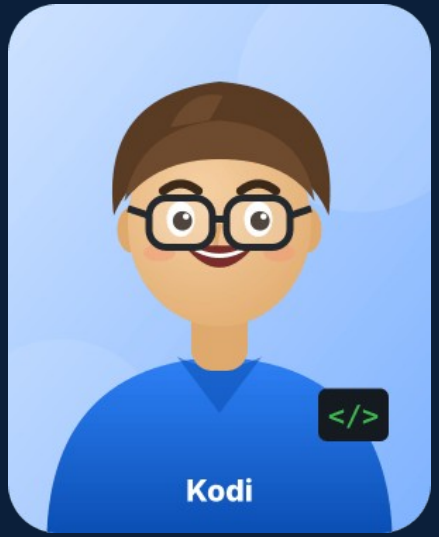
HEF-130 · MISSION 01 / 10

Modèles de performance

Avant de chercher à aller plus vite, il faut comprendre ce que « vite » veut dire, et pourquoi les systèmes ralentissent d'un coup au lieu de ralentir doucement.



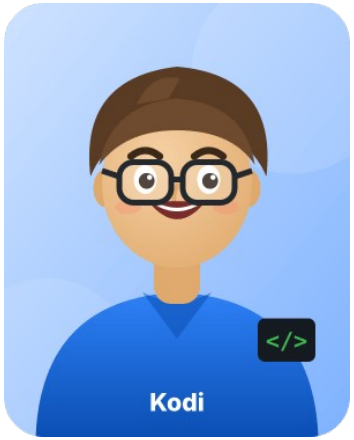


Badge : Cartographe de la lenteur

90 min · avancé

L'histoire

Le jeudi des inscriptions, HashBox ne tombe pas en panne : elle ralentit. À 9 h, les pages s'affichent en 200 ms. À 9 h 20, en 800 ms. À 9 h 30, en 12 secondes. Pourtant le trafic n'a augmenté que de 15 % depuis 9 h 20. Kodi est perplexe : « C'est une courbe, pas une marche. Pourquoi ça explose ? »



> *« Parce qu'un système qui approche de sa capacité ne se comporte pas de façon linéaire, dit Sam. Une file d'attente à 60 % d'utilisation se vide vite. À 95 %, elle ne se vide plus. Il existe des lois simples qui expliquent cela, et on peut les calculer avant de construire. »*

Tu apprends les vocabulaire de la performance (latence, débit, utilisation, saturation), la loi de Little, la loi d'Amdahl, et le comportement d'une file d'attente — avec des simulations pour les voir de tes yeux.

Ta mission

À la fin, je serai capable de…

1. distinguer latence, débit, utilisation et saturation
2. appliquer la loi de Little pour relier concurrence, débit et latence
3. expliquer pourquoi la latence explose près de la saturation (file d'attente)
4. utiliser la loi d'Amdahl pour borner le gain d'un parallélisme
5. raisonner avec des ordres de grandeur de latences matérielles

Défi d'échauffement

Un guichet de poste sert un client toutes les 2 minutes. Si les clients arrivent en moyenne toutes les 4 minutes, la file reste courte. S'ils arrivent toutes les 2,1 minutes ? Toutes les 2 minutes exactement ? Toutes les 1,9 minutes ? Prédis ce qui arrive à la file dans chacun des quatre cas avant de lire la suite.

Le concept

Le vocabulaire

Terme	Définition	Unité
Latence (*latency*)	temps pour **une** opération, de la demande à la réponse	ms
Débit (*throughput*)	nombre d'opérations terminées par unité de temps	requêtes/s
Temps de service	temps de traitement d'une opération **sans attente**	ms
Temps d'attente	temps passé dans la file avant d'être servi	ms
Utilisation	part du temps où la ressource est occupée	%
Saturation	travail en attente parce que la ressource est pleine	longueur de file
Capacité	débit maximal soutenable	requêtes/s

Latence totale = temps d'attente + temps de service.

Le concept

Le vocabulaire (suite)

Latence et débit sont **deux dimensions distinctes** : un camion transporte beaucoup (haut débit) mais lentement (latence élevée) ; une moto est rapide mais transporte peu.

Ordres de grandeur à connaître

Opération	Ordre de grandeur
Accès registre / cache L1	~1 ns
Accès mémoire vive	~100 ns
Lecture aléatoire SSD	~100 µs
Aller-retour dans un centre de données	~0,5 ms
Lecture aléatoire disque dur	~10 ms
Aller-retour Europe ↔ Amérique	~100 ms

Le concept

Ordres de grandeur à connaître (suite)

Chaque niveau est **10 à 1000 fois** plus lent que le précédent. Ce qu'on fait **moins souvent** (évite un accès disque, un aller-retour réseau) pèse plus que ce qu'on fait **plus vite**.

La loi de Little

Pour tout système **stable** :

```
L = λ × W
```

- `L` : nombre moyen de requêtes **dans le système** (concurrence) ;
- `λ` : débit moyen d'arrivée (requêtes/s) ;
- `W` : temps moyen passé dans le système (latence).

Le concept

La loi de Little (suite)

Exemples : 100 requêtes/s avec une latence de 200 ms → **20 requêtes en cours** en moyenne. Une base qui accepte 50 connexions et dont les requêtes durent 100 ms ne peut pas dépasser **500 requêtes/s**.

Pourquoi la latence explose : la file d'attente

Pour une file simple (un serveur, arrivées aléatoires, services aléatoires — modèle **M/M/1**), avec ρ l'utilisation (λ/μ) :

```
temps d'attente moyen = ρ / (μ − λ) = ρ / (1 − ρ) × temps de service
```

Utilisation ρ	Attente / temps de service
50 %	1 ×
80 %	4 ×
90 %	9 ×

Le concept

Pourquoi la latence explose : la file d'attente (suite)

Utilisation ρ	Attente / temps de service
95 %	19 ×
99 %	99 ×

La croissance est **hyperbolique** : près de 100 %, un petit supplément de trafic multiplie l'attente. Règle pratique : **garder l'utilisation de la ressource critique sous 60-70 %**, et prévoir des marges (capacité N+1).

La loi d'Amdahl

Si une fraction p d'un programme peut être parallélisée sur n unités, l'accélération maximale est :

Le concept

La loi d'Amdahl (suite)

```
accélération = 1 / ((1 − p) + p / n)
```

Si 90 % du programme se parallélise, même avec une infinité de cœurs, on ne dépasse pas **×10**. La partie **séquentielle** (10 %) fixe la limite. Corollaire : avant d'ajouter des machines, cherche la partie qui ne se parallélise pas.

Le coût de la coordination (loi d'extensibilité universelle)

Dans la réalité, ajouter des travailleurs a aussi un coût de **contention** (verrous partagés) et de **cohérence** (synchronisation). Au-delà d'un point, **plus de ressources = moins de débit**. La courbe monte, atteint un plafond, puis redescend.

Modèles mentaux utiles

- **Où va le temps ?** CPU, mémoire, disque, réseau, attente de verrous, attente d'un service tiers.

Le concept

Modèles mentaux utiles (suite)

- **Goulot d'étranglement** : la ressource la plus saturée fixe le débit global (théorie des contraintes, HEF-120).
- **Moyenne et queue** : la moyenne cache les requêtes lentes (mission 2).

🧠 ***Astuce*** *: avant d'optimiser, pose-toi trois questions : quelle est la ressource qui sature ? à quel taux d'utilisation ? et qu'est-ce qui attend ? La réponse oriente tout le reste.*

Exemple pas à pas

Problème : vérifier par simulation la loi de Little et la formule de la file M/M/1, voir l'explosion de la latence près de la saturation, et calculer la limite d'Amdahl.

```
import random

def simuler_file(lambda_, mu, n=500000, graine=1):
    # file à un serveur : arrivées de Poisson (taux lambda_), services exponentiel
   s (taux mu)
    rng = random.Random(graine)
    t_arrivee, libre_a = 0.0, 0.0
    attentes, residences = [], []
    for _ in range(n):
        t_arrivee += rng.expovariate(lambda_)
        debut = max(t_arrivee, libre_a)                  # on attend si le serveur
    est occupé
        service = rng.expovariate(mu)
        libre_a = debut + service
        attentes.append(debut - t_arrivee)
```

Exemple pas à pas (suite)

```
        residences.append(libre_a - t_arrivee)          # temps total dans le sys
    tème
     return sum(attentes) / n, sum(residences) / n, t_arrivee

MU = 100.0                                              # le serveur sait traiter
     100 requêtes/s (service moyen : 10 ms)
print("utilisation | attente simulée | attente théorique | latence moyenne")
resultats = {}
for rho in (0.5, 0.8, 0.9, 0.95):
    attente, residence, duree = simuler_file(rho * MU, MU)
    theorique = rho / (MU - rho * MU)                   # ρ / (μ − λ)
    resultats[rho] = (attente, residence)
    print(f"   {rho:4.0%}     |   {attente * 1000:6.1f} ms    |    {theorique * 10
   00:6.1f} ms     |   {residence * 1000:6.1f} ms")
    assert abs(attente - theorique) / theorique < (0.08 if rho < 0.95 else 0.12)
      # la simulation confirme la formule (à quelques % près ; plus de bruit près
   de la saturation)

# la latence explose : à 95 % d'utilisation, elle est beaucoup plus grande qu'à 50
```

Exemple pas à pas (suite)

```
    %
assert resultats[0.95][1] > 8 * resultats[0.5][1]

# ---- loi de Little : L = λ × W ----
lambda_, mu = 80.0, 100.0
attente, residence, duree = simuler_file(lambda_, mu, graine=2)
# nombre moyen dans le système, calculé en intégrant dans le temps via les événeme
   nts (arrivées +1, départs −1)
rng = random.Random(2)
t, libre = 0.0, 0.0
evenements = []
for _ in range(100000):
    t += rng.expovariate(lambda_)
    debut = max(t, libre); libre = debut + rng.expovariate(mu)
    evenements += [(t, +1), (libre, -1)]
evenements.sort()
aire, courant, precedent = 0.0, 0, 0.0
for temps, delta in evenements:
    aire += courant * (temps - precedent); courant += delta; precedent = temps
```

Exemple pas à pas (suite)

```
L_simule = aire / precedent
print(f"Little : λ × W = {lambda_ * residence:.2f} ; L mesuré = {L_simule:.2f}")
assert abs(L_simule - lambda_ * residence) / L_simule < 0.05

# ---- capacité avec contrainte de connexions ----
connexions, duree_requete = 50, 0.100
print("débit maximal d'une base à 50 connexions et 100 ms par requête :", connexio
   ns / duree_requete, "req/s")
assert connexions / duree_requete == 500

# ---- loi d'Amdahl ----
def amdahl(p, n):
    return 1 / ((1 - p) + p / n)

for p in (0.5, 0.9, 0.99):
    print(f"p = {p:4.0%} : " + "  ".join(f"{n:>5d} cœurs → x{amdahl(p, n):6.2f}" f
   or n in (2, 8, 64, 1024)) + f"   | limite : x{1 / (1 - p):.0f}")
assert round(amdahl(0.9, 10 ** 9)) == 10 and amdahl(0.5, 10 ** 9) < 2.0001
```

Exemple pas à pas (suite)

Lecture :

1. La **simulation** de la file M/M/1 confirme la formule $\rho/(\mu-\lambda)$: à 50 % d'utilisation l'attente moyenne est de 10 ms (un temps de service), à 90 % de 90 ms, à 95 % de 190 ms. La latence ne croît pas linéairement : elle **explose** près de la saturation.
2. La **loi de Little** est vérifiée : le nombre moyen de requêtes dans le système est bien $\lambda \times W$.
3. Une base de **50 connexions** avec des requêtes de **100 ms** plafonne à **500 requêtes/s**, quelle que soit sa puissance de calcul : c'est la loi de Little appliquée à une contrainte.
4. **Amdahl** : avec 90 % de code parallélisable, 1024 cœurs ne donnent qu'environ ×9,9 ; avec 50 %, jamais plus de ×2.

À toi, avec aide

Applique la loi de Little à trois situations :

1. Un service reçoit **300 requêtes/s**, chacune dure **120 ms** en moyenne. Combien de requêtes sont en cours simultanément ? Combien de threads minimum faut-il pour les absorber ?
2. Un pool de **40 connexions** à la base. Chaque requête garde une connexion **25 ms**. Quel est le débit maximal ? Si le trafic atteint 2000 requêtes/s, que se passe-t-il ?
3. Un restaurant de **60 places** où chaque client reste en moyenne **45 minutes**. Combien de clients peut-il servir par heure au maximum ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

L = λ × W : convertis tout en secondes.

Indice 2

Le débit maximal d'un pool = taille du pool / durée d'utilisation.

Indice 3

Pour le restaurant : λ = L / W.

À toi, avec aide · réponse

Réponse

```
print("1.", 300 * 0.120, "requêtes en cours (≥ 36 threads, plus une marge)")
print("2.", 40 / 0.025, "req/s maximum")
print("3.", 60 / 0.75, "clients par heure")
assert round(300 * 0.120) == 36 and 40 / 0.025 == 1600 and 60 / 0.75 == 80
```

1. **36 requêtes** en cours en moyenne : il faut au moins 36 threads (en pratique 50-60 pour absorber les pics).
2. Le pool plafonne à **1600 req/s**. À 2000 req/s, 400 req/s de travail ne trouvent pas de connexion : une **file d'attente** se forme et grandit sans limite → timeouts. Remède : augmenter le pool (si la base le supporte), réduire le temps de requête, ou refuser l'excès (délestage).
3. **80 clients par heure** (60 places / 0,75 h). C'est une capacité : pour accueillir plus de monde, il faut réduire la durée de séjour ou ajouter des places.

À toi, en solo

Dimensionnement d'un service. Un service a un temps de service moyen de **20 ms** par requête (un cœur). On veut tenir un **trafic de pointe de 900 requêtes/s** avec une latence moyenne **inférieure à 60 ms**.

1. Calcule avec la formule M/M/1 la latence moyenne (attente + service) d'un serveur pour un taux d'arrivée donné ;
2. répartis le trafic de façon égale sur k serveurs identiques et trouve le **plus petit `k`** qui respecte la contrainte de 60 ms ;
3. calcule l'utilisation par serveur ;
4. ajoute la contrainte de **tolérance à une panne** (N+1) : combien de serveurs prévoir pour tenir encore la contrainte si l'un tombe ?
5. vérifie par simulation (un serveur à la charge calculée) que la latence moyenne est bien sous 60 ms.

À toi, en solo · réponse

Pistes de correction

```
import random

def latence_mm1(lambda_, mu):
    if lambda_ >= mu:
        return float("inf")
    return 1 / (mu - lambda_)                                  # attente + service

MU, TRAFIC, CIBLE = 50.0, 900.0, 0.060                         # 20 ms de service → μ
   = 50 req/s par serveur
k = 1
while latence_mm1(TRAFIC / k, MU) > CIBLE:
    k += 1
rho = TRAFIC / k / MU
print(f"minimum {k} serveurs, utilisation {rho:.0%}, latence {latence_mm1(TRAFIC /
    k, MU) * 1000:.1f} ms")

k_nplus1 = k + 1
while latence_mm1(TRAFIC / (k_nplus1 - 1), MU) > CIBLE:    # il doit tenir avec un
    serveur en moins
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    k_nplus1 += 1
print(f"avec tolérance à une panne : {k_nplus1} serveurs")
assert k == 28 and rho < 0.7

# vérification par simulation
def simuler(lambda_, mu, n=150000, graine=3):
    rng = random.Random(graine); t = libre = 0.0; total = 0.0
    for _ in range(n):
        t += rng.expovariate(lambda_); debut = max(t, libre); libre = debut + rng.
   expovariate(mu); total += libre - t
    return total / n

simulee = simuler(TRAFIC / k, MU)
print(f"latence simulée : {simulee * 1000:.1f} ms")
assert simulee < CIBLE * 1.1
```

À toi, en solo · réponse

Pistes de correction (suite)

Le minimum est **28 serveurs** (utilisation ≈ 64 %, latence théorique ≈ 56 ms). Pour tenir encore la contrainte **si un serveur tombe**, il en faut **29** (les 28 restants suffisent). Le résultat illustre la règle pratique : on ne dimensionne **jamais** à 100 % — on vise 60-75 % et on garde une marge de panne.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Confondre latence et débit	Optimiser l'un en dégradant l'autre sans s'en rendre compte	Mesurer les deux, savoir lequel est la contrainte
Dimensionner à 100 % d'utilisation	La file explose	Viser 60-75 %, marge de panne
Raisonner en moyenne	La queue de latence est invisible	Percentiles (mission 2)
Ajouter des machines sans chercher le goulot	Amdahl : la partie séquentielle limite	Identifier ce qui ne se parallélise pas

Mini-boss

Modèle de capacité de HashBox. L'architecture : un équilibreur, 4 serveurs d'application (20 ms de CPU par requête chacun, 2 cœurs), une base (5 ms par requête, un seul « serveur » logique) et un service de paiement tiers (150 ms, appelé pour 10 % des requêtes).

1. calcule pour chaque composant le débit maximal et la latence moyenne en fonction du trafic (formule M/M/c ou M/M/1 approchée) ;
2. identifie le **goulot** et le trafic au-delà duquel la latence dépasse 300 ms ;
3. écris une simulation par événements discrets de toute la chaîne et compare avec ton modèle ;
4. applique Little pour dimensionner les pools de connexions ;
5. propose 3 changements, chiffre leur effet (Amdahl ou file d'attente) et classe-les par gain/effort ;
6. trace (tableau texte) latence moyenne contre trafic de 100 à 1000 requêtes/s.

Mini-boss · réponse

Pistes de correction

Un bon travail : une fonction par composant (latence = f(trafic)), un tableau où l'on voit la latence plafonner puis diverger au trafic du goulot (ici, probablement les serveurs d'application ou la base selon les paramètres), une simulation qui confirme le modèle à quelques pourcents, des pools dimensionnés par Little (`concurrence = débit × latence`), et trois améliorations (par exemple appeler le paiement en asynchrone, réduire le CPU par requête, ajouter un serveur) avec les gains estimés.

Quiz éclair

Quiz éclair

1. La loi de Little relie :

A. concurrence, débit et latence moyenne
B. CPU, mémoire et disque
C. coût, délai et qualité

2. Près de 100 % d'utilisation d'une ressource, la latence :

A. reste stable
B. explose, car la file d'attente grandit rapidement
C. diminue

3. D'après Amdahl, si 90 % d'un programme est parallélisable, l'accélération maximale est :
A. infinie

Quiz éclair

Quiz éclair (suite)

B. ×90
C. environ ×10

4. 🔁 *Révision HEF-120, mission 06* : Un pipeline CI est modélisé comme :

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

5. 🔁 *Révision HEF-120, mission 09* : Un disjoncteur sert à :

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

Quiz éclair · réponse

Réponses

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

Je réfléchis

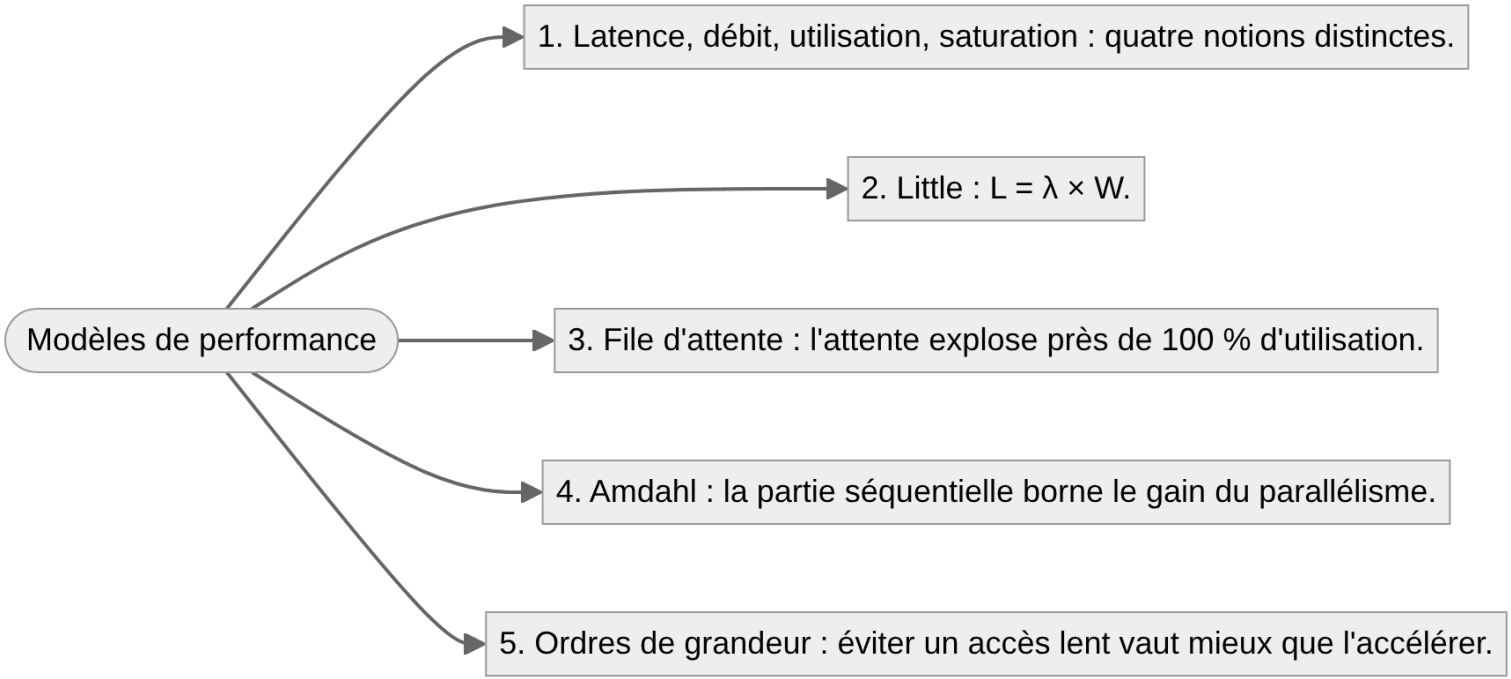
1. Quelle ressource de ton système approche le plus de sa capacité, et comment le saurais-tu ?
2. Pourquoi est-ce raisonnable de laisser 30 % de capacité « inutilisée » ?

À retenir

1. Latence, débit, utilisation, saturation : quatre notions distinctes.
2. Little : $L = \lambda \times W$.
3. File d'attente : l'attente explose près de 100 % d'utilisation.
4. Amdahl : la partie séquentielle borne le gain du parallélisme.
5. Ordres de grandeur : éviter un accès lent vaut mieux que l'accélérer.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Cartographe de la lenteur

Prochaine mission : Métriques et budgets

