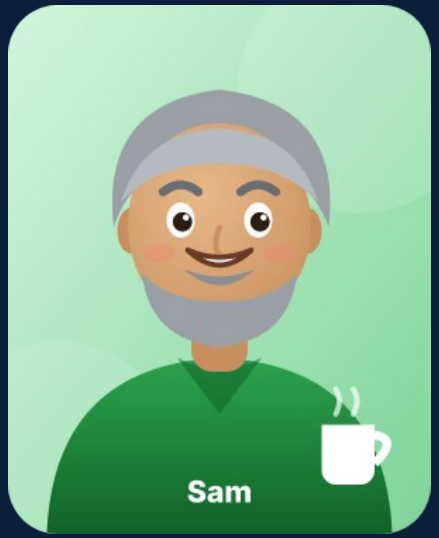
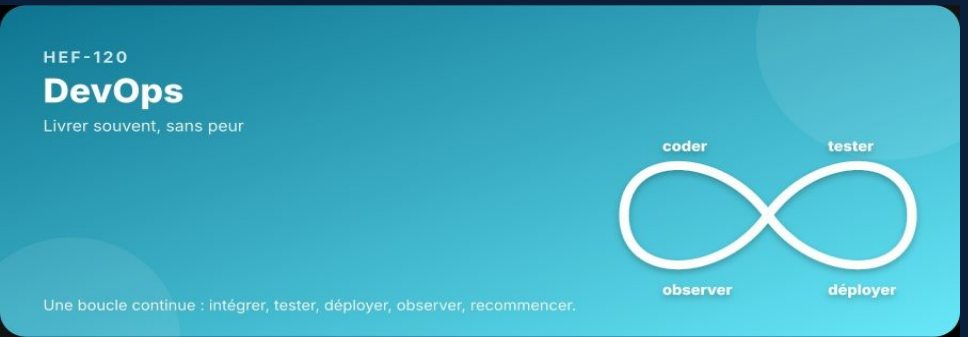
HEF-120 · MISSION 07 / 10

CD et environnements

Livrer en continu, ce n'est pas déployer à l'aveugle : c'est déployer progressivement, avec des garde-fous et la possibilité de revenir en arrière.





Badge : Pilote de déploiement

105 min · intermédiaire

L'histoire

Vendredi 16 h : HashBox remplace d'un coup toutes les instances par la nouvelle version. Une erreur de configuration passe inaperçue en test. Le site est coupé pour tout le monde pendant 40 minutes, le temps de revenir à l'ancienne version… dont la base de données a déjà été modifiée pour la nouvelle.



> *« Le problème n'est pas d'avoir déployé, dit Maya. C'est d'avoir déployé tout, en une fois, sans filet. Une bonne livraison avance par petits pas, mesure, et sait reculer. »*

Tu apprends les environnements et la promotion d'un même artefact, les stratégies de déploiement (rolling, blue/green, canary), les drapeaux de fonctionnalité et les migrations de base de données compatibles.

Ta mission

À la fin, je serai capable de…

1. distinguer livraison continue et déploiement continu
2. promouvoir un même artefact immuable de dev vers staging puis production
3. comparer rolling, blue/green et canary et simuler leurs garanties
4. décider d'une promotion de canary avec un test statistique simple
5. faire évoluer une base de données sans interruption (expand/contract)

Défi d'échauffement

Défi d'échauffement

Un théâtre change le décor pendant que la salle est pleine. Il peut : fermer le rideau et tout changer d'un coup, changer un élément à la fois pendant que le spectacle continue, ou préparer un second décor derrière le rideau et basculer en une seconde. Compare trois approches et leurs risques si le nouveau décor est défectueux.

Le concept

Livraison continue ou déploiement continu ?

	Livraison continue	Déploiement continu
Idée	chaque changement validé est **prêt** à être déployé	chaque changement validé est **déployé automatiquement**
Décision de déployer	humaine (bouton)	automatique
Prérequis	pipeline fiable	pipeline + tests + surveillance + retour arrière automatiques

Environnements

Environnement	Rôle
Développement	essais des développeurs

Le concept

Environnements (suite)

Environnement	Rôle
Test / intégration	tests automatisés partagés
Pré-production (staging)	réplique de la production pour la validation finale
Production	utilisateurs réels

Principe : **un seul artefact** construit une fois (mission 3) est **promu** de dev à production. Seule la **configuration** change (mission 5). On ne reconstruit jamais pour la production. L'environnement de staging doit **ressembler** à la production (mêmes versions, même topologie), sinon il rassure à tort.

Les stratégies de déploiement

Stratégie	Principe	Avantage	Risque / coût
Recreate	arrêter l'ancienne, démarrer la nouvelle	simple	**interruption**

Le concept

Les stratégies de déploiement (suite)

Stratégie	Principe	Avantage	Risque / coût
Rolling	remplacer les instances progressivement	pas d'interruption, peu de surcoût	deux versions coexistent un temps
Blue/green	deux environnements complets ; on bascule le trafic d'un coup	retour arrière instantané	double l'infrastructure
Canary	une petite part du trafic (1 %, 5 %, 25 %…) va vers la nouvelle version, on mesure, puis on étend	détecte les problèmes avec un impact limité	demande de bonnes métriques
Feature flag	le code est déployé mais la fonctionnalité est éteinte, activée progressivement	découple déploiement et mise en service	dette si les drapeaux restent

Le concept

Déploiement progressif (rolling)

Paramètres typiques : **surtension** (*max surge*, instances en plus pendant la mise à jour) et **indisponibilité maximale** (*max unavailable*). L'orchestrateur n'arrête une ancienne instance que lorsqu'une nouvelle est **prête** (test de disponibilité réussi).

Canary : décider avec des chiffres

On compare la **nouvelle** version (canary) et l'**ancienne** (référence) sur les mêmes métriques : taux d'erreur, latence. Un **test de proportions** (z) indique si la différence est plausiblement due au hasard :

```
p1 = erreurs_canary / n1 ; p2 = erreurs_reference / n2
p  = (e1 + e2) / (n1 + n2)
z  = (p1 − p2) / sqrt( p (1 − p) (1/n1 + 1/n2) )
```

Le concept

Canary : décider avec des chiffres (suite)

Décisions : trop peu de données → **continuer à observer** ; z > 1,96 avec canary pire → **retour arrière** ; sinon → **promouvoir** à l'étape suivante.

Drapeaux de fonctionnalité

Un **drapeau** active une fonction pour une fraction d'utilisateurs. Pour qu'un utilisateur ait une expérience **stable** et que la progression soit **monotone** (monter de 10 % à 20 % n'éteint personne), on utilise un **haché stable** de `(drapeau, utilisateur)`.

Les bases de données : expand / contract

Pendant un déploiement progressif, **deux versions du code** tournent sur **la même base**. Une migration doit donc être compatible avec les deux. Méthode **expand/contract** pour renommer une colonne :

1. **Expand** : ajouter la nouvelle colonne (nullable) ;

Le concept

Les bases de données : expand / contract (suite)

2. **Double écriture** : le code écrit dans les deux ; **remplir** (backfill) les anciennes lignes ;
3. **Basculer la lecture** vers la nouvelle colonne ;
4. **Contract** : une fois sûr, supprimer l'ancienne colonne (déploiement ultérieur).

Aucune étape ne casse le code de la version précédente ni celle d'après : **retour arrière toujours possible**.

Approbations et garde-fous

- Validations automatiques : tests de fumée, métriques, vérifications de sécurité.
- Approbation humaine pour les changements sensibles.
- **Fenêtres de gel** (pas de déploiement la veille d'un pic ou d'un week-end) — politique, pas dogme : mieux vaut déployer petit et souvent.
- **Traçabilité** : qui a déployé quoi, quand, avec quel artefact.

Le concept

Approbations et garde-fous (suite)

🧠 ***Astuce*** *: sépare* ***déployer*** *(mettre le code en production) de* ***mettre en service*** *(l'activer pour les utilisateurs). Les drapeaux rendent cette séparation naturelle, et rendent le retour arrière aussi simple qu'un interrupteur.*

Exemple pas à pas

Problème : simuler un déploiement progressif avec disponibilité minimale garantie, une décision de canary par test statistique, et des drapeaux de fonctionnalité stables et monotones.

```
import hashlib, math, random

# ---------------- rolling update ----------------
def rolling(versions, nouvelle, disponibilite_min, sain):
    # versions : liste des versions des instances ; `sain(v)` dit si une instance
   de la version v démarre correctement
    instances = list(versions)
    journal, dispo_min_observee = [], len(instances)
    for i in range(len(instances)):
        if instances[i] == nouvelle:
            continue
        if not sain(nouvelle):                                  # la nouvelle inst
   ance ne devient pas prête : on arrête
            journal.append(f"échec : la nouvelle instance n'est pas saine, arrêt à
    {i} instance(s) mises à jour")
```

Exemple pas à pas (suite)

```
            return instances, journal, dispo_min_observee, False
        instances.append(nouvelle)                                  # surtension : on
    ajoute d'abord la nouvelle
        instances.pop(i)                                            # puis on retire l
    'ancienne
        dispo = len(instances)
        dispo_min_observee = min(dispo_min_observee, dispo)
        assert dispo >= disponibilite_min
        journal.append(f"instance {i} remplacée ({dispo} disponibles)")
        instances.insert(i, instances.pop())                        # garde l'ordre po
    ur la lisibilité
     return instances, journal, dispo_min_observee, True

etat, journal, dispo, ok = rolling(["v1"] * 4, "v2", disponibilite_min=4, sain=lam
    bda v: True)
print(journal[-1], "| versions finales :", set(etat), "| disponibilité minimale :"
    , dispo)
assert ok and set(etat) == {"v2"} and dispo >= 4
etat, journal, dispo, ok = rolling(["v1"] * 4, "v2", disponibilite_min=4, sain=lam
```

Exemple pas à pas (suite)

```
   bda v: v != "v2")
print(journal[-1])
assert not ok and set(etat) == {"v1"}                          # rien n'a été rem
   placé : la production est intacte

# ---------------- canary : décision statistique ----------------
def decision_canary(err_c, n_c, err_r, n_r, minimum=500):
    if n_c < minimum:
        return "continuer", None
    p1, p2 = err_c / n_c, err_r / n_r
    p = (err_c + err_r) / (n_c + n_r)
    denom = math.sqrt(p * (1 - p) * (1 / n_c + 1 / n_r)) if 0 < p < 1 else 0
    z = (p1 - p2) / denom if denom else 0.0
    if z > 1.96:
        return "retour arrière", z
    return "promouvoir", z

for nom, args in {"trop tôt": (1, 100, 20, 9900), "canary sain": (6, 1000, 55, 900
   0), "canary défectueux": (45, 1000, 55, 9000)}.items():
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    d, z = decision_canary(*args)
    print(f"{nom:18s} → {d}" + (f" (z = {z:.1f})" if z is not None else ""))
assert decision_canary(1, 100, 20, 9900)[0] == "continuer"
assert decision_canary(6, 1000, 55, 9000)[0] == "promouvoir"
assert decision_canary(45, 1000, 55, 9000)[0] == "retour arrière"

# ---------------- drapeaux : stables et monotones ----------------
def actif(drapeau, utilisateur, pourcentage):
    h = int(hashlib.sha256(f"{drapeau}:{utilisateur}".encode()).hexdigest()[:8], 1
   6)
    return (h % 10000) / 100 < pourcentage

utilisateurs = [f"u{i}" for i in range(20000)]
etapes = [1, 5, 10, 25, 50, 100]
precedent = set()
for pct in etapes:
    courant = {u for u in utilisateurs if actif("nouvelle-recherche", u, pct)}
    assert precedent <= courant                                    # monotone
    : monter ne coupe personne
```

Exemple pas à pas (suite)

```
    print(f"{pct:3d} % → {len(courant) / len(utilisateurs):6.1%} des utilisateurs"
   )
    assert abs(len(courant) / len(utilisateurs) - pct / 100) < 0.01      # réparti
   tion proche de la cible
    precedent = courant
assert actif("a", "u1", 30) == actif("a", "u1", 30)                      # stable
   : même résultat à chaque appel
assert {u for u in utilisateurs if actif("a", u, 10)} != {u for u in utilisateurs
   if actif("b", u, 10)}      # drapeaux indépendants
```

Lecture :

- **Rolling** : la disponibilité ne descend jamais sous le minimum car on ajoute avant de retirer. Si la nouvelle version n'est pas saine, le déploiement **s'arrête** et la production reste intacte.
- **Canary** : avec trop peu de données, on observe encore ; un taux d'erreur de 0,6 % contre 0,6 % → promotion ; 4,5 % contre 0,6 % → retour arrière ($z > 1{,}96$).

Exemple pas à pas (suite)

- **Drapeaux** : le haché de `(drapeau, utilisateur)` donne une répartition stable et **monotone** ; augmenter le pourcentage n'éteint personne. Chaque drapeau a ses propres groupes (indépendants).

À toi, avec aide

Simule un **blue/green** : deux environnements `bleu` (version 1) et `vert` (version 2) ; un routeur dirige 100 % du trafic vers l'un d'eux. Écris :

1. `Routeur` avec `basculer(vers)` qui n'accepte la bascule que si l'environnement cible a passé son test de santé ;
2. une simulation de 1000 requêtes pendant lesquelles on bascule à la requête 500, puis on **revient en arrière** à la requête 700 parce que le taux d'erreur de `vert` dépasse 5 % ;
3. vérifie qu'**aucune requête n'est perdue** (chaque requête reçoit une réponse) et affiche le nombre de requêtes servies par chaque environnement et le nombre d'erreurs vues par les utilisateurs.

Indices progressifs

Indice 1

Chaque environnement renvoie `ok` ou `erreur` selon une probabilité (graine fixée).

Indice 2

Après la bascule, mesure le taux d'erreur sur une fenêtre de 50 requêtes.

Indice 3

Le retour arrière est une simple bascule inverse.

Réponse

```
import random

class Environnement:
    def __init__(self, nom, version, taux_erreur, sain=True):
        self.nom, self.version, self.taux_erreur, self.sain, self.servies, self.er
   reurs = nom, version, taux_erreur, sain, 0, 0
    def servir(self, rng):
        self.servies += 1
        if rng.random() < self.taux_erreur:
            self.erreurs += 1
            return "erreur"
        return "ok"

class Routeur:
    def __init__(self, actif): self.actif = actif
    def basculer(self, vers):
        if not vers.sain:
            raise RuntimeError(f"{vers.nom} n'a pas passé son test de santé")
        self.actif = vers
```

À toi, avec aide · réponse

Réponse (suite)

```
rng = random.Random(4)
bleu, vert = Environnement("bleu", "v1", 0.002), Environnement("vert", "v2", 0.12)
          # la v2 a un bug caché
routeur = Routeur(bleu)
reponses, fenetre, retour = 0, [], None
for i in range(1000):
    if i == 500:
        routeur.basculer(vert)
    res = routeur.actif.servir(rng)
    reponses += 1
    if routeur.actif is vert:
        fenetre = (fenetre + [res == "erreur"])[-50:]
        if len(fenetre) == 50 and sum(fenetre) / 50 > 0.05 and retour is None:
            routeur.basculer(bleu); retour = i
print(f"bascule à 500, retour arrière à {retour} | bleu {bleu.servies} req / {bleu
   .erreurs} err | vert {vert.servies} req / {vert.erreurs} err")
assert reponses == 1000 and bleu.servies + vert.servies == 1000
assert retour is not None and 500 < retour < 700 and vert.erreurs > 0 and routeur.
```

À toi, avec aide · réponse

Réponse (suite)

```
actif is bleu
```

Le retour arrière est **instantané** car l'ancien environnement est resté intact et prêt. Le coût : double infrastructure pendant la transition. Les erreurs vues par les utilisateurs se limitent à la fenêtre de détection.

À toi, en solo

Migration expand/contract en conditions réelles. Avec `sqlite3`, renomme la colonne `nom` de la table `membres` en `nom_complet` **sans interruption**, alors que deux versions de l'application fonctionnent en même temps :

1. écris `AppV1` (lit/écrit `nom`) et `AppV2` (lit `nom_complet`, écrit les deux pendant la transition) ;
2. enchaîne les phases : (a) expand : ajout de `nom_complet` ; (b) AppV2 déployée avec double écriture ; (c) backfill des anciennes lignes ; (d) AppV2 lit `nom_complet` seul ; (e) contract : suppression de `nom` ;
3. **après chaque phase**, vérifie que V1 et V2 fonctionnent encore (jusqu'à la phase où V1 est retirée) ;
4. montre qu'un renommage direct (`ALTER ... RENAME COLUMN`) casse AppV1 pendant le déploiement.

À toi, en solo · réponse

Pistes de correction

```
import sqlite3

class AppV1:
    def ajouter(self, bd, nom): bd.execute("INSERT INTO membres (nom) VALUES (?)",
    (nom,))
    def lister(self, bd): return sorted(r[0] for r in bd.execute("SELECT nom FROM
   membres"))

class AppV2:
    def __init__(self, double_ecriture=True, lecture_nouvelle=False):
        self.double, self.lecture_nouvelle = double_ecriture, lecture_nouvelle
    def ajouter(self, bd, nom):
        if self.double:
            bd.execute("INSERT INTO membres (nom, nom_complet) VALUES (?, ?)", (no
   m, nom))
        else:
            bd.execute("INSERT INTO membres (nom_complet) VALUES (?)", (nom,))
    def lister(self, bd):
        col = "nom_complet" if self.lecture_nouvelle else "nom"
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        return sorted(r[0] for r in bd.execute(f"SELECT {col} FROM membres"))

bd = sqlite3.connect(":memory:")
bd.execute("CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT NOT NULL)")
v1, anciens = AppV1(), ["ada", "bob"]
for n in anciens: v1.ajouter(bd, n)

# (a) EXPAND : colonne nullable, V1 continue de fonctionner
bd.execute("ALTER TABLE membres ADD COLUMN nom_complet TEXT")
v1.ajouter(bd, "cleo"); assert v1.lister(bd) == ["ada", "bob", "cleo"]

# (b) V2 en double écriture : V1 et V2 coexistent
v2 = AppV2(double_ecriture=True, lecture_nouvelle=False)
v2.ajouter(bd, "dan")
assert v1.lister(bd) == v2.lister(bd) == ["ada", "bob", "cleo", "dan"]

# (c) BACKFILL des anciennes lignes
bd.execute("UPDATE membres SET nom_complet = nom WHERE nom_complet IS NULL")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
# (d) V2 lit la nouvelle colonne ; V1 fonctionne toujours (retour arrière encore p
   ossible)
v2.lecture_nouvelle = True
assert v2.lister(bd) == ["ada", "bob", "cleo", "dan"] and v1.lister(bd) == ["ada",
    "bob", "cleo", "dan"]

# (e) CONTRACT, une fois V1 retirée : V2 n'écrit plus l'ancienne colonne
v2.double = False
bd.execute("ALTER TABLE membres DROP COLUMN nom")
v2.ajouter(bd, "eve")
assert v2.lister(bd) == ["ada", "bob", "cleo", "dan", "eve"]

# contre-exemple : un renommage direct casse V1 pendant le déploiement
bd2 = sqlite3.connect(":memory:")
bd2.execute("CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT NOT NULL)")
v1.ajouter(bd2, "ada")
bd2.execute("ALTER TABLE membres RENAME COLUMN nom TO nom_complet")
try:
    v1.lister(bd2); raise AssertionError("V1 aurait dû échouer")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
except sqlite3.OperationalError as e:
    print("renommage direct → V1 cassée :", e)
print("expand/contract : aucune phase n'a cassé V1 ni V2")
```

Chaque phase est **compatible avec la version précédente et la suivante** du code. C'est ce qui permet le déploiement progressif et le retour arrière sans corruption.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Tout remplacer d'un coup	Une erreur touche tous les utilisateurs	Rolling, canary, blue/green
Staging différent de la production	Il rassure à tort	Mêmes versions, topologie et données représentatives
Migration de base incompatible avec l'ancienne version	Retour arrière impossible	Expand/contract en plusieurs déploiements
Drapeaux jamais retirés	Code mort et combinaisons ingérables	Date de retrait et propriétaire pour chaque drapeau

Mini-boss

Mini-boss

Contrôleur de promotion de HashBox. Écris un simulateur de livraison qui promeut un artefact `hashbox-1.7.0` (haché fixe) à travers `dev → staging → production` :

1. à chaque étape, vérifie que l'artefact déployé a le **même haché** que celui construit ;
2. staging : tests de fumée, puis approbation (simulée) ;
3. production : canary à 5 %, 25 %, 100 % avec la décision statistique ; une version défectueuse est détectée et le **retour arrière automatique** est exécuté ;
4. drapeau de fonctionnalité activé progressivement après le déploiement ;
5. journal d'audit : qui, quoi, quand, haché ;
6. 3 scénarios : version saine, version défectueuse (erreurs), artefact altéré (haché différent → refus).

Mini-boss · réponse

Pistes de correction

Un bon travail : classes `Environnement`, `Artefact(nom, haché)`, `ControleurPromotion` avec `promouvoir(cible)`; refus si `haché_déployé != haché_construit`; canary par étapes avec `decision_canary`; retour arrière automatique rétablissant la version précédente et créant une entrée d'audit; drapeau activé après la promotion complète. Les trois scénarios doivent produire respectivement : promotion complète, retour arrière à 5 % ou 25 %, refus immédiat avec message sur le haché.

Quiz éclair

Quiz éclair

1. Un déploiement canary consiste à :

A. envoyer une petite part du trafic vers la nouvelle version, mesurer, puis étendre
B. remplacer tout d'un coup
C. déployer seulement la nuit

2. Pour pouvoir revenir en arrière après une migration de base, on utilise :

A. un renommage direct de colonne
B. la méthode expand/contract compatible avec les deux versions
C. la suppression de la base

3. Un artefact promu de staging à la production doit être :

A. reconstruit pour la production

Quiz éclair

Quiz éclair (suite)

B. modifié à la main
C. le même, construit une seule fois

4. 🔁 *Révision mission 06* : Un pipeline CI est modélisé comme :

A. un graphe orienté acyclique de jobs
B. une liste sans dépendances
C. une boucle infinie

5. 🔁 *Révision mission 06* : Si un job échoue, ses dépendants sont :

A. exécutés quand même
B. ignorés, tandis que les jobs indépendants continuent
C. supprimés du dépôt

Quiz éclair · réponse

Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

Je réfléchis

1. Quelle est la pire minute possible pour un déploiement chez toi, et que ferait-on si ça échouait à ce moment ?
2. Quelle fonctionnalité de ton projet gagnerait à être déployée derrière un drapeau ?

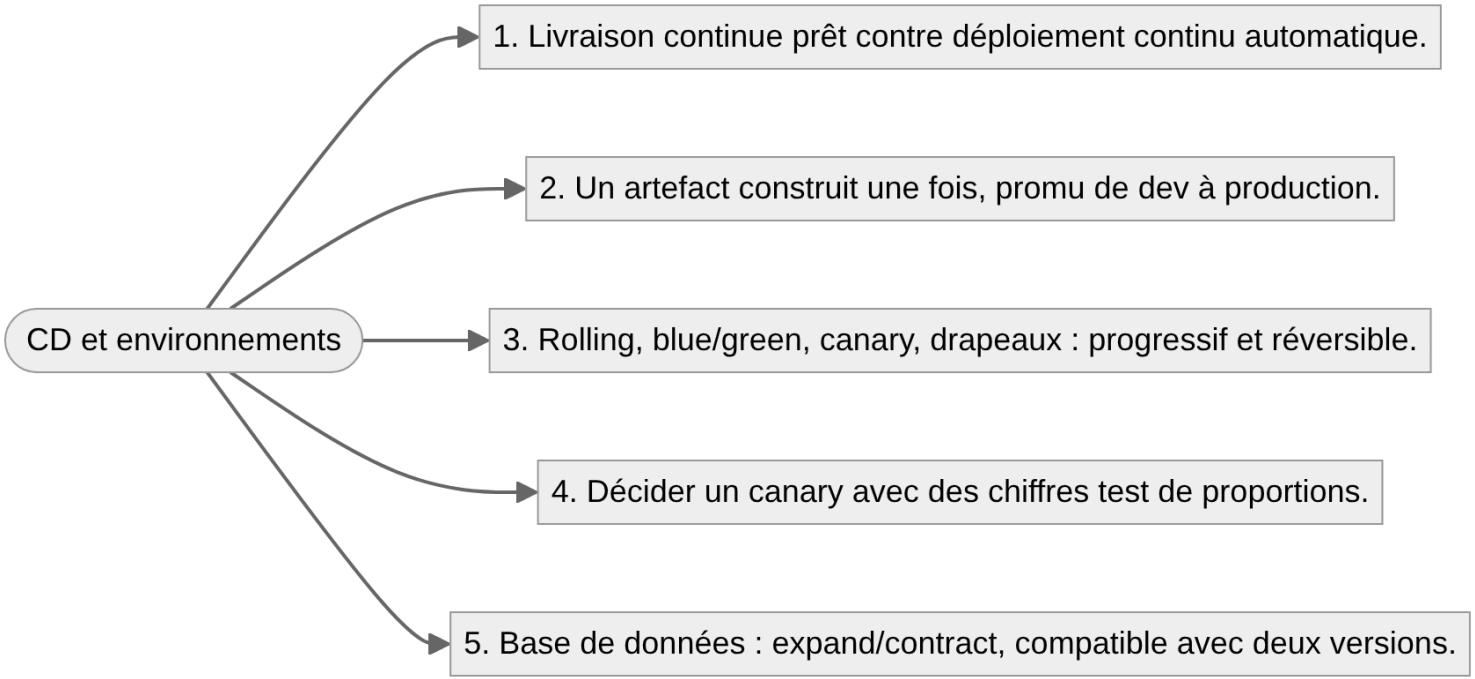
À retenir

À retenir

1. Livraison continue (prêt) contre déploiement continu (automatique).
2. Un artefact construit une fois, promu de dev à production.
3. Rolling, blue/green, canary, drapeaux : progressif et réversible.
4. Décider un canary avec des chiffres (test de proportions).
5. Base de données : expand/contract, compatible avec deux versions.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Pilote de déploiement

Prochaine mission : Observabilité

