checklist de mise en production

- [ ] La CI est **verte** sur le commit déployé ; la PR a été **relue**
- [ ] L'image est identifiée par un **tag précis** (SHA), affiché par `/health`
- [ ] `.env` présent sur le serveur, `chmod 600`, **absent de Git**, comparé à `.env.example`
- [ ] Mots de passe longs et aléatoires (`openssl rand -hex 24`), différents du dev
- [ ] Seul le reverse proxy publie des ports ; HTTP redirige vers HTTPS
- [ ] `nginx -t` OK avant tout rechargement
- [ ] Tous les services `healthy` (`docker compose ps`) ; smoke test `/health` vert
- [ ] Une sauvegarde existe **et** a été restaurée au moins une fois
- [ ] La procédure de **rollback** est écrite et a été testée
- [ ] La supervision (Uptime Kuma) surveille `/health` et sait prévenir quelqu'un
- [ ] Documentation à jour : `README.md`, `docs/ARCHITECTURE.md`, `docs/DEPLOIEMENT.md` ; petites PR

> *À recopier dans votre `docs/DEPLOIEMENT.md`.*

Le post-mortem sans blâme

Objectif : que l'incident ne se reproduise pas — **pas** trouver un coupable. On parle de **processus**, pas de personnes.

Rubrique	Contenu
Résumé	2-3 phrases : ce qui s'est passé, impact pour les utilisateurs
Chronologie	Heure par heure : déclencheur, premier signal, diagnostic, correction, retour à la normale
Cause racine	Les « 5 pourquoi » : demander « pourquoi ? » jusqu'à une cause qu'on peut corriger
Ce qui a bien / moins bien marché	Ce qui a aidé, ce qui a ralenti
Actions	Tableau : action, responsable, échéance — **concrètes et vérifiables**

> *« Pierre a mal tapé la variable » → blâme, rien ne change. « Le `.env` est modifié à la main sans vérification » → action : comparer automatiquement avec `.env.example`.*

Modèle : `docs/INCIDENTS.md` (dans la solution du fil rouge)

TP fil rouge — Étape 5 : analyse d'un incident (25 min)

Énoncé : `tp/fil-rouge-taskflow/etape5-incident.md`

Le scénario : juste après un merge, le site affiche « 502 Bad Gateway ». Des logs vous sont fournis, et l'incident est **reproductible** sur votre production locale.

1. **Reproduire** l'incident en suivant l'énoncé
2. **Diagnostiquer** avec la méthode : symptôme → logs du proxy → `compose ps` → logs de l'api → cause
3. **Corriger**, redéployer, vérifier : `/health` et `ps` au vert
4. **Rédiger** le post-mortem dans `docs/INCIDENTS.md` : chronologie, cause racine (5 pourquoi), **au moins deux actions**

> *Notez chaque commande tapée et ce qu'elle vous a appris : c'est votre chronologie. Et c'est un excellent exemple à raconter en mise en situation.*

Récap du jour 4

- **Serveur** : VPS, PaaS, cloud ; SSH par **clé**, utilisateur non-root, pare-feu. Dans le TP, le poste joue le serveur : mêmes gestes
- **Production** : `compose.prod.yaml` (images ghcr.io, `.env` hors Git), un **reverse proxy** Nginx seul exposé, HTTP → HTTPS
- **HTTPS** : certificat signé par une autorité (Let's Encrypt en vrai, auto-signé ici) ; **sauvegarde** `pg_dump` restaurée
- **Déploiement continu** : runner self-hosted → copie → `deploy.sh <sha>` → smoke test ; rollback par tag
- **Supervision** : logs JSON, healthchecks, Uptime Kuma ; méthode de diagnostic ; post-mortem sans blâme

Quiz du jour 4

1. Dans `compose.prod.yaml`, pourquoi seul le service `proxy` publie-t-il des ports ?
2. Que fait la ligne `return 301 https://$host$request_uri;` ?
3. Pourquoi utiliser un runner **self-hosted** dans notre TP, et quel risque pose le montage du socket Docker ?
4. Le dernier déploiement a cassé la production. Comment revenir à la version précédente ?
5. Le site renvoie « 502 Bad Gateway ». Dans quel ordre menez-vous l'enquête ?

Quiz du jour 4 — Réponses

1. Un **seul point d'entrée** : le proxy gère HTTPS ; front, api et db restent **inaccessibles** de l'extérieur (surface d'attaque réduite)
2. Elle **redirige définitivement** (301) toute requête HTTP vers la même adresse en **HTTPS**
3. Les runners de GitHub ne peuvent pas joindre notre poste ; le runner self-hosted **va chercher** les jobs. Le socket Docker donne l'équivalent des **droits root** sur le poste : sur un dépôt public, approbation obligatoire des workflows venant de *forks*
4. **Rollback par tag** : *Run workflow* de `release.yml` avec le SHA précédent (ou `./deploy.sh <ancien-sha>`) ; restaurer une sauvegarde si les données ont été touchées
5. Symptôme (`curl /health`) → logs du **proxy** → `docker compose ps` → logs de l'**api** → cause racine → correction → vérification → post-mortem

Glossaire du jour 4 (1/2)

Terme	Définition
VPS	*Virtual Private Server* : serveur virtuel loué, que vous administrez entièrement
PaaS	*Platform as a Service* : plateforme qui héberge votre code sans que vous gériez le serveur
Cloud	Infrastructure louée à la demande chez un grand fournisseur (AWS, Azure, Google Cloud)
SSH	*Secure Shell* : connexion chiffrée à une machine distante, idéalement par clé
Clé publique / privée	Le cadenas qu'on dépose sur le serveur / la clé qui ne quitte jamais votre poste
Reverse proxy	Point d'entrée unique qui reçoit les requêtes et les transmet aux services internes
Nginx	Serveur web très utilisé comme reverse proxy
HTTP / HTTPS	Protocole du web / sa version chiffrée par TLS
TLS	*Transport Layer Security* : chiffrement et authentification des échanges
Certificat / CA	Carte d'identité numérique d'un site / autorité de certification qui la signe
Let's Encrypt / certbot	Autorité gratuite / outil qui obtient et renouvelle ses certificats

Glossaire du jour 4 (2/2)

Terme	Définition
`pg_dump`	Outil de PostgreSQL qui exporte une base en fichier SQL restaurable
Rotation	Suppression automatique des sauvegardes les plus anciennes
Runner self-hosted	Machine à vous qui exécute les jobs GitHub Actions
Socket Docker	Fichier `/var/run/docker.sock` qui permet de piloter Docker (équivaut aux droits root)
Environment GitHub	Cible de déploiement (`production`) avec historique, secrets et approbations
Smoke test	Vérification rapide qu'une application déployée répond (`/health`)
Rollback	Retour à une version précédente (ici : redéployer un ancien tag)
Push / pull	Le pipeline pousse le déploiement / le serveur tire les nouvelles images (Watchtower)
Uptime Kuma	Outil de surveillance qui vérifie une URL et alerte en cas de panne
Post-mortem sans blâme	Analyse écrite d'un incident, centrée sur les processus et non sur les personnes

Évaluation (1/3) — Le QCM formatif (16h00, 15 min)

- **20 questions**, une seule bonne réponse parmi quatre, sans document
- Couvre les 6 séquences : DevOps, Git, Docker, GitHub Actions, déploiement, supervision
- **Formatif** : il ne compte pas dans la validation, il vous situe
- Correction express collective à la fin de la journée

Sujet : `evaluation/qcm.md`

> *Les quiz de fin de journée des slides sont le meilleur entraînement : relisez-les pendant la pause.*

Évaluation (2/3) — Présenter votre pipeline

5 min de présentation + 3 min de questions, comme une revue de fin de sprint ou un entretien d'embauche.

Minute	Contenu attendu
0:00 – 0:45	**Le problème** et la vue d'ensemble (schéma de `docs/ARCHITECTURE.md`)
0:45 – 1:45	**Conteneurs** : un point fort par Dockerfile, `compose.yaml` (volume, healthcheck)
1:45 – 4:15	**Démo live** : modification → PR → CI verte → merge → build & push → deploy → smoke test → la production a changé
4:15 – 4:45	**Exploitation** : logs, `/health`, Uptime Kuma, sauvegarde, rollback, l'incident de l'étape 5
4:45 – 5:00	**Bilan** : ce que vous amélioreriez, et pourquoi

- **Lancez la PR au tout début** de la démo : le CI tourne pendant que vous commentez
- **Plan B** : captures d'écran prêtes (PR verte, run de `release.yml`, smoke test, site en ligne)

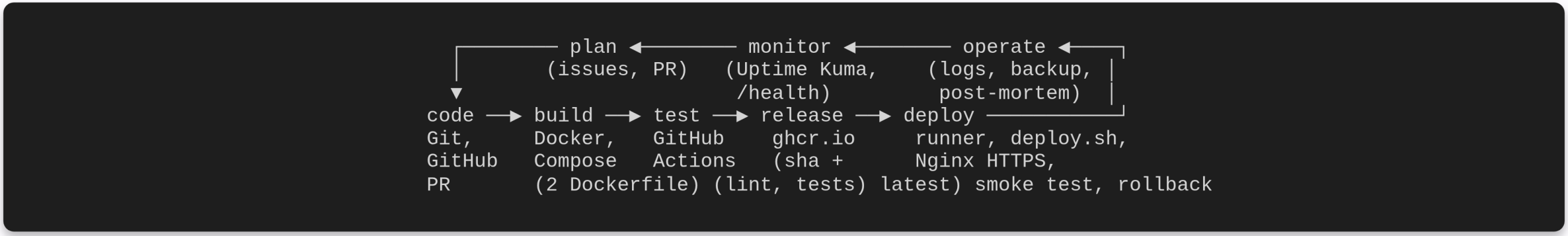
Consignes complètes : `evaluation/mise-en-situation.md`

Évaluation (3/3) — Ce qui est évalué

Critère	Points
Dockerfile fonctionnel (api + front, bonnes pratiques)	15
Fichier Docker Compose (db + api + front, volume, healthcheck, `.env`)	15
Dépôt GitHub configuré (branche protégée, PR revues, commits propres, tag)	15
Workflow GitHub Actions (lint, tests, build, push d'image)	20
Déploiement automatique fonctionnel (HTTPS, smoke test)	15
Documentation technique (architecture + procédure de déploiement)	10
Présentation du pipeline et justification des choix	10

- **Validation** : au moins **50/100** et chacun des **5 premiers critères** au moins partiellement atteint
- Une brique manque ? Présentez ce qui fonctionne et expliquez **honnêtement** comment vous la termineriez
- Grille détaillée : `evaluation/grille-evaluation.md`
- Votre pipeline est une **preuve** pour le dossier professionnel DWWM (compétence « Préparer et exécuter le déploiement d'une application web »)

Ce que vous avez construit en 4 jours



- Une application qui ne tournait que « sur votre machine » est désormais **conteneurisée**, **vérifiée** à chaque PR, **publiée**, **déployée automatiquement** et **supervisée**
- Vous avez fait **à la main**, puis **automatisé** : vous savez ce que fait chaque étape du pipeline

Pour aller plus loin

Sujet	Pour quoi faire	Par où commencer
Kubernetes	Orchestrer des dizaines de conteneurs sur plusieurs machines	k3d ou minikube sur votre poste
Terraform	Créer les serveurs eux-mêmes par du code (*Infrastructure as Code*)	Créer un VPS chez un hébergeur en quelques lignes
Prometheus + Grafana	Métriques chiffrées, tableaux de bord, alertes	Exposer `/metrics` dans l'API TaskFlow
GitLab CI	Le même pipeline sur l'autre grande plateforme	Traduire `ci.yml` en `.gitlab-ci.yml`
PaaS (Render, Railway, Fly.io)	Mettre TaskFlow en ligne pour de vrai, gratuitement	Déployer l'image ghcr.io
Un vrai VPS	Rejouer le jour 4 avec SSH, Let's Encrypt, pare-feu	`docs/DEPLOIEMENT.md` + démo `demos/demo-ssh-serveur/`

Merci et bonne continuation !

Ressources

`tp/fil-rouge-taskflow/etape4-deploiement.md` · `tp/fil-rouge-taskflow/etape5-incident.md`
`exercices/jour4/` · `ressources/lab/runner/` · `ressources/cheatsheet-linux-ssh-nginx.md`
`evaluation/` · docs.docker.com · docs.github.com/actions · nginx.org/en/docs