

Matthieu DARTOIS - 21113417

Vinh-Trung THIEU - 21415515

Rapport de projet SRE

Responsable de l'UE : Naceur MALOUCH

Professeurs : Binh-Minh BUI-XUAN, Alfred DEIVASSAGAYAME,
Arthur ESCRIOU

UE : Cloud et Réseaux Virtuels

Année universitaire : 2024-2025

Sommaire

Sommaire.....	2
I. Introduction.....	3
II. Informations analysées.....	3
III. Analyse des scénarios.....	5
i. Premier scénario : Pings.....	5
ii. Deuxième scénario : Pending Connections.....	11
iii. Troisième scénario : Write / Read.....	14
iv. Quatrième scénario : Test complet.....	17
IV. Optimisation des ressources du cluster.....	19
i. Redistribution des ressources.....	19
ii. Comparaison des performances.....	19
1. Rejeu du premier scénario.....	19
2. Rejeu du second scénario.....	22
3. Rejeu du troisième scénario.....	24
4. Rejeu du dernier scénario.....	26
V. Conclusion.....	28

I. Introduction

L'objectif de ce projet est d'évaluer les performances du cluster Kubernetes que nous avons mis en place au précédent projet. Pour ce faire, nous allons appliquer un scénario de test afin d'observer le comportement de nos applications lors d'un pic de charge. Ainsi, nous pourrons déduire le débit maximum des différents types de requêtes, la charge optimale ainsi que le comportement de l'application lors du scaling horizontal de nos Pods. Il sera ainsi possible de redistribuer les ressources en fonction de nos observations pour repousser les limites de notre infrastructure. Le code utilisé pour ce projet est disponible sur GitHub :

- [Code et scripts du cluster](#)
- [Scripts utilisés pour tester le cluster.](#)

Les instructions pour utiliser le cluster et les scripts de tests sont disponibles dans les fichiers *README.md* à la racine de chacun des projets.

Les ressources allouées aux Pods Node-Redis, Redis et Redis-Replica doivent consommer moins de 2Go de RAM et 1 cœur de CPU afin que l'on ait une configuration similaire à celle des VMs d'AWS EC2. Les ressources allouées à nos Pods pour les premiers tests sont les suivantes :

Type de Pod	CPU	RAM	Nombre de Pods
Redis	180m	310Mi	1
Redis-Replica	150m	260Mi	2
Node-Redis	150m	500Mi	2

Au total et dans le pire des cas, les Pods consommeront 750m (0.75 cœur de CPU) et 1830Mi ≈ 1.92Go de RAM. Les limites sont donc respectées.

II. Informations analysées

Notre cluster Minikube possède différents services exportant des métriques à Prometheus. Nous utilisons ensuite Grafana pour mettre en forme certaines de ses métriques et analyser le comportement de notre cluster. Ainsi, notre tableau de bord Grafana contient 13 panneaux nous aidant à analyser les métriques Prometheus :

- Le nombre de Pods de chaque service de notre cluster à chaque seconde
- L'état des Pods de nos services à chaque seconde, qui permet d'observer l'évolution du cycle de vie des Pods
- Les ressources maximales que les Pods pourront utiliser (CPU et mémoire RAM)
- L'utilisation effective des ressources de nos Pods en fonction du temps
- Le nombre de clients simultanés sur la base de données Redis en fonction du temps



Screenshot d'exemple du tableau de bord Grafana

III. Analyse des scénarios

i. Premier scénario : *Pings*

Dans ce scénario, nous cherchons à atteindre les limites du serveur en envoyant un grand nombre de requêtes au serveur Node-Redis, sans solliciter la base de données. Pour effectuer nos tests, nous utilisons d'abord un processus qui fait 10000 requêtes par groupe de 100 toutes les 200 millisecondes. Le serveur ne crash pas et la latence des requêtes semble stable (0.5 ms en moyenne). On ne peut pas observer de montée en charge du service sur ce test. Nous pensons que le temps d'attente de 200 ms avant chaque groupe de requêtes laisse suffisamment de temps au serveur pour répondre sans être surchargé. Pour tester cette théorie, nous supprimons cette attente dans les tests et nous parvenons alors à observer un nouveau Pod se créer. Ce nouveau Pod permet de théoriquement doubler le débit de traitement des requêtes. Le script de test nous informe que les 10000 requêtes sont traitées en 69 secondes, soit à un débit d'environ 145 requêtes par seconde. Les ressources CPU des Pods Node-Redis ne sont pas utilisées au maximum, mais les serveurs peinent à répondre si on tente d'envoyer plus de requêtes. Le serveur Node-Redis n'est pas configuré pour envoyer des erreurs HTTP, mais on peut observer dans les logs des tests et de l'Ingress-Controller que beaucoup de connexions sont fermées prématurément, résultant en un grand nombre d'erreurs. Nous n'avons pas réussi à trouver l'exacte cause de ce problème, mais nous avons plusieurs pistes :

- Les serveurs Node-Redis n'ont pas assez de ressources et un pic de charge trop soudain leur fait fermer les connexions existantes sans que l'intervalle de scrapping Prometheus ne permette de l'observer
- L'Ingress-Controller est trop limité en CPU et les connexions TCP ne sont plus synchronisées
- L'Ingress-Controller tente de créer des connexions avec le backend, mais la connexion échoue après plusieurs tentatives.

Le dernier point est très probablement une des causes puisque nous observons ses essais de connexions en utilisant la commande *netstat* dans le Pod d'Ingress-Controller, mais nous n'arrivons pas à déterminer la cause de l'échec.

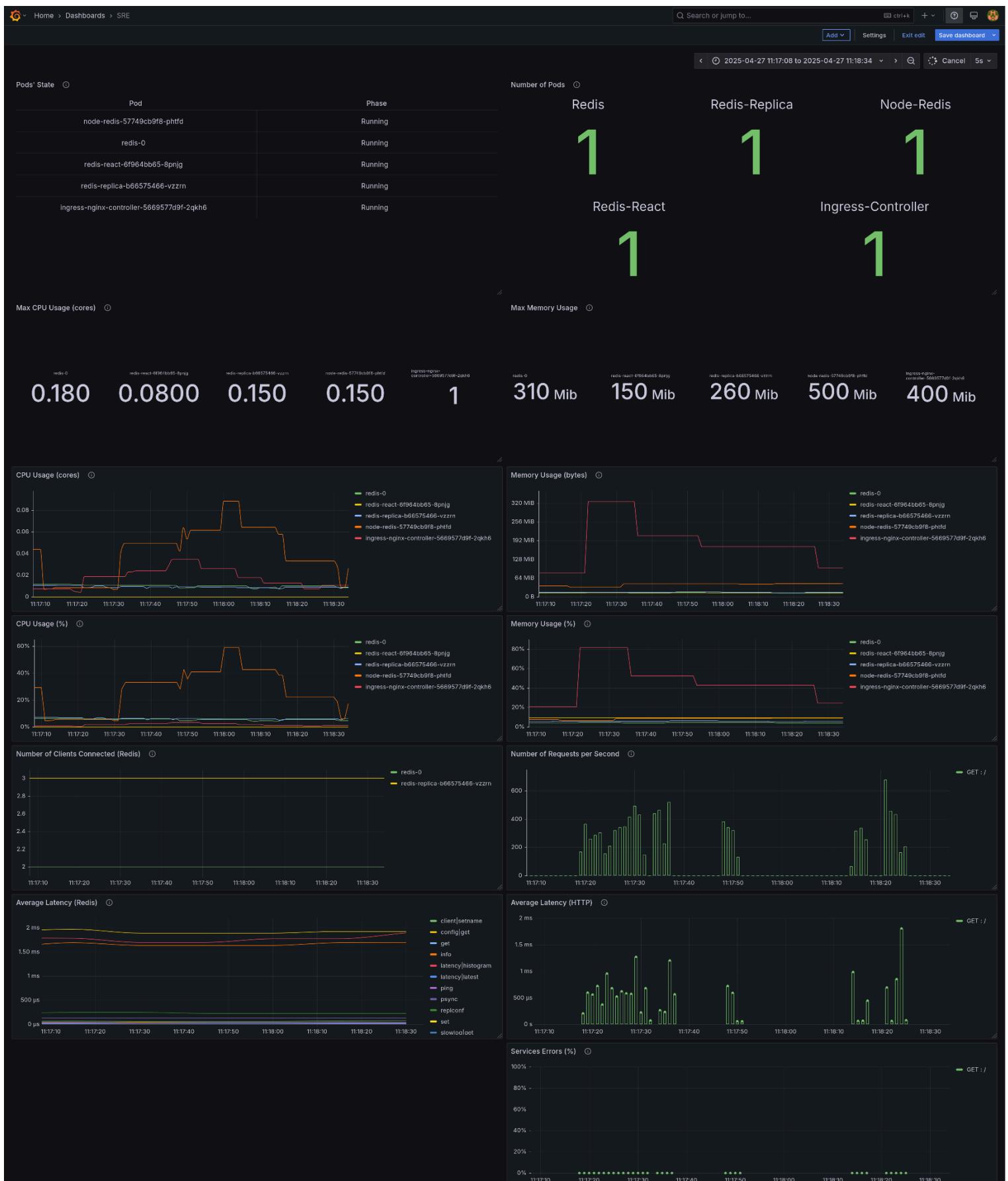
La consommation des ressources CPU apparente des Pods est importante en CPU pour Node-Redis, mais pas pour l'Ingress-Controller tandis que c'est plutôt l'inverse pour la consommation de mémoire RAM.

```
🚀 Cluster test script
URL: http://192.168.49.200/node-redis
Concurrent processes: 1
Test type: server
📊 Starting server test with 1 processes at 1745745436
  - Started server test process 1 with PID 74888
  - Progress: 1/1 processes completed
✅ server test completed successfully
📈 Statistics:
  - Total requests: 10000
  - Total success: 10000
  - Test duration: 69 seconds
  - Throughput: 144.92 requests/second
📝 Test logs available in /tmp/tmp.fpE0vBm5dc directory
```

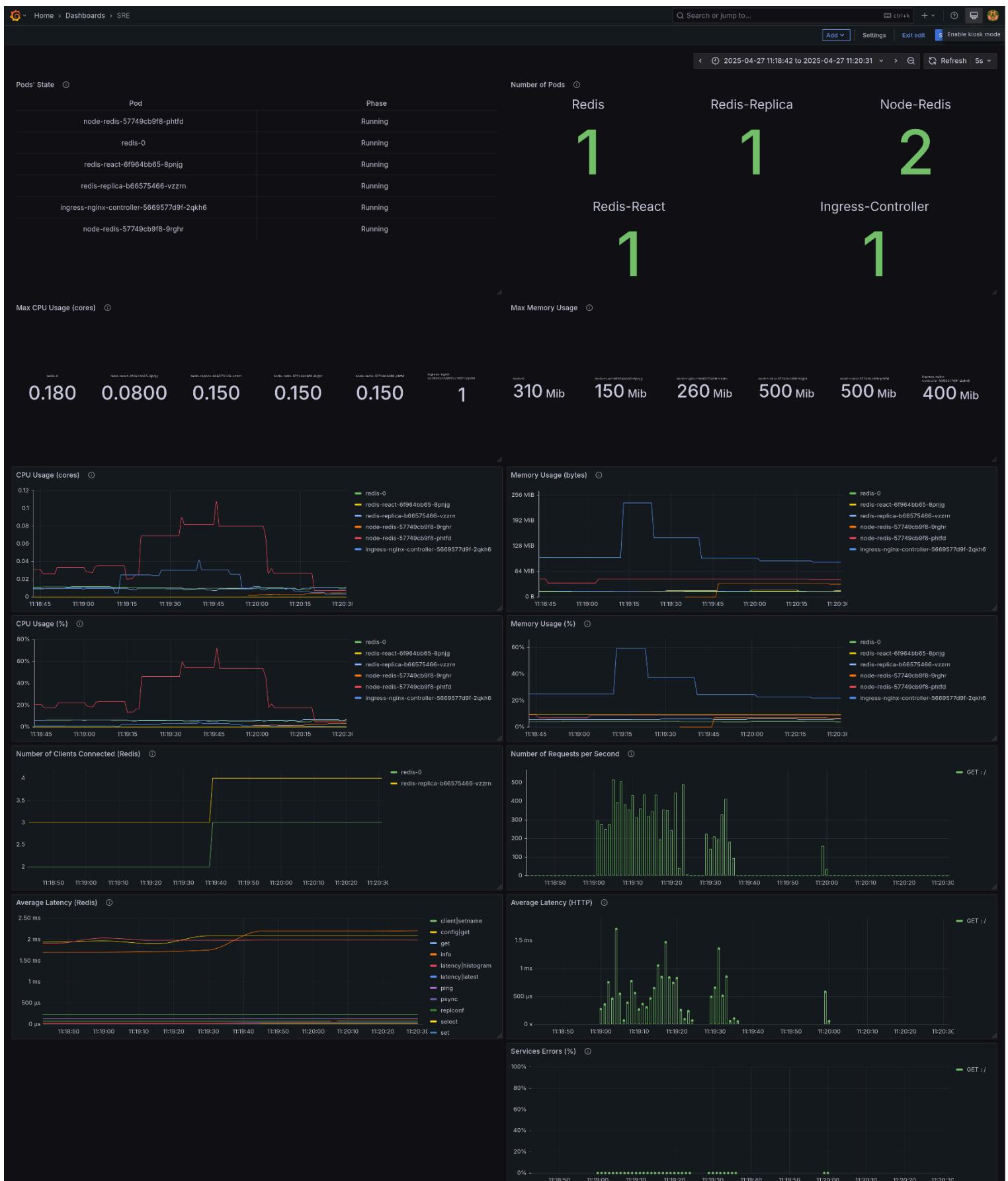
Screenshot des résultats de notre script de test avec 1 processus et 200 ms de temps d'attente entre chaque groupe de 100 requêtes, pour 10000 requêtes

```
🚀 Cluster test script
URL: http://192.168.49.200/node-redis
Concurrent processes: 1
Test type: server
📊 Starting server test with 1 processes at 1745745538
  - Started server test process 1 with PID 75452
  - Progress: 1/1 processes completed
✅ server test completed successfully
📈 Statistics:
  - Total requests: 10000
  - Total success: 10000
  - Test duration: 61 seconds
  - Throughput: 163.93 requests/second
📝 Test logs available in /tmp/tmp.nts4WnUqLg directory
```

Screenshot des résultats de notre script de test avec 1 processus et aucun temps d'attente entre chaque groupe de 100 requêtes, pour 10000 requêtes



Screenshot du tableau de bord Grafana contenant les résultats de notre premier scénario avec 1 processus et 200 ms de temps d'attente entre chaque groupe de 100 requêtes, pour 10000 requêtes



Screenshot du tableau de bord Grafana contenant les résultats de notre premier scénario avec 1 processus et aucun temps d'attente entre chaque groupe de 100 requêtes, pour 10000 requêtes

On observe clairement sur les résultats suivants que toutes les requêtes n'arrivent pas au serveur Node-Redis. Pour autant, le nombre de requêtes reçues ne correspond pas au nombre de succès du script de test, il y a donc une erreur qui impacte les connexions existantes, probablement au niveau de l'Ingress-Controller. Nous verrons si augmenter les ressources peut suffir pour repousser cette limite. Pour le moment, on peut supposer que le débit maximum des requêtes doit être aux alentours de 200 requêtes par seconde.

```
🚀 Cluster test script
URL: http://192.168.49.200/node-redis
Concurrent processes: 2
Test type: server
📊 Starting server test with 2 processes at 1745749037
  - Started server test process 1 with PID 93349
  - Started server test process 2 with PID 93351
  - Progress: 2/2 processes completed
✅ server test completed successfully
📝 Statistics:
  - Total requests: 20000
  - Total success: 285
  - Test duration: 15 seconds
  - Throughput: 1333.33 requests/second
📝 Test logs available in /tmp/tmp.GaJ6fsrZ15 directory
```

Screenshot des résultats de notre script de test avec 2 processus et 200 ms de temps d'attente entre chaque groupe de 100 requêtes, pour 20000 requêtes



Screenshot du tableau de bord Grafana contenant les résultats de notre premier scénario avec 2 processus et 200 ms de temps d'attente entre chaque groupe de 100 requêtes, pour 20000 requêtes

ii. Deuxième scénario : *Pending Connections*

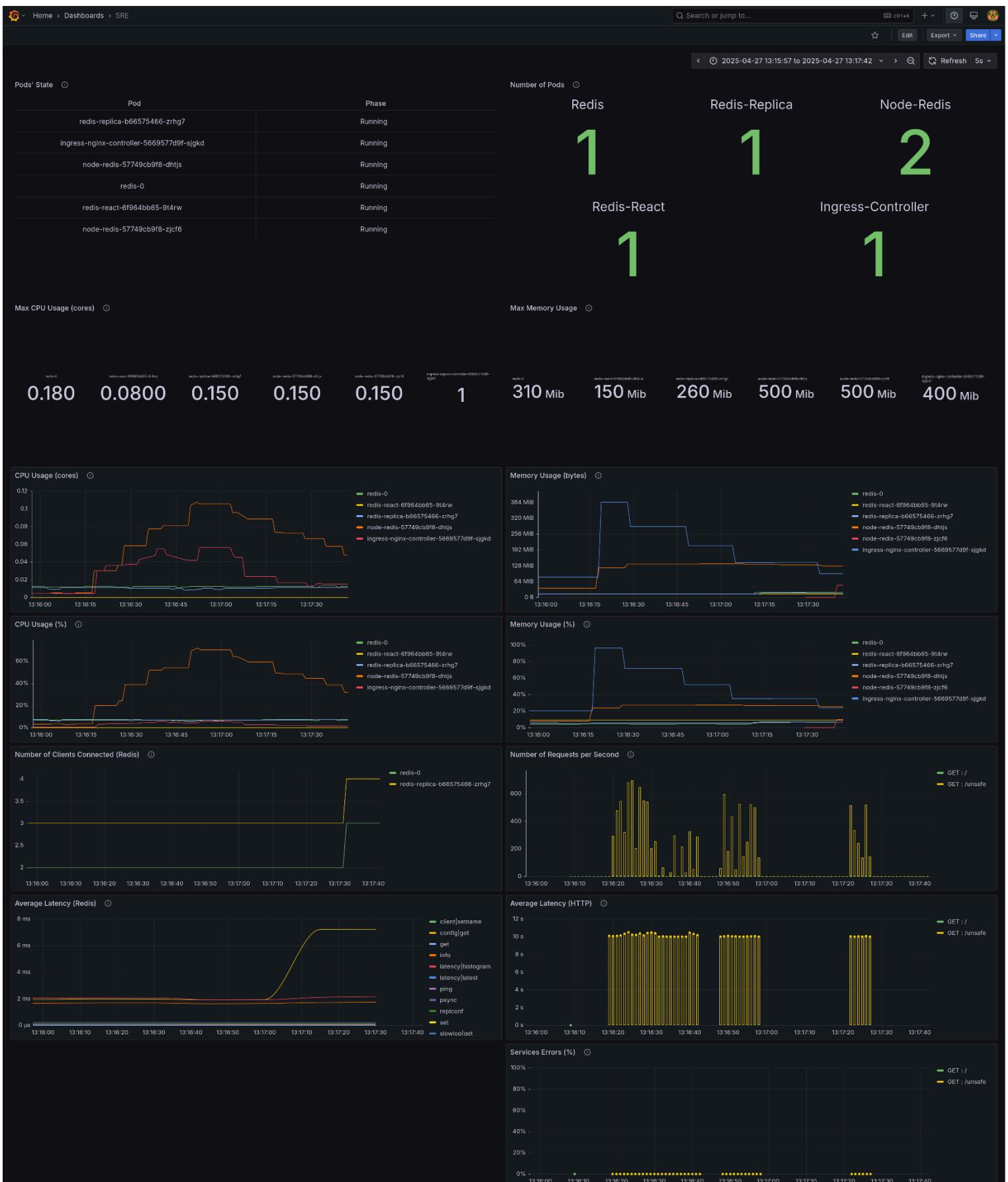
Dans ce second scénario, nous cherchons à atteindre les limites de connexions simultanées sur notre cluster. Pour ce faire, notre script de test crée 12, puis 13 processus qui ouvrent chacun 1000 connexions avec un timeout de 10 secondes. L'objectif est d'atteindre la limite d'utilisateurs en simultané. On observe cette limite entre 12000 et 13000 connexions : le même problème que précédemment survient lorsque l'on passe de 12 à 13 processus. En revanche, on observe ici que la mémoire RAM de l'Ingress-Controller sature dès le début du test avec 12 processus et que le test ne dure pas 10 secondes comme prévu : l'Ingress-Controller doit attendre 15 réponses avant de transmettre le reste des requêtes. On peut remarquer dans le test avec 13 processus que seulement 5 requêtes sont transmises au backend. On observe bien une croissance rapide de la mémoire RAM consommée par le Pod d'Ingress-Controller, mais elle reste bien inférieure à celle du test avec 12 processus. Il est possible que la mémoire ait saturé entre deux intervalles de scrapping Prometheus, ne nous laissant pas la possibilité d'observer la consommation maximum et/ou la source du problème. La configuration de l'Ingress-Controller indique que le Pod devrait pouvoir gérer jusqu'à 100000 connexions, mais une autre limitation (ressources CPU ou mémoire dans le cluster, gestion du Pod par Kubernetes, limites de connexions des conteneurs) pourrait l'empêcher d'être aussi performant qu'attendu. On remarque également que la consommation du CPU du Pod Node-Redis croît très rapidement quand même, ce qui pourrait être très limitant si les requêtes étaient acheminées correctement.

```
✓ pending test completed successfully
✗ Statistics:
  - Total requests: 12000
  - Total success: 12000
  - Test duration: 79 seconds
  - Throughput: 151.89 requests/second
📝 Test logs available in /tmp/tmp.qaB8u9NvYx directory
```

Screenshot des résultats de notre script de test avec 12 processus

```
✓ pending test completed successfully
✗ Statistics:
  - Total requests: 13000
  - Total success: 5
  - Test duration: 17 seconds
  - Throughput: 764.70 requests/second
📝 Test logs available in /tmp/tmp.ApsDQvPPt1 directory
```

Screenshot des résultats de notre script de test avec 13 processus



Screenshot du tableau de bord Grafana contenant les résultats de notre second scénario avec 12 processus



Screenshot du tableau de bord Grafana contenant les résultats de notre second scénario avec 13 processus

iii. Troisième scénario : Write / Read

Dans ce scénario, nous essayons de stresser la base de données en effectuant un grand nombre d'écriture et de lecture. La proportion d'écritures par rapport aux lectures est aléatoire pour simuler un comportement réaliste d'utilisateurs. Le test utilise 64 clés différentes ainsi que des tailles aléatoires entre 1 et 50 Ko. Nous effectuons le test avec 8, puis 9 processus, chacun effectuant 1000 requêtes par groupe de 10 (écritures et lectures confondues) toutes les 200 millisecondes. Il y a au moins 1 écriture et lecture par groupe de requêtes.

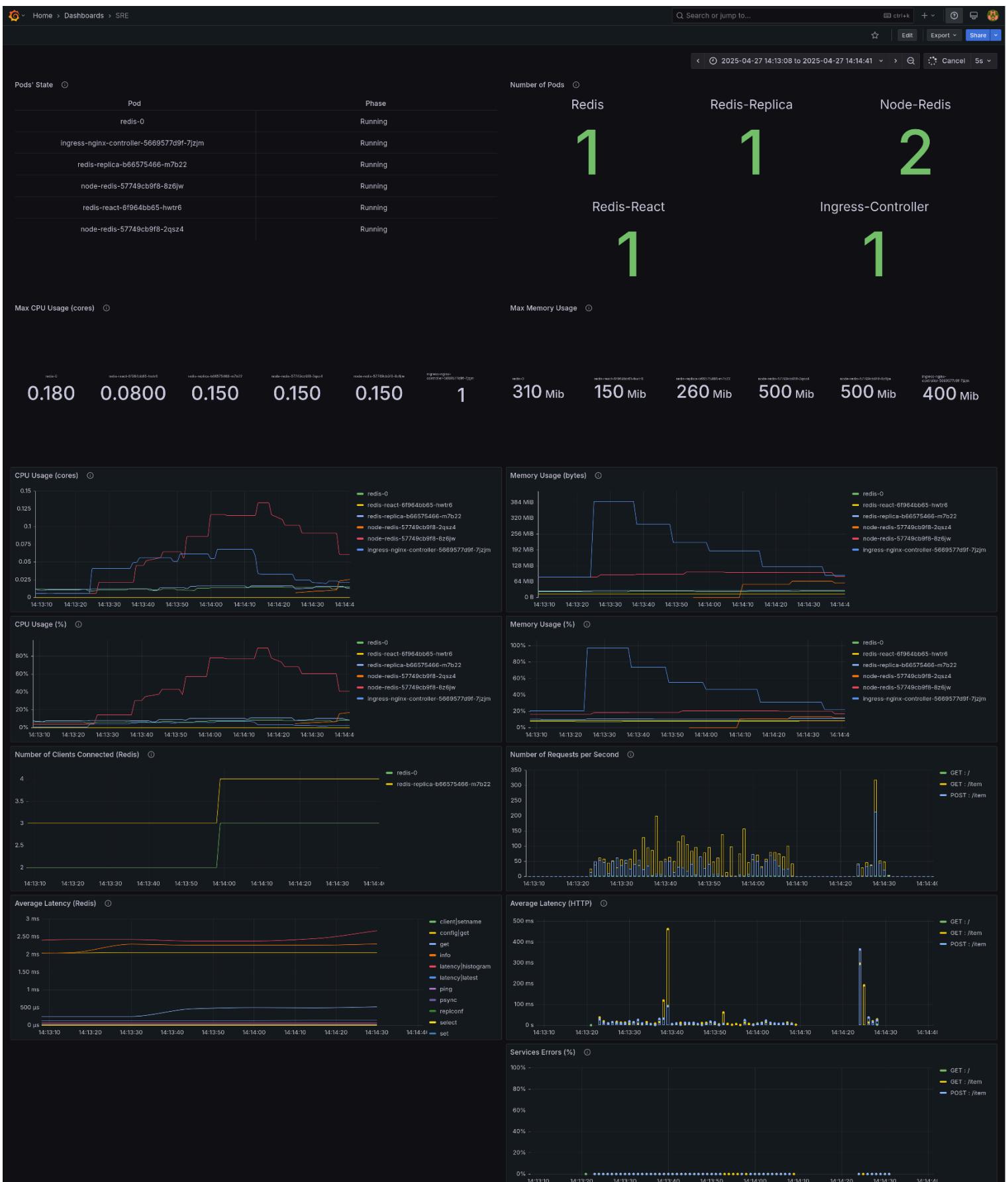
Notre test a montré que l'Ingress-Controller est bel et bien le goulot d'étranglement de notre cluster : toutes les données envoyées et reçues sont stockées temporairement dans son cache et sa mémoire RAM sature complètement jusqu'au crash du service. On voit bien que le serveur n'a pas reçu plus de 300 requêtes avant le crash de l'Ingress-Controller. La consommation du CPU des Pods Node-Redis semble être conséquente, ce qui pourrait impacter les performances à cause du *CPU throttling* de Kubernetes avec des charges plus élevées. À noter que Redis ne semble pas limité en CPU ou en mémoire.

```
✓ writeRead test completed successfully
☒ Statistics:
  - Total requests: 8000
  - Total success: 8000
  - Test duration: 145 seconds
  - Throughput: 55.17 requests/second
☒ Test logs available in /tmp/tmp.V3nkpVvnIe directory
```

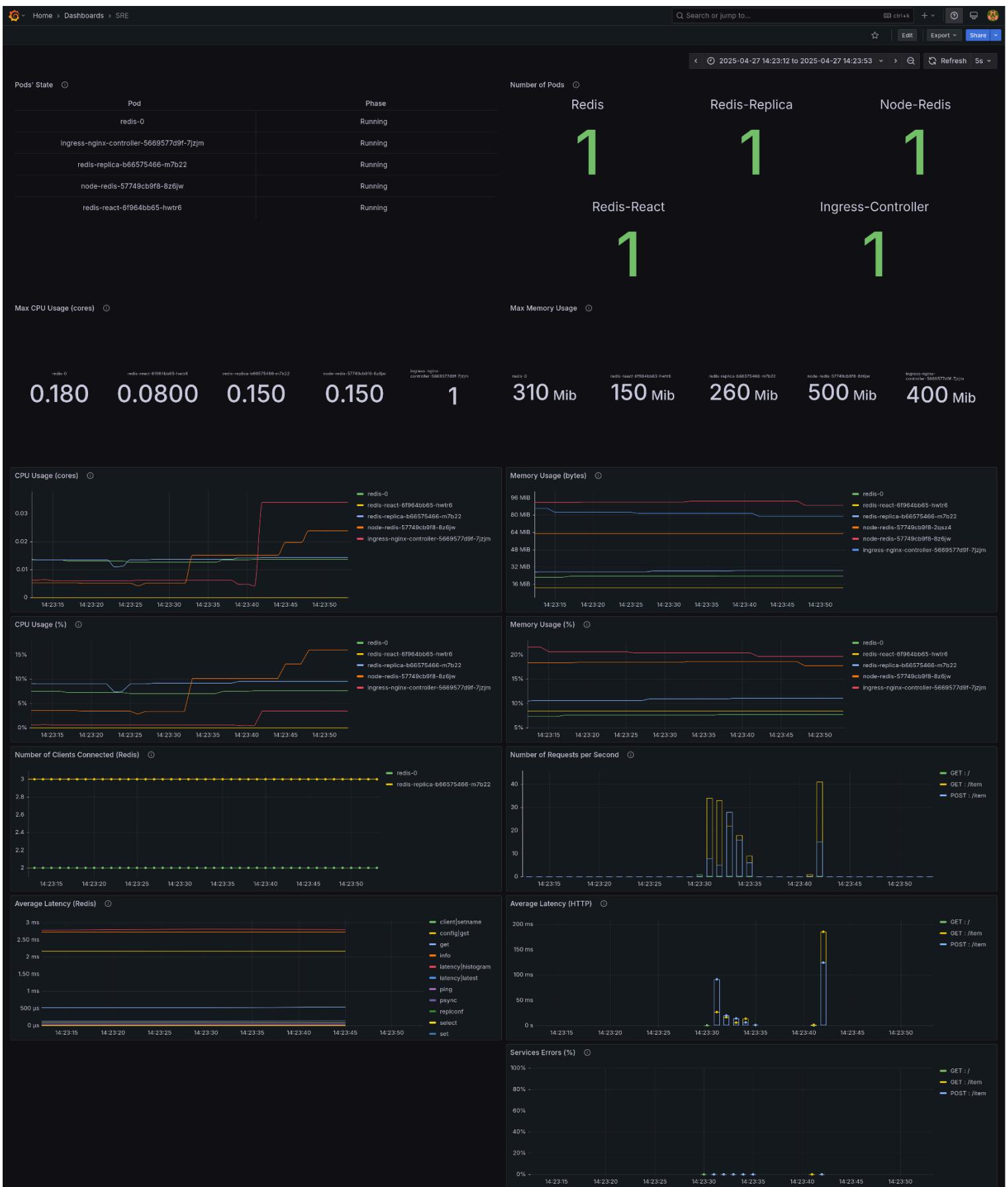
Screenshot des résultats de notre script de test avec 8 processus

```
✓ writeRead test completed successfully
☒ Statistics:
  - Total requests: 9000
  - Total success: 1
  - Test duration: 4 seconds
  - Throughput: 2250.00 requests/second
☒ Test logs available in /tmp/tmp.oC2UDma0ZN directory
```

Screenshot des résultats de notre script de test avec 9 processus



Screenshot du tableau de bord Grafana contenant les résultats de notre troisième scénario avec 8 processus



Screenshot du tableau de bord Grafana contenant les résultats de notre troisième scénario avec 9 processus

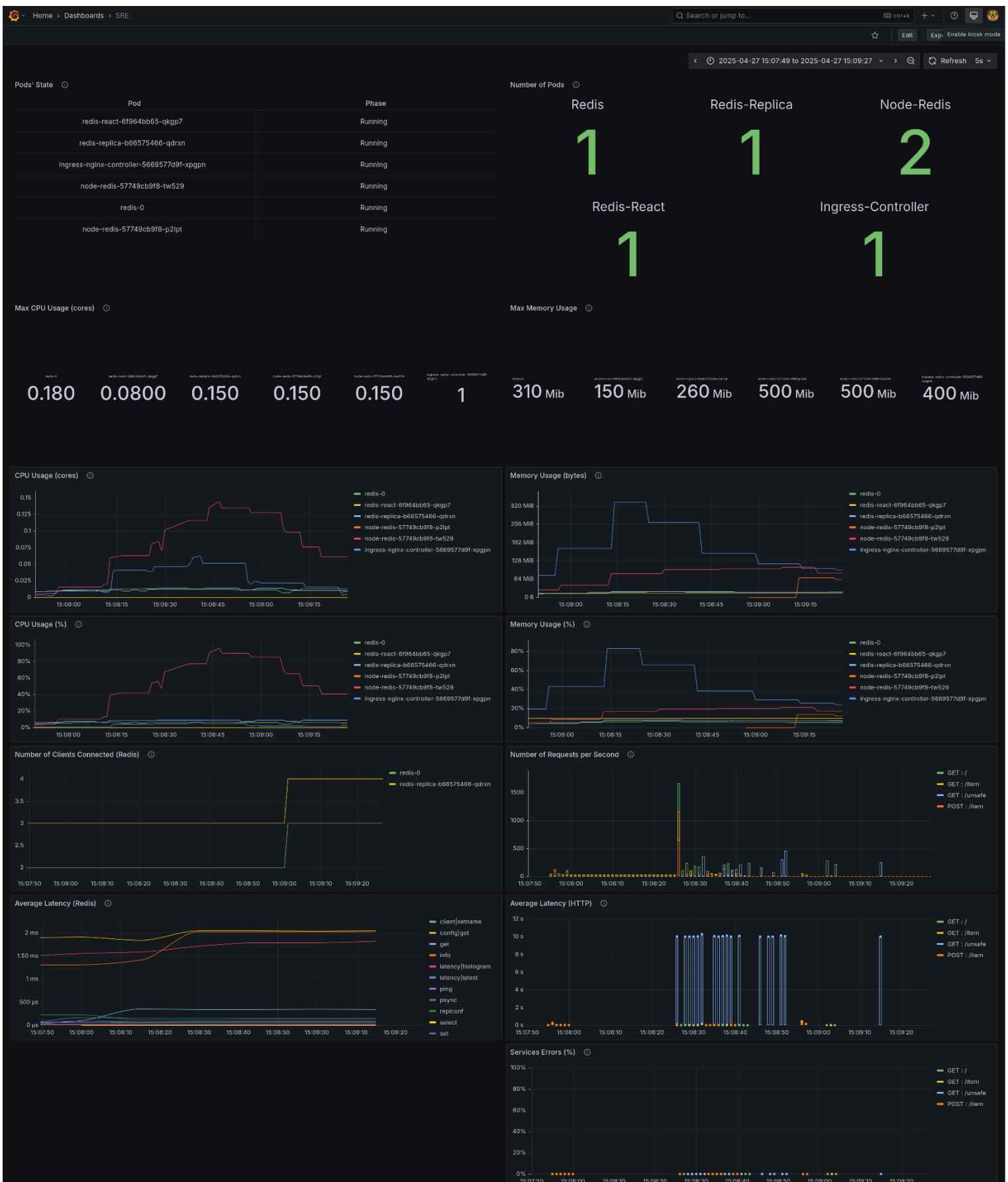
iv. Quatrième scénario : Test complet

Ce quatrième et dernier test sert à simuler une utilisation de tous les services simultanément en les lançant aléatoirement sur plusieurs processus. Nous avons baissé le nombre de requêtes par test de type *server* pour pouvoir avoir un trafic plus varié avant d'atteindre une limite.

Nous commençons par tester 11 processus, le trafic est très varié et nous voyons que la mémoire de l’Ingress-Controller atteint déjà les 80%. Même constat pour la consommation du CPU du premier Pod Node-Redis, qui atteint les 100%, mais fournit quand même les services. Notre cluster répond donc mieux à la charge lorsque le trafic est varié, certaines requêtes sont évidemment plus gourmandes en ressources que d’autres (*writeRead* et *pending* notamment).

```
✓ combined test completed successfully
📈 Statistics:
- Total requests: 11000
- Total success: 11000
- Test duration: 145 seconds
- Throughput: 75.86 requests/second
📝 Test logs available in /tmp/tmp.7axnfhpySk directory
```

Screenshot du tableau de bord Grafana contenant les résultats de notre dernier scénario avec 11 processus



Screenshot du tableau de bord Grafana contenant les résultats de notre dernier scénario avec 11 processus

IV. Optimisation des ressources du cluster

i. Redistribution des ressources

Nous avons donc appris de nos tests que nous avions la nécessité d'augmenter les ressources allouées à l'Ingress-Controller. De plus, le service Redis-React n'est pas utilisé et ne consomme presque pas de ressources : nous pouvons donc réduire ses ressources allouées pour les redistribuer. De même, les services Redis et Redis-Replica consomment peu de leurs ressources : nous pouvons les réduire d'au moins 25% pour les allouer au service Node-Redis, qui en a bien plus besoin. Il est important de souligner le fait que les Pods Redis, Redis-Replica et Node-Redis doivent toujours consommer au total moins de 2Go de RAM et 1 cœur de CPU pour avoir des spécifications similaires aux VMs d'AWS EC2. Voici les ressources que nos Pods utilisent maintenant dans le pire des cas (par Pod) :

Type de Pod	CPU	RAM	Nombre de Pods
Redis	140m	220Mi	1
Redis-Replica	130m	240Mi	2
Node-Redis	200m	400Mi	3

Au total et dans le pire des cas, nos Pods utilisent 1 cœur de CPU et 1900 Mi \approx 1.99 Go. Les limites du scénario sont donc toujours respectées.

Nous avons également revu à la baisse les fenêtres de stabilisation des HorizontalPodsAutoscalers pour que les services soient plus réactifs en cas de pics de charge.

L'Ingress-Controller possède maintenant 2 coeurs de CPU (donc deux workers au lieu de 1 pour gérer les connexions) ainsi que 2 Gi de mémoire RAM (\approx 2.15 Go).

ii. Comparaison des performances

Après avoir redistribué les ressources, nous effectuons la même série de tests que précédemment afin de comparer les résultats. À chaque test, nous prenons le cas extrême qui faisait crash l'Ingress-Controller.

1. Rejet du premier scénario

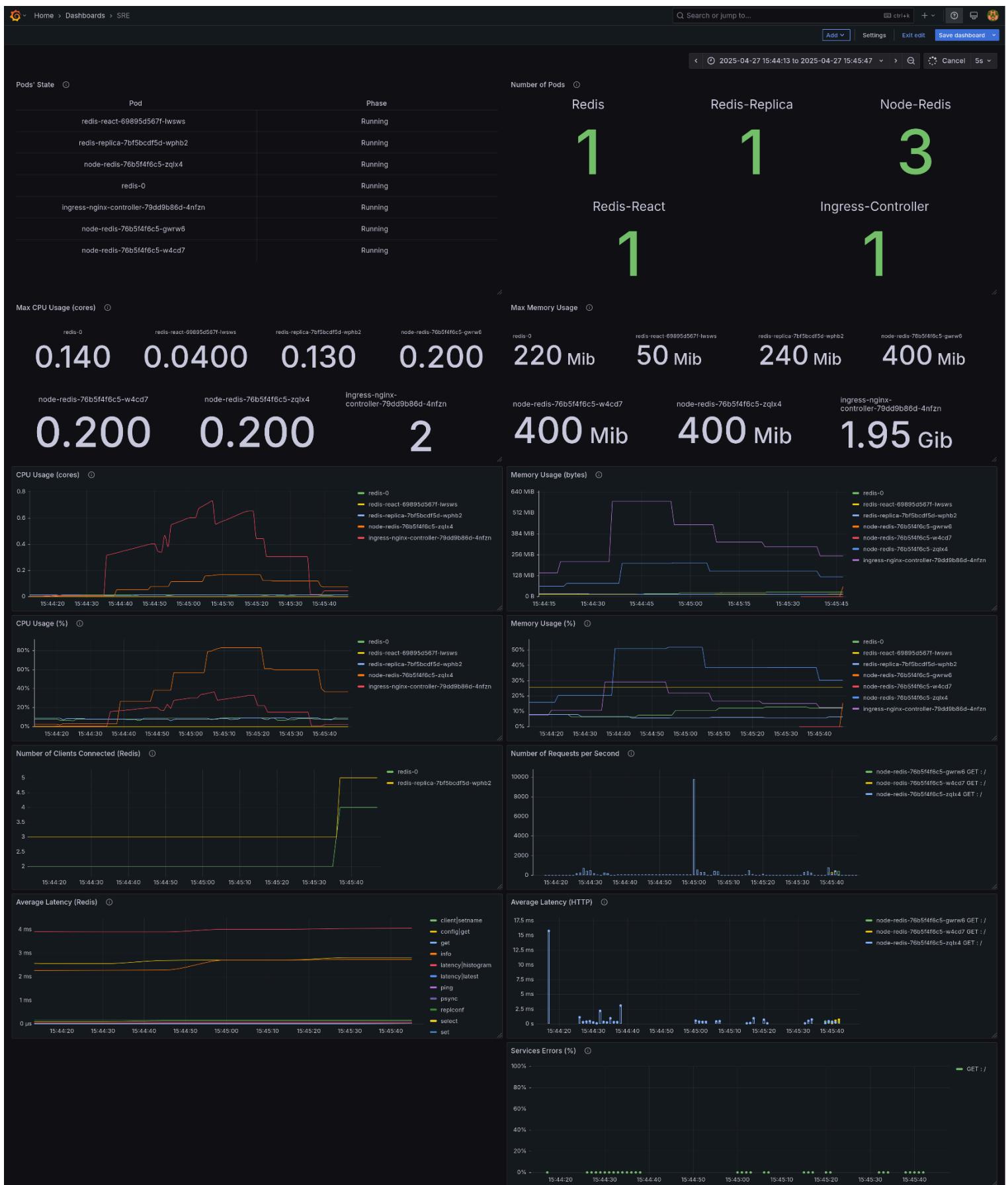
Pour rappel, nous utilisons 2 processus faisant chacun 10000 requêtes par groupe de 100, toutes les 200 ms.

On observe ici que le test passe. Il y a effectivement un gros pic de l'utilisation de la mémoire RAM de l'Ingress-Controller et le Pod Node-Redis initial suffit pour traiter toutes les requêtes, sans même atteindre sa limite d'utilisation. Nous pouvons observer ce dernier point en modifiant légèrement le tableau de bord Grafana pour

grouper les requêtes par Pod. Étonnamment, nous pouvons observer sur le tableau de bord que le backend reçoit presque l'entièreté des 10000 requêtes en moins d'une seconde. Nous ne sommes pas sûrs de la raison pour laquelle les requêtes sont toutes traitées en même temps, mais ce comportement est problématique puisque le cluster pourrait accepter une charge plus importante si les requêtes étaient traitées au rythme de leur réception par l'Ingress-Controller. Nous pouvons aussi émettre l'hypothèse que le Pod est trop sollicité et qu'il n'actualise pas les métriques, donnant la priorité au traitement des requêtes entrantes.

```
✓ server test completed successfully
✗ Statistics:
  - Total requests: 20000
  - Total success: 20000
  - Test duration: 87 seconds
  - Throughput: 229.88 requests/second
📝 Test logs available in /tmp/tmp.5HZcBnY1j0 directory
```

Screenshot du tableau de bord Grafana contenant les nouveaux résultats de notre premier scénario avec 2 processus



Screenshot du tableau de bord Grafana contenant les nouveaux résultats de notre premier scénario avec 2 processus

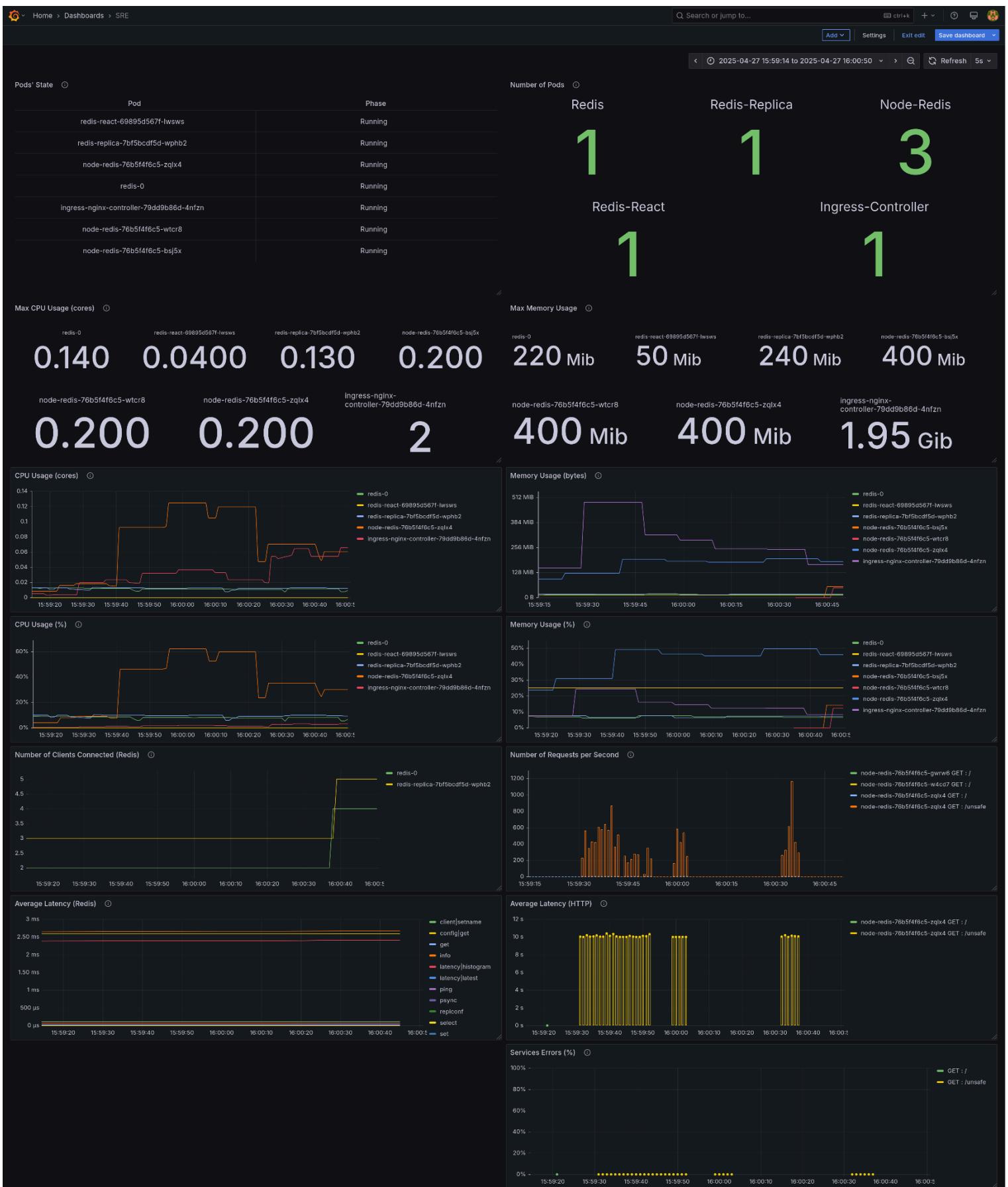
2. Rejet du second scénario

Pour rappel, nous utilisons 13 processus ouvrant chacun 1000 connexions avec un timeout de 10 secondes.

Là encore, le test passe et on peut observer que de nouveaux Pods Node-Redis sont créés, mais qu'ils ne répondent à aucune requête. Ses ressources CPU ne dépassent pas 60% et sa consommation de mémoire RAM ne dépasse pas les 50%. L'Ingress-Controller ne semble plus du tout être le facteur limitant dans ce scénario.

```
✓ pending test completed successfully
✗ Statistics:
  - Total requests: 13000
  - Total success: 13000
  - Test duration: 78 seconds
  - Throughput: 166.66 requests/second
✗ Test logs available in /tmp/tmp.0YW02PvxXN directory
```

Screenshot du tableau de bord Grafana contenant les nouveaux résultats de notre second scénario avec 13 processus



Screenshot du tableau de bord Grafana contenant les nouveaux résultats de notre second scénario avec 13 processus

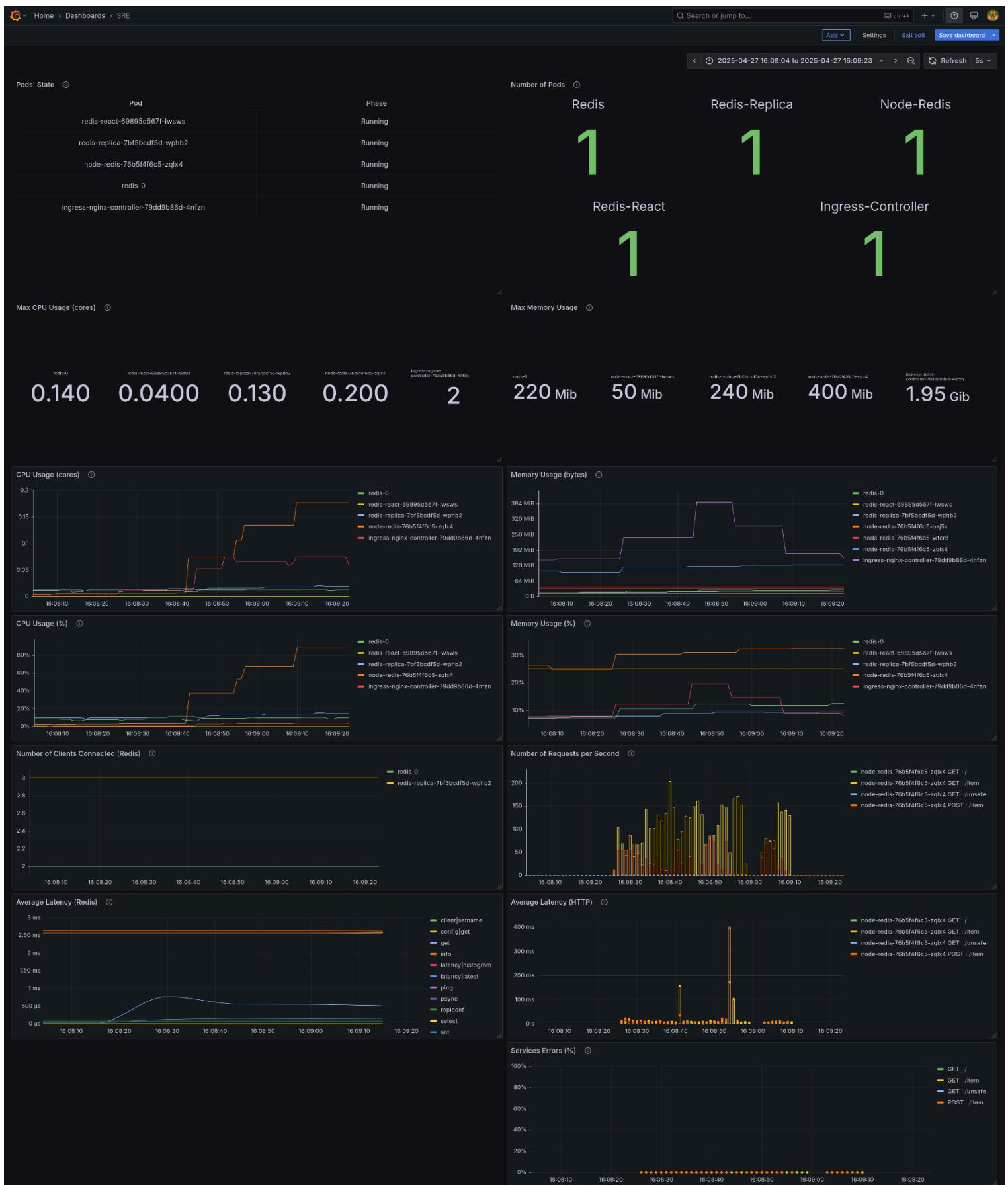
3. Rejet du troisième scénario

Pour rappel, le test utilise 64 clés différentes ainsi que des tailles aléatoires entre 1 et 50 Ko pour chaque requête d'écriture ou de lecture. Nous effectuons le test avec 9 processus, chacun effectuant 1000 requêtes par groupe de 10 (écritures et lectures confondues) toutes les 200 millisecondes.

Le test passe et le backend semble plus sollicité que dans les tests précédents en atteignant 80% d'utilisation du CPU, mais terminant le traitement des requêtes avant que d'autres Pods ne soient créés. Là encore, l'Ingress-Controller réagit très bien à la charge et ne consomme pas beaucoup de ses ressources.

```
✓ writeRead test completed successfully
✗ Statistics:
  - Total requests: 9000
  - Total success: 9000
  - Test duration: 122 seconds
  - Throughput: 73.77 requests/second
💡 Test logs available in /tmp/tmp.MfFEdPRhQa directory
```

Screenshot du tableau de bord Grafana contenant les nouveaux résultats de notre troisième scénario avec 9 processus



Screenshot du tableau de bord Grafana contenant les nouveaux résultats de notre troisième scénario avec 9 processus

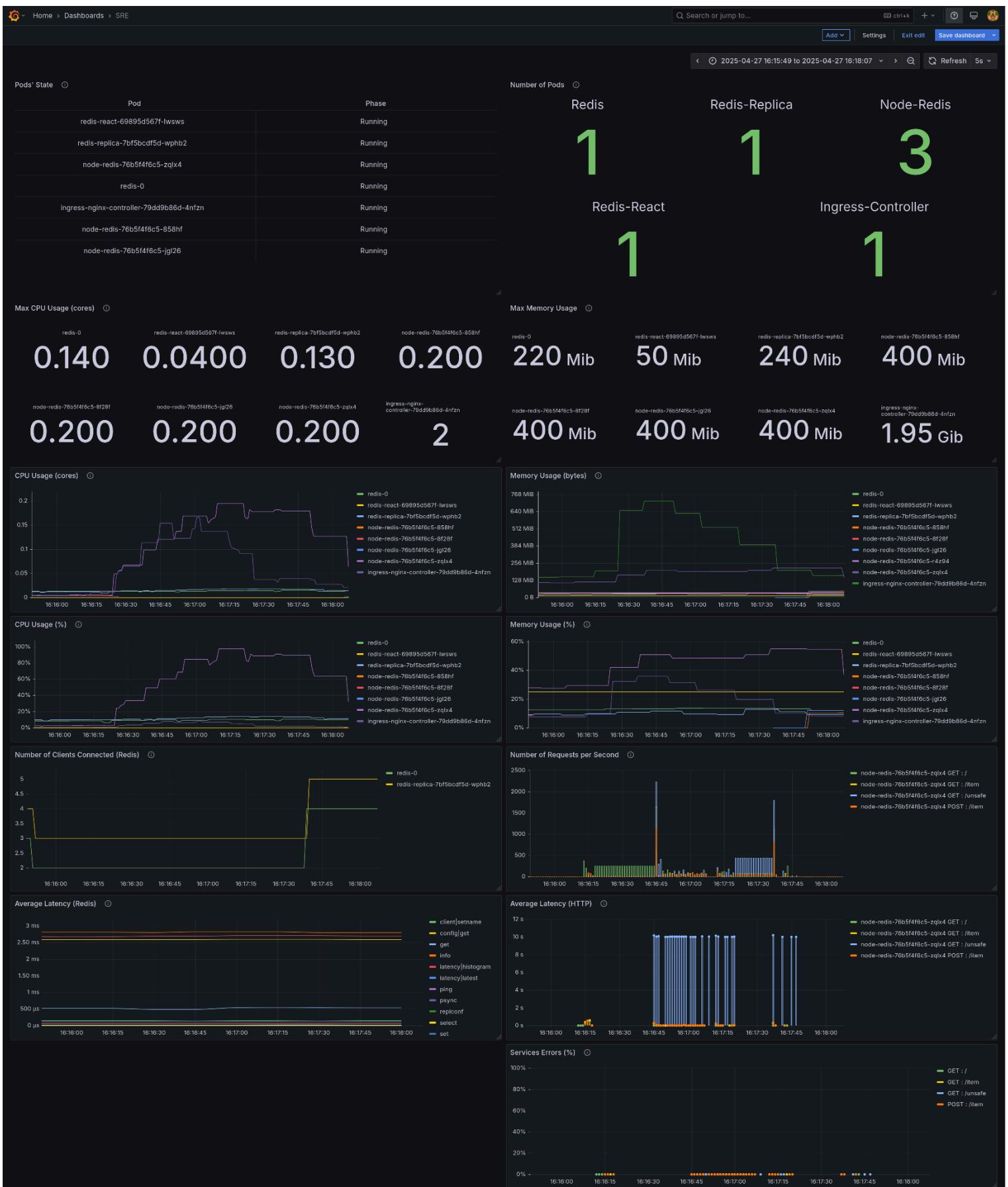
4. Rejet du dernier scénario

Pour ce test final, nous essayons de retrancher le cluster dans ses limites en doublant le nombre de processus que nous avions utilisés auparavant, soit 20 processus effectuant un des trois tests précédents aléatoirement. La seule variation est dans le nombre de requêtes envoyées par les processus de tests *server* qui passe de 10000 à 1000.

Le test passe et comme pour les tests précédents, le premier Pod Node-Redis termine de traiter toutes les requêtes avant que d'autres Pods ne soient prêts à en accepter. Son utilisation du CPU atteint les 100%, mais grâce au *CPU throttling* de Kubernetes, le Pod de crash pas et ralentit simplement son activité pour respecter les limites de sa configuration. L'Ingress-Controller supporte très bien le nombre élevé de requêtes, mais utilise tout de même 60% de sa mémoire RAM allouée. De plus, le trafic est varié pour ce test ce qui signifie que les résultats sont plutôt représentatifs d'un cas d'utilisation réel.

```
✓ combined test completed successfully
✗ Statistics:
  - Total requests: 20000
  - Total success: 20000
  - Test duration: 166 seconds
  - Throughput: 120.48 requests/second
💡 Test logs available in /tmp/tmp.Pkg5Cp1dZ0 directory
```

Screenshot du tableau de bord Grafana contenant les nouveaux résultats de notre dernier scénario avec 20 processus



Screenshot du tableau de bord Grafana contenant les nouveaux résultats de notre troisième scénario avec 20 processus

V. Conclusion

Bien que l’Ingress-Controller ne semble pas consommer un grand pourcentage de ses ressources CPU, il est important de garder cette allocation pour que les requêtes TCP restent synchronisées et que l’ordonnanceur ne le prive pas d’un temps précieux pour traiter les requêtes. En tant que goulot d’étranglement de notre cluster, il est primordial que le maximum de ressources lui soit alloué. Ainsi, nous pourrions lui en allouer plus si nous n’avions pas autant de services annexes dans le cluster (addons Minikube *metallb*, *server-metrics* et *dashboard*) ainsi que *kube-state-metrics* et *node-exporter*, pour avoir des métriques supplémentaires pour Prometheus. Grafana et Prometheus utilisent également une quantité non négligeable de ressources dans le cluster.

Ce projet nous a permis d’expérimenter l’importance du Site Reliability Engineering (**SRE**). En effet, il est primordial pour une infrastructure Cloud de pouvoir détecter les pannes et les montées en charge afin d’assurer la livraison de ressources à la demande pour le bon fonctionnement des services. De plus, c’est une discipline très importante pour les entreprises afin d’optimiser les ressources allouées aux différents services afin de réduire les coûts en ne payant pas des ressources inutiles, et pour maximiser les performances des services. Dans le cadre de notre projet, nous avons donc pu maximiser les capacités de notre cluster en distribuant les ressources efficacement.