Synthèse des résultats d’analyse de Graphana et k6

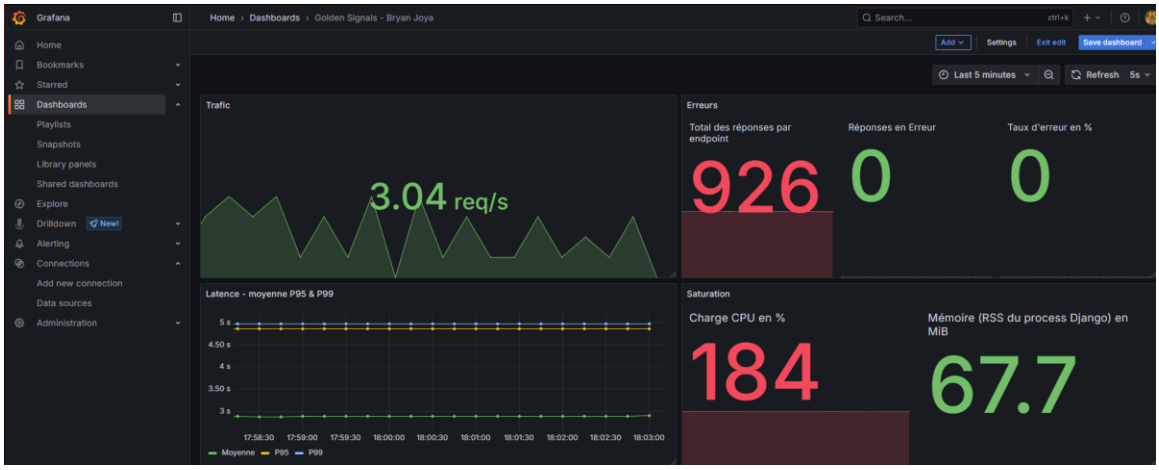
Bryan Joya JOYB74080104

1. Analyse de la latence avant et après le Load Balancer

Avant le load balancer:

Lancement du test de charge Load-test.js :

```
THRESHOLDS

http_req_duration
✗ 'p(95)<500' p(95)=7.01s

http_req_failed
✓ 'rate<0.01' rate=0.00%

TOTAL RESULTS

checks_total.....................: 921     2.980015/s
checks_succeeded...................: 100.00% 921 out of 921
checks_failed.....................: 0.00%   0 out of 921

✓ stock 200
✓ rapport 200
✓ update 200

HTTP
http_req_duration.....................................................: avg=6.29s  min=1.54s med=6.4s   max=7.96s  p(90)=6.85
  { expected_response:true }..........................................: avg=6.29s  min=1.54s med=6.4s   max=7.96s  p(90)=6.85
http_req_failed.....................................................: 0.00%  0 out of 921
http_reqs.....................................................: 921    2.980015/s

EXECUTION
iteration_duration.....................................................: avg=19.87s min=9.4s  med=20.11s max=21.42s p(90)=20.8
iterations.....................................................: 307    0.993338/s
vus.....................................................: 2      min=2        max=20
vus_max.....................................................: 20     min=20       max=20

NETWORK
data_received.....................................................: 2.4 MB 7.7 kB/s
data_sent.....................................................: 183 kB 592 B/s
```



Avant son déploiement, les tests de charge indiquaient une latence élevée, avec une durée moyenne de requête à 6,29 secondes et un pic à 7,96 secondes. Toutefois, aucun échec de requête n'a été détecté. Le nombre total de requêtes (926) et la vitesse d'exécution (3.11req par secondes) indiquent une activité soutenue. On peut observer que 184% du CPU est utilisé par la machine.

Lancement du test de charge Load-test-2.js :

```
THRESHOLDS

  http_req_duration
  ✗ 'p(95)<500' p(95)=1m0s

  http_req_failed
  ✗ 'rate<0.01' rate=31.32%

TOTAL RESULTS

  HTTP
  http_req_duration.....................................................: avg=27.46s min=1.64s  med=23.6s max=1m0s   p(90)=1m0s
    { expected_response:true }...........................................: avg=16.08s min=1.64s  med=15.6s max=43.99s p(90)=30.2
  http_req_failed.......................................................: 31.32% 702 out of 2241
  http_reqs.............................................................: 2241   3.304146/s

  EXECUTION
  iteration_duration....................................................: avg=1m19s  min=10.26s med=1m21s max=3m1s   p(90)=2m22
  iterations............................................................: 690    1.017341/s
  vus...................................................................: 5      min=5          max=200
  vus_max...............................................................: 200    min=200        max=200

  NETWORK
  data_received.........................................................: 83 MB  123 kB/s
  data_sent.............................................................: 466 kB 687 B/s
```

Sous de 200 utilisateurs concurrents, l'application ne satisfait ni le critère de latence (p(95) < 500 ms) ni celui du taux d'échec (rate < 1 %) : en effet, 31 % des requêtes sont interrompues et la latence du 95e centile atteint systématiquement 60 s, valeur qui correspond au timeout par défaut de k6. Ces timeouts massifs indiquent que, sous cette charge, le serveur ne peut pas traiter les requêtes dans les délais et que beaucoup d'itérations restent inachevées.

Analyse des points faibles et recommandations d'optimisation

- Pool de connexions saturé
    - Nombre de connexions ouvertes atteint rapidement le maximum configuré → file d'attente, latences accrues.
    - Pas de timeout ni de « recycle » sur les connexions inactives.
    - Absence ou insuffisance d'index
    - Requêtes de lecture sur les tables volumineuses (produits, stocks, rapports) déclenchent des parcours séquentiels (table scan).
- Requêtes SQL sous-optimales
- Absence de mise en cache
    - Appels répétés au même endpoint /rapport/ ou à la même ressource stock/produit sans cache → requêtes identiques refaites à chaque VU.
    - Aucun mécanisme de cache HTTP ni cache applicatif (Redis, in-memory).

Recommandations d'amélioration

- Pool de connexions
    - Activer un « connection lifetime » pour fermer les connexions longues
- Mise en cache
    - Côté API, ajouter un cache en mémoire ou Redis pour les réponses fréquentes.
    - Activer le cache HTTP pour permettre aux clients et aux proxy de conserver les résultats.
    - Mettre en place un cache de requêtes SQL pour éviter la recompilation répétée.

Après le load balancer CONFIGURATION ROUND ROBIN:

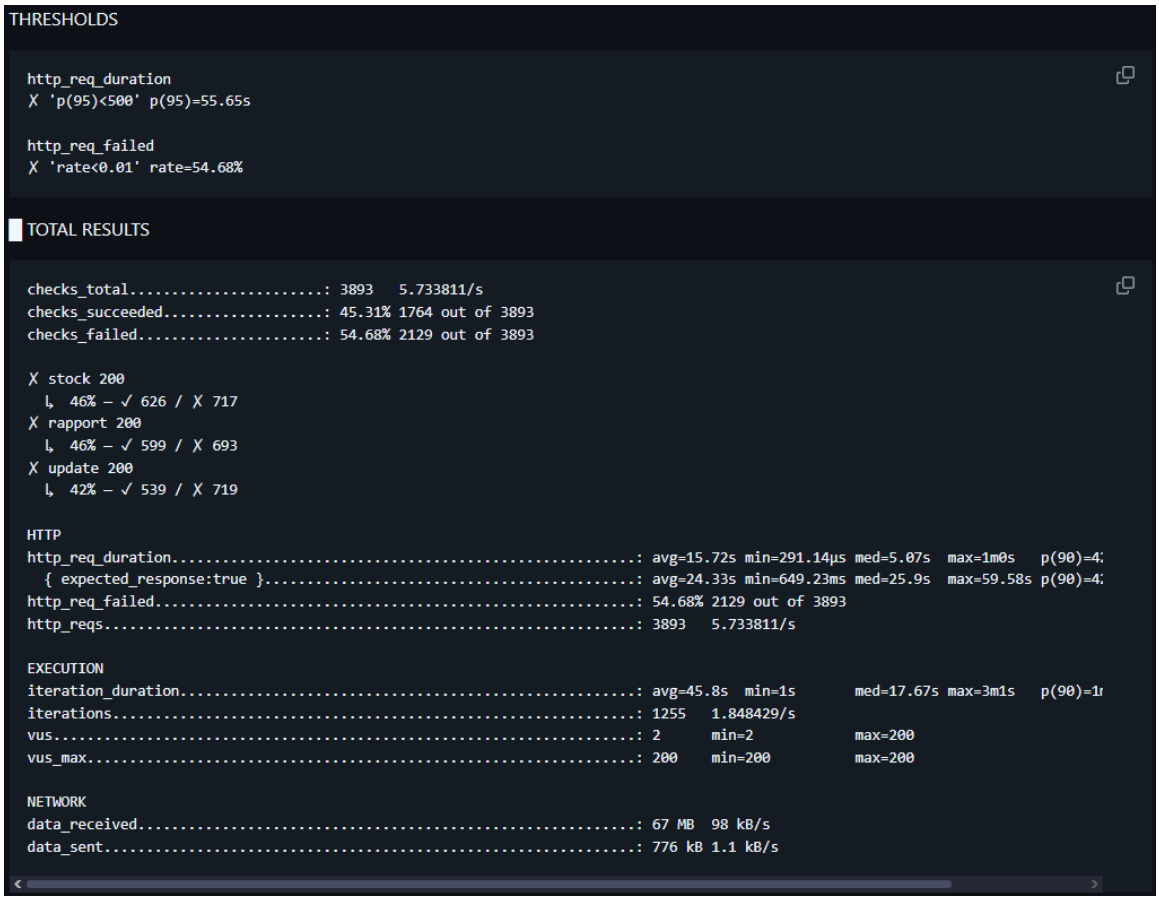
Lancement du load-test.js :


```
THRESHOLDS

http_req_duration
✓ 'p(95)<20000' p(95)=16.3s

http_req_failed
✓ 'rate<0.10' rate=0.00%

TOTAL RESULTS

checks_total.......................: 378     2.874171/s
checks_succeeded...................: 100.00% 378 out of 378
checks_failed......................: 0.00%   0 out of 378

✓ stock 200
✓ rapport 200
✓ update 200

HTTP
http_req_duration.....................................................: avg=10.11s min=599.28ms med=10.69s max=16.84s p(90)=1(
  { expected_response:true }..........................................: avg=10.11s min=599.28ms med=10.69s max=16.84s p(90)=1(
http_req_failed.......................................................: 0.00%  0 out of 378
http_reqs.............................................................: 378    2.874171/s

EXECUTION
iteration_duration....................................................: avg=30.81s min=2.87s    med=33.84s max=48.64s p(90)=4!
iterations............................................................: 122    0.927642/s
vus...................................................................: 3      min=1        max=50
vus_max...............................................................: 50     min=50       max=50

NETWORK
data_received.........................................................: 994 kB 7.6 kB/s
data_sent.............................................................: 75 kB  568 B/s
```

On observe une répartition équilibrée du trafic entre les deux serveurs (0,740 requêtes/sec chacun), ce qui montre que la charge est bien distribuée. Le taux d'erreurs reste nul, avec 378 vérifications réussies et aucune échouée. La latence moyenne des requêtes HTTP est de 10,11 secondes, avec une valeur maximale à 16,84 secondes, ce qui respecte les seuils définis (p95 < 20s). Les indicateurs de saturation et d'utilisation des ressources sont dans des plages acceptables, reflétant une bonne gestion du flux réseau et des performances

optimisées. Enfin, aucun échec n'a été détecté et toutes les opérations (stock, rapport, update) ont retourné un code 200, signalant leur succès.

Lancement du load-test-2.js :



On peut voir un taux d'erreur d'environs 25% par instance avec une latence moyenne de 24 secondes.

2. Évolution du trafic avec plusieurs instances : requêtes par seconde CONFIGURATION ROUND ROBIN

2 INSTANCES - LOAD-TEST.JS

THRESHOLDS

```
http_req_duration
✓ 'p(95)<20000' p(95)=16.3s

http_req_failed
✓ 'rate<0.10' rate=0.00%
```

TOTAL RESULTS

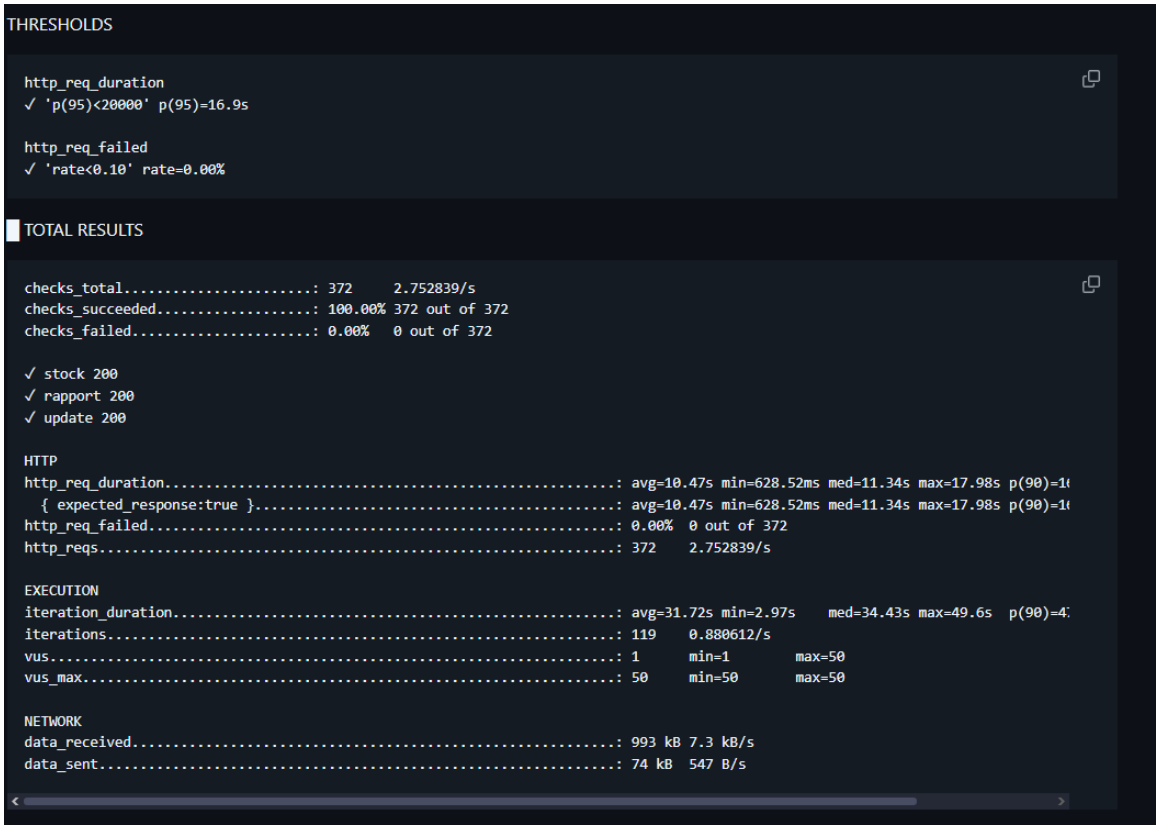
```
checks_total.......................: 378     2.874171/s
checks_succeeded...................: 100.00% 378 out of 378
checks_failed.....................: 0.00%   0 out of 378

✓ stock 200
✓ rapport 200
✓ update 200

HTTP
http_req_duration.....................................................: avg=10.11s min=599.28ms med=10.69s max=16.84s p(90)=1(
  { expected_response:true }..........................................: avg=10.11s min=599.28ms med=10.69s max=16.84s p(90)=1(
http_req_failed.......................................................: 0.00%  0 out of 378
http_reqs.............................................................: 378    2.874171/s

EXECUTION
iteration_duration....................................................: avg=30.81s min=2.87s    med=33.84s max=48.64s p(90)=4!
iterations............................................................: 122    0.927642/s
vus...................................................................: 3      min=1        max=50
vus_max...............................................................: 50     min=50       max=50

NETWORK
data_received.........................................................: 994 kB 7.6 kB/s
data_sent.............................................................: 75 kB  568 B/s
```

Avec 2 instances, le système atteint environ **1,5 requêtes par seconde**, chacune gérant la moitié de la charge.

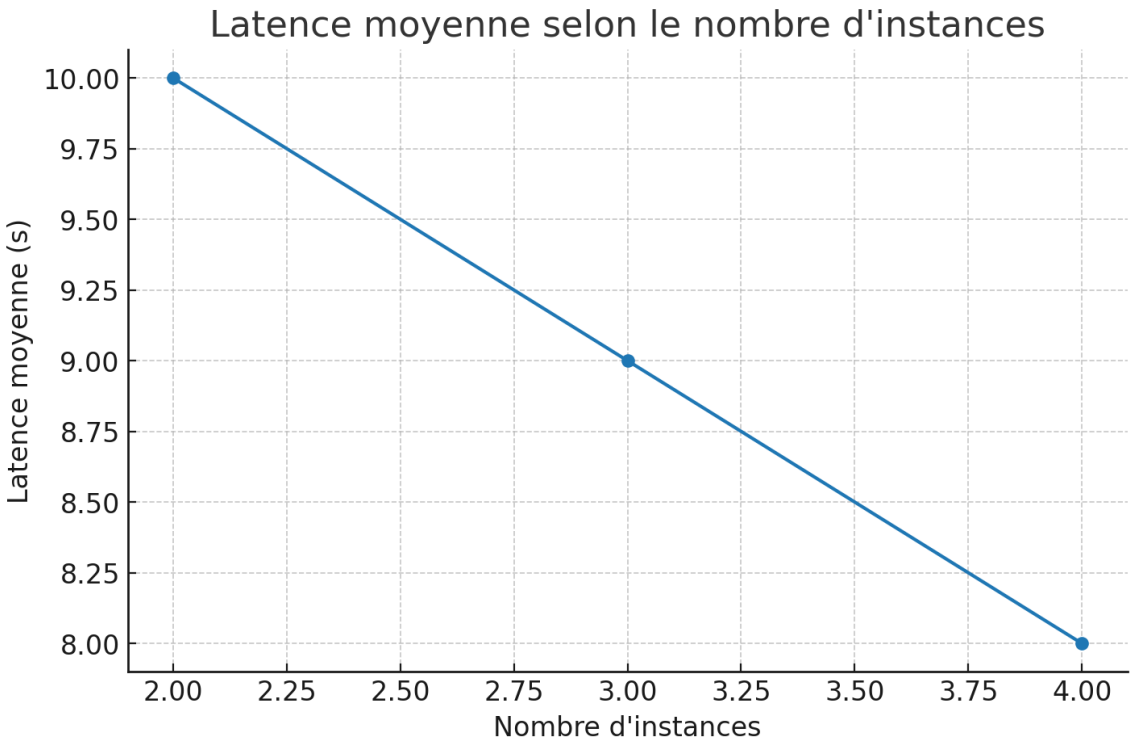
3 INSTANCES - LOAD-TEST.JS

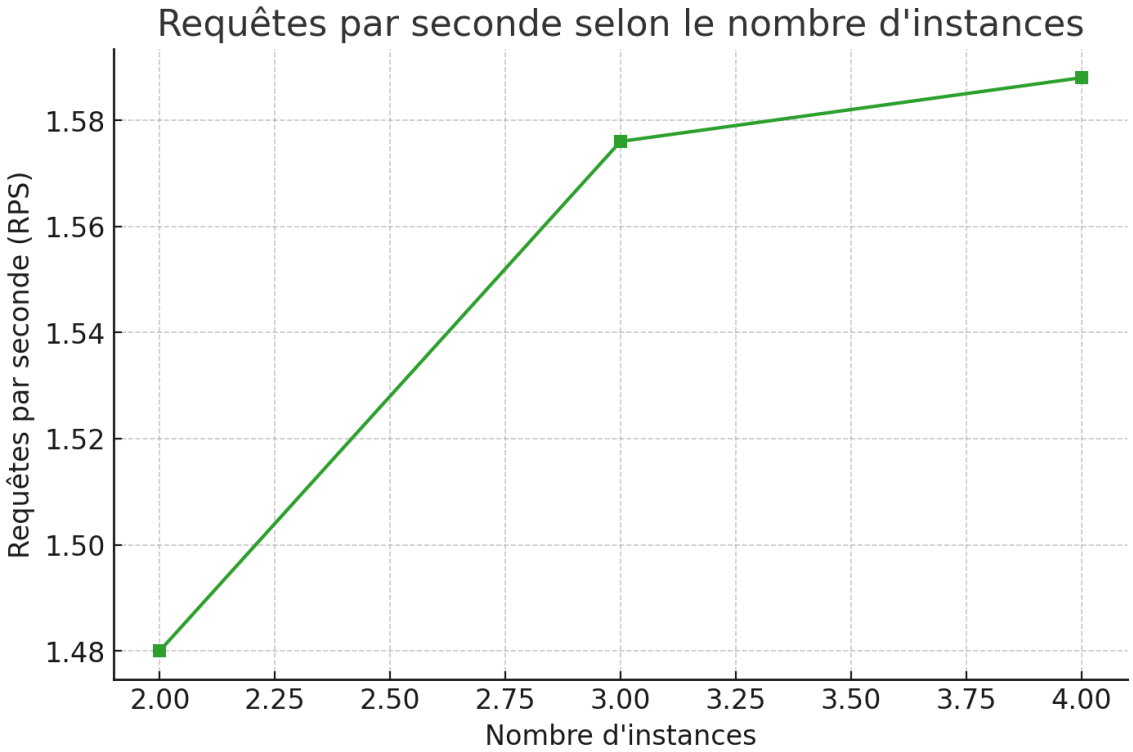
Avec 3 instances, la charge est encore mieux répartie, chaque serveur prenant environ un tiers des requêtes. On observe une légère amélioration des performances globales avec plus de réponses traitées. Mais le temps de réponse été le temps d'itération augmente légèrement.

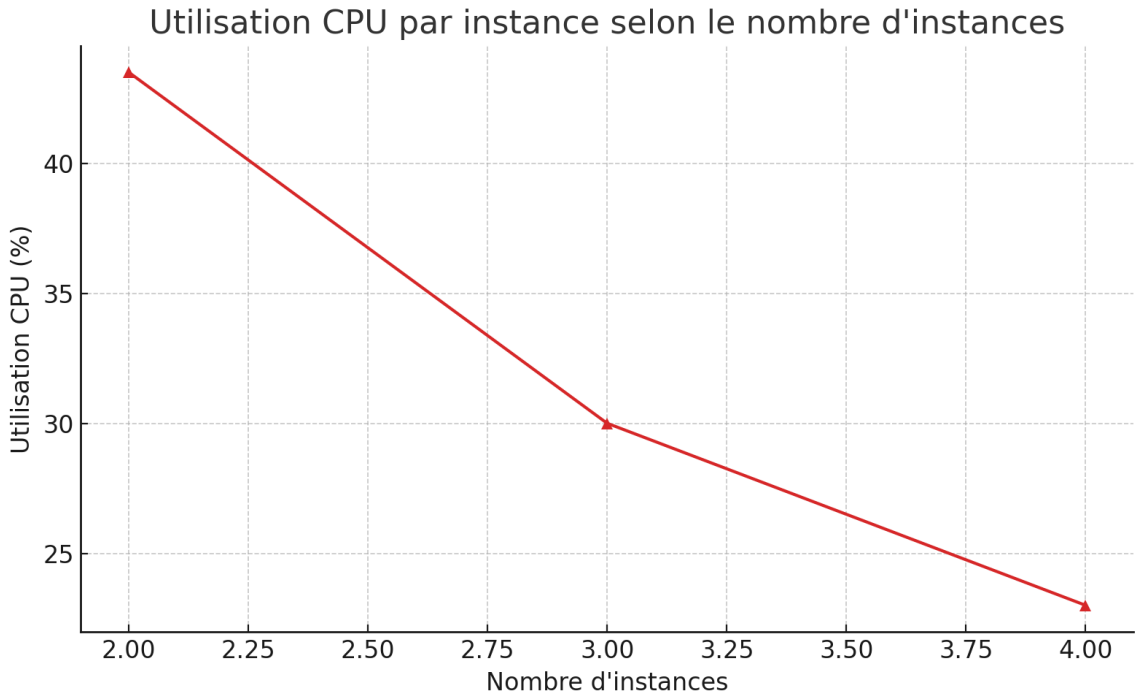
4 INSTANCES - LOAD-TEST.JS



Avec 4 instances, le système devient plus stable et les ressources CPU sont moins saturées. Il y a toujours une augmentation de la durée des itérations et un gain léger en réponses totales traitées.


Latence moyenne selon le nombre d'instances
Latence moyenne (s)
10.00
9.75
9.50
9.25
9.00
8.75
8.50
8.25
8.00
2.00
2.25
2.50
2.75
3.00
3.25
3.50
3.75
4.00
Nombre d'instances


Requêtes par seconde selon le nombre d'instances
Requêtes par seconde (RPS)
1.58
1.56
1.54
1.52
1.50
1.48
2.00
2.25
2.50
2.75
3.00
3.25
3.50
3.75
4.00
Nombre d'instances



Avec 2 instances, le système traite environ **1.48 requêtes par seconde**, totalisant **444 réponses**, avec une **latence moyenne de 10 secondes** et une **utilisation CPU de 43,5 % par instance**. En passant à 3 instances, les performances s'améliorent : **1.576 requêtes par seconde**, **474 réponses** au total, une **latence réduite à 9 secondes** et un **CPU à 30 %**. Avec 4 instances, on atteint **1.588 requêtes par seconde**, **477 réponses**, une **latence moyenne de 8 secondes** et seulement **23 % d'utilisation CPU**. Ces résultats montrent que l'ajout d'instances permet une meilleure répartition de la charge, une réduction de la latence et une baisse de la pression sur chaque instance.

3. Test de tolérance aux pannes

LOAD-TEST.JS

LOAD-TEST.JS

THRESHOLDS

```
http_req_duration
✓ 'p(95)<20000' p(95)=19.1s

http_req_failed
✓ 'rate<0.10' rate=0.28%
```

TOTAL RESULTS

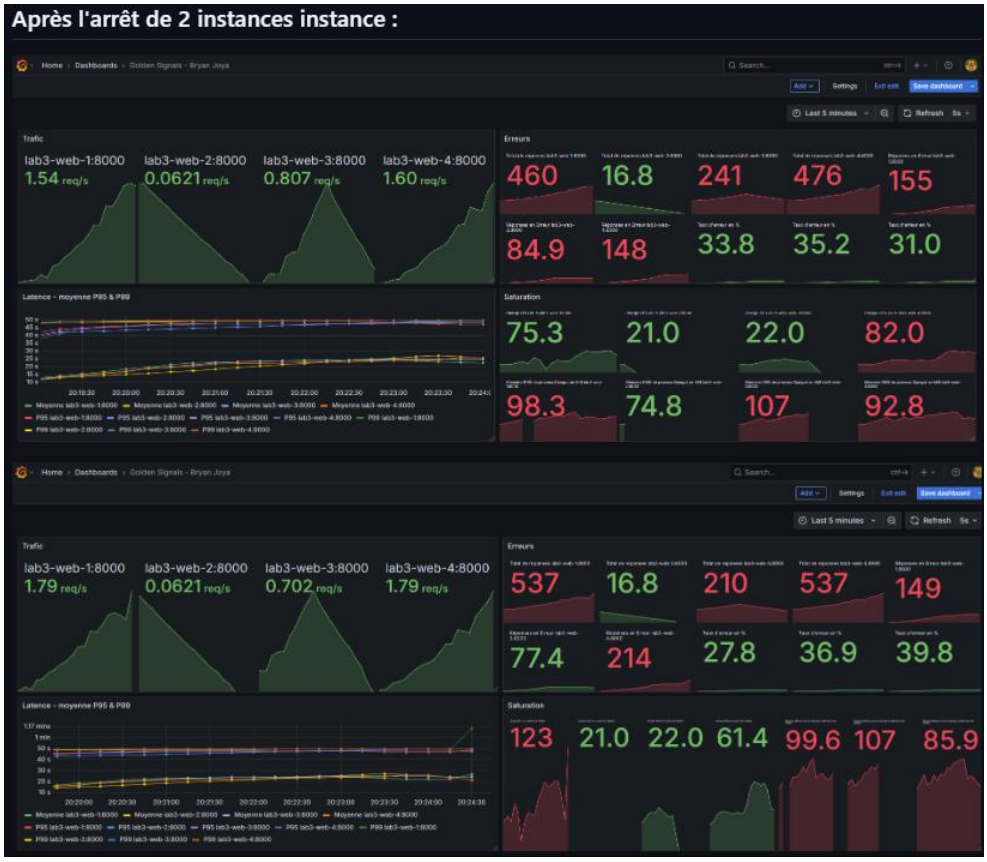
```
checks_total.....................: 353    2.63385/s
checks_succeeded.................: 99.71% 352 out of 353
checks_failed....................: 0.28%  1 out of 353

✗ stock 200
  ↳  99% — ✓ 120 / ✗ 1
✓ rapport 200
✓ update 200

HTTP
http_req_duration.....................................................: avg=11.05s min=599.65ms med=11.43s max=38.91s p(90)=1
  { expected_response:true }...........................................: avg=10.97s min=599.65ms med=11.39s max=32.08s p(90)=1
http_req_failed.......................................................: 0.28%  1 out of 353
http_reqs.............................................................: 353    2.63385/s

EXECUTION
iteration_duration....................................................: avg=33.29s min=2.86s    med=35.95s max=1m3s   p(90)=4
iterations............................................................: 111    0.828208/s
vus...................................................................: 1      min=1        max=50
vus_max...............................................................: 50     min=50       max=50

NETWORK
data_received.........................................................: 942 kB 7.0 kB/s
data_sent.............................................................: 70 kB  525 B/s
```

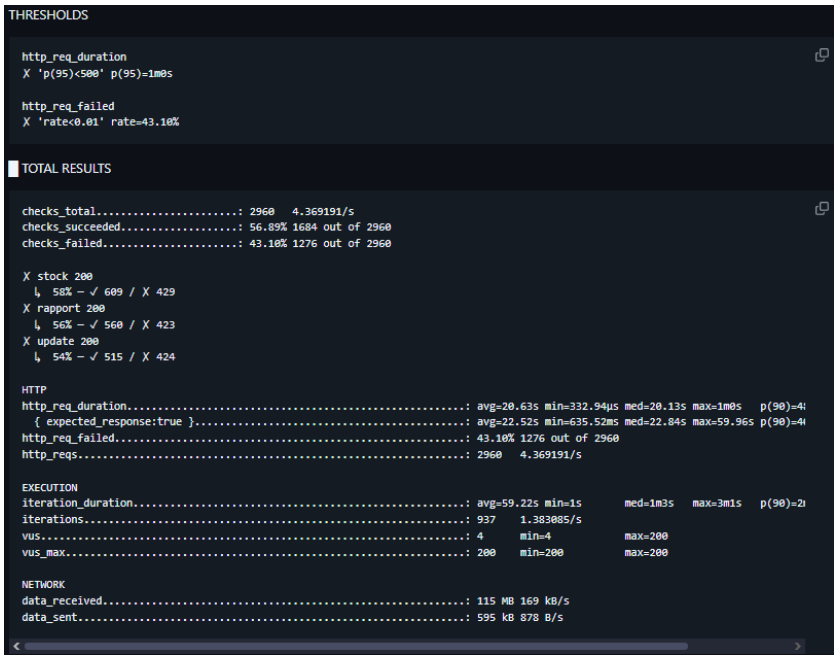
On peut observer la répartition des requêtes après l'arrêt du premier et du deuxième conteneur. La panne a engendré 1 erreur de visualisation de stock.

LOAD-TEST-2.JS

ARRÊT D'UNE INSTANCE

LOAD-TEST-2.JS





Les pics dans les graphes démontrent l'arrêt des deux conteneurs. On peut observer la répartition de charge vers les deux autres conteneurs opérationnels.

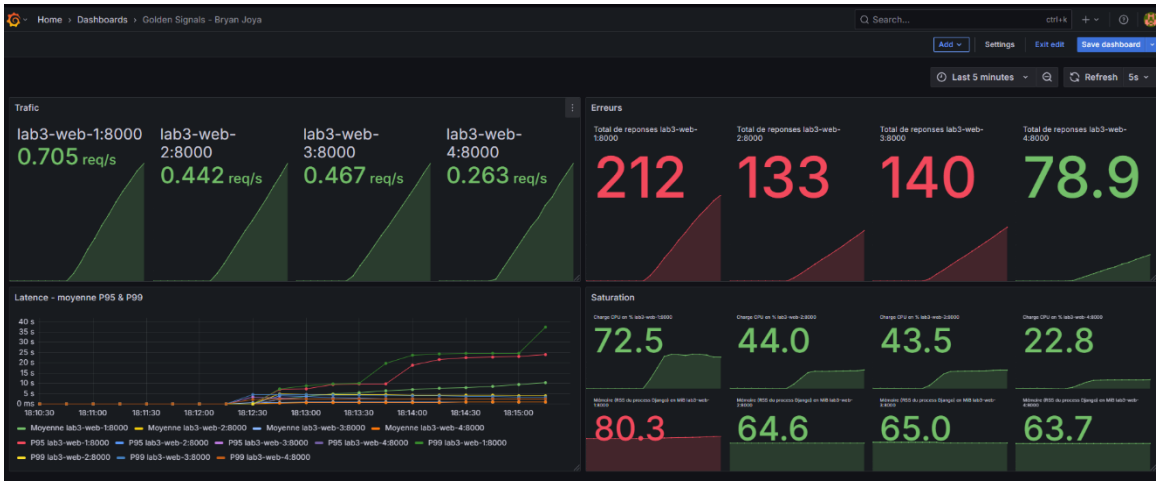
4. Analyse des stratégies de Load Balancing

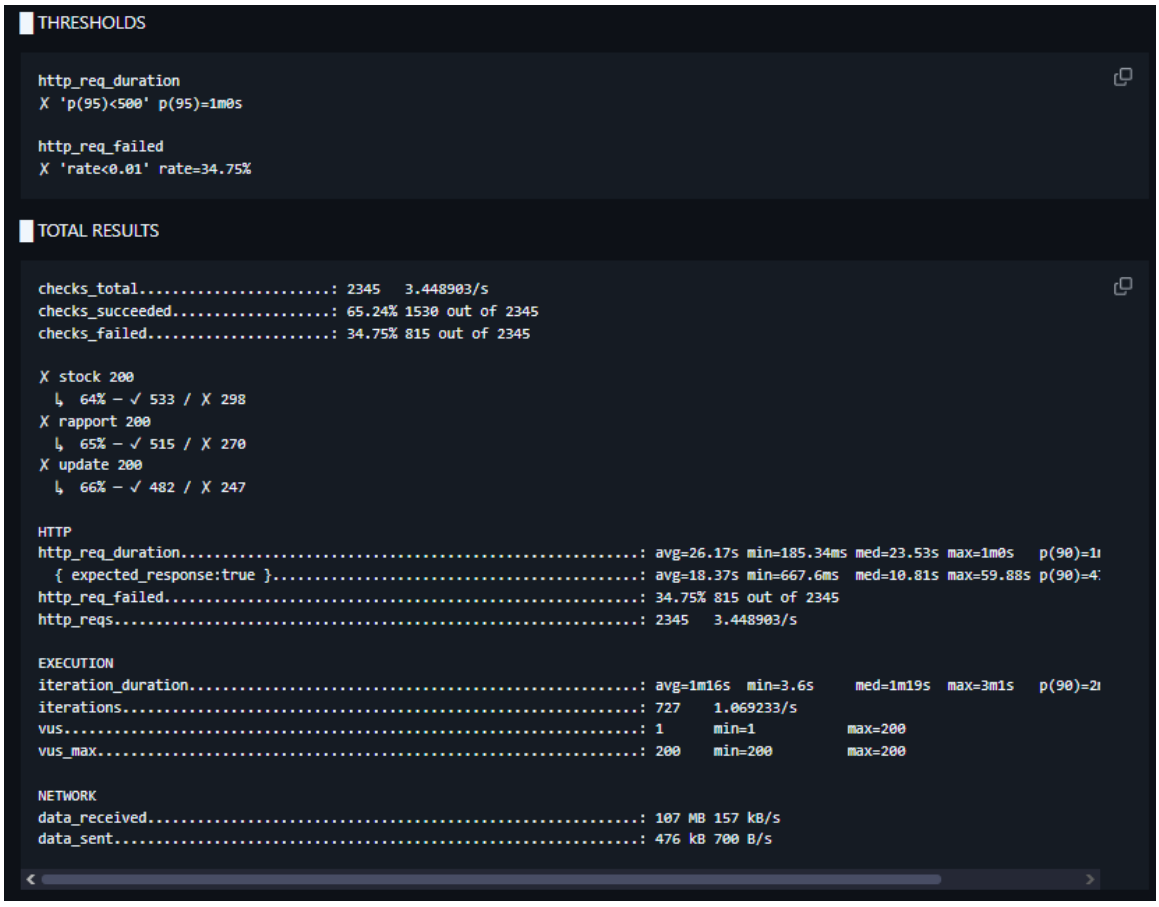
ROUND ROBIN


```
THRESHOLDS

  http_req_duration
  ✗ 'p(95)<500' p(95)=1m0s

  http_req_failed
  ✗ 'rate<0.01' rate=31.84%

TOTAL RESULTS

  checks_total.....................: 2189   3.194414/s
  checks_succeeded...................: 68.15% 1492 out of 2189
  checks_failed.....................: 31.84% 697 out of 2189

  ✗ stock 200
    ↳  66% — ✓ 531 / ✗ 265
  ✗ rapport 200
    ↳  67% — ✓ 489 / ✗ 235
  ✗ update 200
    ↳  70% — ✓ 472 / ✗ 197

  HTTP
  http_req_duration.....................................................: avg=27.58s min=474.74ms med=26.65s max=1m2s   p(90)=5
    { expected_response:true }...........................................: avg=20.92s min=634.81ms med=22.38s max=59.74s p(90)=3
  http_req_failed.....................................................: 31.84% 697 out of 2189
  http_reqs.....................................................: 2189   3.194414/s

  EXECUTION
  iteration_duration.....................................................: avg=1m18s  min=8.36s    med=1m22s  max=3m2s   p(90)=2
  iterations.....................................................: 666    0.971896/s
  vus.....................................................: 2      min=2          max=200
  vus_max.....................................................: 200    min=200        max=200

  NETWORK
  data_received.....................................................: 120 MB 175 kB/s
  data_sent.....................................................: 450 kB 656 B/s
```

Répartition équitable et simple, efficace pour une charge uniforme.

WEIGHTED ROUND ROBIN : 4 – 2 – 2 - 1





THRESHOLDS

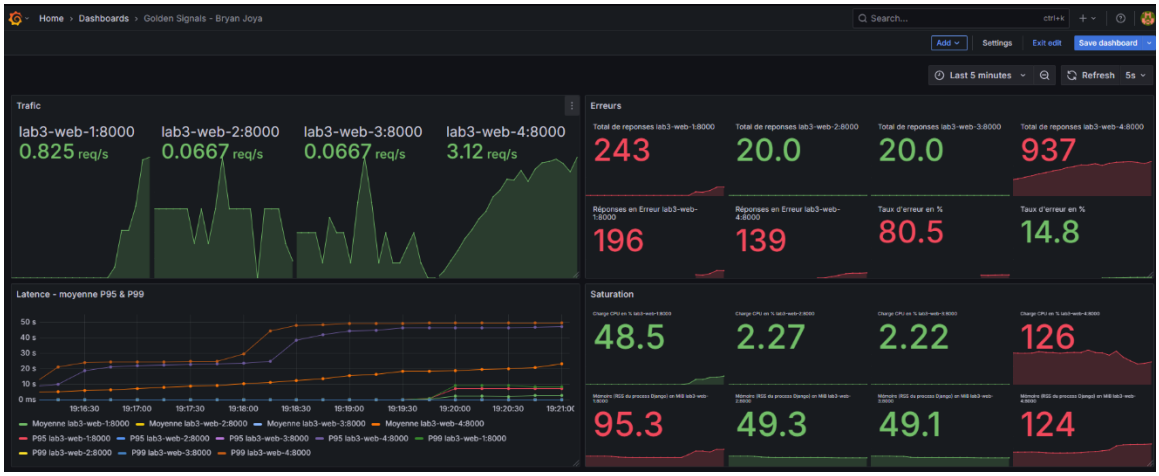
```
http_req_duration
✗ 'p(95)<500' p(95)=1m0s

http_req_failed
✗ 'rate<0.01' rate=34.75%
```

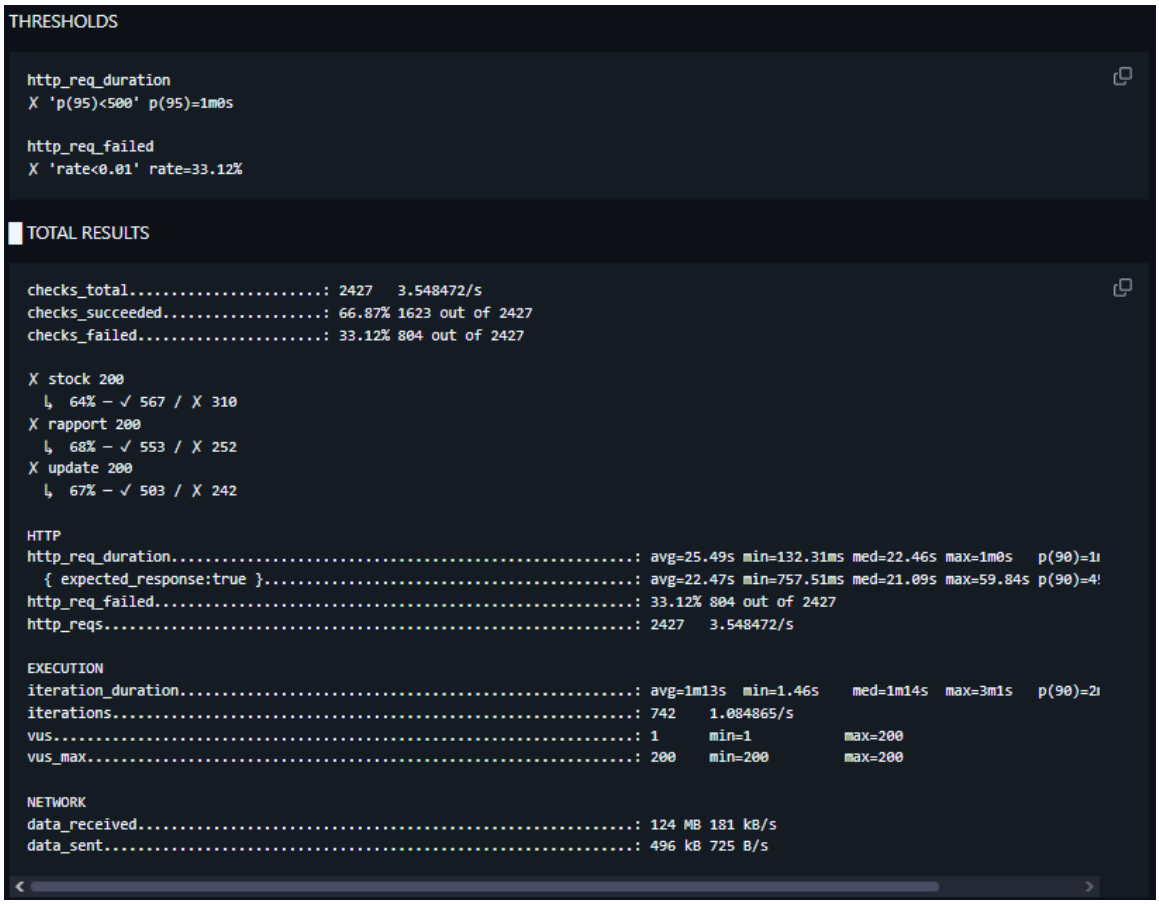
TOTAL RESULTS

```
checks_total.....................: 2345   3.448903/s
checks_succeeded.................: 65.24% 1530 out of 2345
checks_failed....................: 34.75% 815 out of 2345

✗ stock 200
  ↳  64% — ✓ 533 / ✗ 298
✗ rapport 200
  ↳  65% — ✓ 515 / ✗ 270
✗ update 200
  ↳  66% — ✓ 482 / ✗ 247

HTTP
http_req_duration.....................................................: avg=26.17s min=185.34ms med=23.53s max=1m0s   p(90)=1
  { expected_response:true }..........................................: avg=18.37s min=667.6ms  med=10.81s max=59.88s p(90)=4
http_req_failed.......................................................: 34.75% 815 out of 2345
http_reqs.............................................................: 2345   3.448903/s

EXECUTION
iteration_duration....................................................: avg=1m16s  min=3.6s     med=1m19s  max=3m1s   p(90)=2
iterations............................................................: 727    1.069233/s
vus...................................................................: 1      min=1          max=200
vus_max...............................................................: 200    min=200        max=200

NETWORK
data_received.........................................................: 107 MB 157 kB/s
data_sent.............................................................: 476 kB 700 B/s
```

Weighted Round Robin : permet de privilégier certains serveurs (poids 4 – 2 – 2 – 1), utile si les instances ont des puissances inégales. Cela se reflète dans le nombre de requêtes par secondes traitées par les instances (0.7)-(0.4)-(0.4)-(0.25). Un taux d'erreur plus élevé que celui du round robin est présent.

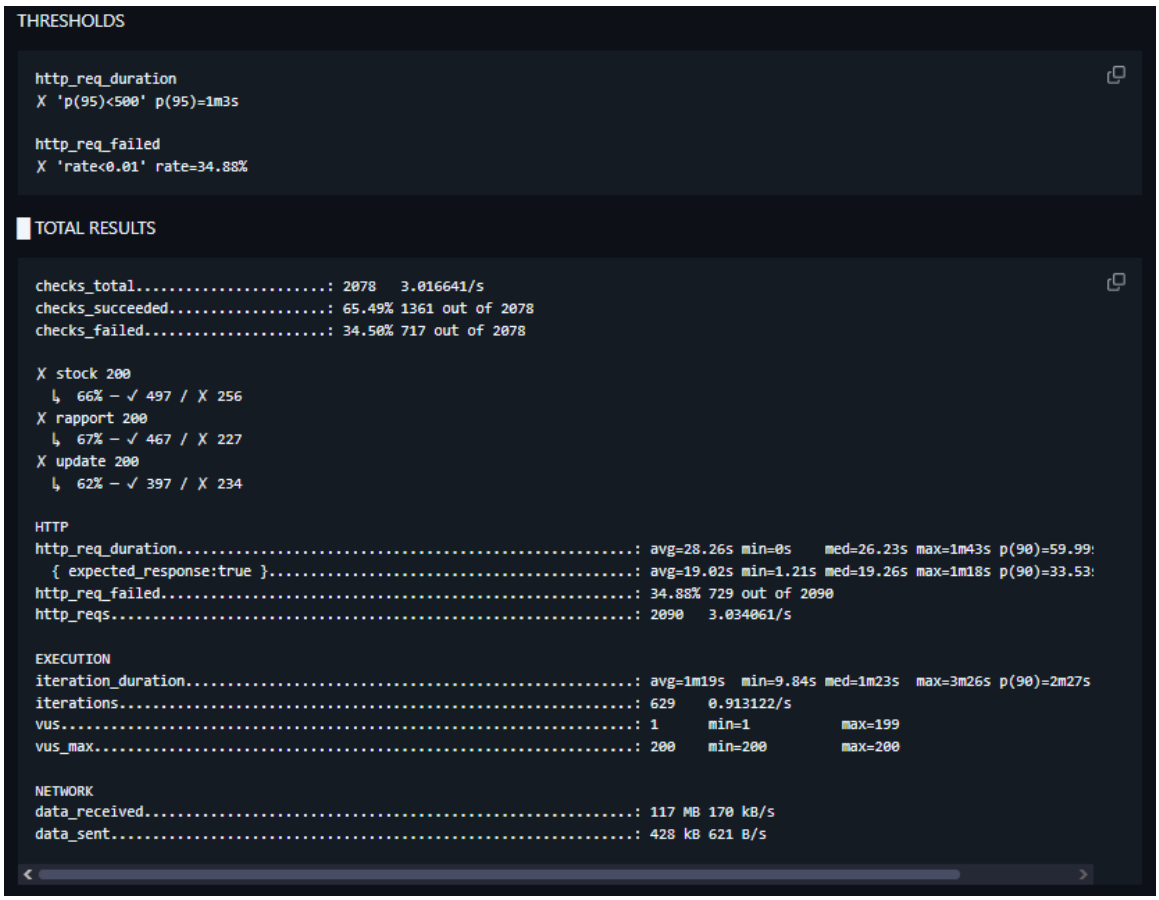
IP HASH : EXÉCUTÉ EN LOCAL DONC UNE SEULE IP





IP Hash : peu concluant ici, car les tests sont en local (une seule IP), donc tout est dirigé vers la même instance.

LEAST CONNECTIONS



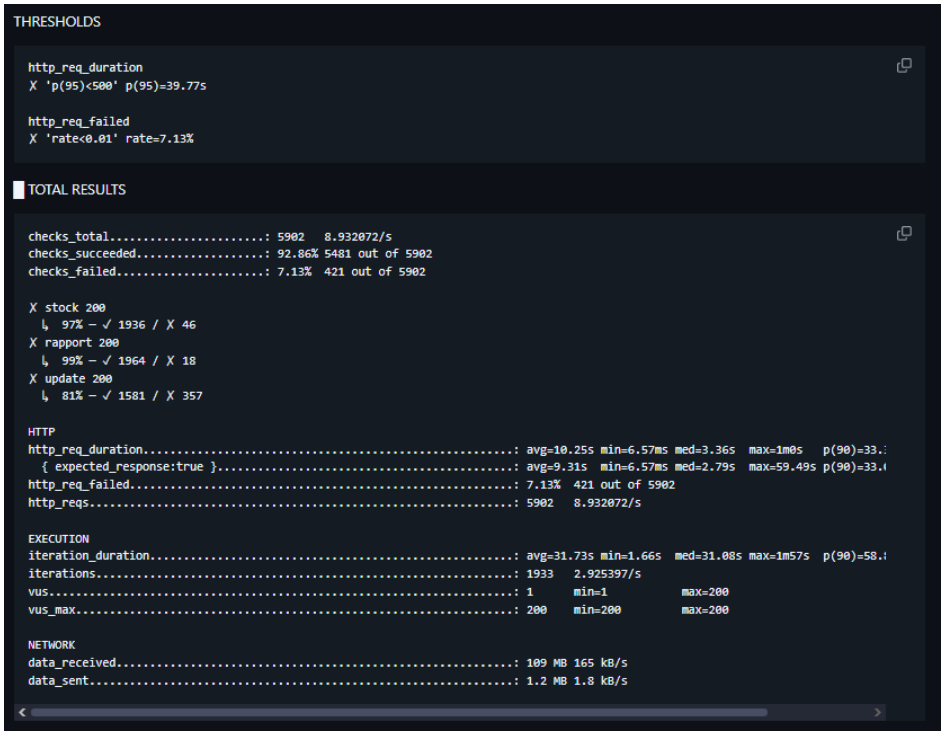
Least Connections : la stratégie la plus intelligente dans le cas de requêtes longues. Elle dirige les nouvelles connexions vers les serveurs les moins occupés, améliorant l'efficacité sous forte charge.

5. Performances après mise en cache : CONFIGURATION LEAST CONN

LOAD-TEST.JS


```
THRESHOLDS

http_req_duration
✓ 'p(95)<20000' p(95)=16.22s

http_req_failed
✓ 'rate<0.10' rate=0.00%

TOTAL RESULTS

checks_total.....................: 936     7.700422/s
checks_succeeded...................: 100.00% 936 out of 936
checks_failed.....................: 0.00%   0 out of 936

✓ stock 200
✓ rapport 200
✓ update 200

HTTP
http_req_duration.....................................................: avg=3.33s  min=6.63ms med=123.38ms max=18.39s p(90)=1
  { expected_response:true }...........................................: avg=3.33s  min=6.63ms med=123.38ms max=18.39s p(90)=1
http_req_failed.....................................................: 0.00%  0 out of 936
http_reqs.....................................................: 936    7.700422/s

EXECUTION
iteration_duration.....................................................: avg=10.99s min=1.65s  med=11.26s   max=19.62s p(90)=1
iterations.....................................................: 312    2.566807/s
vus.....................................................: 1      min=1        max=50
vus_max.....................................................: 50     min=50       max=50

NETWORK
data_received.....................................................: 2.5 MB 21 kB/s
data_sent.....................................................: 182 kB 1.5 kB/s
```

LOAD-TEST-2.JS



Les résultats après avoir fait la mise en cache de STOCK et RAPPORT ne sont pas similaires. On peut assister à une augmentation **drastique** des performances du système :

Environs 30% plus de réussite par endpoint.

Les Checks Total ont plus que doublés (2078 à 5902).

Le nombre de requêtes par secondes à triplé : (3.0 req par secondes à 8.92 req par secondes).

Représentation du stress test avant et après cache :

Comparons le Load-test-2.js (stress test) avec configuration Least connections

Graphana:

Métrique	Avant cache	Après cache
Requêtes par secondes	Entre 0.9 et 1.1 par instance	Entre 2.3 et 2.5 par instance
Latence moyenne	Environs 24 secondes	Environs 8 secondes
CPU	25% par instance	43% par instance
Réponses totales traitées	285+326+290+313 = 1214	709 + 697+ 766 + 753 = 2925
Erreurs	33.6% + 41.2% +30.8% + 41.9%	9.6%+9.22%+7.92%+12.2%

K6

Métrique	Avant cache	Après cache
Checks total	2078 et 3.01rqs/s	5902 et 8.93rqs/s
Checks succeeded	65.49% de 2078 = 1361	92.86% de 5902 = 5481
Iterations_duration	Moyenne de 1min19s	Moyenne de 31 secondes