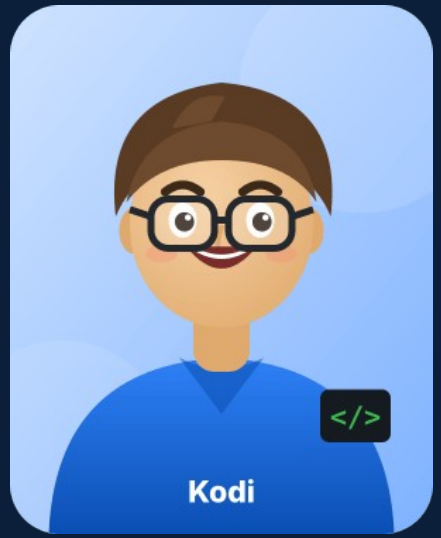
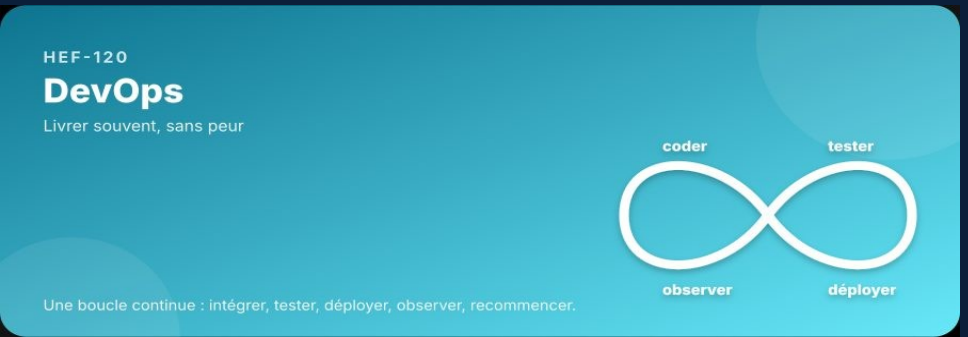
HEF-120 · MISSION 09 / 10

Rollback et résilience opérationnelle

Tout finit par tomber en panne. La question est : combien de temps, quel impact, et à quelle vitesse revient-on à un état sain ?





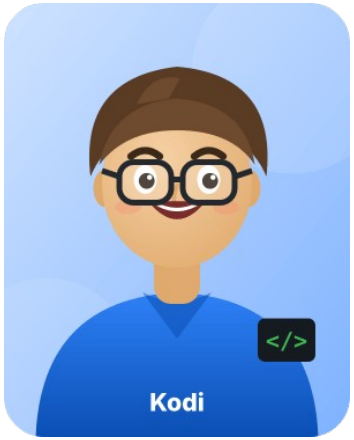
Badge : Funambule avec filet

105 min · intermédiaire

L'histoire

Un jeudi soir, un déploiement de HashBox augmente le temps de réponse de la base. Les requêtes s'accumulent, les clients relancent, la charge double, la base s'effondre. Une panne locale devient une panne totale : c'est la **cascade**. Il faut une heure pour reprendre la main.

> *« Le déploiement n'est pas le vrai problème, dit Maya. Le vrai problème, c'est que chaque composant a continué de harceler un voisin déjà en difficulté. Un système résilient se protège : il abandonne vite, ne retente pas bêtement, et sait revenir en arrière. »*



Tu apprends à revenir en arrière proprement, et les mécanismes de résilience : délais d'attente, tentatives avec recul exponentiel, disjoncteur, dégradation progressive, sauvegardes, exercices de panne.

Ta mission

À la fin, je serai capable de…

1. choisir entre retour arrière et correction en avant (rollback / roll-forward)
2. implémenter des tentatives avec recul exponentiel et gigue (jitter)
3. implémenter un disjoncteur (circuit breaker) avec états fermé, ouvert, semi-ouvert
4. calculer RTO et RPO d'un plan de sauvegarde
5. préparer et mener un exercice de panne contrôlé (chaos engineering)

Défi d'échauffement

Défi d'échauffement

Dans un immeuble, quand un appareil électrique surchauffe, le disjoncteur coupe le courant de ce circuit pour protéger toute l'installation. Après réparation, on le réarme. Que se passerait-il sans disjoncteur ? Quel équivalent logiciel imagines-tu quand un service dont tu dépends tombe en panne ?

Le concept

Revenir en arrière

Stratégie	Quand	Condition
Rollback (retour arrière)	un déploiement dégrade le service	version précédente toujours disponible ; base compatible (mission 7)
Roll-forward (correction en avant)	le correctif est petit et rapide, ou le retour arrière est impossible (données migrées)	pipeline rapide et fiable
Désactivation par drapeau	la fonctionnalité est derrière un drapeau	le plus rapide : un interrupteur

Le **temps de rétablissement** (MTTR, mission 1) dépend de la **simplicité du retour arrière**. Il se prépare avant, pas pendant l'incident.

Le concept

Revenir en arrière (suite)

Contrôles de santé : *liveness* (« le processus vit-il ? » sinon redémarrage) et *readiness* (« est-il prêt à recevoir du trafic ? » sinon retiré du routage). Un déploiement automatique s'arrête si les contrôles échouent.

Les pannes en cascade

Un composant lent ou en panne → ses clients attendent → leurs ressources (threads, connexions) se remplissent → ils tombent aussi → la panne remonte. Les relances automatiques **aggravent** la situation (tempête de tentatives, *retry storm*).

Les mécanismes de défense

Mécanisme	Idée
Délai d'attente (*timeout*)	ne jamais attendre indéfiniment ; mieux vaut échouer vite

Le concept

Les mécanismes de défense (suite)

Mécanisme	Idée
Tentatives avec recul exponentiel	réessayer, mais en espaçant de plus en plus (1 s, 2 s, 4 s…)
Gigue (*jitter*)	ajouter de l'aléa pour que les clients ne retentent pas tous au même instant
Budget de tentatives	limiter le nombre total (3 essais, pas 30)
Disjoncteur	après N échecs, arrêter d'appeler pendant un temps, puis tester prudemment
Cloisonnement (*bulkhead*)	réserver des ressources par fonction pour qu'une panne ne contamine pas tout
Dégradation progressive	continuer avec un service réduit (cache, valeur par défaut)
Limitation de débit / délestage	refuser une partie du trafic pour préserver le reste
Idempotence	pouvoir rejouer une opération sans effet double

Le concept

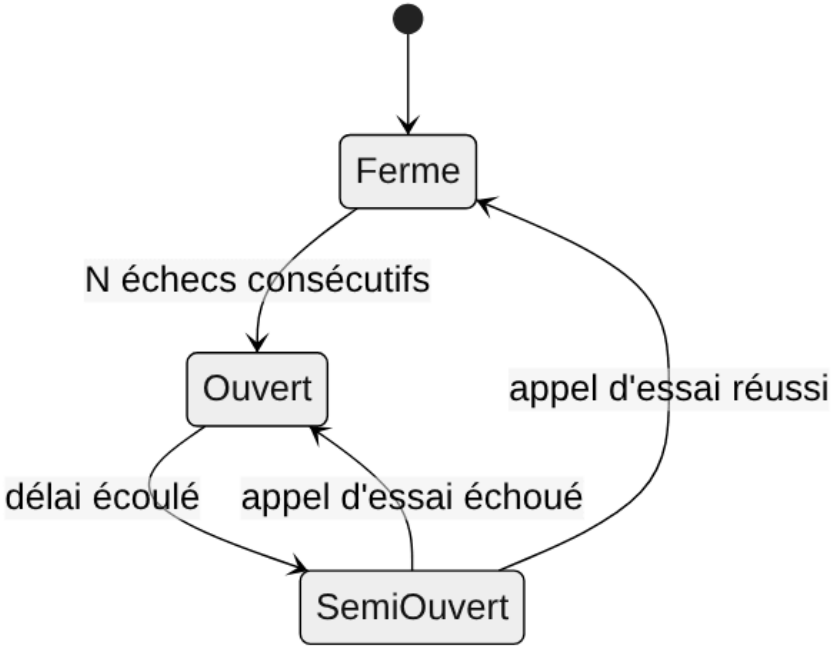
Les mécanismes de défense (suite)

On ne retente que des opérations **idempotentes** et des erreurs **transitoires** (délai, 503), jamais des erreurs définitives (400, 404).

Le disjoncteur

Le concept

Le disjoncteur (suite)



Le concept

Le disjoncteur (suite)

Description du schéma : trois états. Fermé (les appels passent). Après N échecs consécutifs : ouvert (les appels sont refusés immédiatement). Après un délai : semi-ouvert (un appel d'essai est autorisé). Si l'essai réussit : retour à fermé ; s'il échoue : retour à ouvert.

Sauvegardes et reprise

Sigle	Sens	Question
RPO (*Recovery Point Objective*)	perte de données maximale acceptable	combien d'heures de données peut-on perdre ?
RTO (*Recovery Time Objective*)	délai maximal de rétablissement	en combien de temps doit-on être de nouveau en service ?

- Sauvegarde toutes les 24 h → **RPO ≈ 24 h** ; avec réplication continue → RPO de quelques secondes.
- Règle **3-2-1** : 3 copies, sur 2 supports différents, dont 1 hors site.

Le concept

Sauvegardes et reprise (suite)

- **Une sauvegarde non restaurée n'est pas une sauvegarde** : tester régulièrement la restauration.

Ingénierie du chaos

On **provoque volontairement** des pannes (instance arrêtée, latence ajoutée, dépendance coupée) dans un cadre **contrôlé** pour vérifier que les défenses fonctionnent :

1. définir l'**état stable** mesurable (SLI) ;
2. formuler une **hypothèse** (« si le cache tombe, le taux de succès reste > 99 % ») ;
3. **injecter** la panne, à petite échelle d'abord, avec un bouton d'arrêt ;
4. **observer**, comparer à l'hypothèse ;
5. **corriger** et recommencer.

Le concept

Procédures d'urgence (*runbooks*)

Pour chaque alerte : un document court, à jour, avec symptômes, vérifications, actions, personnes à joindre, et critères d'escalade. Un exercice périodique (« jour de jeu ») les valide.

🧠 ***Astuce** : chaque fois que tu écris un appel à un autre service, écris en même temps : le délai d'attente, ce qu'on fait en cas d'échec, et si l'opération peut être rejouée sans danger.*

Exemple pas à pas

Problème : implémenter des tentatives avec recul exponentiel et gigue, un disjoncteur, puis montrer par simulation qu'ils évitent la cascade.

```
import random

# ---------------- horloge simulée et dépendance instable ----------------
class Horloge:
    def __init__(self): self.t = 0.0
    def __call__(self): return self.t
    def dormir(self, s): self.t += s

class DependanceEnPanne:
    def __init__(self, horloge, pannes):                # pannes : liste d'interv
   alles [(debut, fin)]
        self.h, self.pannes, self.appels = horloge, pannes, 0
    def appeler(self):
        self.appels += 1
        if any(a <= self.h() < b for a, b in self.pannes):
```

Exemple pas à pas (suite)

```
            raise ConnectionError("dépendance indisponible")
        return "ok"

# ---------------- tentatives avec recul exponentiel et gigue ----------------
def avec_tentatives(fonction, horloge, essais=4, base=0.5, plafond=8.0, rng=None):
    rng = rng or random.Random(0)
    for numero in range(essais):
        try:
            return fonction()
        except ConnectionError:
            if numero == essais - 1:
                raise
            attente = min(plafond, base * 2 ** numero)
            horloge.dormir(rng.uniform(0, attente))                    # « gigue co
   mplète » : entre 0 et le recul
    raise AssertionError

h = Horloge()
dep = DependanceEnPanne(h, [(0, 1.0)])                                 # panne brèv
```

Exemple pas à pas (suite)

```
   e : 1 seconde
assert avec_tentatives(dep.appeler, h, rng=random.Random(3)) == "ok"
print(f"succès après {dep.appels} appel(s), horloge = {h.t:.2f} s")

# ---------------- disjoncteur ----------------
class Disjoncteur:
    def __init__(self, horloge, seuil=3, delai=10.0):
        self.h, self.seuil, self.delai = horloge, seuil, delai
        self.etat, self.echecs, self.ouvert_a = "ferme", 0, None
    def appeler(self, fonction):
        if self.etat == "ouvert":
            if self.h() - self.ouvert_a >= self.delai:
                self.etat = "semi-ouvert"
            else:
                raise RuntimeError("disjoncteur ouvert : appel refusé sans toucher
    la dépendance")
        try:
            resultat = fonction()
        except ConnectionError:
```

Exemple pas à pas (suite)

```
            self.echecs += 1
            if self.etat == "semi-ouvert" or self.echecs >= self.seuil:
                self.etat, self.ouvert_a = "ouvert", self.h()
            raise
        self.etat, self.echecs = "ferme", 0
        return resultat

h = Horloge()
dep = DependanceEnPanne(h, [(0, 30.0)])                          # panne de
   30 s
dj = Disjoncteur(h, seuil=3, delai=10.0)
refus, vers_dependance = 0, 0
for _ in range(200):                                             # un appel
   toutes les 0,25 s pendant 50 s
    h.dormir(0.25)
    en_panne = h() <= 30.0
    avant = dep.appels
    try:
        dj.appeler(dep.appeler)
```

Exemple pas à pas (suite)

```
    except RuntimeError:
        refus += 1
    except ConnectionError:
        pass
    if en_panne:
        vers_dependance += dep.appels - avant
print(f"pendant la panne de 30 s (120 appels) : {refus} refusés localement, seulem
   ent {vers_dependance} envoyés à la dépendance")
assert vers_dependance < 15 and refus > 100
assert dj.etat == "ferme"                                          # la dépend
   ance est revenue, le disjoncteur s'est refermé

# ---------------- la cascade : avec ou sans protection ----------------
def charge(avec_protection, graine=1):
    rng, h = random.Random(graine), Horloge()
    dep = DependanceEnPanne(h, [(5.0, 15.0)])
    dj = Disjoncteur(h, seuil=3, delai=3.0)
    sollicitations_dependance = 0
    for _ in range(400):                                           # 400 clien
```

Exemple pas à pas (suite)

```
    ts, 1 requête / 0,05 s
         h.dormir(0.05)
         for _tentative in range(1 if avec_protection else 10):          # sans pro
    tection : 10 relances immédiates
             avant = dep.appels
             try:
                 (dj.appeler(dep.appeler) if avec_protection else dep.appeler())
                 break
             except (RuntimeError, ConnectionError):
                 pass
             finally:
                 sollicitations_dependance += dep.appels - avant
     return sollicitations_dependance

sans, avec = charge(False), charge(True)
print(f"appels reçus par la dépendance en difficulté : sans protection {sans}, ave
    c disjoncteur {avec}")
assert sans > 3 * avec
```

Exemple pas à pas (suite)

Lecture :

- **Recul exponentiel avec gigue** : après une panne brève, la nouvelle tentative réussit sans marteler la dépendance.
- **Disjoncteur** : pendant une panne de 30 s, plus de 100 appels sur 120 sont refusés *localement* (instantanément, sans toucher la dépendance), alors que seuls quelques appels d'essai (trois échecs pour ouvrir, puis un essai toutes les dix secondes) sont envoyés. Une fois la dépendance revenue, le disjoncteur se referme tout seul.
- **Cascade** : sans protection, la dépendance en difficulté reçoit plusieurs fois plus d'appels (relances immédiates) que d'habitude, ce qui l'empêche de se rétablir. Avec le disjoncteur, elle est laissée tranquille pendant la panne.

À toi, avec aide

À toi, avec aide

Calcule RPO et RTO pour trois plans de sauvegarde et compare à un besoin de **RPO ≤ 15 min** et **RTO ≤ 1 h** :

Plan	Sauvegarde	Restauration
A	complète toutes les 24 h à minuit	3 h (téléchargement + restauration)
B	complète toutes les 24 h + journaux de transactions toutes les 5 min	1 h 30
C	réplique synchrone vers un site secondaire + bascule automatique	5 min

Pour chacun : RPO et RTO pires cas, verdict, et recommandation de coût (A, B, C selon le besoin).

À toi, avec aide · réponse

Indices progressifs

Indice 1

RPO pire cas = intervalle entre deux points de sauvegarde (pour A : 24 h).

Indice 2

Pour B, le RPO est l'intervalle des journaux (5 min), mais le RTO inclut le rejeu.

Indice 3

Un plan peut respecter le RPO sans respecter le RTO.

À toi, avec aide · réponse

Réponse

```
besoin = {"rpo_min": 15, "rto_min": 60}
plans = {"A": {"rpo": 24 * 60, "rto": 180}, "B": {"rpo": 5, "rto": 90}, "C": {"rpo
   ": 0, "rto": 5}}
for nom, p in plans.items():
    ok_rpo, ok_rto = p["rpo"] <= besoin["rpo_min"], p["rto"] <= besoin["rto_min"]
    print(f"plan {nom} : RPO {p['rpo']} min {'✅' if ok_rpo else '❌'} | RTO {p['rto
   ']} min {'✅' if ok_rto else '❌'}")
assert [n for n, p in plans.items() if p["rpo"] <= 15 and p["rto"] <= 60] == ["C"]
```

Plan A : jusqu'à **24 h de données perdues** et 3 h d'arrêt — ne respecte rien. Plan B : RPO excellent (5 min) mais **RTO de 1 h 30** : il faut l'améliorer (restauration automatisée, serveur de secours préparé) ou accepter 1 h 30. Plan C : respecte les deux, mais coûte une infrastructure en double. **Recommandation** : C si le besoin est strict ; B + automatisation de la restauration si le budget est limité et qu'on peut ajuster le RTO à 1 h 30. Dans tous les cas : **tester la restauration** régulièrement.

À toi, en solo

Mini ingénierie du chaos. Construis un banc d'essai :

1. un service `Recherche` qui dépend d'un `Index` (lent ou en panne par intermittence) et d'un `Cache` ;
2. un injecteur de pannes avec probabilité et durée configurables ;
3. une version « fragile » (pas de timeout, pas de repli) et une version « résiliente » (timeout simulé, repli sur le cache, disjoncteur) ;
4. un protocole en 5 étapes (état stable, hypothèse, injection, observation, conclusion) exécuté pour des pannes de 10 %, 30 % et 60 % de l'index ;
5. un rapport comparant le taux de succès et la latence p95 des deux versions ;
6. identifie le seuil de panne au-delà duquel même la version résiliente ne tient plus son SLO de 99 %.

Pistes de correction

```
import random, math

def p95(v):
    v = sorted(v); return v[math.ceil(0.95 * len(v)) - 1]

def scenario(version, taux_panne, n=3000, graine=2):
    rng = random.Random(graine)
    cache = {f"q{i}": f"résultat {i}" for i in range(50)}              # le cache
   couvre les requêtes fréquentes
    ouvert_jusqua, echecs_consecutifs, succes, latences = -1, 0, 0, []
    for i in range(n):
        q = f"q{rng.randrange(60)}"                                    # 50/60 d
   es requêtes sont en cache
        if version == "fragile":
            if rng.random() < taux_panne:
                latences.append(30000)                                 # attend
   30 s puis échoue
                continue
            latences.append(rng.uniform(20, 60)); succes += 1
```

À toi, en solo · réponse

Pistes de correction (suite)

```
            continue
        if i < ouvert_jusqua:                                   # disjonc
    teur ouvert : repli immédiat
            latences.append(2)
            if q in cache: succes += 1
            continue
        if rng.random() < taux_panne:                           # timeout
     de 200 ms, puis repli
            latences.append(200); echecs_consecutifs += 1
            if echecs_consecutifs >= 5: ouvert_jusqua = i + 50
            if q in cache: succes += 1
        else:
            latences.append(rng.uniform(20, 60)); succes += 1; echecs_consecutifs
    = 0
     return succes / n, p95(latences)

for taux in (0.0, 0.1, 0.3, 0.6):
    f, r = scenario("fragile", taux), scenario("resiliente", taux)
    print(f"panne {taux:4.0%} | fragile : succès {f[0]:6.1%}, p95 {f[1]:7.0f} ms |
```

Pistes de correction (suite)

```
    résiliente : succès {r[0]:6.1%}, p95 {r[1]:4.0f} ms")
assert scenario("resiliente", 0.3)[1] < 300 < scenario("fragile", 0.3)[1]
assert scenario("resiliente", 0.3)[0] > scenario("fragile", 0.3)[0]
# seuil à partir duquel même la version résiliente passe sous 99 % : dès qu'une pa
   nne existe, car 1/6 des requêtes ne sont pas en cache
seuil = next(t for t in (0.0, 0.05, 0.1, 0.2, 0.3, 0.5, 0.6) if scenario("resilien
   te", t)[0] < 0.99)
print("la version résiliente passe sous 99 % dès une panne de", f"{seuil:.0%}")
```

La version résiliente **limite la latence** (timeout + disjoncteur + repli) et garde un taux de succès nettement plus élevé, mais elle ne fait pas de miracle : les requêtes absentes du cache échouent tant que l'index est en panne. Le chaos révèle précisément cette limite — ici, la **couverture du cache** — et donne une action concrète (réchauffer le cache, élargir le repli).

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Relancer immédiatement et sans limite	Tempête de tentatives, cascade	Recul exponentiel, gigue, budget de tentatives
Appels sans délai d'attente	Les ressources s'épuisent à attendre	Timeout partout, plus court que celui de l'appelant
Retenter une opération non idempotente	Double facturation, doublons	Clés d'idempotence, ou ne pas retenter
Sauvegardes jamais restaurées	On découvre qu'elles sont inutilisables le jour de la panne	Exercices de restauration planifiés

Mini-boss

Mini-boss

Exercice de résilience de HashBox. Prépare et mène un « jour de jeu » simulé :

1. décris le système (api → service paiement + base + cache) et ses défenses (timeouts, tentatives, disjoncteur, repli) ;
2. écris un banc de simulation avec injection de pannes (latence, indisponibilité, perte de paquets) sur chaque dépendance ;
3. définis l'état stable (SLI) et 4 hypothèses ;
4. exécute 6 expériences, et pour chacune compare l'hypothèse au résultat ;
5. simule un déploiement fautif avec **retour arrière automatique** déclenché par les contrôles de santé, et mesure le temps de rétablissement ;
6. calcule RPO/RTO pour un plan de sauvegarde et simule une restauration (en recréant une base SQLite depuis une sauvegarde) ;
7. rédige le rapport : ce qui a tenu, ce qui a cédé, 5 actions correctives priorisées.

Mini-boss · réponse

Pistes de correction

Un bon travail : classes `Horloge`, `Disjoncteur`, `avec_tentatives`, un injecteur de pannes paramétrable, 6 expériences avec hypothèses explicites, un contrôleur de déploiement qui surveille la santé et revient en arrière, une restauration réelle via `sqlite3.Connection.backup()` vérifiée par comptage de lignes, et un rapport honnête incluant au moins une hypothèse **réfutée** (c'est là que l'on apprend le plus).

Quiz éclair

Quiz éclair

1. Un disjoncteur (circuit breaker) sert à :

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

2. Le recul exponentiel avec gigue sert à :

A. augmenter le nombre de requêtes
B. cacher les erreurs
C. éviter que tous les clients retentent au même instant

3. Le RPO représente :

A. la perte de données maximale acceptable

Quiz éclair (suite)

B. le temps de rétablissement

C. le nombre de serveurs

4. 🔁 *Révision mission 08* : Un budget d'erreurs de 99,9 % sur 30 jours correspond à environ :
A. 4 heures

B. 43 minutes d'indisponibilité

C. 4 secondes

5. 🔁 *Révision mission 08* : Pour mesurer la latence ressentie par les moins chanceux, on regarde :
A. la moyenne uniquement

B. le nombre de serveurs

C. les percentiles élevés (p95, p99)

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle est ta dépendance la plus fragile ? Que fait ton code si elle répond en 30 secondes ?
2. À quand remonte la dernière restauration réelle d'une sauvegarde chez toi ?

À retenir

À retenir

1. Rollback, roll-forward ou drapeau : préparés avant l'incident.
2. Cascade : relances, attentes sans limite, ressources saturées.
3. Timeouts, recul exponentiel + gigue, disjoncteur, dégradation progressive.
4. RPO (données perdues) et RTO (temps de reprise) ; restaurer pour de vrai.
5. Chaos : hypothèse, injection contrôlée, observation, correction.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Funambule avec filet

Prochaine mission : Pipeline complet

