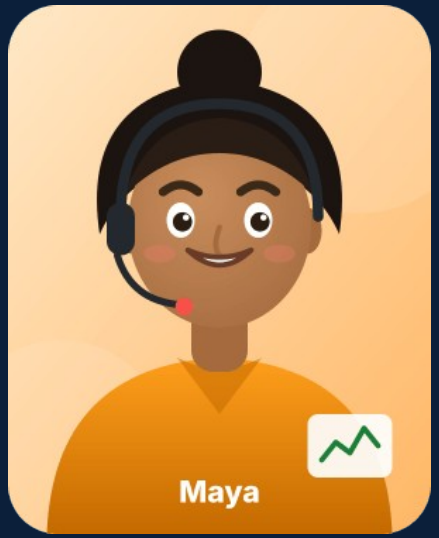
HEF-150 · MISSION 06 / 10

# Architecture orientée produit

*La bonne architecture n'est pas la plus élégante : c'est celle qui permet d'apprendre vite aujourd'hui sans interdire de grandir demain.*







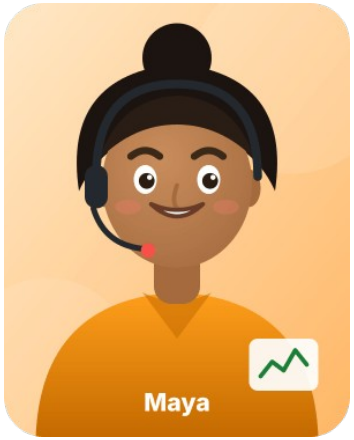
**Badge : Architecte pragmatique**

105 min · avancé

# L'histoire

Dès le premier jour de l'espace formateurs, un ingénieur propose une architecture en quatorze microservices, un bus d'événements, trois bases de données différentes et un orchestrateur. « C'est ce qu'il faudra quand on aura un million d'utilisateurs. » Aujourd'hui, HashBox en a quarante pour ce produit, et l'équipe compte quatre personnes. Pendant que l'on installe l'infrastructure, personne ne parle à un formateur.

*« Une architecture se juge par rapport à ce que le produit doit faire maintenant et à ce qu'il a une chance raisonnable de devenir, dit Sam. Elle doit rendre les décisions faciles à changer, surtout celles dont on sait qu'on se trompera. Et elle ne doit pas compliquer l'apprentissage. »*



L'histoire

# L'histoire (suite)

**Tu apprends à partir des attributs de qualité demandés par le produit, à comparer des options avec une matrice pondérée robuste, à décider entre construire et acheter, à choisir entre monolithe modulaire et microservices, à vérifier automatiquement des règles d'architecture, et à distinguer décisions réversibles et irréversibles.**

Ta mission

# À la fin, je serai capable de…

1. dériver les attributs de qualité d'une architecture des besoins du produit
2. comparer des options avec une matrice pondérée et tester sa robustesse aux poids
3. comparer construire et acheter sur un horizon de coût total de possession
4. comprendre monolithe modulaire, microservices et squelette ambulant, et leur coût
5. écrire des « fonctions d'aptitude » qui vérifient automatiquement des règles d'architecture

Défi d'échauffement

# Défi d'échauffement

Tu construis une maison pour une famille qui ne sait pas encore si elle aura un enfant, deux, ou aucun. Veux-tu faire dès maintenant cinq chambres et un sous-sol aménagé, ou une maison simple avec des murs porteurs bien placés qui permettent d'agrandir ? Qu'est-ce qui est facile à changer plus tard (peinture, mobilier), et qu'est-ce qui est très difficile (fondations, plan) ? Applique la distinction à un logiciel.

Le concept

# L'architecture au service du produit

L'**architecture** est l'ensemble des décisions **difficiles à changer** : structure en composants, choix des technologies principales, frontières de données, modes de déploiement. Elle découle de trois entrées :

| Entrée | Exemples pour HashBox |
|---|---|
| **Besoins fonctionnels** | publier un cours, s'inscrire, payer |
| **Attributs de qualité** | disponibilité, sécurité, performance, évolutivité, coût, facilité de changement |
| **Contraintes** | équipe de 4, six semaines, budget, conformité (RGPD), compétences |

## Les attributs de qualité, hiérarchisés

Le concept

# Les attributs de qualité, hiérarchisés (suite)

On ne peut pas tout maximiser. Un **arbre d'utilité** relie chaque qualité à des **scénarios précis** et chiffrés :

| Qualité | Scénario | Importance | Difficulté |
|---|---|---|---|
| Disponibilité | « le site répond à 99,5 % des requêtes sur 30 jours » | Élevée | Moyenne |
| Changeabilité | « ajouter un moyen de paiement en moins de 3 jours » | Élevée | Élevée |
| Performance | « page d'un cours en moins de 800 ms (p95) » | Moyenne | Faible |
| Coût | « moins de 0,05 € d'infrastructure par membre actif et par mois » | Moyenne | Moyenne |

Les qualités **importantes et difficiles** guident l'architecture. Pour un MVP, la **changeabilité** (apprendre vite) l'emporte presque toujours sur l'évolutivité extrême.

Le concept

# Comparer des options : la matrice pondérée

1. lister les **critères** (attributs de qualité, coût, risque) ;
2. attribuer un **poids** à chacun ;
3. noter chaque option sur chaque critère ;
4. calculer le score pondéré.

Piège : les poids sont des **opinions**. On teste la **robustesse** : si l'on fait varier les poids de ±30 %, le gagnant change-t-il ? Une décision stable face aux variations est solide ; une décision qui bascule demande plus d'information.

## Construire ou acheter

Le concept

# Construire ou acheter (suite)

| Construire | Acheter / réutiliser |
|---|---|
| contrôle total, différenciation | rapidité, maintenance par un tiers |
| coût initial élevé + maintenance | licence (par utilisateur ou forfait), dépendance |
| nécessaire pour ce qui est **différenciant** | judicieux pour ce qui est **banal** (paiement, authentification, e-mail) |

Comparer le **coût total de possession** (TCO) sur 3 ans : construire = développement + exploitation + maintenance ; acheter = licences (croissantes avec les utilisateurs) + intégration + dépendance (coût de sortie).

## Monolithe modulaire ou microservices ?

| | Monolithe modulaire | Microservices |
|---|---|---|
| Déploiement | un seul artefact | un par service |

Le concept

# Monolithe modulaire ou microservices ? (suite)

| | Monolithe modulaire | Microservices |
|---|---|---|
| Frontières | modules dans un code unique, règles de dépendance | réseau |
| Avantages | simple, rapide à changer, une transaction | équipes autonomes, mise à l'échelle indépendante |
| Coûts | discipline nécessaire | **toute la complexité distribuée** (HEF-140) : réseau, pannes, cohérence, observabilité |
| Quand | **équipe petite, produit en découverte** | grandes organisations, domaines stabilisés |

Loi de Conway : l'architecture reflète l'organisation. Règle utile : **commencer par un monolithe bien découpé en modules**, extraire un service seulement quand une raison précise le justifie (échelle différente, équipe distincte, contrainte de sécurité).

Le concept

# Le squelette ambulant

Une implémentation **minimale mais de bout en bout** qui traverse toutes les couches (interface, logique, stockage, déploiement, observation) : elle **valide l'architecture tôt** et permet de livrer dès la première semaine.

## Décisions réversibles et irréversibles

| Type | Définition | Attitude |
|---|---|---|
| **Porte à deux sens** | réversible à faible coût (choix d'une bibliothèque derrière une interface) | décider **vite** |
| **Porte à sens unique** | coûteuse à défaire (modèle de données, format public d'API, fournisseur de cloud) | décider **avec soin**, documenter |

Le concept

# Décisions réversibles et irréversibles (suite)

Stratégie : transformer les décisions irréversibles en réversibles — **interfaces**, **couches d'isolation**, **migrations progressives**.

## Fonctions d'aptitude (architecture évolutive)

Une **fonction d'aptitude** (*fitness function*) est un **test automatique** d'une propriété d'architecture :

| Propriété | Fonction d'aptitude |
|---|---|
| Frontières de modules | un module `domaine` ne peut pas importer `infrastructure` |
| Pas de cycles | le graphe de dépendances est acyclique |
| Taille | aucune fonction de plus de 60 lignes |
| Performance | temps de réponse p95 < budget en CI |
| Sécurité | aucune dépendance avec faille critique |

Le concept

# Fonctions d'aptitude (architecture évolutive) (suite)

Elles s'exécutent en intégration continue : l'architecture **ne se dégrade pas sans qu'on le voie**.

## Coût par utilisateur

Un produit doit être **viable** : `coût d'infrastructure par utilisateur actif` doit rester inférieur à `revenu par utilisateur`. Un modèle simple : coût fixe (environnement, supervision) + coût variable par utilisateur. Les microservices augmentent le **coût fixe** (chaque service a son environnement, sa supervision).

> 🧠 ***Astuce** : écris l'architecture comme des **décisions** (ADR) avec leurs **alternatives écartées** et **les signaux** qui te feraient la revoir (« si nous dépassons 50 000 membres actifs, extraire le service de paiement »).*

# Exemple pas à pas

**Problème** : comparer trois architectures par une matrice pondérée avec test de robustesse (simulation des poids), chiffrer construire contre acheter, appliquer des fonctions d'aptitude (dépendances interdites, cycles), et modéliser le coût par utilisateur d'un monolithe modulaire et de microservices.

```
import ast, random

# ---------------- 1. matrice pondérée et robustesse aux poids ----------------
CRITERES = ["changeabilité", "coût de démarrage", "disponibilité", "évolutivité",
   "simplicité d'exploitation"]
POIDS = {"changeabilité": 0.30, "coût de démarrage": 0.20, "disponibilité": 0.15,
   "évolutivité": 0.10, "simplicité d'exploitation": 0.25}      # produit en décou
   verte, équipe de 4
OPTIONS = {   # notes de 1 à 5
    "monolithe modulaire": {"changeabilité": 5, "coût de démarrage": 5, "disponibi
   lité": 3, "évolutivité": 3, "simplicité d'exploitation": 5},
    "microservices":       {"changeabilité": 3, "coût de démarrage": 1, "disponibi
   lité": 4, "évolutivité": 5, "simplicité d'exploitation": 1},
    "tout fournisseur (plateforme sans serveur)": {"changeabilité": 4, "coût de dé
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
   marrage": 4, "disponibilité": 5, "évolutivité": 4, "simplicité d'exploitation":
    5},
}
def score(option, poids): return sum(poids[c] * OPTIONS[option][c] for c in CRITER
   ES)
for o in OPTIONS: print(f"  {o:45s} {score(o, POIDS):.2f}")
gagnant = max(OPTIONS, key=lambda o: score(o, POIDS)); print("gagnant :", gagnant)

def robustesse(essais=5000, bruit=0.30, graine=1):
    rng, victoires = random.Random(graine), {o: 0 for o in OPTIONS}
    for _ in range(essais):
        w = {c: POIDS[c] * rng.uniform(1 - bruit, 1 + bruit) for c in CRITERES}
        total = sum(w.values()); w = {c: v / total for c, v in w.items()}
        # on renormalise
        victoires[max(OPTIONS, key=lambda o: score(o, w))] += 1
    return {o: v / essais for o, v in victoires.items()}
r = robustesse()
print("proportion de victoires quand les poids varient de ±30 % :", {o.split(" (")
   [0]: f"{v:.0%}" for o, v in r.items()})
```

# Exemple pas à pas (suite)

```
assert r["microservices"] < 0.02 and 0.2 < max(r.values()) < 0.95                #
    le gagnant dépend un peu des poids : on le sait

# ---------------- 2. construire ou acheter (coût total sur 36 mois) -------------
   ---
def tco_construire(mois, dev=60000, maintenance_mensuelle=2500, hebergement_mensue
   l=300):
    return dev + mois * (maintenance_mensuelle + hebergement_mensuel)
def tco_acheter(mois, utilisateurs, integration=8000, par_utilisateur=0.90, forfai
   t=600):
    total = integration
    for m in range(1, mois + 1):
        total += forfait + par_utilisateur * utilisateurs(m)
    return total
croissance = lambda m: 1000 * 1.08 ** m
    # +8 % de membres par mois
print("mois | construire | acheter")
point_de_bascule = None
for mois in (6, 12, 18, 24, 30, 36):
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
    c, a = tco_construire(mois), tco_acheter(mois, croissance)
    print(f" {mois:3d} |  {c:8.0f}  | {a:8.0f}  {'← acheter coûte plus' if a > c a
   nd point_de_bascule is None and not point_de_bascule else ''}")
    if a > c and point_de_bascule is None: point_de_bascule = mois
bascule = next(m for m in range(1, 200) if tco_acheter(m, croissance) > tco_constr
   uire(m))
print("l'achat devient plus cher que la construction à partir du mois", bascule)
assert tco_acheter(12, croissance) < tco_construire(12) and 24 <= bascule <= 40

# ---------------- 3. fonctions d'aptitude : dépendances interdites et cycles ----
   ------------
MODULES = {
    "domaine.cours": "from domaine.inscription import Inscription\nclass Cours: pa
   ss",
    "domaine.inscription": "import infrastructure.base\nclass Inscription: pass",
                # ❌ le domaine importe l'infrastructure
    "application.service": "from domaine.cours import Cours\nimport infrastructure
   .base",
    "infrastructure.base": "import sqlite3",
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
    "infrastructure.mail": "from application.service import creer",
                # ❌ cycle potentiel via application
}
def dependances(sources):
    graphe = {}
    for nom, code in sources.items():
        deps = set()
        for noeud in ast.walk(ast.parse(code)):
            if isinstance(noeud, ast.Import): deps |= {a.name for a in noeud.names
   }
            elif isinstance(noeud, ast.ImportFrom) and noeud.module: deps.add(noeu
   d.module)
        graphe[nom] = {d for d in deps if d in sources}
    return graphe
def verifier(sources, interdits):
    g, violations = dependances(sources), []
    for nom, deps in g.items():
        for d in deps:
            for source, cible in interdits:
```

# Exemple pas à pas (suite)

```
                if nom.startswith(source) and d.startswith(cible): violations.appe
   nd(f"{nom} → {d} (interdit : {source} ne doit pas dépendre de {cible})")
    # cycles
    etat = {}
    def visiter(n, chemin):
        etat[n] = 1
        for d in g[n]:
            if etat.get(d) == 1: violations.append("cycle : " + " → ".join(chemin
   + [n, d]))
            elif d not in etat: visiter(d, chemin + [n])
        etat[n] = 2
    for n in g:
        if n not in etat: visiter(n, [])
    return violations
for v in verifier(MODULES, [("domaine", "infrastructure"), ("domaine", "applicatio
   n")]): print("  ✗", v)
assert len(verifier(MODULES, [("domaine", "infrastructure"), ("domaine", "applicat
   ion")])) == 1
assert verifier({"domaine.cours": "class Cours: pass", "application.service": "fro
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
    m domaine.cours import Cours", "infrastructure.base": "import sqlite3"}, [("dom
    aine", "infrastructure")]) == []

# ---------------- 4. coût par utilisateur : monolithe modulaire contre microservi
    ces ----------------
def cout_mensuel(utilisateurs, services, fixe_par_service=350, variable_par_1000=1
    8, socle=250):
     return socle + services * fixe_par_service + variable_par_1000 * utilisateurs
    / 1000
print("membres actifs | monolithe (1 déploiement) | 12 microservices | coût par me
    mbre : monolithe / micro")
for u in (500, 5000, 50000, 500000):
     m, s = cout_mensuel(u, 1), cout_mensuel(u, 12)
     print(f"  {u:7d}      |        {m:9.0f} €         |    {s:9.0f} €    |  {m / u
    :6.3f} € / {s / u:6.3f} €")
assert cout_mensuel(500, 12) / 500 > 7 * cout_mensuel(500, 1) / 500 and cout_mensu
    el(500000, 12) / cout_mensuel(500000, 1) < 1.5
```

# Exemple pas à pas (suite)

**Lecture** :

1. La **matrice pondérée** place le monolithe modulaire en tête avec les poids d'un produit en découverte. Le **test de robustesse** montre que ce choix n'est pas absolument stable (la plateforme sans serveur gagne dans une partie des tirages) alors que les microservices ne gagnent presque jamais : l'information utile est la **marge**, pas le score.
2. **Construire ou acheter** : l'achat est moins cher pendant les deux premières années et demie (peu de membres), puis la licence croît avec les utilisateurs et dépasse le coût de construction vers le mois 32 : le choix dépend de l'horizon et de la croissance, et une **stratégie de sortie** s'impose.
3. Les **fonctions d'aptitude** détectent une dépendance interdite (le domaine qui importe l'infrastructure) automatiquement dans l'intégration continue.
4. Pour 500 membres, 12 microservices coûtent **plus de sept fois plus par membre** qu'un monolithe ; à 500 000 membres, l'écart n'est plus que d'environ 40 % : la complexité distribuée se justifie quand l'échelle ou les équipes l'exigent.

# À toi, avec aide

Une équipe hésite entre construire un système de notifications (développement : 40 000 €, maintenance : 1 500 €/mois, hébergement : 200 €/mois) et acheter un service (intégration : 5 000 €, 300 €/mois de forfait + 0,02 € par notification ; 200 000 notifications par mois au départ, +5 % par mois).

1. Calcule le coût cumulé des deux options à 12, 24, 36 et 48 mois.
2. À partir de quel mois construire devient-il moins cher ?
3. Que se passe-t-il si la croissance est de +10 % par mois ? De 0 % ?
4. Quel facteur non chiffré ici ferait pencher la décision vers acheter ou construire ?

À toi, avec aide · réponse

# Indices progressifs

## Indice 1

Cumule mois par mois dans une boucle.

## Indice 2

Le coût variable de l'achat croît avec le volume.

## Indice 3

Pense à la dépendance, au coût de sortie, à la différenciation.

# Réponse

```
def construire(mois): return 40000 + mois * (1500 + 200)
def acheter(mois, croissance):
    total, volume = 5000, 200000
    for m in range(mois):
        total += 300 + 0.02 * volume; volume *= 1 + croissance
    return total
for c in (0.05, 0.10, 0.0):
    bascule = next(m for m in range(1, 400) if construire(m) < acheter(m, c))
    print(f"croissance {c:4.0%} : construire devient moins cher au mois {bascule}"
   )
for m in (12, 24, 36, 48): print(f"{m} mois : construire {construire(m):8.0f} | ac
   heter {acheter(m, 0.05):8.0f}")
assert construire(36) < acheter(36, 0.05) < acheter(36, 0.10)
```

# Réponse (suite)

Plus la croissance est forte, plus la licence variable pèse et plus **construire** devient intéressant tôt ; à croissance nulle, l'achat reste compétitif longtemps. Facteurs non chiffrés : **différenciation** (si les notifications sont un avantage concurrentiel, construire), **dépendance et coût de sortie** (changer de fournisseur plus tard), **compétences** disponibles, **vitesse de mise sur le marché** (acheter permet de livrer en une semaine), **fiabilité** (un fournisseur spécialisé est-il meilleur que nous ?).

# À toi, en solo

**Vérificateur d'architecture modulaire.** Pour un monolithe modulaire représenté par un dictionnaire `chemin_module → code source` :

1. chaque module expose une **API publique** (liste `__all__`) ; les autres modules ne peuvent importer que des noms de cet `__all__` (`from billing import charge` autorisé si `charge` est dans `billing.__all__`) ;
2. règles de couches : `web → application → domaine`, `infrastructure → domaine` ; toute autre dépendance est interdite ;
3. détecte : import d'un nom privé, dépendance de couche interdite, cycle entre modules, module sans `__all__` ;
4. calcule des métriques de couplage par module : couplage afférent (qui dépend de moi), efférent (de qui je dépends), instabilité I = Ce / (Ca + Ce) ;
5. teste sur un projet d'exemple de 7 modules avec 4 violations injectées ; vérifie que chaque violation est détectée et qu'un projet propre ne produit aucune alerte.

À toi, en solo · réponse

# Pistes de correction

```
import ast

COUCHES = {"web": 0, "application": 1, "domaine": 2, "infrastructure": 1}
AUTORISE = {("web", "application"), ("application", "domaine"), ("infrastructure",
    "domaine"), ("web", "domaine")}

def analyse(sources):
    exports, graphe, noms_importes, alertes = {}, {m: set() for m in sources}, [],
    []
    for module, code in sources.items():
        arbre = ast.parse(code); tout = None
        for n in arbre.body:
            if isinstance(n, ast.Assign) and any(getattr(t, "id", "") == "__all__"
    for t in n.targets): tout = {e.value for e in n.value.elts}
        if tout is None: alertes.append(f"{module} : pas d'API publique (__all__)"
   ); tout = set()
        exports[module] = tout
        for n in ast.walk(arbre):
            if isinstance(n, ast.ImportFrom) and n.module in sources:
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
                graphe[module].add(n.module); noms_importes += [(module, n.module,
 a.name) for a in n.names]
            elif isinstance(n, ast.Import):
                for a in n.names:
                    if a.name in sources: graphe[module].add(a.name)
 for src, cible, nom in noms_importes:
    if nom not in exports[cible]: alertes.append(f"{src} importe le nom privé
{cible}.{nom}")
 for src, deps in graphe.items():
    for d in deps:
        couche_src, couche_d = src.split(".")[0], d.split(".")[0]
        if couche_src != couche_d and (couche_src, couche_d) not in AUTORISE:
alertes.append(f"couche interdite : {src} → {d}")
 etat = {}
 def visiter(n, chemin):
    etat[n] = 1
    for d in graphe[n]:
        if etat.get(d) == 1: alertes.append("cycle : " + " → ".join(chemin + [
n, d]))
```

# Pistes de correction (suite)

```
            elif d not in etat: visiter(d, chemin + [n])
        etat[n] = 2
    for n in graphe:
        if n not in etat: visiter(n, [])
    ca = {m: sum(1 for d in graphe.values() if m in d) for m in graphe}
    ce = {m: len(graphe[m]) for m in graphe}
    instabilite = {m: (ce[m] / (ca[m] + ce[m]) if ca[m] + ce[m] else 0.0) for m in
    graphe}
    return alertes, instabilite

PROPRE = {
    "web.pages": "from application.service import inscrire\n__all__ = ['afficher']
   ",
    "application.service": "from domaine.cours import Cours\n__all__ = ['inscrire'
   ]",
    "domaine.cours": "__all__ = ['Cours']\nclass Cours: pass",
    "infrastructure.base": "from domaine.cours import Cours\n__all__ = ['sauver']"
   ,
}
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
alertes, inst = analyse(PROPRE)
print("projet propre :", alertes or "aucune alerte", "| instabilité :", {m: round(
   i, 2) for m, i in inst.items()})
assert alertes == [] and inst["domaine.cours"] == 0.0
SALE = dict(PROPRE)
SALE["domaine.cours"] = "from infrastructure.base import sauver\n__all__ = ['Cours
   ']\nclass Cours: pass"           # couche interdite + cycle
SALE["web.pages"] = "from application.service import _brouillon\n__all__ = ['affic
   her']"                          # nom privé
SALE["infrastructure.mail"] = "import smtplib"
                                   # pas de __all__
alertes, _ = analyse(SALE)
for a in alertes: print("  ✗", a)
assert any("nom privé" in a for a in alertes) and any("couche interdite" in a for
   a in alertes) and any("cycle" in a for a in alertes) and any("pas d'API publiqu
   e" in a for a in alertes)
```

À toi, en solo · réponse

# Pistes de correction (suite)

L'**instabilité** `I = Ce/(Ca+Ce)` vaut 0 pour un module dont tout le monde dépend et qui ne dépend de personne (le domaine : stable), proche de 1 pour un module qui dépend de beaucoup et dont personne ne dépend (une page web : instable). Le principe : **les dépendances vont vers la stabilité**. Ces contrôles tournent en intégration continue et empêchent la lente dégradation de la structure.

# Pièges classiques

| Piège | Pourquoi c'est un problème | Comment l'éviter |
|---|---|---|
| **Concevoir pour un million d'utilisateurs avec quarante** | Complexité et coût prématurés, apprentissage retardé | Monolithe modulaire, squelette ambulant, signaux de revue |
| **Choisir une technologie par mode** | Décision irréversible non justifiée | Critères, matrice robuste, ADR avec alternatives |
| **Matrice pondérée avec des poids intouchables** | Fausse précision | Tester la robustesse aux poids |
| **Règles d'architecture seulement écrites** | Elles s'érodent | Fonctions d'aptitude en intégration continue |

# Mini-boss

**Dossier d'architecture du MVP HashBox « espace formateurs ».**

1. arbre d'utilité : 6 scénarios de qualité chiffrés avec importance et difficulté ; identifie les 2 qui guident l'architecture ;
2. matrice pondérée de 3 options (monolithe modulaire, microservices, plateforme gérée) avec **test de robustesse** (5000 tirages) ; conclusion argumentée ;
3. analyse **construire / acheter** pour deux composants (paiement, recherche) avec TCO sur 36 mois et sensibilité à la croissance ;
4. définition du **squelette ambulant** (5 composants minimaux) et plan de livraison en 2 semaines ;
5. au moins 5 **fonctions d'aptitude** exécutables (frontières, cycles, taille de fonction, présence de tests, dépendances interdites) appliquées à un petit projet de ton choix ;
6. classification des 8 décisions principales en **portes à deux sens / à sens unique**, avec pour chacune du sens unique une façon de la rendre réversible ;

Mini-boss

# Mini-boss (suite)

7. 3 ADR avec alternatives écartées et **signaux de revue** chiffrés (« si X dépasse Y, extraire Z »).

Mini-boss · réponse

# Pistes de correction

Un bon dossier : scénarios de qualité mesurables, matrice robuste (gagnant et probabilité), TCO avec courbes et bascule, squelette ambulant crédible, fonctions d'aptitude qui échouent quand on viole une règle, tableau de réversibilité (le format public de l'API et le modèle de données sont à sens unique : on les protège par des versions et des migrations progressives), ADR avec signaux de revue précis (par exemple « extraire la facturation en service si l'équipe dépasse 8 personnes ou si le trafic de facturation dépasse 10 % de la CPU »).

Quiz éclair

# Quiz éclair

**1.** Pour un produit en découverte avec une petite équipe, l'architecture recommandée est généralement :
A. quatorze microservices dès le premier jour
B. un monolithe modulaire, avec possibilité d'extraire des services plus tard
C. aucune structure

**2.** Une fonction d'aptitude (fitness function) est :

A. une métrique de performance du processeur
B. un algorithme génétique
C. un test automatique d'une propriété d'architecture

**3.** Une décision « à sens unique » est une décision :

A. coûteuse à défaire, à prendre avec soin

Quiz éclair

# Quiz éclair (suite)

B. qu'on peut changer en cinq minutes
C. prise par une seule personne

**4.** 🔁 *Révision mission 05* : Un test d'utilisabilité avec 5 personnes sert surtout à :

A. mesurer précisément un taux de réussite
B. découvrir des problèmes d'utilisabilité majeurs
C. remplacer les données d'usage

**5.** 🔁 *Révision mission 05* : Le rapport de contraste minimal AA pour un texte normal est :

A. 2:1
B. 10:1
C. 4,5:1

Quiz éclair · réponse

# Réponses

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

# Je réfléchis

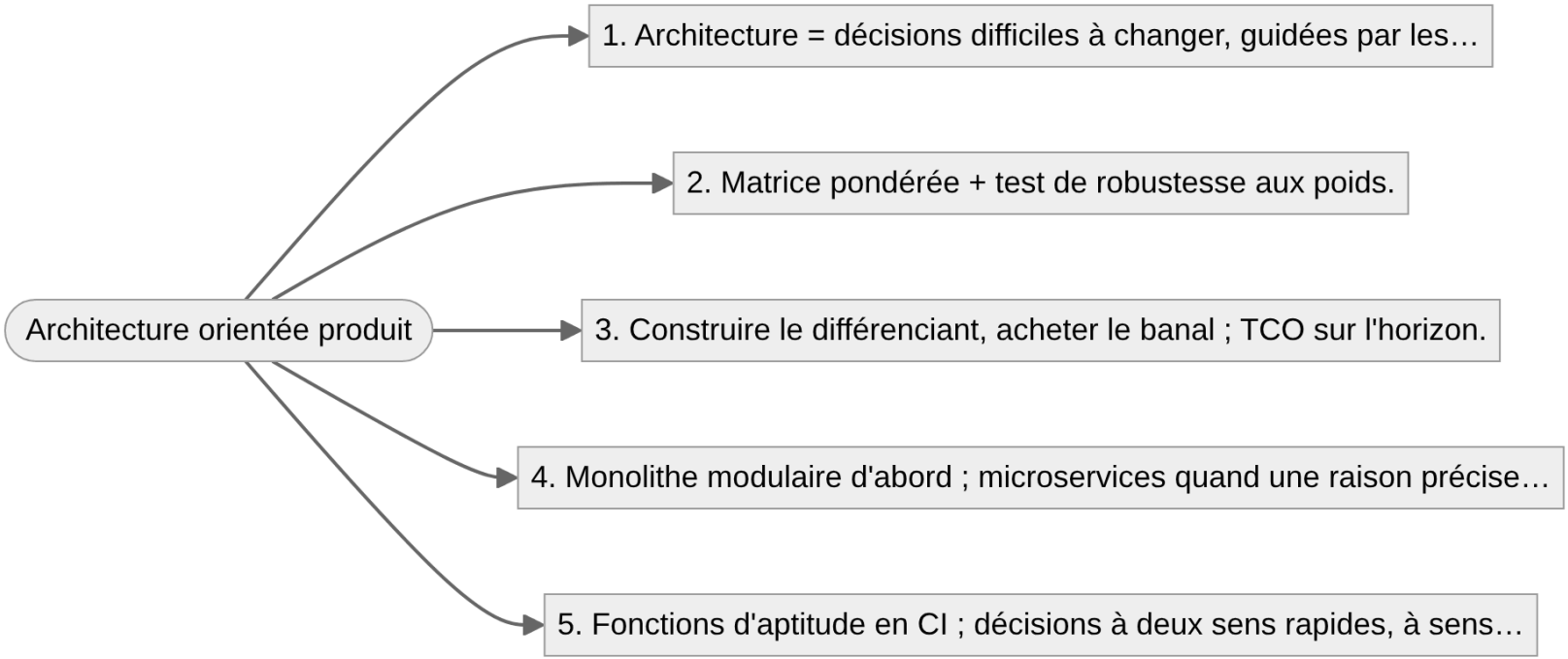
1. Quelle décision d'architecture de ton produit serait très coûteuse à défaire, et l'as-tu documentée ?
2. Quelle complexité de ton système répond à un problème d'échelle que tu n'as pas encore ?

# À retenir

1. Architecture = décisions difficiles à changer, guidées par les qualités importantes et difficiles.
2. Matrice pondérée + test de robustesse aux poids.
3. Construire le différenciant, acheter le banal ; TCO sur l'horizon.
4. Monolithe modulaire d'abord ; microservices quand une raison précise l'exige.
5. Fonctions d'aptitude en CI ; décisions à deux sens rapides, à sens unique soignées.

À retenir

# Le schéma de la mission



MISSION ACCOMPLIE

# +100 XP

## Badge débloqué : Architecte pragmatique

Prochaine mission : Priorisation et roadmap

