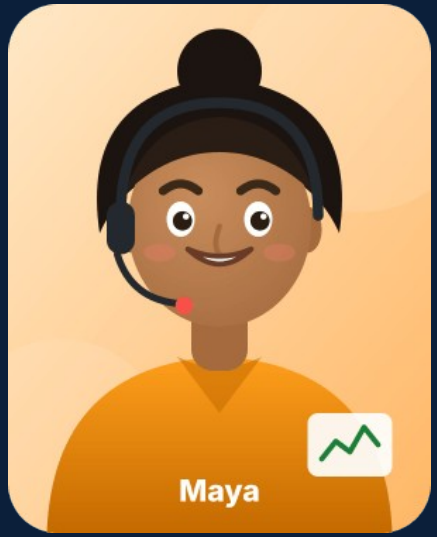
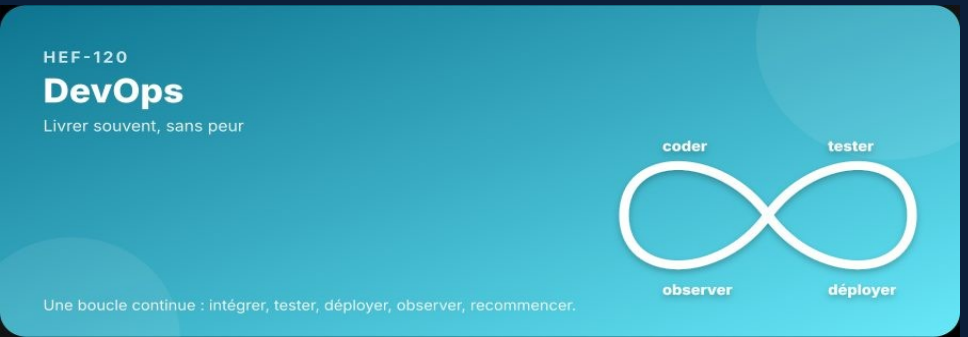
HEF-120 · MISSION 10 / 10

Pipeline complet

Assembler toutes les pièces : du commit à la production, avec tests, artefact, déploiements progressifs, observation et retour arrière automatique.





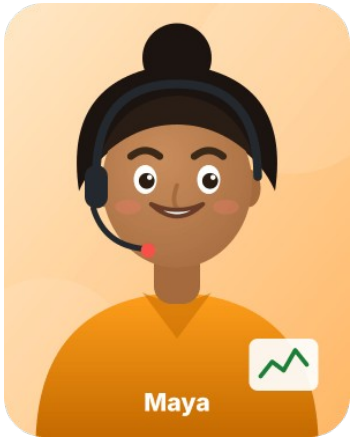
Badge : Architecte de la livraison

115 min · intermédiaire

L'histoire

Un mercredi matin, Kodi corrige un bug d'arrondi dans les remises. Il pousse son commit à 10 h 02. À 10 h 04, les tests sont verts. À 10 h 06, l'artefact est publié. À 10 h 09, il est en préproduction, où le test de fumée réussit. À 10 h 12, 5 % du trafic le reçoit ; à 10 h 25, tout le monde. Kodi n'a pas cliqué sur un seul bouton.

Le lendemain, un autre changement dégrade la latence. À la deuxième étape, le contrôle des métriques détecte la dérive : le déploiement recule seul, en deux minutes, avant que la plupart des clients ne remarquent quoi que ce soit.



« Voilà ce qu'on cherchait depuis le début, sourit Sam. Non pas aller vite en prenant des risques, mais aller vite parce que chaque étape est vérifiée et réversible. »

L'histoire

L'histoire (suite)

Dans cette dernière mission, tu assembles tout le module : Git, build reproductible, CI, configuration, déploiement progressif, observabilité, retour arrière — dans un pipeline cohérent et mesuré.

Ta mission

À la fin, je serai capable de…

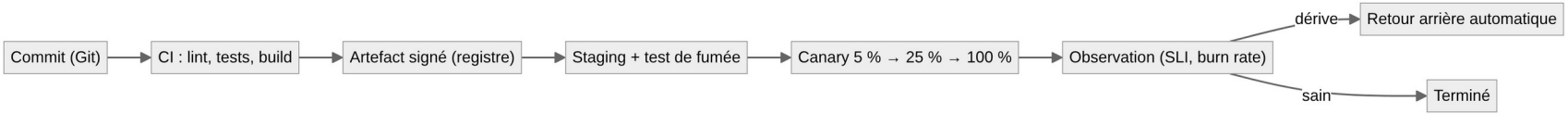
1. assembler les briques du module en un pipeline de bout en bout
2. appliquer des politiques de livraison sous forme de code (gel, approbations, empreintes)
3. automatiser la décision de promotion ou de retour arrière
4. mesurer le pipeline avec les indicateurs DORA
5. documenter le pipeline et ses procédures d'exploitation

Défi d'échauffement

Dessine sur une feuille la chaîne complète d'un changement dans ton idéal : de la touche « enregistrer » jusqu'à l'utilisateur. Marque 4 points de contrôle et pour chacun : ce qu'on vérifie, et ce qu'on fait si le contrôle échoue. Où se trouvent, selon toi, les endroits où l'on ne peut pas revenir en arrière ?

Le concept

La chaîne de bout en bout



Description du schéma : un commit déclenche la CI (style, tests, construction). L'artefact est publié dans un registre, déployé en préproduction avec un test de fumée, puis en production par étapes canary. L'observation décide : dérive → retour arrière automatique ; sain → fin du déploiement.

Les briques et leur rôle

Brique	Mission	Garantit
Git, PR, branche protégée	2	relecture, historique, déclencheur

Le concept

Les briques et leur rôle (suite)

Brique	Mission	Garantit
Build reproductible, verrouillage	3	même source → même artefact
Conteneurs	4	même environnement d'exécution
Configuration et secrets	5	un artefact, plusieurs environnements
CI (graphe de jobs)	6	vérification rapide et automatique
CD (canary, drapeaux, migrations)	7	livraison progressive et réversible
Observabilité (SLI, burn rate)	8	décision fondée sur des mesures
Résilience (retour arrière, chaos)	9	récupération rapide

Politique de livraison comme code

Les règles de l'équipe s'écrivent en **code vérifiable**, évaluées à chaque déploiement :

Le concept

Politique de livraison comme code (suite)

Règle	Exemple
Empreinte	l'artefact déployé = l'artefact testé
Approbation	deux relectures pour les changements sensibles
Gel	pas de déploiement hors horaires d'astreinte ou la veille d'un pic
Budget d'erreurs	pas de déploiement risqué si le budget est épuisé
Santé de l'environnement	pas de déploiement pendant un incident ouvert
Sécurité	aucune faille critique dans l'artefact (HEF-100)

Une politique **automatisée** est appliquée à tous de la même façon, sans négociation.

Le concept

Pipeline et indicateurs

Chaque exécution laisse une **trace** (événements horodatés) : on en déduit les indicateurs DORA (mission 1) **automatiquement**, plutôt que par déclaration.

Documenter pour exploiter

Document	Contenu
Architecture du pipeline	schéma, étapes, outils, propriétaires
Guide d'intervention (*runbook*)	par alerte : symptômes, vérifications, actions, escalade
Procédure de retour arrière	commandes, critères, qui décide
Registre des décisions	pourquoi telle stratégie, tel seuil
Plan d'amélioration	prochains goulots à traiter

Le concept

Maturité : où en es-tu ?

Niveau	Caractéristique
1	build et déploiement manuels
2	CI automatisée, déploiement manuel
3	livraison continue vers staging, production par bouton
4	déploiement progressif automatique avec observation et retour arrière
5	amélioration continue mesurée, exercices de panne réguliers

🧠 ***Astuce*** *: automatise d'abord ce qui te fait peur ou ce qui te prend le plus de temps, mesure, puis recommence. Un pipeline parfait n'existe pas ; un pipeline qui s'améliore, si.*

Exemple pas à pas

Problème : assembler un pipeline complet simulé — politique de livraison, build reproductible, CI, staging, canary avec décision statistique, observation, retour arrière automatique, et DORA calculés à partir du journal d'événements.

```
import hashlib, math, random
from datetime import datetime, timedelta

# ================ artefact et politique ================
def construire(sources):
    return hashlib.sha256("|".join(f"{k}={v}" for k, v in sorted(sources.items()))
   .encode()).hexdigest()[:12]

def politique(contexte):
    violations = []
    if contexte["empreinte_deployee"] != contexte["empreinte_testee"]: violations.
   append("l'artefact déployé n'est pas celui qui a été testé")
    if contexte["relecteurs"] < 1: violations.append("aucune relecture")
    if contexte["maintenant"].weekday() == 4 and contexte["maintenant"].hour >= 16
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    : violations.append("gel : pas de déploiement le vendredi après 16 h")
     if contexte["budget_restant"] <= 0: violations.append("budget d'erreurs épuisé
     : seuls les correctifs de fiabilité sont autorisés")
     if contexte["failles_critiques"] > 0: violations.append("faille critique prése
    nte")
     return violations

# ================= CI simulée =================
def ci(sources):
    etapes = [("lint", "def " in sources.get("app.py", "")), ("tests", sources.get
    ("bug", False) is False)]
    return all(ok for _, ok in etapes), etapes

# ================= production simulée =================
class Production:
    def __init__(self, graine=5):
        self.rng, self.version, self.t = random.Random(graine), "v1", 0
    def trafic(self, version, n, defectueuse):
        erreurs = sum(1 for _ in range(n) if self.rng.random() < (0.08 if (defectu
```

Exemple pas à pas (suite)

```
    euse and version == "v2") else 0.004))
        return erreurs, n

def decision(err_c, n_c, err_r, n_r):
    if n_c < 400: return "continuer"
    p1, p2, p = err_c / n_c, err_r / n_r, (err_c + err_r) / (n_c + n_r)
    z = (p1 - p2) / math.sqrt(p * (1 - p) * (1 / n_c + 1 / n_r)) if 0 < p < 1 else
    0
    return "retour" if z > 1.96 else "promouvoir"

# ================ le pipeline complet ================
def livrer(sources, defectueuse, journal, maintenant, contexte_extra=None):
    t0 = maintenant
    journal.append((t0, "commit"))
    reussi, _ = ci(sources)
    journal.append((t0 + timedelta(minutes=2), "ci_ok" if reussi else "ci_echec"))
    if not reussi:
        return "arrêté à la CI"
    empreinte = construire(sources)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    contexte = dict(empreinte_deployee=empreinte, empreinte_testee=empreinte, rele
cteurs=1, maintenant=t0, budget_restant=10, failles_critiques=0)
    contexte.update(contexte_extra or {})
    violations = politique(contexte)
    if violations:
        journal.append((t0 + timedelta(minutes=3), "refus_politique")); return "re
fusé : " + " ; ".join(violations)
    journal.append((t0 + timedelta(minutes=4), "staging_ok"))           # test de
    fumée réussi
    prod = Production()
    for etape, part in enumerate((0.05, 0.25, 1.0), 1):
        horodatage = t0 + timedelta(minutes=5 + 8 * etape)
        n_c = int(2000 * part)
        e_c, n_c = prod.trafic("v2", max(n_c, 1), defectueuse)
        e_r, n_r = prod.trafic("v1", 2000, defectueuse)
        d = decision(e_c, n_c, e_r, n_r)
        journal.append((horodatage, f"canary_{int(part * 100)}%_{d}"))
        if d == "retour":
            journal.append((horodatage + timedelta(minutes=1), "retour_arriere"));
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    return f"retour arrière à l'étape {int(part * 100)} %"
    journal.append((t0 + timedelta(minutes=30), "deploye"))
    return "déployé à 100 %"

journal = []
debut = datetime(2025, 6, 4, 10, 2)                                  # un mercre
   di
resultats = {
    "changement sain":       livrer({"app.py": "def f(): ..."}, False, journal, de
   but),
    "version défectueuse":   livrer({"app.py": "def f(): ..."}, True, journal, deb
   ut + timedelta(days=1)),
    "tests rouges":          livrer({"app.py": "def f(): ...", "bug": True}, False
   , journal, debut + timedelta(days=2)),
    "vendredi 17 h":         livrer({"app.py": "def f(): ..."}, False, journal, da
   tetime(2025, 6, 6, 17, 0)),
    "artefact altéré":       livrer({"app.py": "def f(): ..."}, False, journal, de
   but + timedelta(days=4), {"empreinte_deployee": "ffffffffffff"}),
}
```

Exemple pas à pas (suite)

```
for nom, res in resultats.items():
    print(f"{nom:20s} → {res}")
assert resultats["changement sain"] == "déployé à 100 %"
assert resultats["version défectueuse"].startswith("retour arrière")
assert resultats["tests rouges"] == "arrêté à la CI"
assert "gel" in resultats["vendredi 17 h"] and "pas celui" in resultats["artefact
   altéré"]

# ================ DORA calculés à partir du journal ================
deploiements = [t for t, e in journal if e == "deploye"]
retours = [t for t, e in journal if e == "retour_arriere"]
print(f"déploiements réussis : {len(deploiements)} | retours arrière : {len(retour
   s)} | événements journalisés : {len(journal)}")
assert len(deploiements) == 1 and len(retours) == 1
duree = (next(t for t, e in journal if e == "deploye") - debut).total_seconds() /
   60
print(f"délai commit → production du changement sain : {duree:.0f} min")
assert duree == 30
```

Exemple pas à pas (suite)

Lecture : cinq cas passent par le **même** pipeline :

1. **Changement sain** : CI → staging → canary 5 % → 25 % → 100 % : déployé en 30 minutes, sans intervention.
2. **Version défectueuse** : le contrôle statistique détecte la dérive à une étape de canary et déclenche le **retour arrière** automatique, avant que tout le trafic soit touché.
3. **Tests rouges** : arrêté à la CI, rien n'est construit ni déployé.
4. **Vendredi 17 h** : la politique de gel refuse le déploiement.
5. **Artefact altéré** : l'empreinte ne correspond pas à l'artefact testé → refus.

Chaque événement est journalisé, d'où les indicateurs DORA se calculent **automatiquement**.

À toi, avec aide

À partir du journal de l'exemple (liste de `(horodatage, événement)`), écris `dora_depuis_journal(journal)` qui calcule :

- le nombre de déploiements réussis (`deploye`) ;
- le nombre de retours arrière ;
- le **taux d'échec des changements** = retours arrière / (déploiements + retours) ;
- le délai moyen entre un `commit` et son `deploye` (pour les changements qui sont allés au bout) ;
- le temps moyen de rétablissement : entre l'événement `canary_..._retour` et le `retour_arriere` suivant.

Teste avec un journal construit à la main de 4 changements (2 réussis, 1 retour arrière, 1 arrêté en CI).

À toi, avec aide · réponse

Indices progressifs

Indice 1

Associe chaque `commit` à l'événement final qui le suit avant le commit suivant.

Indice 2

Parcours le journal dans l'ordre et conserve le dernier commit rencontré.

Indice 3

Le délai est la différence entre l'horodatage du `deploye` et celui du `commit`.

À toi, avec aide · réponse

Réponse

```
from datetime import datetime, timedelta

def dora_depuis_journal(journal):
    journal = sorted(journal)
    deployes, retours, delais, retablissements = 0, 0, [], []
    dernier_commit, detection = None, None
    for t, e in journal:
        if e == "commit":
            dernier_commit, detection = t, None
        elif e.startswith("canary_") and e.endswith("_retour"):
            detection = t
        elif e == "retour_arriere":
            retours += 1
            if detection is not None:
                retablissements.append((t - detection).total_seconds() / 60)
        elif e == "deploye":
            deployes += 1
            delais.append((t - dernier_commit).total_seconds() / 60)
    taux = retours / (deployes + retours) if deployes + retours else 0.0
```

À toi, avec aide · réponse

Réponse (suite)

```
    return {"deploye": deploye, "retours": retours, "taux_echec": taux,
            "delai_moyen_min": sum(delais) / len(delais) if delais else 0.0,
            "mttr_min": sum(retablissements) / len(retablissements) if retablissem
   ents else 0.0}

t = datetime(2025, 6, 2, 9, 0)
J = [(t, "commit"), (t + timedelta(minutes=30), "deploye"),
     (t + timedelta(hours=2), "commit"), (t + timedelta(hours=2, minutes=20), "can
   ary_25%_retour"), (t + timedelta(hours=2, minutes=21), "retour_arriere"),
     (t + timedelta(hours=4), "commit"), (t + timedelta(hours=4, minutes=26), "dep
   loye"),
     (t + timedelta(hours=6), "commit"), (t + timedelta(hours=6, minutes=2), "ci_e
   chec")]
m = dora_depuis_journal(J)
print(m)
assert m["deploye"] == 2 and m["retours"] == 1 and round(m["taux_echec"], 3) == 0
   .333
assert m["delai_moyen_min"] == 28 and m["mttr_min"] == 1
```

Réponse (suite)

Le délai moyen est de 28 minutes (30 et 26 min), le temps de rétablissement de **1 minute** grâce à l'automatisation, le taux d'échec de 1/3 sur ce petit échantillon.

À toi, en solo

Politique de livraison comme code, extensible. Écris un moteur de politiques :

1. chaque règle est une fonction `(contexte) -> None | str` (message de violation) enregistrée avec un nom, une gravité (`bloquant` / `avertissement`) et une **exception** possible (dispense signée avec date d'expiration) ;
2. au moins 8 règles : empreinte, relectures (2 pour un changement touchant `securite/`), gel, budget d'erreurs, failles critiques, couverture minimale, migration compatible (pas de `DROP COLUMN` sans phase `contract`), taille de PR ;
3. un rapport clair (règle, gravité, résultat, dispense éventuelle) ;
4. des tests : un changement conforme passe ; chaque règle peut échouer isolément ; une dispense expirée n'est plus acceptée ;
5. une règle « méta » : aucune dispense ne peut s'appliquer à la règle d'empreinte.

À toi, en solo · réponse

Pistes de correction

```
from datetime import date

class Politiques:
    def __init__(self): self.regles, self.dispenses = [], {}
    def regle(self, nom, gravite="bloquant"):
        def deco(f):
            self.regles.append((nom, gravite, f)); return f
        return deco
    def dispenser(self, nom, jusqua, par):
        if nom == "empreinte":
            raise ValueError("aucune dispense possible pour la règle d'empreinte")
        self.dispenses[nom] = {"jusqua": jusqua, "par": par}
    def evaluer(self, ctx):
        rapport, bloque = [], False
        for nom, gravite, f in self.regles:
            message = f(ctx)
            if message is None:
                rapport.append((nom, gravite, "ok", None)); continue
            d = self.dispenses.get(nom)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
            if d and ctx["date"] <= d["jusqua"]:
                rapport.append((nom, gravite, "dispensé", d["par"])); continue
            rapport.append((nom, gravite, "violation", message))
            bloque = bloque or gravite == "bloquant"
        return (not bloque), rapport

P = Politiques()
@P.regle("empreinte")
def _(c): return None if c["empreinte_deployee"] == c["empreinte_testee"] else "ar
   tefact différent de celui testé"
@P.regle("relectures")
def _(c):
    exige = 2 if any(f.startswith("securite/") for f in c["fichiers"]) else 1
    return None if c["relecteurs"] >= exige else f"{exige} relecture(s) exigée(s),
    {c['relecteurs']} obtenue(s)"
@P.regle("gel")
def _(c): return "gel du vendredi après 16 h" if c["date"].weekday() == 4 and c["h
   eure"] >= 16 else None
@P.regle("budget")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
def _(c): return None if c["budget_restant"] > 0 else "budget d'erreurs épuisé"
@P.regle("failles")
def _(c): return None if c["failles_critiques"] == 0 else "faille critique"
@P.regle("couverture")
def _(c): return None if c["couverture"] >= 80 else f"couverture {c['couverture']}
    % < 80 %"
@P.regle("migration")
def _(c): return "DROP COLUMN sans phase contract" if "DROP COLUMN" in c["sql"] an
   d not c.get("phase_contract") else None
@P.regle("taille_pr", gravite="avertissement")
def _(c): return None if c["lignes"] <= 400 else f"PR de {c['lignes']} lignes : di
   fficile à relire"

BON = dict(empreinte_deployee="a1", empreinte_testee="a1", relecteurs=1, fichiers=
   ["app/x.py"], date=date(2025, 6, 4), heure=10,
           budget_restant=5, failles_critiques=0, couverture=85, sql="", lignes=12
   0)
ok, rapport = P.evaluer(BON); assert ok and all(r[2] == "ok" for r in rapport)
for cle, valeur, regle in [("empreinte_deployee", "zz", "empreinte"), ("relecteurs
```

À toi, en solo · réponse

Pistes de correction (suite)

```
   ", 0, "relectures"), ("budget_restant", 0, "budget"),
                         ("failles_critiques", 2, "failles"), ("couverture", 60,
    "couverture"), ("sql", "ALTER TABLE t DROP COLUMN c", "migration")]:
    ok, rapport = P.evaluer(dict(BON, **{cle: valeur}))
    assert not ok and [r[0] for r in rapport if r[2] == "violation"] == [regle], r
   egle
ok, rapport = P.evaluer(dict(BON, fichiers=["securite/auth.py"], relecteurs=1)); a
   ssert not ok
ok, rapport = P.evaluer(dict(BON, date=date(2025, 6, 6), heure=17)); assert not ok
ok, _ = P.evaluer(dict(BON, lignes=900)); assert ok                          # av
   ertissement seulement
P.dispenser("couverture", date(2025, 6, 10), "Maya")
ok, rapport = P.evaluer(dict(BON, couverture=60)); assert ok and ("couverture", "b
   loquant", "dispensé", "Maya") in rapport
ok, _ = P.evaluer(dict(BON, couverture=60, date=date(2025, 6, 20))); assert not ok
       # dispense expirée
try:
    P.dispenser("empreinte", date(2030, 1, 1), "Kodi"); raise AssertionError
except ValueError: pass
```

À toi, en solo · réponse

Pistes de correction (suite)

```
print("moteur de politiques validé")
```

Écrire la politique comme du code la rend **testable**, **appliquée à tous**, et **historisée** : une dispense a un nom, une date d'expiration, et certaines règles — comme l'empreinte — n'en admettent jamais.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Automatiser sans mesurer	On ne sait pas si on s'améliore	Calculer DORA depuis les événements du pipeline
Retour arrière jamais répété	Il échoue le jour où on en a besoin	Exercer régulièrement (jours de jeu)
Politiques par courtoisie, pas par code	Appliquées inégalement	Politiques exécutables, dispenses tracées et expirantes
Pipeline que personne ne comprend	Un seul expert, risque d'équipe	Documentation, schéma, guide d'intervention, propriétaires

Mini-boss

Mini-boss

Le pipeline de livraison de HashBox, de bout en bout. Livre un dossier complet :

1. un simulateur `pipeline.py` qui enchaîne : commit → CI (graphe de jobs avec chemin critique) → build reproductible (empreinte) → politique de livraison → staging (test de fumée) → canary (5, 25, 100 %) avec décision statistique → observation (SLI, burn rate) → retour arrière automatique, avec journal d'événements ;
2. 8 scénarios de test (sain, défectueux, tests rouges, vendredi, artefact altéré, budget épuisé, faille critique, dispense valide) ;
3. calcul automatique des indicateurs DORA sur 30 jours de changements simulés (graine fixée) ;
4. un schéma du pipeline (Mermaid) et un `RUNBOOK.md` : 5 alertes avec symptômes, vérifications, actions, critères de retour arrière ;
5. un exercice de panne : un composant de la chaîne tombe (registre indisponible, staging cassé), et le pipeline se comporte de façon sûre ;

Mini-boss (suite)

6. un plan d'amélioration à 3 mois fondé sur tes mesures, avec le goulot actuel, le gain attendu et l'indicateur qui prouvera le résultat ;
7. une rétrospective de 10 lignes : quelle brique du module a eu le plus d'effet sur le délai ? sur la fiabilité ?

Mini-boss · réponse

Pistes de correction

Un bon dossier : le pipeline assemble les fonctions de toutes les missions (`chemin_critique`, build déterministe, `Politiques`, `decision_canary`, `burn_rate`, retour arrière), un journal unique d'événements, un calcul DORA qui retrouve fréquence / délai / taux d'échec / MTTR sur 30 jours; le runbook décrit pour chaque alerte (par exemple « burn rate élevé après déploiement ») les premières vérifications (dernier déploiement, comparaison canary/référence), l'action (retour arrière par drapeau ou version précédente) et l'escalade; le plan d'amélioration cible le goulot réel mesuré (par exemple l'approbation manuelle ou le job E2E) avec un indicateur de preuve (délai médian < 1 h).

Quiz éclair

Quiz éclair

1. Dans un pipeline mature, une politique de livraison est :

A. exécutée automatiquement et appliquée à tous les changements
B. un document que personne n'applique
C. une décision verbale

2. Les indicateurs DORA peuvent être calculés :

A. uniquement par questionnaire
B. automatiquement à partir du journal d'événements du pipeline
C. jamais

3. Le retour arrière automatique est déclenché par :

A. l'heure de la journée

Quiz éclair

Quiz éclair (suite)

B. le nombre de commits
C. la détection d'une dérive des métriques pendant le déploiement progressif

4. 🔁 *Révision mission 09* : Un disjoncteur (circuit breaker) sert à :

A. cesser d'appeler une dépendance en panne pour la laisser récupérer
B. accélérer les requêtes
C. chiffrer les échanges

5. 🔁 *Révision mission 09* : Le RPO représente :

A. le temps de rétablissement
B. la perte de données maximale acceptable
C. le nombre de serveurs

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Dans ta propre chaîne de livraison, quelle étape est encore manuelle et fait peur ?
2. Quel indicateur DORA aimerais-tu améliorer d'abord, et quelle action y contribuerait le plus ?

À retenir

À retenir

1. Un pipeline relie Git, build, CI, configuration, CD, observabilité et retour arrière.
2. Les politiques de livraison s'écrivent en code, avec des dispenses tracées.
3. La promotion ou le retour arrière sont décidés par des mesures.
4. Les indicateurs DORA se calculent depuis le journal d'événements.
5. Documenter, exercer, améliorer : un pipeline vit.

Félicitations

Félicitations

Tu as terminé **HEF-120 — DevOps et livraison**. Tu sais utiliser Git comme outil de collaboration, produire des builds reproductibles, conteneuriser un service, gérer configuration et secrets, concevoir un pipeline CI/CD, déployer progressivement et revenir en arrière, observer un système avec des SLO, et le rendre résilient.
→ Prochain module : **HEF-130 — Performance**

MISSION ACCOMPLIE

+120 XP

Badge débloqué : Architecte de la livraison

Fin du module HEF-120. Bravo !

