CFA Production - Nginx autoscaling report

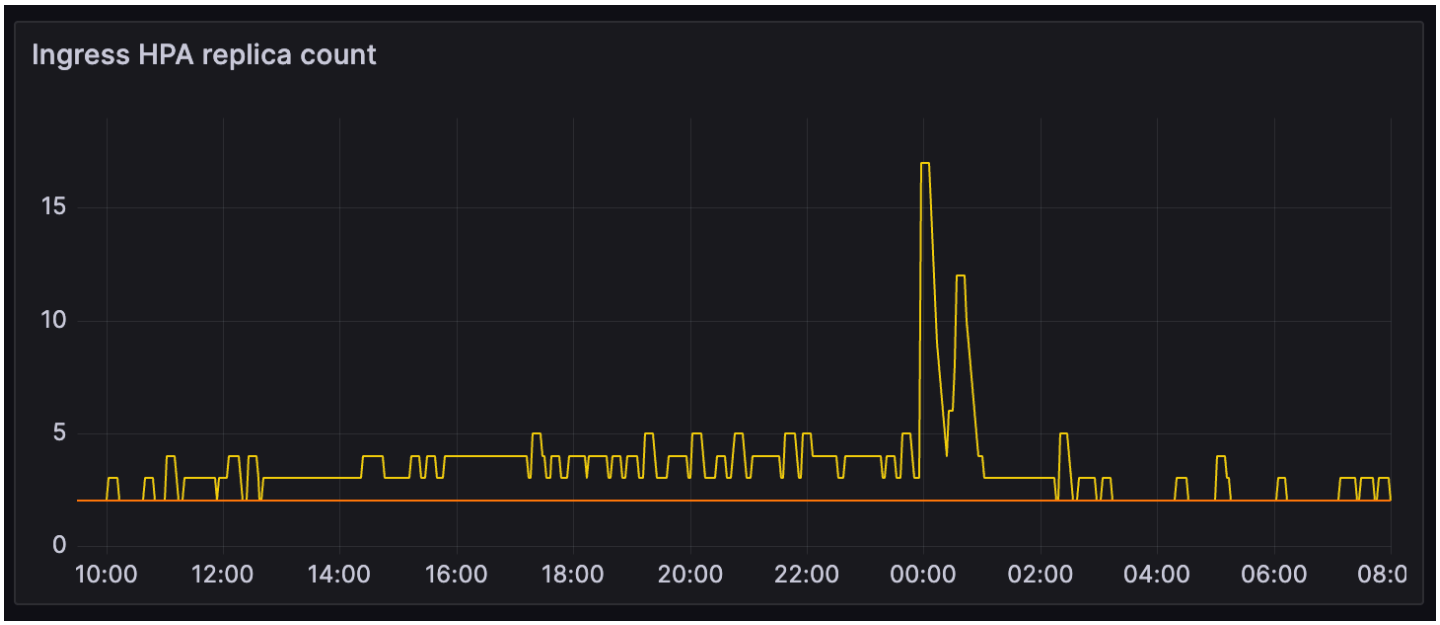
Summary

This is a **follow-up** report of the NGS-22 rollout safety report, where we finished with an Nginx observation. The conslusion of the observation was that since Nginx is autoscaling, every **scale down event** aborts **hundreds** of persistent **websocket** connections, which is triggering **auto-reconnects** on the client (SDK) sides generating a **thundering herd** effect on our upsteam services, hydra-subscription.

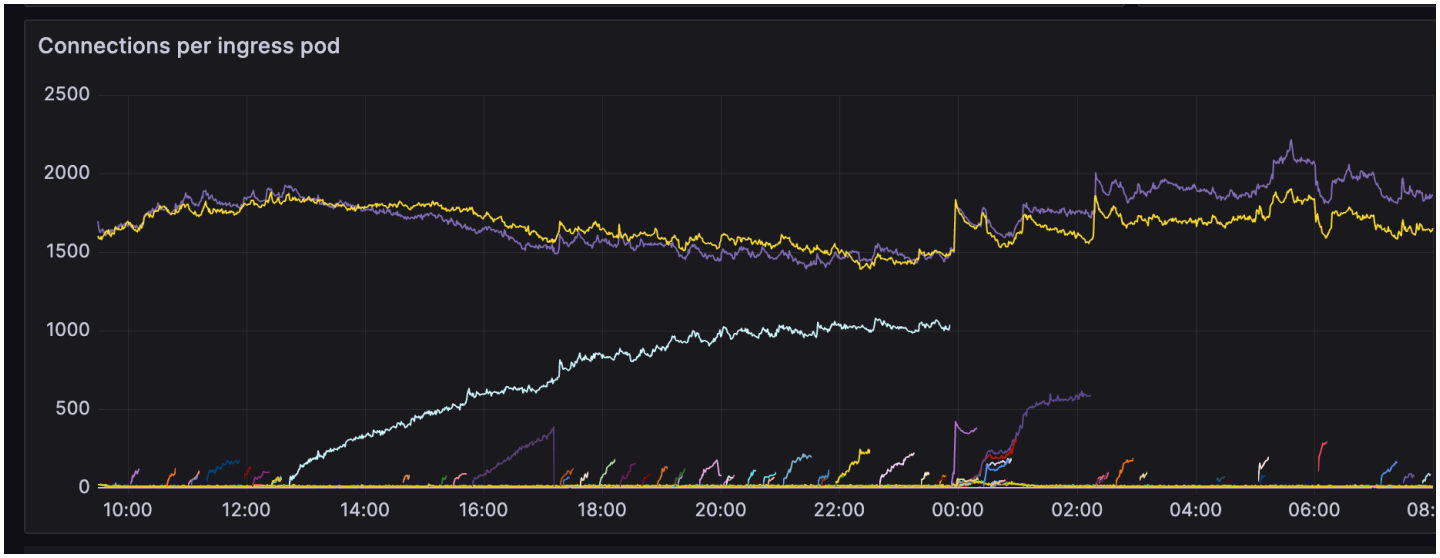
In this document we're **reviewing** the details of the **Nginx** configuration in CFA prod focusing on the **HPA** (Horizontal Pod Autoscaler) and the auotscaling feature, and provide suggestion to resolve this problem.

Recap

We'll start a quick **refresher**, here is an example of last friday's Nginx autoscaling events in CFA prod:



Every single **scale down** event is **killing** a handful of websocket **connections**, the lower the total replica count, the more (as the websocket connections per pod will be higher). The HPA is configured with minimum of **2 replicas** (**orange** line, **baseline**), whenever the HPA is stepping between 2/3/4 (yellow line), these are the most **impactful**, damaging events. Let's see this period from the **connections** per nginx pod point of view:



Twe two top lines are the **minimum 2 replicas**, keeping websocket connections alive, while at the **bottom** we see the newly spawned nginx pods collecting new connections and then getting killed with a scale down event. Whenever the line ends **abruptly**, that's a scale down event, on average it kills **~1-200 connections**, but here we're seeing even **~500**, even **~1000** connections getting killed because of the scale down event.

CFA Prod, Nginx review

We have a few Nginx deployments in CFA Prod:

```
sendai@J9LXF7X3N2 ~ % kubectl get deploy -n ingress
NAME                                            READY   UP-TO-DATE   AVAILABLE   AGE
ingress-nginx-controller                        2/2     2            2           2y66d
ingress-nginx-defaultbackend                    1/1     1            1           2y66d
login-nginx-ingress-nginx-controller            2/2     2            2           2y66d
login-nginx-ingress-nginx-defaultbackend        1/1     1            1           2y66d
public-nginx-ingress-nginx-controller           3/3     3            3           2y66d
public-nginx-ingress-nginx-defaultbackend       1/1     1            1           2y66d
public-streaming-nginx-ingress-nginx-controller 2/2     2            2           2y66d
public-streaming-nginx-ingress-nginx-defaultbackend 1/1 1            1           2y66d
```

But we're going to **focus** on the one that's taking the majority of the traffic, `public-nginx-ingress-nginx-controller`, and we're going to focus on aspects only **relevant** to the autoscaler. These are:

- generic overview
- resource allocation
- node allocation
- autoscaler configuration

Overview

It's worth noting that our Nginx deployment is having **2 containers:**

- the **nginx** image itself
- signal sciences **agent**

Both containers have resource requests and limits set (which in my opinion is a good practice, it puts the containers to the guaranteed QoS class), yet it's interesting that the **nginx** is allocated **2 CPU** cores, while the **signal sciences** Nginx module **4 CPU** cores. Signal sciences is a WAF (Web Application Firewall) now owned by Fastly and unfortunately we don't have access to its dashboards, only CFA. I'd recommend to **request** a read-only **access** to this **dashboard**, as having visibility to all components leading up to our upstream services can supercharge our investigations during odd, unusual symptoms.

Nginx container resource limit:

```
resources:
  limits:
    cpu: "2"
    memory: 6Gi
  requests:
    cpu: "2"
    memory: 2Gi
```

Signal sciences resource limit:

```
resources:
  limits:
    cpu: "4"
    memory: 200Mi
  requests:
    cpu: "4"
    memory: 10Mi
```

Node allocation

The Nginx deployments in CFA are using a **nodeSelector** to get assigned to specific nodes:

```
nodeSelector:
  app.[redacted]tier: ingress
  kubernetes.io/os: linux
```

This nodegroup consist of c5n.2xlarge EC2 machines, each EC2 is equipped with **8 CPUs**, **~21G memory** and up to **25 Gbps** (Gigabit per second) network bandwidth. This means that one Nginx pod will only fit one VM at a time, as it requests 6 CPUs out of the total 8 during scheduling. I also would like to make a comment about the nodeSelectors. NodeSelectors are amazing to separate workload, as we do in CFA Prod. **Ingresses** go to the **ingress tier**, system (like kube-system) go to the infrastucture tier, and the app goes to the app tier. Each **tier** is having its **own node labels** hence each tier has its own **dedicates** set of **nodes**.

The only observation I'd like to make here is that even though this works, yet what was surprising to me that we don't use taints & tolerations as well. With node selectors we can achieve to assign workload to specific node groups, but using it with taints & tolerations we can guarantee that nothing that's explicitly allowed will get there either. This isn't a problem at CFA now, as we're meticulously using node selector everywhere, but if anyone would start anything without a node selector, that might easily could end up on the ingress nodes as well (because the lack of taints). Usually these (nodeSelector, taints & tolerations) are configured in pair, to get explicit rules on node assignments, when that's the use case.

Nginx is also configured with multithreading and

```
worker_processes 2;
worker_connections  16384;
```

which are ideal in this current context. Please note the **worker_processes**, which is recommended to **match** the allocated **CPU** cores, so if we'd like to increase that, we should increase worker_processes with it as well.

Horizontal Pod Autoscaler

Next, we can move to **reviewing** the HPA configuration. We're using Keda for autoscaler, but **only** for **this** Nginx deployment. All the **others** are using the **default HPA**. Let's see the HPA config:

```
  metrics:
```

```
  - external:
      metric:
        name: s1-prometheus
        selector:
          matchLabels:
            scaledobject.keda.sh/name: public-nginx
      target:
        averageValue: "600"
        type: AverageValue
    type: External
  - external:
      metric:
        name: s2-prometheus
        selector:
          matchLabels:
            scaledobject.keda.sh/name: public-nginx
      target:
        averageValue: "5"
        type: AverageValue
    type: External
  - resource:
      name: cpu
      target:
        averageUtilization: 50
        type: Utilization
    type: Resource
  minReplicas: 2
```

We can see it has **three metrics** defined as potential **triggers**, one **internal** (resource, CPU) and two **external**, s1-prometheus with the threshold value of 600 and s2-prometheus with the threshold value of 5. As you can see the **minimum replica** count is **2**.

If we check the **status** of the **autoscaler**, it shows something highly **revealing**:

```
  Type    Reason             Age                   From                       Message
  ----    ------             ----                  ----                       -------
  Normal  SuccessfulRescale  47m (x1451 over 91d)  horizontal-pod-autoscaler  New size: 2;
reason: All metrics below target
  Normal  SuccessfulRescale  46m (x1102 over 90d)  horizontal-pod-autoscaler  New size: 3;
reason: external metric s1-
prometheus(&LabelSelector{MatchLabels:map[string]string{scaledobject.keda.sh/name: public-
nginx,},MatchExpressions:[]LabelSelectorRequirement{},}) above target
```

These two events are the scale down and scale up events. These lines tells us the followings:

- that we scaled up & down **1451 times** in the **past 91 days**, which is ~**15 scale events per day**.
- the mass **majority** of the scale events are indeed between **2-4 replicas**, which is the most **impactful**

- the **source trigger** for all the events are coming from **s1-prometheus**, external metric

s1-prometheus

s1-prometheus is an **external**, **prometheus** metric:

```
  - metadata:
      query: avg(sum(irate(container_network_receive_bytes_total{job="kubelet",
metrics_path="/metrics/cadvisor",namespace="ingress"}[2m]) * on (namespace,pod)
group_left(workload,workload_type) namespace_workload_pod:kube_pod_owner:relabel{
namespace="ingress", workload=~"public-nginx-ingress-nginx-controller",
workload_type="deployment"}) by (pod)) / 1024
      serverAddress: http://prometheus-operated.monitoring:9090
      threshold: "600"
      unsafeSsl: "true"
    type: prometheus
```

Let's break down this **metric**, so it's **easier** to understand. It consist of **two queries** and a left **join** via the namespace and pod labels. The first **query**:

```
sum(irate(container_network_receive_bytes_total{job="kubelet",
metrics_path="/metrics/cadvisor",namespace="ingress"}[2m])
```

This is showing the i**ncoming network bytes** of the **container** with 2m time aggregation period.

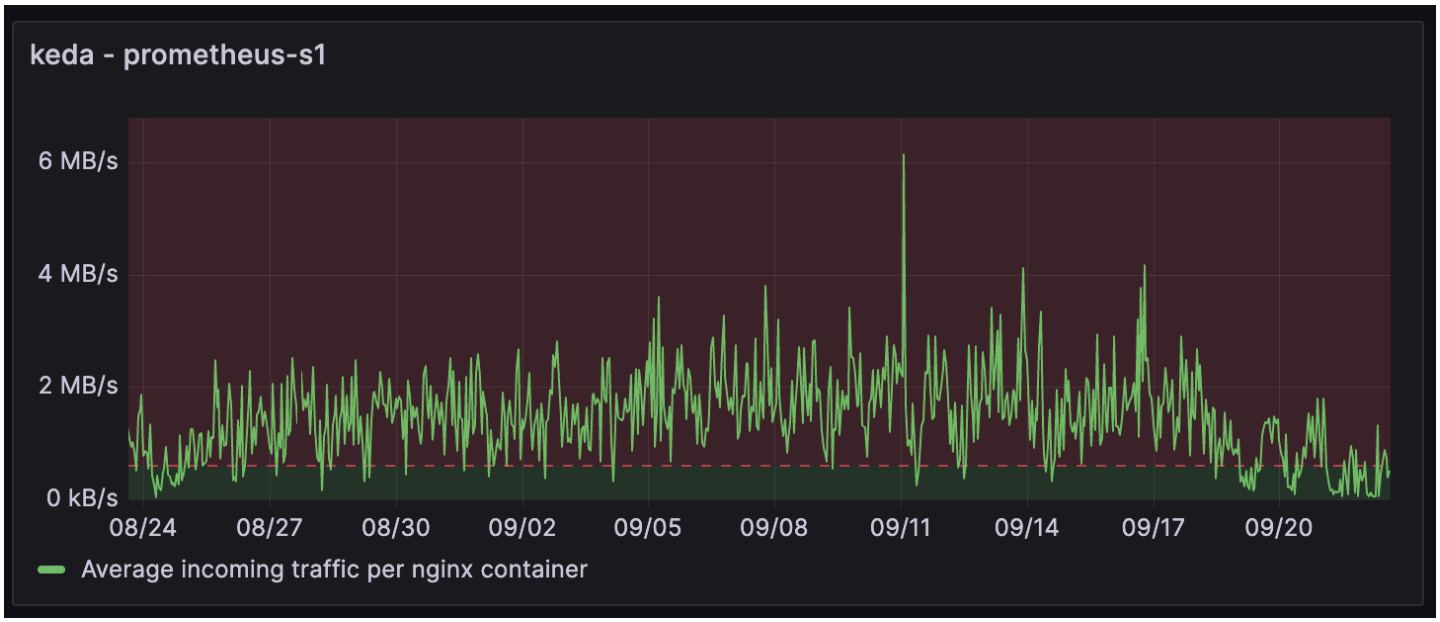
```
namespace_workload_pod:kube_pod_owner:relabel{ namespace="ingress", workload=~"public-nginx-
ingress-nginx-controller", workload_type="deployment"}
```

This metric is **querying** the **pod names** and **deployments**, related to the nginx deployment. The reason why we need to do this, because the **first metric** even though contains the pod names, it doesn't **contain** the **deployment names**. Using these two queries and joining them, we can **query** the **incoming traffic bytes** by the **deployment name**.

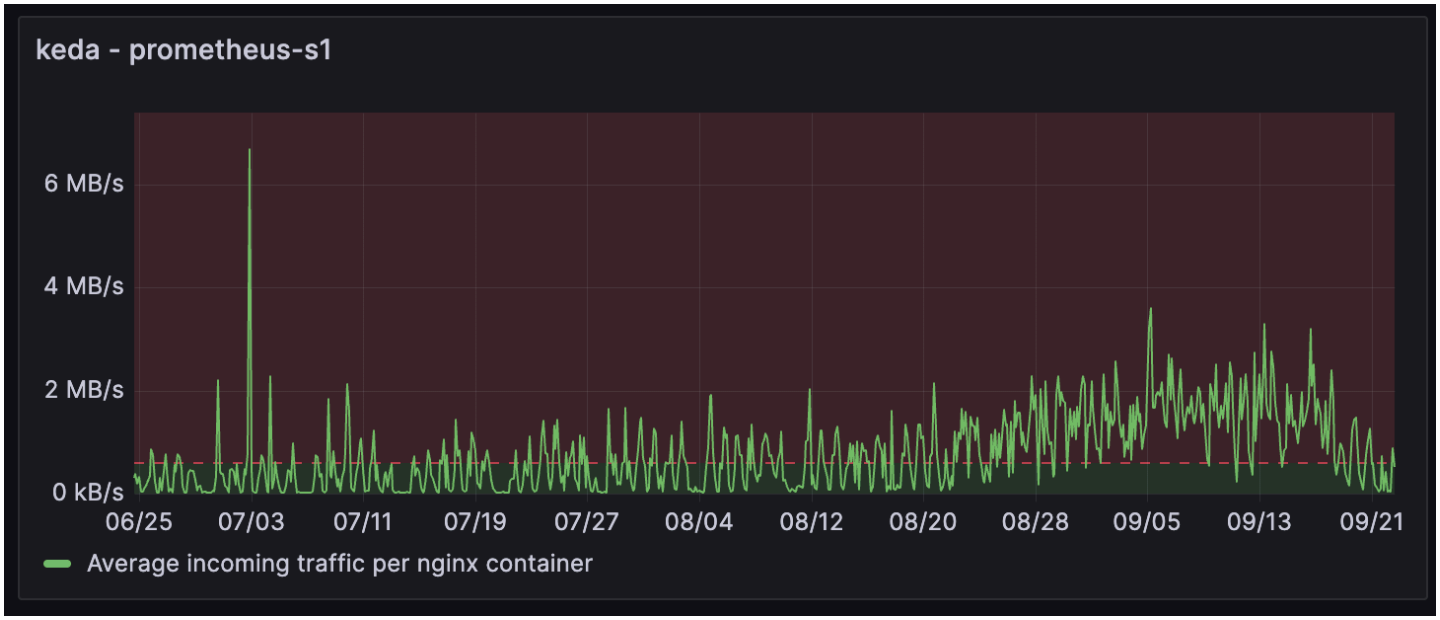
Once we have these, it **adds** them up and gets the **average**, finally **divide** by 1024 to get **kilobytes** instead if bytes.

In short, **if** the **average incoming traffic rate** (per second) of the **nginx pods** gets above **600kbyte/s, trigger a** scale up** event. My opinion is that in **production** grade machines it's **rare** to see network related scale up triggers, for two reasons. One, production scale VMs usually have shockingly **high bandwidth capacities** (don't forget, we pay for outgoing traffic in AWS, they don't want to limit paying their customers), two, if for some reason the bandwidth is not sufficient, we should have an alert on it and move to bigger VMs with more bandwidth (but, again, I've never seen this being a problem, so this is all theoretical).

Let's **visualize** this, last 30 days, the **red dashed** line shows the **trigger threshold** of 600kbytes:



And 90 days:

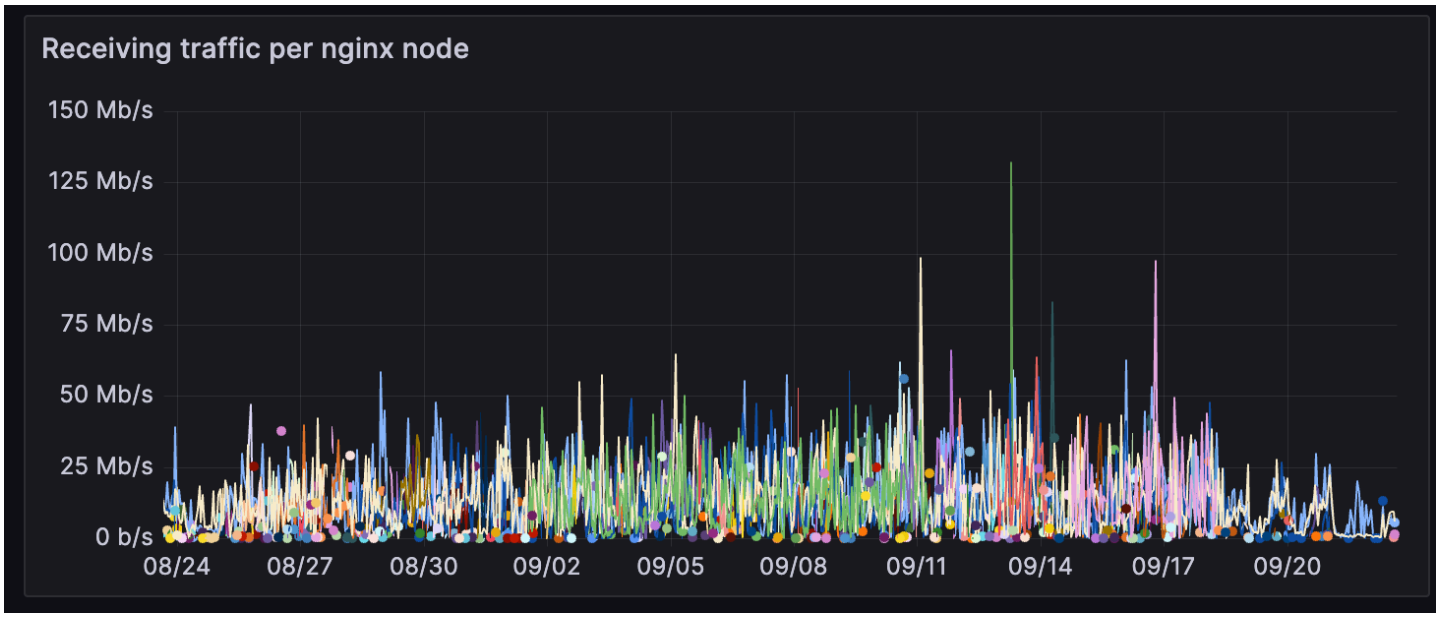


We just **confirmed** that we've been hitting the **600 kbyte/s** threshold **constantly**.

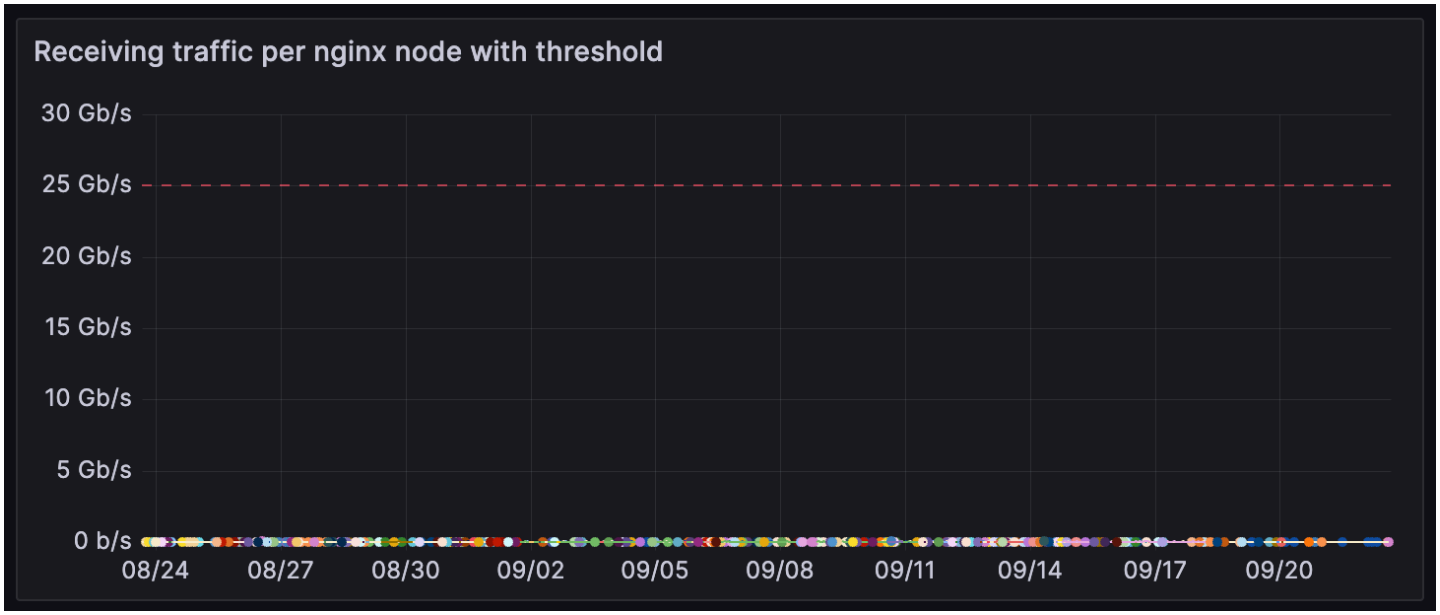
Actual workload

The proposed fix for this is to **disable autoscaling** and configure **alerting** for a reasonable (~70%) threshold, so we'd be notified if the Nginx would struggle, despite Nginx being super lightweight and efficient. In order to recommend this, I need to put into **perspective** our current Nginx **load** in CFA production. Please note that so far we've been talking about bytes as the container is using bytes as a metric, from here on we're talking about bits, as the EC2 bandwidth spec is in bits.

So, let's see our incoming traffic per ingress EC2, last 30 days:



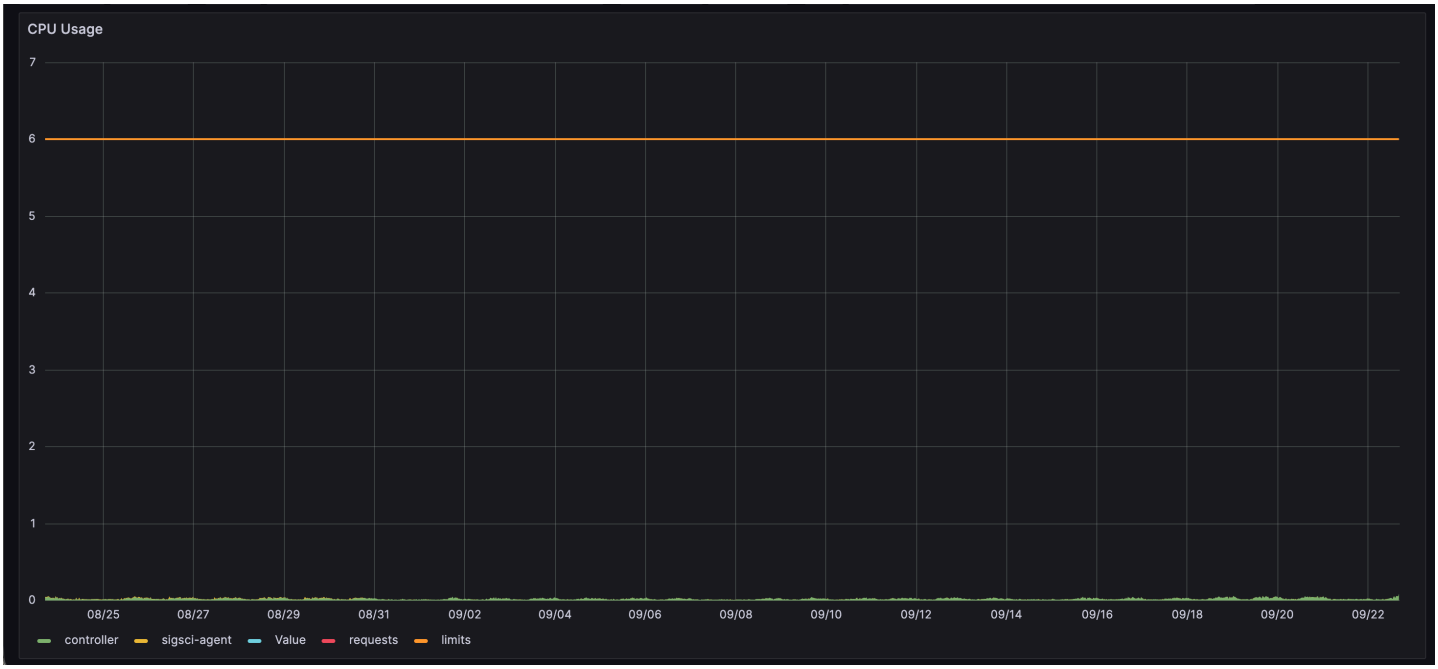
These are **nodes** (not individual containers) and the traffic metric is in **bits**. Our EC2 VM type support bandwidth up to **25Gbps**, and we're **barely** getting up to 100Mbps. To put this into perspective, I created another chart, **same data** but now the scale is set to 30Gbps with a red dashed threshold line of **25Gbps**:



These charts **show** that we're **safe** and **far** from being **saturated** on our **incoming network bandwidth**, our traffic level barely shows up on the chart, **compared** to the **capacities** of our **EC2 types** we're using.

CPU

Just to be fair, I mentioned before that the **Nginx container** in the Nginx pod only have **2 CPUs allocated**, while the signal sciences **agent** has **4 CPUs**, so I had to check the CPU utilization of both containers, just to ensure **we're safe** to disable **HPA** for CPU as well. For this, I used the regular CPU chart from the Kubernetes dashboards checking for last 30 days, and the **green** shows the **CPU usage** (in millicores) of **Nginx container** and the **yellow** shows the **sigsci agent**, while the chart is scaled to 6 cores (2 core for Nginx + 4 core for sigsci).



So, we're **safe** with **CPU** as well.

Action plan

The proposal is that since our **workload isn't** a good **fit** for **autoscaling**, we're going to remove autoscaling. We're going to do it in **two** steps:

Step #1

- **scale up** the Nginx **deployment** to **4 replicas**
- effectively **disable** the autoscaler
    - by setting both **min** and **max** to **4 replicas**
- to ensure we're **not missing out** on the **principles** of the autoscaling **triggers**, make sure we'll **alert** on them
    - **review** if we already have **alerts** in place (I'd think we do for most, like CPU and network threshold)
    - if not, **configure** them as **alerts**, with sensible thresholds

This step is about **disabling** the **HPA** (by setting min/max the same), **preparing** for **removal** (by scaling up the deployment) and **making sure** we're **not missing out** on anything that the **HPA** intended to **protect** Nginx from (via alerts). Please note that Keda supports literally pausing the HPA via annotations, but setting min/max the same value is a clearer way to achieve this. This way if someone looks at the configuration can clearly see that it's not meant to scale, it's easy to miss a Keda specific annotation, but min/max is the first thing we all check.

Also, **why** do it in **two steps** instead of one?

- we only do **gradual changes** to CFA Production with **clear rollback options**
- if we'd just remove the HPA, that'd **leave** the deployment replica **count** exactly where it was **when** the **HPA** was **removed**, which we can't control. This is an **uncertainty** factor we don't do in **production**. There is a feature in Keda which **controls** whether the replica count should be **restored** to the original during a removal, but it's **disabled** by default and we don't have it enabled either.

Step #2

- remove **Horizontal Pod Autoscaler** for Nginx

I'd wait for a **month** to ensure everything is fine and **settled**, this is a **cleanup** at this point, the system is reliable and fully functional, even without this step.

Extra steps

- propose to **remove** the **HPA** for **Valet** ingress as well
  - Valet is not using Keda, it is using regular HPAs
  - @[redacted] already created a **ticket** to **track** this, PLA-561

Side effects

Finally, I'd like to take a moment to mention that once we **apply** these, one **side effect** will **disappear**. These constant connection killer scale down events today help to **rebalance** the connections between our ingresses, as from time to time it forces our clients to reconnect. By fixing this problem, the side effect will disappear as well. I assume this shouldn't be a problem, yet we should be aware of it, as it may **affect** some **chart patterns** we've been used to.

sendai @[redacted] 2025