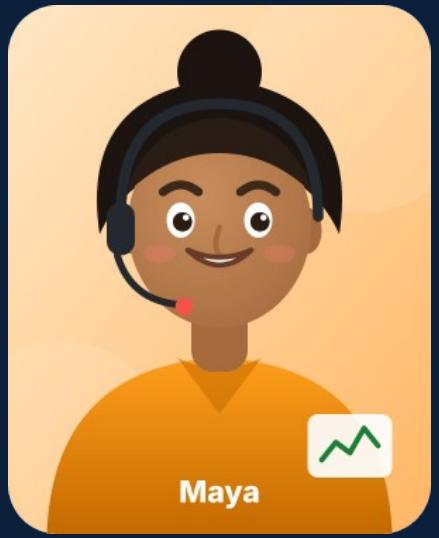
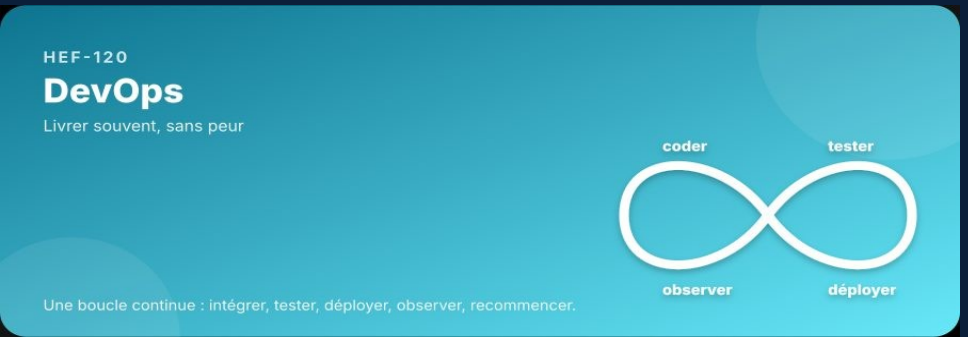
HEF-120 · MISSION 06 / 10

Intégration continue (CI)

Un pipeline est un graphe de tâches : certaines peuvent tourner en même temps, d'autres doivent attendre. Comprendre ce graphe, c'est comprendre pourquoi un pipeline est lent.



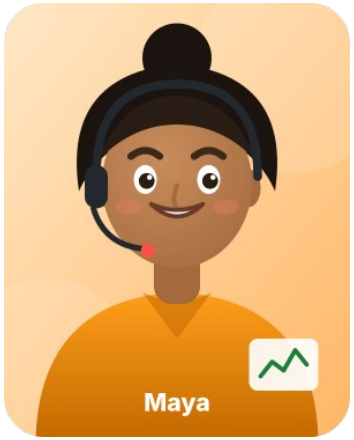


Badge : Chef d'orchestre du pipeline

100 min · intermédiaire

L'histoire

Le pipeline de HashBox dure 25 minutes. Quand Kodi regarde de plus près, il s'aperçoit que les tests unitaires attendent la fin de la vérification de style, alors qu'ils n'ont aucun lien. Les tests sur Windows et sur Linux s'exécutent l'un après l'autre. Et tout recompile à chaque fois, même quand seul le README a changé.



> *« Un pipeline, c'est un graphe de dépendances, dit Sam. Ce qui compte, c'est le chemin le plus long. Tout ce qui est en dehors de ce chemin peut tourner en parallèle sans coûter une seconde de plus. »*

Tu apprends à modéliser un pipeline CI comme un graphe, à paralléliser avec des matrices, à utiliser le cache, et à trouver le chemin critique qui détermine la durée.

Ta mission

À la fin, je serai capable de…

1. décrire un pipeline CI comme un graphe orienté acyclique de jobs
2. simuler son exécution avec un nombre limité d'exécuteurs (runners)
3. calculer le chemin critique et savoir quelles optimisations comptent
4. utiliser les matrices pour tester plusieurs environnements
5. employer le cache à clé de contenu et propager les échecs correctement

Défi d'échauffement

Défi d'échauffement

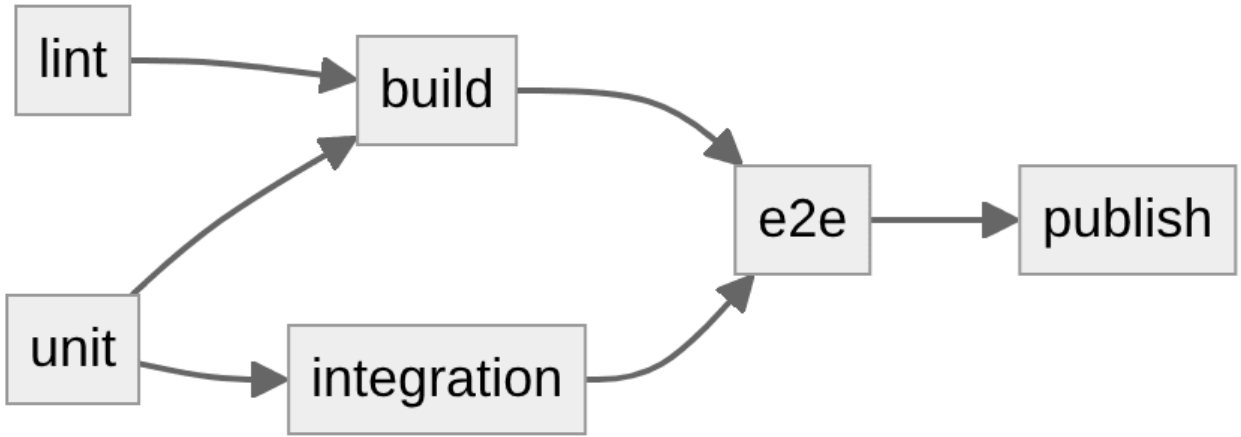
Pour préparer un dîner, tu dois : éplucher (10 min), cuire les pâtes (12 min, après éplucher), faire la sauce (15 min, après éplucher), dresser (3 min, après pâtes et sauce). Combien de temps faut-il seul ? À deux ? Quelle tâche, si tu l'accélères, fait gagner du temps ? Laquelle ne change rien ?

Le concept

Le pipeline comme graphe

Un pipeline CI est un ensemble de **jobs** reliés par des dépendances (needs). Il forme un **graphe orienté acyclique** (DAG) : pas de boucle.

Le pipeline comme graphe (suite)



Description du schéma : six jobs. `lint` et `unit` démarrent en parallèle. `build` attend les deux. `integration` attend `unit`. `e2e` attend `build` et `integration`. `publish` attend `e2e`.

Le concept

Le pipeline comme graphe (suite)

Vocabulaire :

Terme	Sens
Job	tâche exécutée sur un exécuteur (build, tests…)
Étape (*step*)	commande à l'intérieur d'un job
Exécuteur (*runner*)	machine ou conteneur qui exécute un job
Déclencheur	événement qui lance le pipeline (push, PR, planification)
Artefact	fichier produit et transmis d'un job à un autre
Cache	résultats réutilisés d'une exécution à l'autre
Matrice	un même job répété pour plusieurs combinaisons (système, version)

Le concept

Le chemin critique

La durée d'un pipeline avec **assez d'exécuteurs** est celle du **chemin critique** : le chemin le plus long dans le graphe (en sommant les durées).

- Accélérer un job **sur** le chemin critique raccourcit le pipeline ;
- accélérer un job **hors** du chemin critique ne change rien (jusqu'à ce qu'il devienne critique).

Avec peu d'exécuteurs, les jobs parallèles s'ordonnent dans une file : la durée augmente.

Optimisations qui comptent

Technique	Effet
Paralléliser les jobs indépendants	raccourcit le chemin

Le concept

Optimisations qui comptent (suite)

Technique	Effet
Échouer vite : jobs rapides d'abord	économise l'attente
Cache de dépendances et de builds	évite de refaire
Découpage des gros jobs (tests en lots)	raccourcit le chemin critique
Sélection des tests	ne lance que le nécessaire
Runners plus puissants pour le chemin critique	gain direct

Les matrices

Pour tester sur plusieurs combinaisons :

Le concept

Les matrices (suite)

```
matrice: python = [3.10, 3.12] ; système = [linux, windows]
→ 4 jobs indépendants (produit cartésien), avec des exclusions éventuelles
```

Les jobs sont indépendants donc **parallélisables** : ils multiplient la couverture sans allonger la durée (si les exécuteurs suffisent).

Le cache par clé de contenu

Comme pour le build (mission 3) : clé = haché des entrées (`requirements.lock`, sources). Si la clé existe, on restaure ; sinon on calcule et on enregistre. Une clé trop large (la date) ne sert à rien ; trop étroite (le numéro de commit) ne réutilise jamais.

Propagation des échecs

Le concept

Propagation des échecs (suite)

Si un job échoue, ses **dépendants sont ignorés** (*skipped*), mais les jobs indépendants continuent (on veut voir tous les problèmes en une exécution).

Bonnes pratiques

- Pipeline **défini par du code** versionné avec le projet.
- **Délais maximums** (*timeouts*) sur chaque job.
- **Secrets** fournis par la plateforme (mission 5), jamais affichés.
- **Permissions minimales** pour le jeton du pipeline.
- Jobs **idempotents** et sans effet de bord sur les environnements partagés.
- Exécution **sur chaque demande de fusion** ; branche principale protégée.

Le concept

Bonnes pratiques (suite)

🧠 ***Astuce*** *: trace ton pipeline sur papier. Le chemin le plus long est ton premier suspect pour toute optimisation.*

Exemple pas à pas

Problème : simuler l'exécution d'un pipeline avec un nombre limité d'exécuteurs, trouver le chemin critique, mesurer l'effet du cache, de la matrice et d'un échec.

```
import hashlib, heapq
from graphlib import TopologicalSorter
from itertools import product

PIPELINE = {
    "lint":        {"needs": [],                          "duree": 20},
    "unit":        {"needs": [],                          "duree": 60},
    "integration": {"needs": ["unit"],                    "duree": 120},
    "build":       {"needs": ["lint", "unit"],            "duree": 90},
    "e2e":         {"needs": ["build", "integration"],    "duree": 240},
    "publish":     {"needs": ["e2e"],                     "duree": 30},
}

def chemin_critique(pipeline, durees=None):
    durees = durees or {n: j["duree"] for n, j in pipeline.items()}
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    fin, precedent = {}, {}
    for nom in TopologicalSorter({n: j["needs"] for n, j in pipeline.items()}).sta
   tic_order():
        debut = max((fin[d] for d in pipeline[nom]["needs"]), default=0)
        fin[nom] = debut + durees[nom]
        if pipeline[nom]["needs"]:
            precedent[nom] = max(pipeline[nom]["needs"], key=lambda d: fin[d])
    dernier = max(fin, key=fin.get)
    chemin = [dernier]
    while chemin[-1] in precedent:
        chemin.append(precedent[chemin[-1]])
    return fin[dernier], chemin[::-1]

def simuler(pipeline, runners, echecs=frozenset(), durees=None):
    # simulation à événements discrets : un job démarre quand ses dépendances sont
    finies ET qu'un runner est libre
    durees = durees or {n: j["duree"] for n, j in pipeline.items()}
    etat, fin, temps = {}, {}, 0
    restants = {n: set(j["needs"]) for n, j in pipeline.items()}
```

Exemple pas à pas (suite)

```
    en_cours = []                                                  # (heure de fin
, nom)
    while len(etat) < len(pipeline):
        for nom in sorted(n for n in pipeline if n not in etat and n not in [x[1]
for x in en_cours]):
            deps = pipeline[nom]["needs"]
            if any(etat.get(d) in ("echec", "ignore") for d in deps):
                etat[nom] = "ignore"; fin[nom] = temps             # un parent a é
choué : on ignore
            elif all(etat.get(d) == "ok" for d in deps) and len(en_cours) < runner
s:
                heapq.heappush(en_cours, (temps + durees[nom], nom))
        if len(etat) == len(pipeline):
            break
        if not en_cours:
            continue
        temps, nom = heapq.heappop(en_cours)
        etat[nom] = "echec" if nom in echecs else "ok"; fin[nom] = temps
    return max(fin.values()), etat
```

Exemple pas à pas

Exemple pas à pas (suite)

```
total, chemin = chemin_critique(PIPELINE)
print(f"chemin critique : {' → '.join(chemin)} = {total} s")
assert chemin == ["unit", "integration", "e2e", "publish"] and total == 450

for r in (1, 2, 3, 6):
    print(f"{r} runner(s) : {simuler(PIPELINE, r)[0]:4d} s")
assert simuler(PIPELINE, 1)[0] == sum(j["duree"] for j in PIPELINE.values())
       # séquentiel : somme
assert simuler(PIPELINE, 6)[0] == total
           # assez de runners : chemin critique

# ----- optimiser un job HORS du chemin critique ne change rien ; sur le chemin, ç
   a compte -----
base = {n: j["duree"] for n, j in PIPELINE.items()}
hors = dict(base, lint=5)
sur = dict(base, e2e=120)
print("lint 20→5 s :", chemin_critique(PIPELINE, hors)[0], "s   |   e2e 240→120 s
   :", chemin_critique(PIPELINE, sur)[0], "s")
```

Exemple pas à pas (suite)

```
assert chemin_critique(PIPELINE, hors)[0] == total and chemin_critique(PIPELINE, s
   ur)[0] == total - 120

# ----- échec : les dépendants sont ignorés, les indépendants continuent -----
duree, etat = simuler(PIPELINE, 6, echecs={"unit"})
print("si unit échoue :", etat)
assert etat["unit"] == "echec" and etat["integration"] == "ignore" and etat["build
   "] == "ignore" and etat["lint"] == "ok" and etat["publish"] == "ignore"

# ----- cache par clé de contenu -----
CACHE = set()
def cle(*entrees):
    return hashlib.sha256("|".join(entrees).encode()).hexdigest()[:8]

def installer_dependances(lockfile):
    k = cle("deps", lockfile)
    if k in CACHE:
        return 5                                              # restauration
    du cache : 5 s
```

Exemple pas à pas (suite)

```
    CACHE.add(k); return 90                                          # installation
    complète : 90 s

durees_installation = [installer_dependances("lock-v1"), installer_dependances("lo
   ck-v1"), installer_dependances("lock-v2"), installer_dependances("lock-v2")]
print("installations successives :", durees_installation)
assert durees_installation == [90, 5, 90, 5]

# ----- matrice -----
MATRICE = {"python": ["3.10", "3.12"], "systeme": ["linux", "windows"]}
EXCLUSIONS = [{"python": "3.10", "systeme": "windows"}]
combinaisons = [dict(zip(MATRICE, valeurs)) for valeurs in product(*MATRICE.values
   ())]
combinaisons = [c for c in combinaisons if c not in EXCLUSIONS]
print(len(combinaisons), "jobs :", [f"{c['python']}-{c['systeme']}" for c in combi
   naisons])
assert len(combinaisons) == 3
```

Exemple pas à pas (suite)

Lecture :

- Le **chemin critique** (`unit → integration → e2e → publish`, 450 s) fixe la durée minimale, quelle que soit la quantité d'exécuteurs. Avec 1 exécuteur, c'est la somme (560 s) ; avec 6, on atteint le chemin critique.
- Gagner 15 s sur `lint` (hors chemin) ne change rien ; gagner 120 s sur `e2e` (sur le chemin) fait gagner 120 s.
- Un **échec** de `unit` fait ignorer ses descendants, tandis que `lint` s'exécute quand même.
- Le **cache** à clé de contenu réutilise l'installation tant que le fichier de verrouillage ne change pas.
- La **matrice** crée trois jobs parallèles (une combinaison exclue).

À toi, avec aide

Écris `etendre_matrice(matrice, exclure=(), inclure=())` qui produit la liste des combinaisons d'une matrice de jobs (produit cartésien), retire celles qui correspondent à une exclusion (une exclusion est un dictionnaire partiel : toute combinaison qui contient ces valeurs est retirée), et ajoute les inclusions. Teste : `{"python": ["3.10","3.11","3.12"], "systeme": ["linux","windows","macos"]}` avec exclusion `{"python":"3.10","systeme":"macos"}` et exclusion `{"systeme":"windows","python":"3.11"}`, et l'inclusion `{"python":"3.13","systeme":"linux"}`.

À toi, avec aide · réponse

Indices progressifs

Indice 1

`itertools.product` sur les valeurs.

Indice 2

Une exclusion partielle correspond si toutes ses clés sont égales dans la combinaison : `all(c[k] == v for k, v in excl.items())`.

Indice 3

Compte : 9 − 2 + 1 = 8.

À toi, avec aide · réponse

Réponse

```
from itertools import product

def etendre_matrice(matrice, exclure=(), inclure=()):
    combinaisons = [dict(zip(matrice, v)) for v in product(*matrice.values())]
    combinaisons = [c for c in combinaisons if not any(all(c.get(k) == v for k, v
   in e.items()) for e in exclure)]
    return combinaisons + [dict(i) for i in inclure]

M = {"python": ["3.10", "3.11", "3.12"], "systeme": ["linux", "windows", "macos"]}
res = etendre_matrice(M, exclure=[{"python": "3.10", "systeme": "macos"}, {"system
   e": "windows", "python": "3.11"}],
                      inclure=[{"python": "3.13", "systeme": "linux"}])
assert len(res) == 8 and {"python": "3.13", "systeme": "linux"} in res
assert {"python": "3.10", "systeme": "macos"} not in res and {"python": "3.11", "s
   ysteme": "windows"} not in res
print(len(res), "jobs")
```

À toi, avec aide · réponse

Réponse (suite)

Une matrice de 3 × 3 exécutée en parallèle ne coûte pas plus longtemps qu'un seul job (avec assez de runners), mais elle multiplie la confiance dans la compatibilité.

À toi, en solo

Optimiseur de pipeline. À partir du graphe de l'exemple :

1. calcule pour **chaque job** le « gain si on le réduisait de 50 % » sur la durée totale (avec assez de runners) ;
2. identifie les jobs « critiques » (ceux dont la réduction améliore la durée) ;
3. teste trois stratégies : (a) couper `e2e` en deux jobs parallèles de 130 s chacun ; (b) mettre en cache `build` (90 s → 10 s) ; (c) faire passer `integration` avant `unit`… (impossible : explique pourquoi la validation du graphe doit refuser de modifier l'ordre sans retirer la dépendance) ;
4. écris `valider(pipeline)` qui détecte : dépendance inconnue, cycle, job sans durée maximale (`timeout`), job isolé inutile ;
5. teste la validation avec 4 pipelines défectueux.

À toi, en solo · réponse

Pistes de correction

```
from graphlib import TopologicalSorter, CycleError

def duree_totale(pipeline, durees):
    fin = {}
    for nom in TopologicalSorter({n: j["needs"] for n, j in pipeline.items()}).sta
   tic_order():
        fin[nom] = max((fin[d] for d in pipeline[nom]["needs"]), default=0) + dure
   es[nom]
    return max(fin.values())

P = {"lint": {"needs": [], "duree": 20}, "unit": {"needs": [], "duree": 60}, "inte
   gration": {"needs": ["unit"], "duree": 120},
     "build": {"needs": ["lint", "unit"], "duree": 90}, "e2e": {"needs": ["build",
    "integration"], "duree": 240}, "publish": {"needs": ["e2e"], "duree": 30}}
base = {n: j["duree"] for n, j in P.items()}
ref = duree_totale(P, base)
gains = {n: ref - duree_totale(P, dict(base, **{n: base[n] / 2})) for n in P}
print({n: g for n, g in sorted(gains.items(), key=lambda kv: -kv[1])})
critiques = sorted(n for n, g in gains.items() if g > 0)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
assert critiques == ["e2e", "integration", "publish", "unit"]

# (a) couper e2e en deux jobs parallèles de 130 s
P_a = dict(P); del P_a["e2e"]
P_a["e2e_a"] = {"needs": ["build", "integration"], "duree": 130}
P_a["e2e_b"] = {"needs": ["build", "integration"], "duree": 130}
P_a["publish"] = {"needs": ["e2e_a", "e2e_b"], "duree": 30}
d_a = duree_totale(P_a, {n: j["duree"] for n, j in P_a.items()})
# (b) cache de build : 90 s → 10 s (build n'est pas critique : integration l'est)
d_b = duree_totale(P, dict(base, build=10))
print(f"référence {ref} s | e2e découpé {d_a} s | build en cache {d_b} s")
assert d_a == ref - 110 and d_b == ref

def valider(pipeline):
    problemes = []
    for n, j in pipeline.items():
        for d in j["needs"]:
            if d not in pipeline: problemes.append(f"{n} dépend de {d} qui n'exist
   e pas")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        if "timeout" not in j: problemes.append(f"{n} sans timeout")
    try:
        list(TopologicalSorter({n: [d for d in j["needs"] if d in pipeline] for n,
    j in pipeline.items()}).static_order())
    except CycleError as e:
        problemes.append(f"cycle : {e.args[1]}")
    utilises = {d for j in pipeline.values() for d in j["needs"]}
    for n, j in pipeline.items():
        if not j["needs"] and n not in utilises and len(pipeline) > 1:
            problemes.append(f"{n} est isolé (ni entrée ni sortie)")
    return problemes

ok = {"a": {"needs": [], "duree": 1, "timeout": 5}, "b": {"needs": ["a"], "duree":
    1, "timeout": 5}}
assert valider(ok) == []
assert any("n'existe pas" in p for p in valider({"a": {"needs": ["z"], "duree": 1,
    "timeout": 5}}))
assert any("cycle" in p for p in valider({"a": {"needs": ["b"], "duree": 1, "timeo
   ut": 5}, "b": {"needs": ["a"], "duree": 1, "timeout": 5}}))
```

À toi, en solo · réponse

Pistes de correction (suite)

```
assert "a sans timeout" in valider({"a": {"needs": [], "duree": 1}, "b": {"needs":
    ["a"], "duree": 1, "timeout": 5}})
assert any("isolé" in p for p in valider({"a": {"needs": [], "duree": 1, "timeout"
   : 5}, "b": {"needs": [], "duree": 1, "timeout": 5}}))
print("validation du graphe : 4 défauts détectés")
```

Découper `e2e` raccourcit le pipeline (c'est sur le chemin critique), tandis que mettre `build` en cache **ne change rien** : `integration` est plus longue que `build`, donc `build` n'est pas critique. Changer l'ordre de `integration` et `unit` supposerait de retirer la dépendance ; le graphe dit pourquoi un ordre est imposé, la validation protège de ces modifications incohérentes.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Tout enchaîner en série	Pipeline lent sans raison	Paralléliser les jobs indépendants
Optimiser un job hors chemin critique	Aucun gain	Calculer le chemin critique d'abord
Cache sans clé de contenu	Résultats périmés ou jamais réutilisés	Clé = haché des entrées pertinentes
Pipeline sans timeout	Un job bloqué occupe un runner indéfiniment	Un délai maximal par job

Mini-boss

Moteur de pipeline de HashBox. Écris `mini_pipeline.py` qui lit une définition (dictionnaire de jobs avec `needs`, `duree`, `timeout`, `cache` optionnel, `matrice` optionnelle) et :

1. valide la définition (dépendances, cycles, timeouts) ;
2. développe les matrices en jobs ;
3. simule l'exécution avec N runners, avec cache et échecs injectés ;
4. produit une chronologie (qui a tourné quand, sur quel runner) et la durée totale ;
5. calcule le chemin critique et propose la meilleure optimisation ;
6. 12 tests : cycle, dépendance inconnue, échec propagé, cache, matrice avec exclusion, runner unique égal à la somme, assez de runners égal au chemin critique.

Mini-boss · réponse

Pistes de correction

Un bon travail : fonctions `valider`, `etendre_matrice`, `simuler`, `chemin_critique` de la mission combinées; chronologie sous forme de tableau (job, runner, début, fin, état); suggestion « découper le job critique le plus long ». Tests : un runner → somme des durées; assez de runners → chemin critique; un échec ignore tous les descendants; un cache chaud réduit la durée; une exclusion de matrice retire la bonne combinaison.

Quiz éclair

Quiz éclair

1. Un pipeline CI est modélisé comme :

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

2. La durée minimale d'un pipeline (avec assez de runners) est celle :

A. du premier job
B. de la somme de tous les jobs
C. du chemin critique

3. Si un job échoue, ses dépendants sont :

A. ignorés, tandis que les jobs indépendants continuent

Quiz éclair

Quiz éclair (suite)

B. exécutés quand même
C. supprimés du dépôt

4. 🔁 *Révision mission 05* : Un secret est :

A. un nom de variable
B. une configuration sensible dont la fuite cause un dommage
C. un commentaire dans le code

5. 🔁 *Révision mission 05* : Pour changer un secret sans interruption, on :

A. remplace brutalement le secret
B. attend la panne
C. accepte l'ancien et le nouveau pendant la migration, puis révoque l'ancien

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quel est le chemin critique de ton pipeline actuel ? L'as-tu déjà dessiné ?
2. Quel job lent pourrait être découpé ou mis en cache sans risque ?

À retenir

À retenir

1. Pipeline = graphe de jobs ; DAG sans cycle.
2. Chemin critique = durée minimale ; optimiser seulement ce qui est dessus.
3. Cache à clé de contenu ; matrices pour la compatibilité.
4. Un échec ignore les dépendants, pas les indépendants.
5. Timeouts, permissions minimales, pipeline défini par du code.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Chef d'orchestre du pipeline

Prochaine mission : CD et environnements

