



2023 - 1C

75.73 - Arquitectura de Software

TP N°2 - Grupo DDL

Integrantes:

- Garibotti, Borja (106124)
- Langer, Santiago Tobías (107912)
- Lofano, Tomás (101721)
- Sotelo Guerreño, Lucas Nahuel (102730)

Fecha de entrega: 29/06/23

Índice

Introducción	2
Desarrollo	3
Ataque	4
GUID	5
Gráficos de componentes y conectores	7
Useless Facts	9
Useless Facts - Caso base	9
Useless Facts - Cache	13
Useless Facts - Réplicas	17
Useless Facts - Rate limiting	21
Useless Facts - Análisis	24
Metar	26
Metar - Caso base	26
Metar - Cache	30
Metar - Réplicas	34
Metar - Rate limiting	38
Metar - Análisis	41
Space news	42
Space news - Caso base	42
Space news - Cache	46
Space news - Réplicas	50
Space news - Rate limiting	53
Space news - Análisis	56
Conclusiones	57
Referencias	59

Introducción

En este segundo trabajo práctico evaluamos cómo se comportan los escenarios estudiados en el TP1, pero esta vez utilizando Cloud Computing. Además, a cada una de nuestras respuestas le agregaremos un GUID¹ identificatorio provisto por otro servicio.

Al igual que en el primer informe, evaluaremos los atributos de Disponibilidad, Rendimiento y Escalabilidad. Para esto utilizaremos algunas herramientas nuevas como Datadog, Terraform, Ansible y PM2.

Finalmente, compararemos las conclusiones de este informe con el Trabajo Práctico 1.

¹ Globally Unique Identifier

Desarrollo

Como ya mencionamos al principio del informe, utilizamos las máquinas virtuales de Microsoft Azure para contener nuestra aplicación (desarrollada en el TP1). Para conectarnos a nuestras computadoras en la nube utilizamos SSH² y el par de claves generado por Azure. Luego, para evitar gestionar manualmente mediante la interfaz web de Azure, utilizamos Terraform.

Terraform gestiona todas las VMs que uno cree con ella, y permite escribir **declarativamente** cómo queremos que estas se generen. Así, Terraform gestiona qué necesitamos y cómo, encargándose ella misma de proveer lo que habíamos solicitado en las variables de configuración.

A su vez, para instalar automáticamente todo el software necesario en cada VM usamos Ansible. Esta herramienta permite detallar tareas a ejecutar en cada máquina de manera declarativa, por lo que le indicaremos qué software debe instalarse y cómo se debe ejecutar la aplicación.

Por último, para la visualización y análisis de datos usamos Datadog.

Datadog es una potente herramienta que permite, entre otras cosas, recibir la información registrada por nuestras máquinas virtuales y Artillery, formar métricas, detectar errores y reportar incidencias. A lo largo de todo este trabajo práctico visualizaremos la información mediante gráficos de Datadog.

Para este trabajo no usamos un load-balancer de NGINX ya que Azure provee uno. De la misma manera, Azure provee el caché de Redis. Estos recursos son fácilmente configurables mediante Terraform.

Durante el desarrollo del TP surgieron dos problemas inesperados. Primero, el servicio proveedor de GUIDs no lograba satisfacer nuestras necesidades en ciertas circunstancias (véase apartado “GUID”). Segundo, sufrimos un ataque de un agente desconocido.

² Secure SHell

Ataque

Mientras estudiábamos la información en Datadog, notamos que nuestra aplicación tuvo llamados inesperados a ciertas direcciones que no usábamos en Artillery (como `/portal/redlion`, `./env` o `/files`).

Jun 24 01:29:45.984	🌐 pm2	GET	/phpMyAdmin/index.php	385 µs
Jun 24 01:29:45.428	🌐 pm2	GET	/sql/phpMyAdmin/index.php	642 µs
Jun 24 01:29:44.587	🌐 pm2	GET	/db/?/index.php	528 µs
Jun 24 01:29:44.299	🌐 pm2	GET	/phpmy/index.php	496 µs
Jun 24 01:29:43.453	🌐 pm2	GET	/mysql/sqlmanager/index...	461 µs
Jun 24 01:29:41.762	🌐 pm2	GET	/db/websql/index.php	938 µs
Jun 24 01:29:41.208	🌐 pm2	GET	/?/index.php	508 µs
Jun 24 01:29:40.644	🌐 pm2	GET	/?/index.php	459 µs
Jun 24 01:29:39.508	🌐 pm2	GET	/pma/index.php	658 µs
Jun 24 01:29:39.220	🌐 pm2	GET	/db/?/index.php	483 µs
Jun 24 01:29:38.442	🌐 pm2	GET	/administrator/phpmyadmi...	463 µs
Jun 24 01:29:37.879	🌐 pm2	GET	/db/?/index.php	627 µs
Jun 24 01:29:37.591	🌐 pm2	GET	/program/index.php	566 µs
Jun 24 01:29:36.180	🌐 pm2	GET	/?/index.php	441 µs
Jun 24 01:29:35.903	🌐 pm2	GET	/?/index.php	630 µs
Jun 24 01:29:35.060	🌐 pm2	GET	/administrator/web/index...	741 µs
Jun 24 01:29:30.633	🌐 pm2	GET	/administrator/phpMyAdmi...	828 µs
Jun 24 01:29:29.233	🌐 pm2	GET	/mysql/index.php	628 µs
Jun 24 01:29:28.944	🌐 pm2	GET	/administrator/web/index...	516 µs
Jun 24 01:29:27.532	🌐 pm2	GET	/mysql-admin/index.php	676 µs
Jun 24 01:29:27.250	🌐 pm2	GET	/sql/?/index.php	464 µs

Algunas requests a direcciones inesperadas

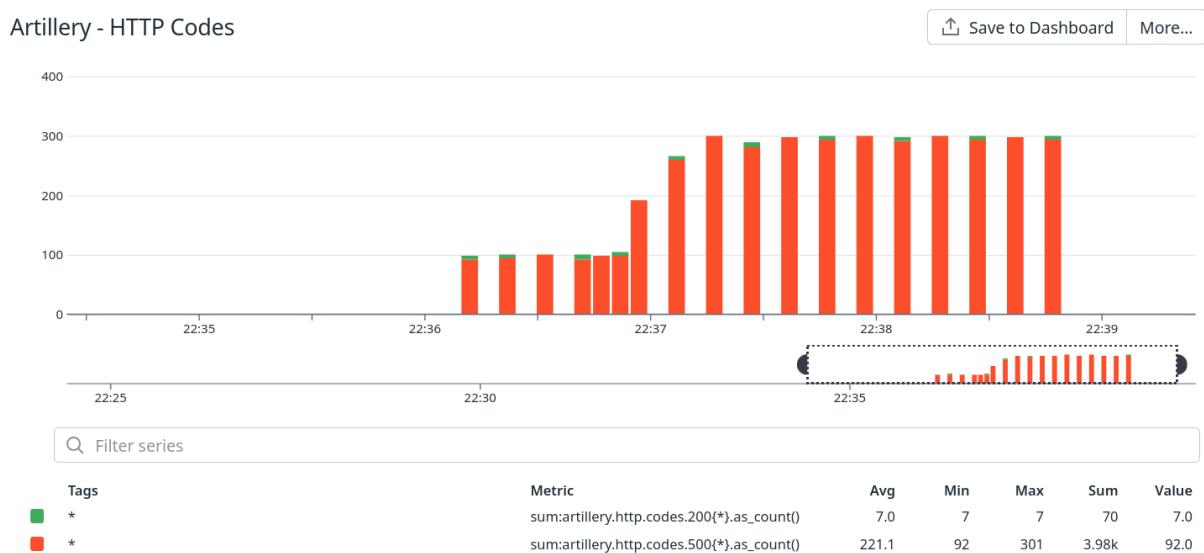
Concluimos que alguien intentó vulnerar nuestra aplicación en busca de exploits y conseguir información sensible. Afortunadamente no tuvo un mayor impacto en el desarrollo del trabajo práctico, no causaron costos inesperados y no se comprometió información privada.

Esta experiencia fue una demostración en la vida real de la importancia de la ciberseguridad, usar claves de acceso, cerrar nuestros puertos y armar VPNs, entre otras posibles medidas de seguridad.

GUID

Para satisfacer el pedido de la consigna, agregamos un nuevo servicio que provee un GUID (Identificador único global) a cada respuesta de la API. Este servicio corre en un contenedor de Azure que se levantó gracias a la configuración de Terraform a partir de una imagen de Docker Hub.

Al ejecutar las primeras pruebas, notamos que nuestra aplicación comienza a fallar porque el servicio nos pone un rate-limit después de hacer 7 solicitudes seguidas.



Ejemplo del rate-limit del servicio de GUID

Es evidente que debemos encontrar la manera de sobreponer este problema para que el sistema siga funcionando cuando el servicio no esté disponible.

En este caso, aplicar rate limit no sirve porque el servicio puede dejar de funcionar si más de 7 usuarios hacen una solicitud al mismo tiempo.

Se podría comenzar a solicitar GUID desde que el sistema se inicie e ir almacenarlos en la caché, pero tienen que descartarse luego de usarse (ya que los identificadores son únicos) y no podemos asegurar tener suficientes disponibles cuando aumente la cantidad de solicitudes.

La replicación parece una solución posible. Teniendo más instancias que nos proveen un mismo servicio, cuando una deja de estar disponible las otras pueden respaldarlas. Sin embargo, consideramos otra opción que resultó más simple que configurar un load balancer y aumentar la cantidad de instancias.

Finalmente, agregamos a la aplicación el paquete `uuid3` de npm que permite generar un GUID de manera local. Por lo tanto, cuando el servicio remoto de GUID no está disponible, le pedimos a la aplicación que genere uno.

³ Universally Unique IDentifier

Entendemos que esta solución tiene sus límites: desconocemos la implementación del servicio original, por lo cual puede ser que haya acciones que nuestra aplicación no pueda hacer. Por ejemplo, si el servicio remoto llevaba un registro de los GUID generados, nosotros ahora no lo estamos haciendo y no podemos verificar si un JSON fue generado por nuestro sistema o no.

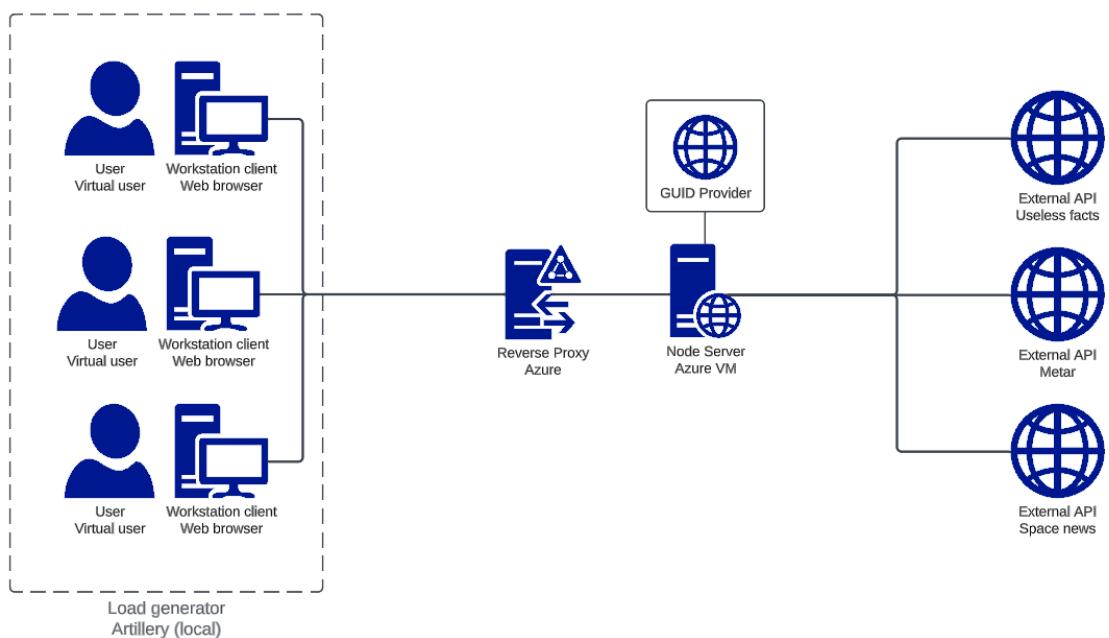
Cuando el servicio remoto de GUID aplica el rate-limit, nos devuelve el código de estado "429 - Too many requests". Además provee en el encabezado de la respuesta la cantidad de segundos en que se puede reintentar la comunicación.

Para disminuir la cantidad de solicitudes fallidas con este código de estado, calculamos en qué momento el servicio volverá a estar disponible, y mientras no lo esté, generamos el GUID de manera local.

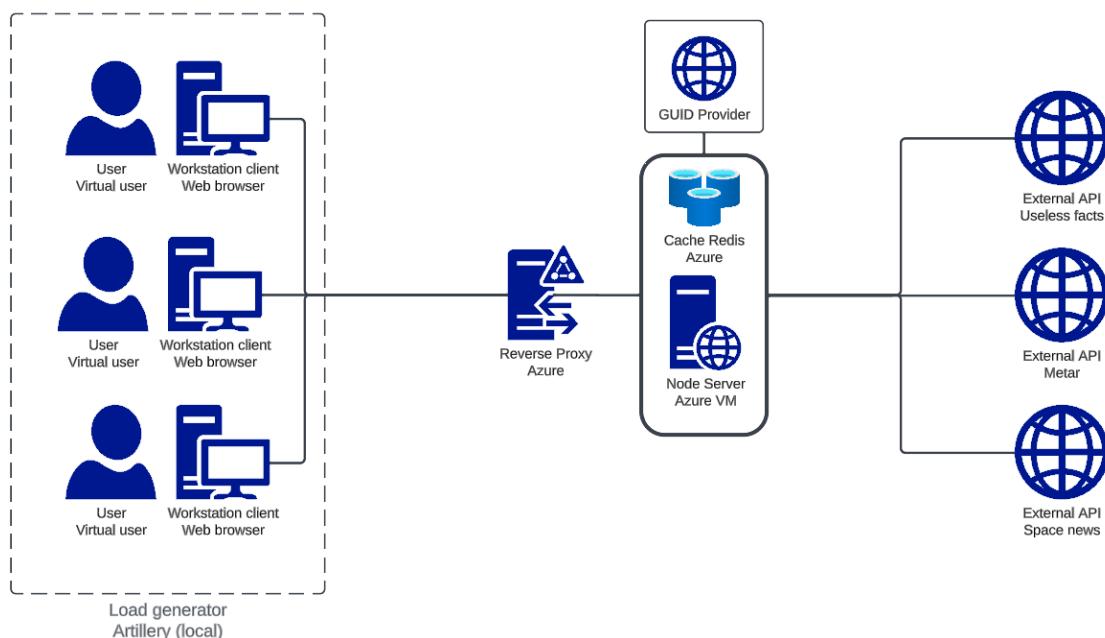


Ejemplo luego de solucionar el rate-limit del servicio de GUID

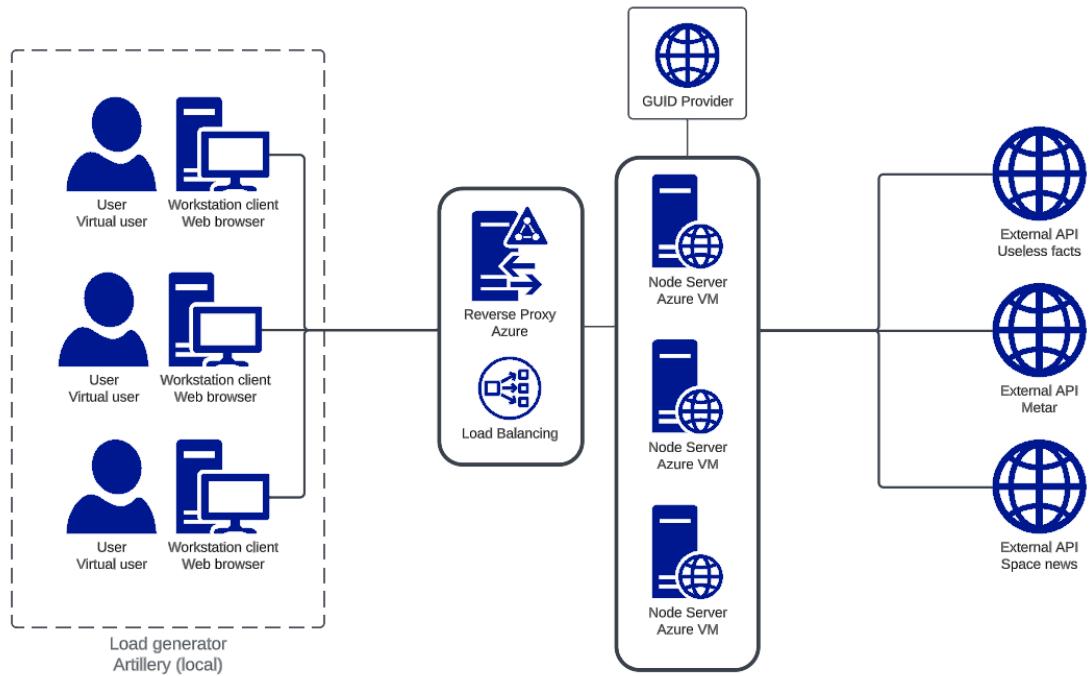
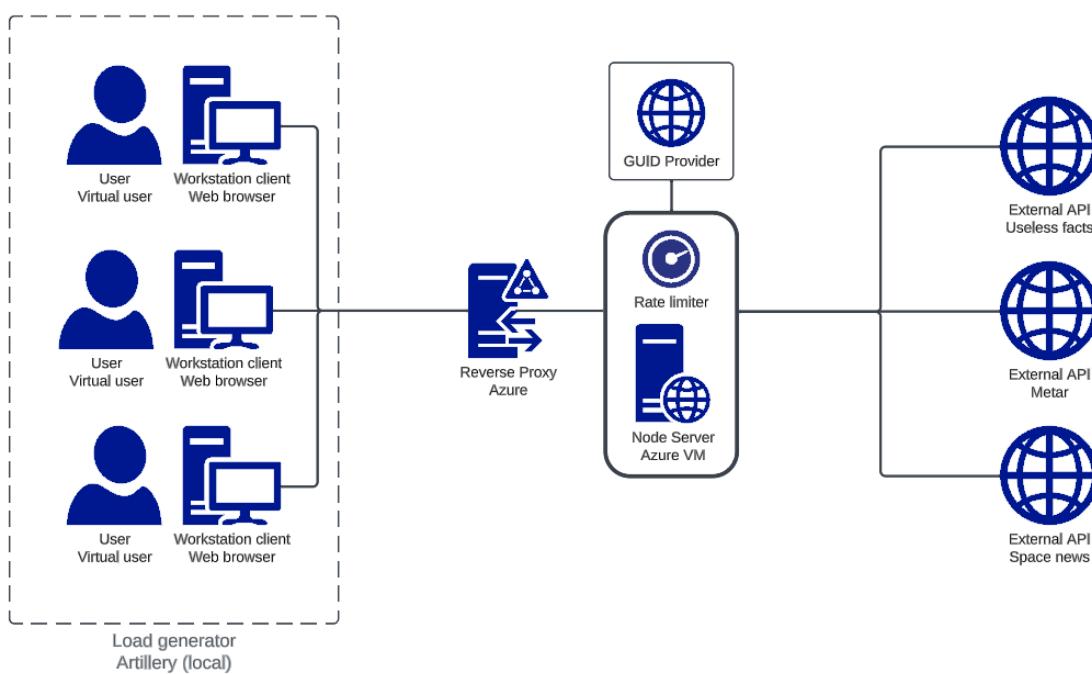
Gráficos de componentes y conectores



Caso base



Estilo arquitectónico cache

*Estilo arquitectónico replication**Táctica rate limiting*

Useless Facts

Se consume un servicio que devuelve distintos hechos inútiles en cada request.

En los siguientes gráficos de Datadog (y generados con escenarios de Artillery) se muestra cómo responde la aplicación y qué impacto tienen distintas tácticas, dados los siguientes niveles de carga:

- **Warm Up:** 10 requests por segundo durante 45 segundos.
- **Ramp Up:** aumenta la cantidad de requests progresivamente hasta llegar a 85 requests.
- **Plain:** 85 requests por segundo durante 90 segundos.

Useless Facts - Caso base

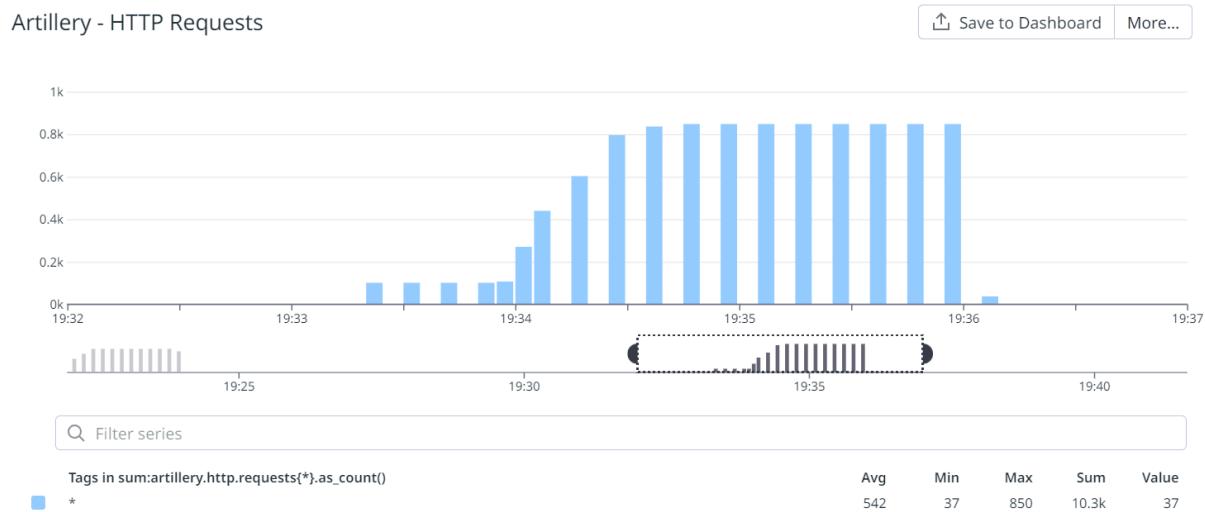


Gráfico 1 de Facts caso base - Número de requests

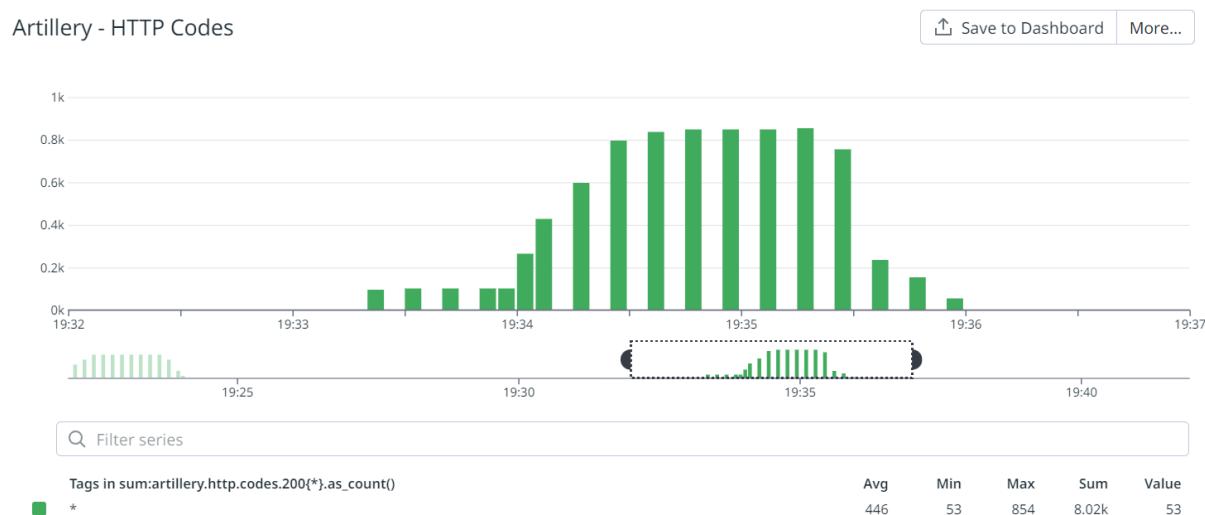


Gráfico 2 de Facts caso base - Códigos de respuesta de requests

Artillery - Error Type

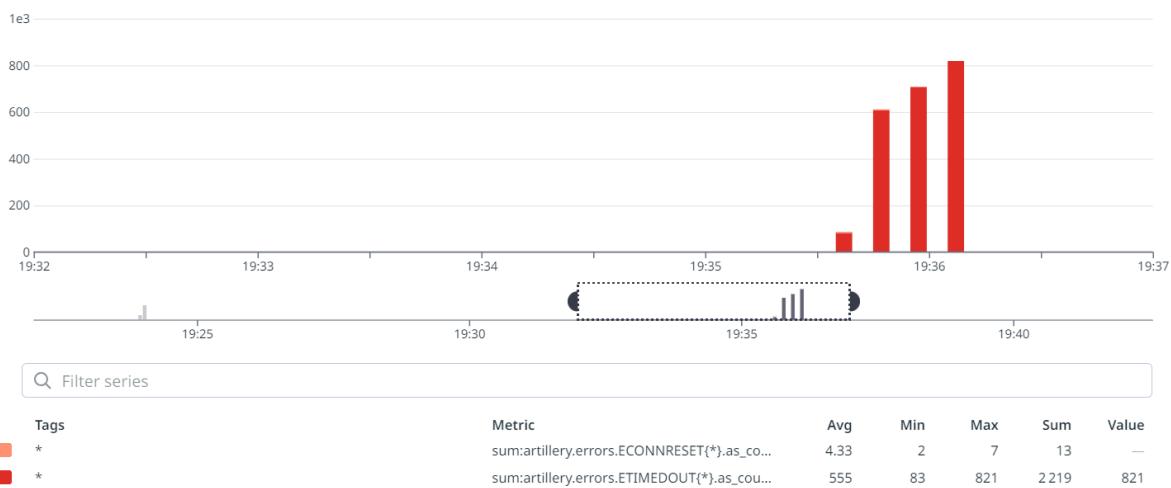
[Save to Dashboard](#) [More...](#)


Gráfico 3 de Facts caso base - Tipos de error

System CPU Usage

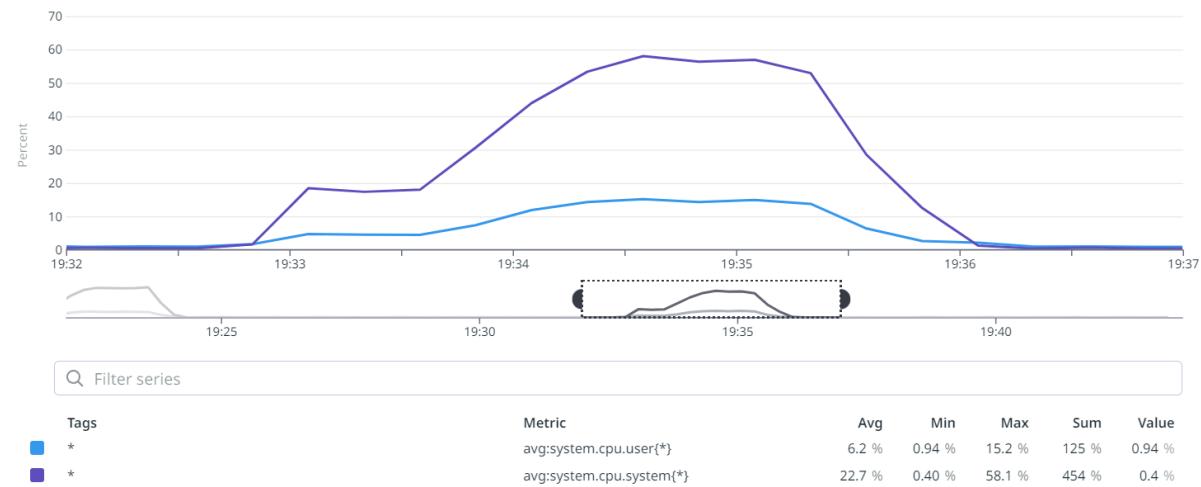
[Save to Dashboard](#) [More...](#)


Gráfico 4 de Facts caso base - Uso de CPU

System Memory Usage

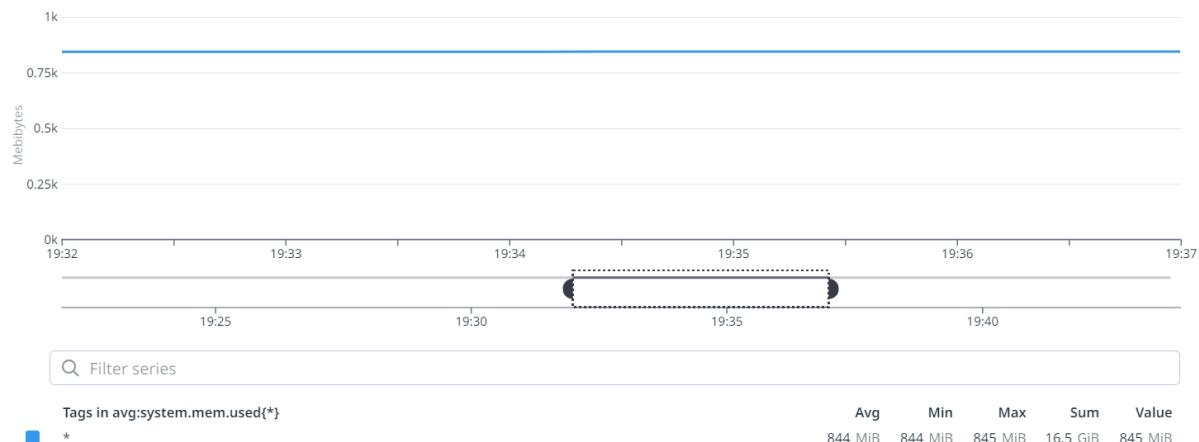
[Save to Dashboard](#) [More...](#)


Gráfico 5 de Facts caso base - Uso de memoria

Response time

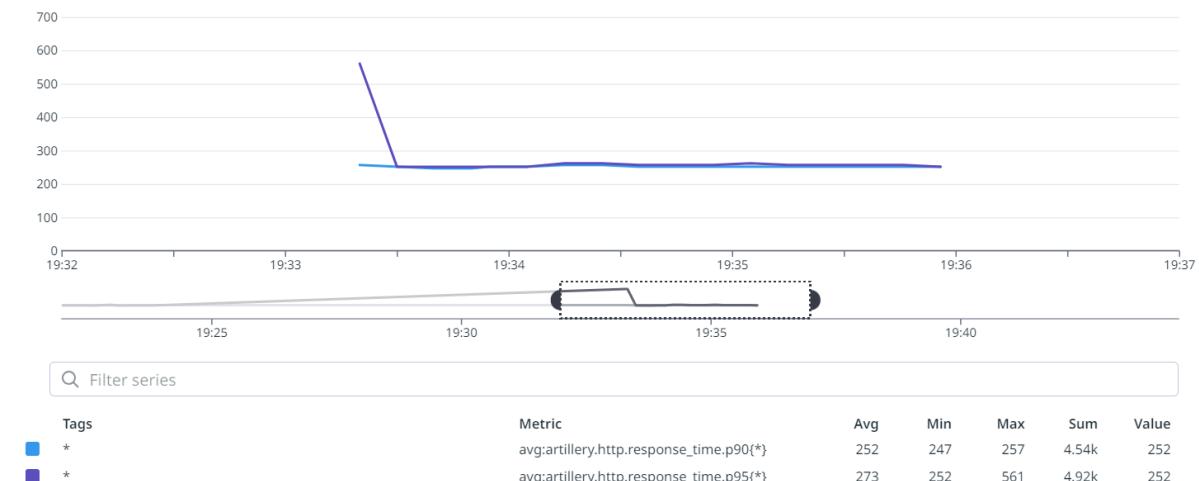
[Save to Dashboard](#) [More...](#)


Gráfico 6 de Facts caso base - Tiempo de respuesta

Max Latency (fact_provider)

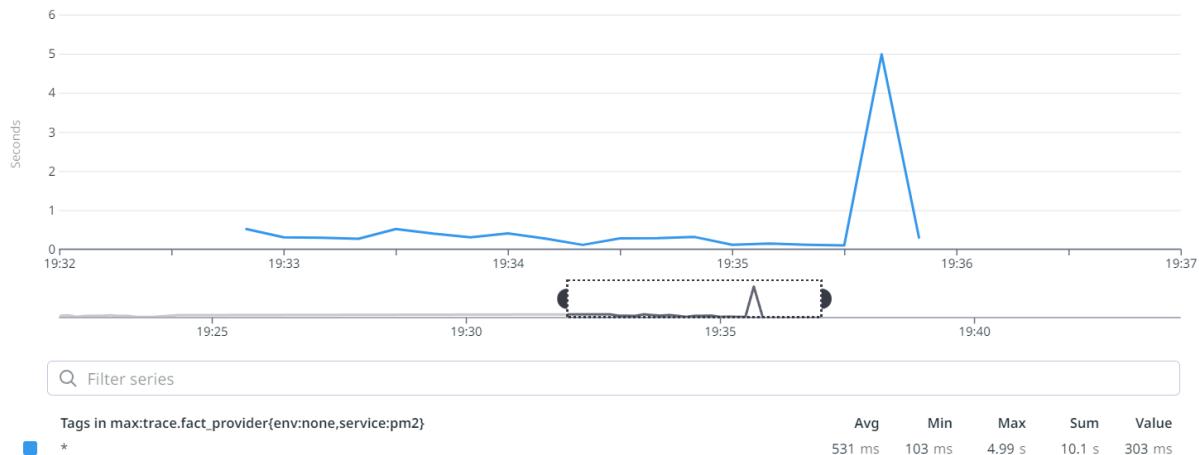
[Save to Dashboard](#) [More...](#)


Gráfico 7 de Facts caso base - Latencia (fact provider)

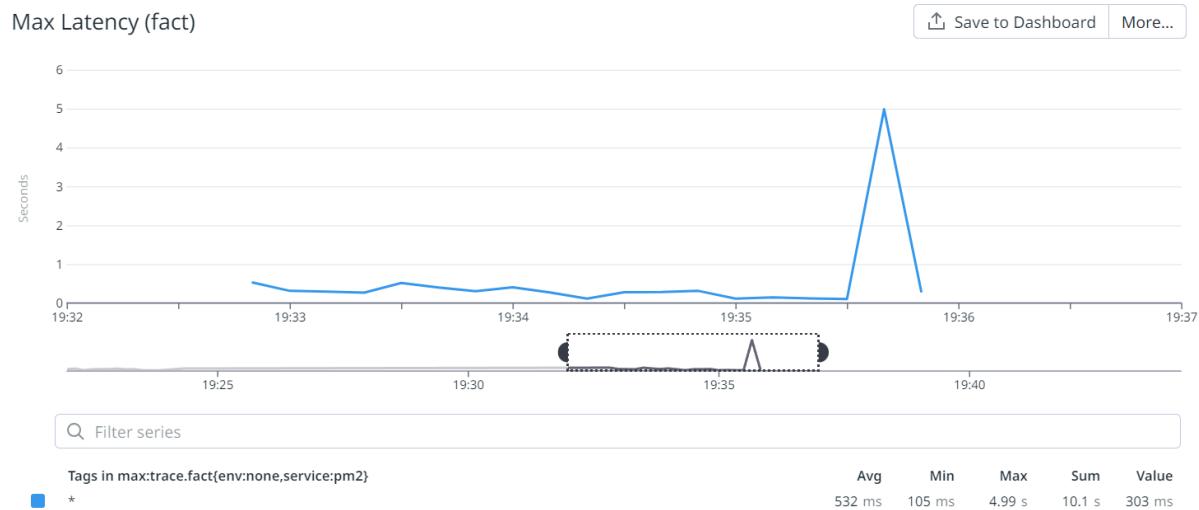


Gráfico 8 de Facts caso base - Latencia (fact)

Este escenario de carga genera 10300 requests, de las cuales el caso base responde exitosamente 8000, pero fallan 2300 requests al finalizar el escenario.

En cuanto a consumo de recursos, el uso del CPU escala junto a la demanda hasta llegar al 58%. El uso de memoria se mantiene constante a lo largo de todo el escenario, en 840 mb.

En cuanto a response time, el P90 se mantiene estable en 250 ms, mientras que el P95 tiene un pico de 560 ms al principio.

La latencia se mantiene constante entre los 100 y los 500 ms, hasta que tiene un gran pico de 5000 ms al final del escenario.

Useless Facts - Cache

Al igual que en el TP1, aplicamos *active population* trayendo y cacheando una cierta cantidad de respuestas de la API Useless Facts, luego a nuestros clientes les devolvemos las respuestas directo del cache. De esta manera, estamos buscando aumentar la performance de nuestra aplicación respondiendo con un elemento del cache en vez de comunicarse con la API y esperar su respuesta. Cuando se terminan las respuestas almacenadas, buscamos nuevas.

Además, podemos seguir dando servicio (por un tiempo limitado) cuando la API externa deje de responder, aumentando nuestra **disponibilidad**.

Para este caso, realizamos un *active population* de 100 facts por vez (es decir, llenamos el cache con 100 facts cuando se vacía).

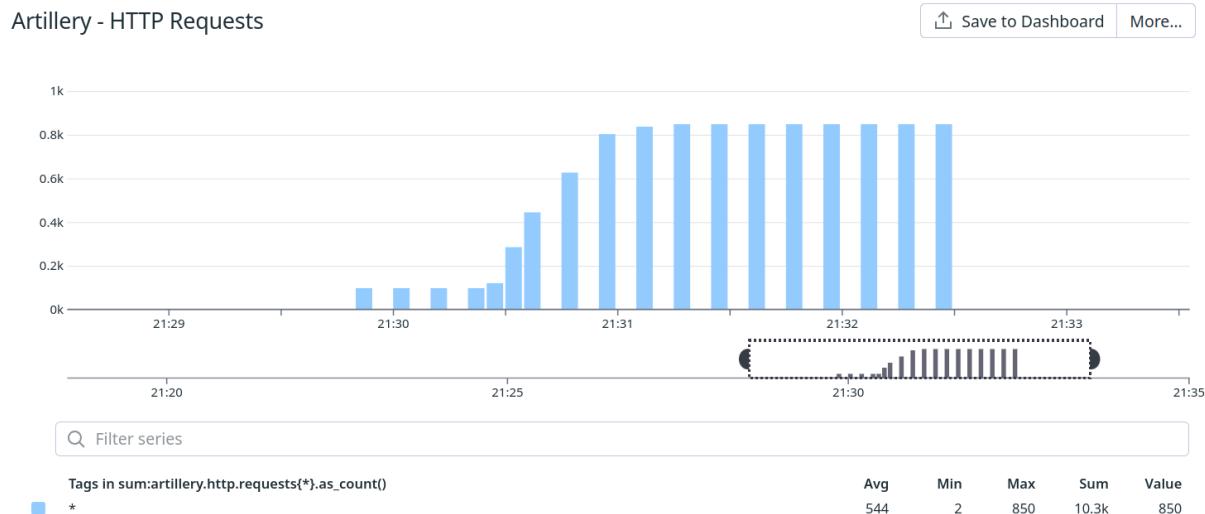


Gráfico 1 de Facts cache - Número de requests

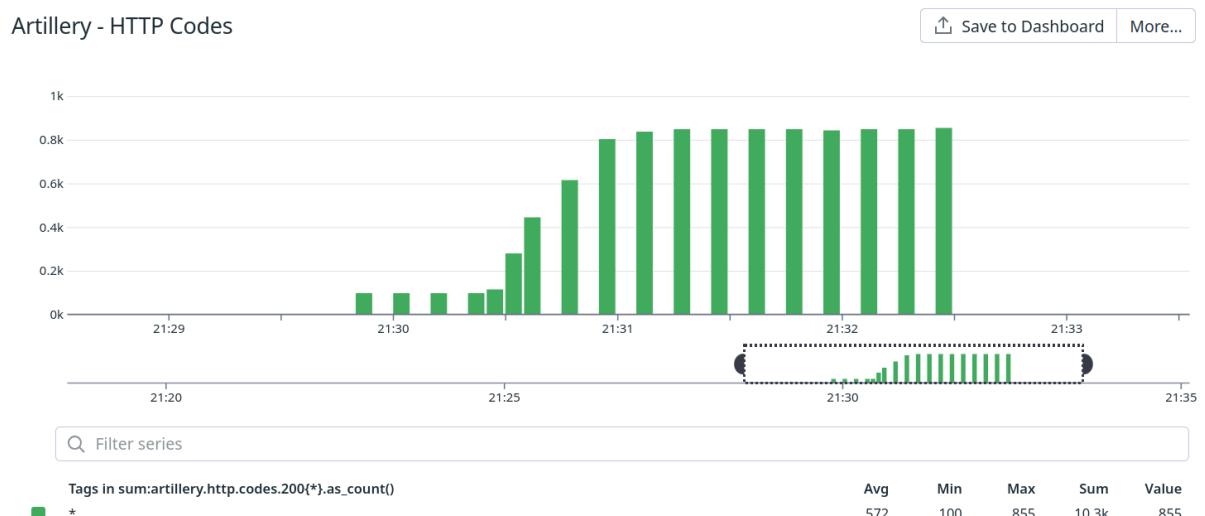


Gráfico 2 de Facts cache - Códigos de respuesta de requests

System CPU Usage

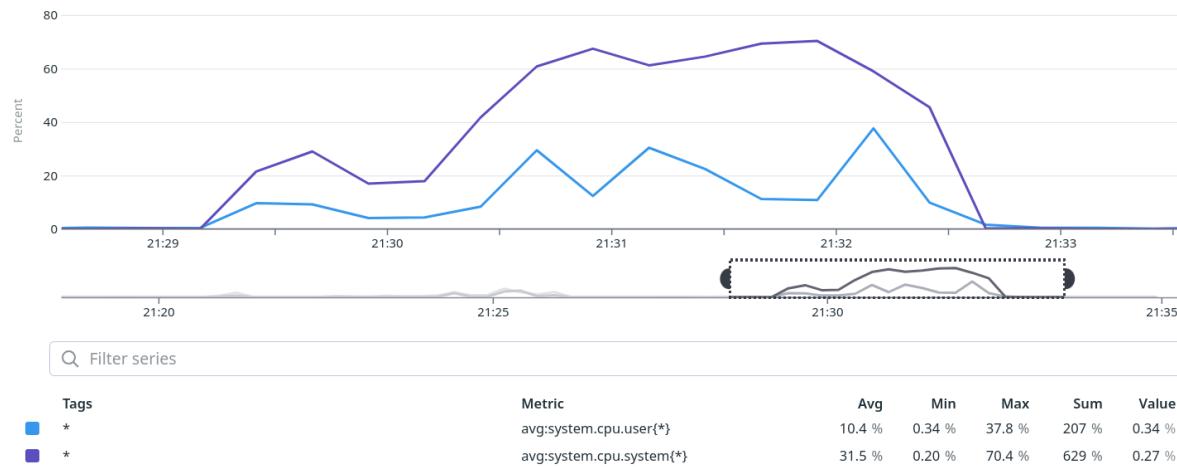
[Save to Dashboard](#) [More...](#)


Gráfico 3 de Facts cache - Uso de CPU

System Memory Usage

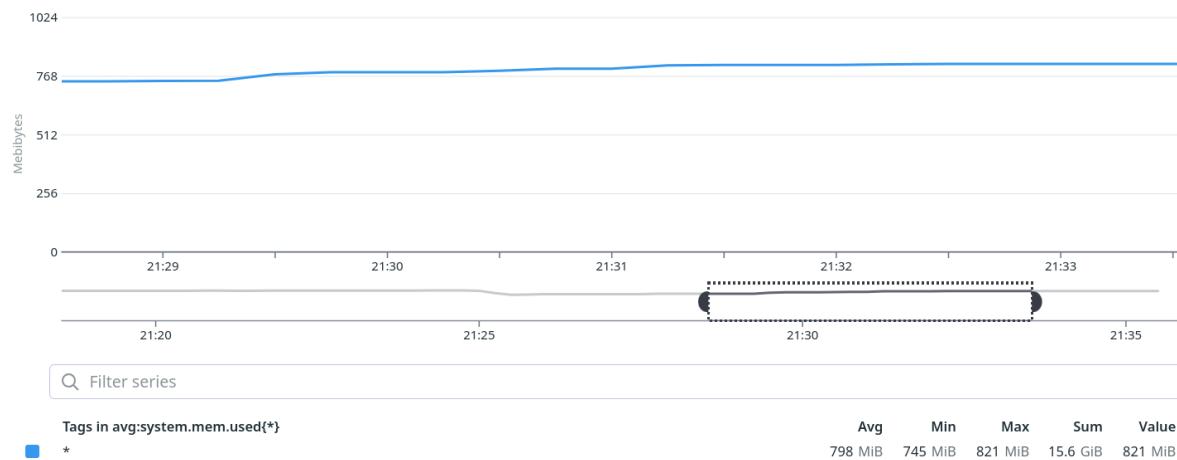
[Save to Dashboard](#) [More...](#)


Gráfico 4 de Facts cache - Uso de memoria

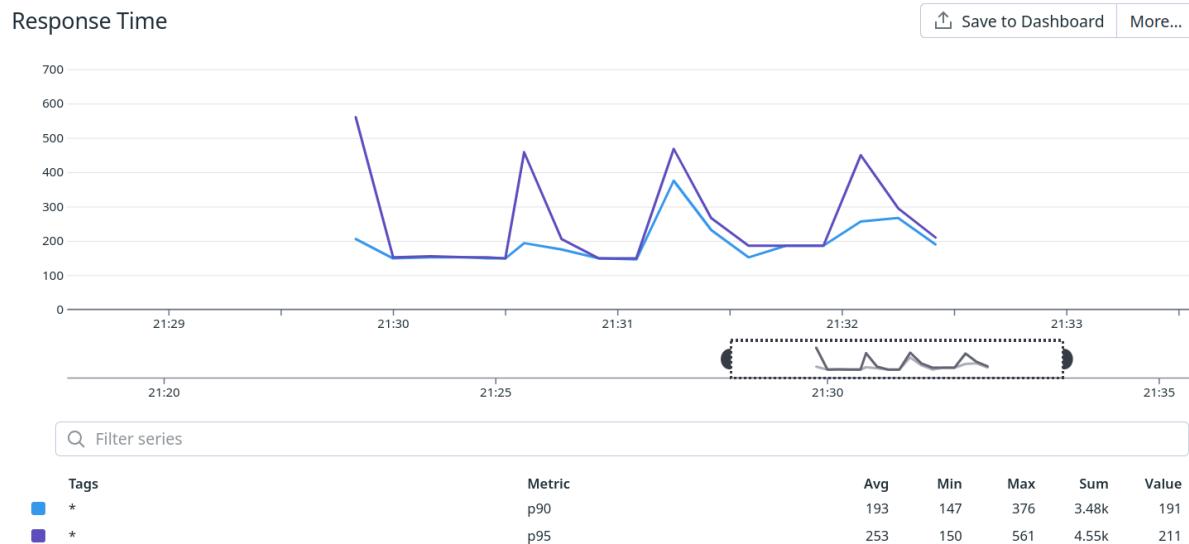


Gráfico 5 de Facts cache - Tiempo de respuesta

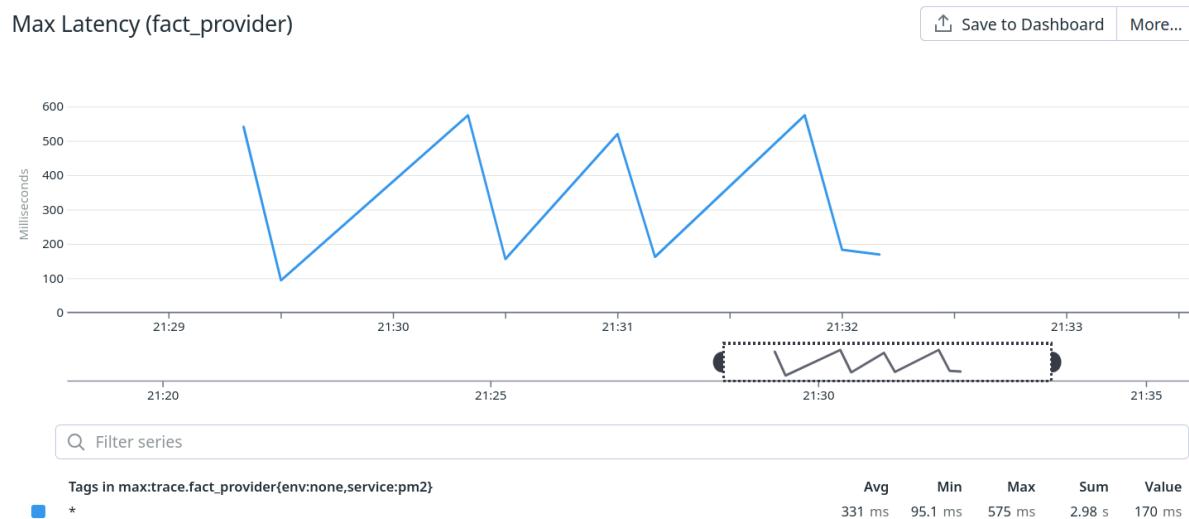


Gráfico 6 de Facts cache - Latencia (fact provider)

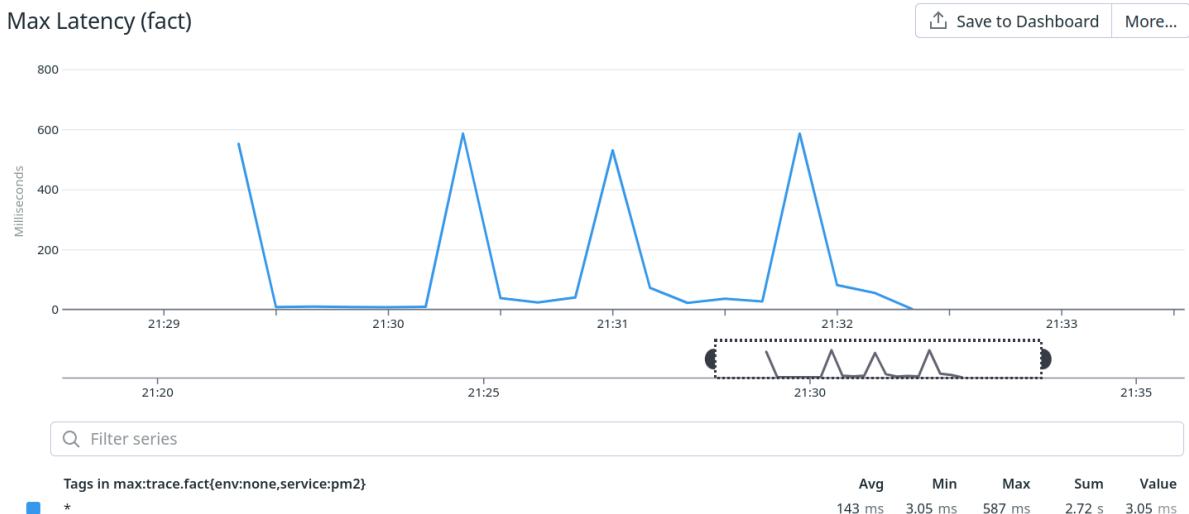


Gráfico 7 de Facts cache - Latencia (fact)

A diferencia del caso base, al correr este escenario con cache implementado ninguna request falla.

Sin embargo, el uso de CPU aumentó 12 puntos desde 58% a 70%. El uso de memoria, sin embargo, se mantuvo constante y similar al caso base.

En el response time y la latencia se pueden notar las mayores ventajas de cache. El tiempo de respuesta baja a 150 ms, un 40% menos que el caso base (aunque tiene picos de 450 ms). La latencia muestra picos similares al response time. Los gráficos de latencia de fact_provider y fact son distintos, en este último el mínimo es de 3 ms y el máximo de 600 ms.

Podemos notar que al agregar cache mejoramos la performance, puesto que para el p95 tenemos un response time alrededor de los ~253ms, en promedio, comparado a los ~273ms del caso base (es decir, un ~10% de mejora). De todas formas, debemos tener en cuenta que muchas requests están por encima y por debajo de los valores mencionados, ya que este es un promedio.

Es importante notar los picos de los gráficos 5, 6 y 7. Interpretamos que los tiempos aumentan cada cierto tiempo porque el servicio sale a buscar más respuestas para llenar el cache. Cuando hay respuestas guardadas en el cache, las requests se resuelven más rápido que en el caso base.

Useless Facts - Rélicas

Aquí replicaremos el servidor en tres nodos y Azure proveerá un load-balancer para distribuir la carga entre ellos.

Artillery - HTTP Requests

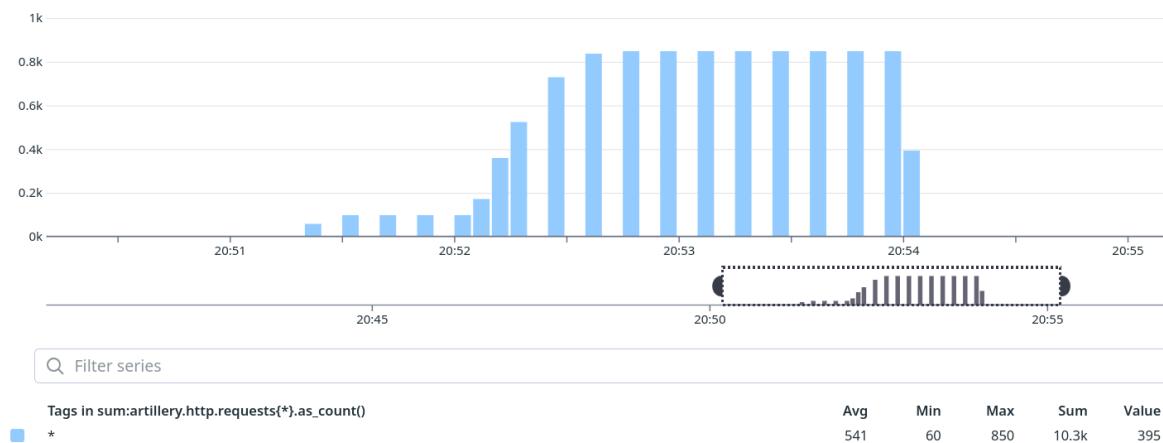
[Save to Dashboard](#) [More...](#)


Gráfico 1 de Facts réplicas - Número de requests

Artillery - HTTP Codes

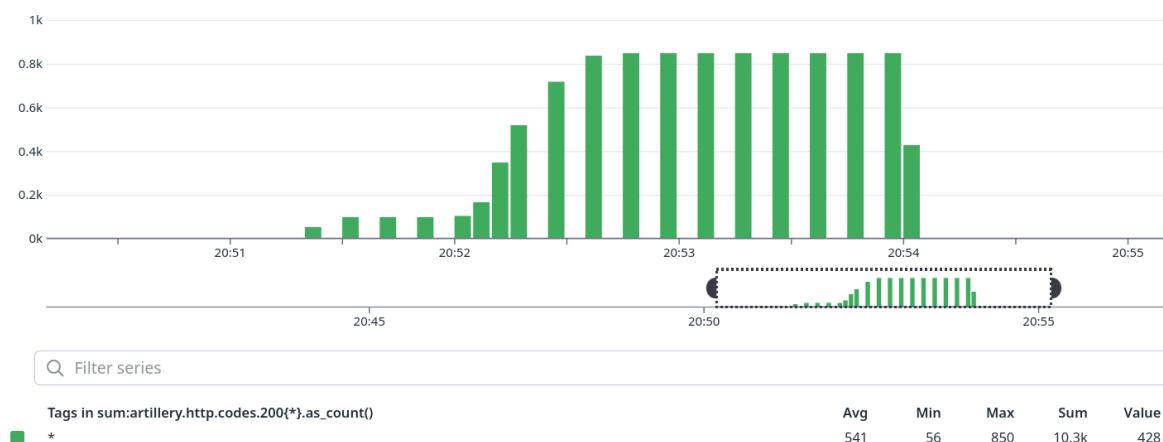
[Save to Dashboard](#) [More...](#)


Gráfico 2 de Facts réplicas - Códigos de respuesta de requests

Artillery - Error Type

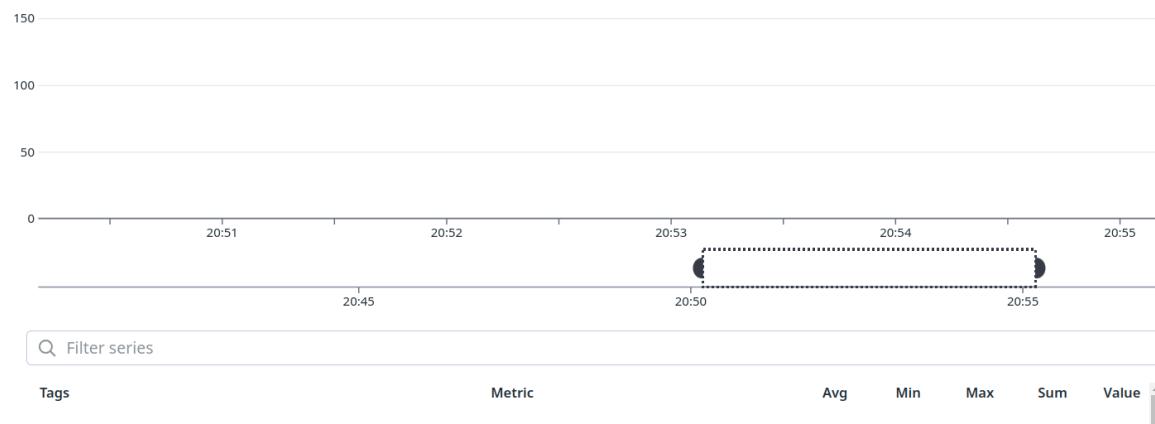
[Save to Dashboard](#) More...

Gráfico 3 de Facts réplicas - Tipos de error

System CPU Usage

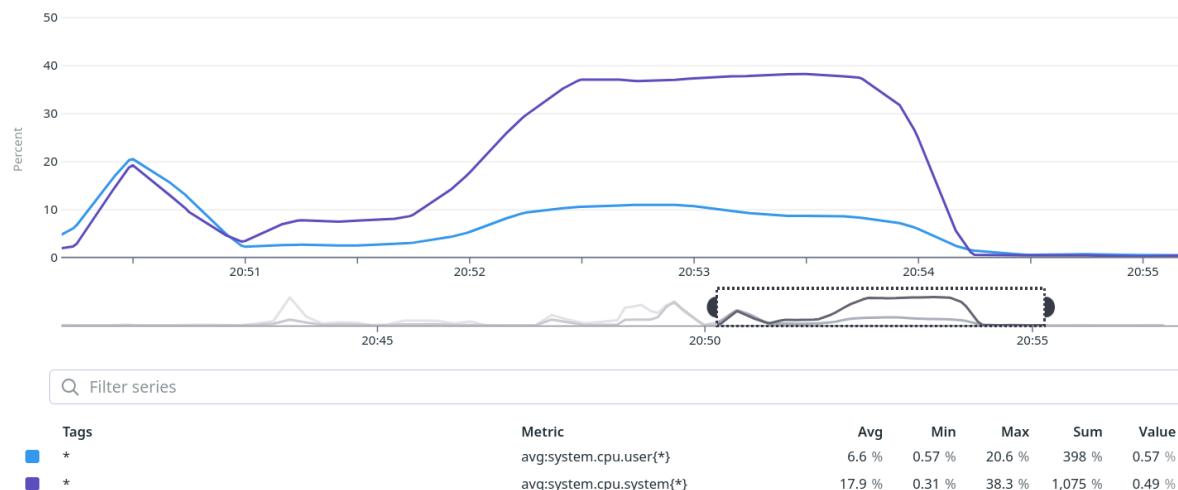
[Save to Dashboard](#) More...

Gráfico 4 de Facts réplicas - Uso de CPU

System Memory Usage

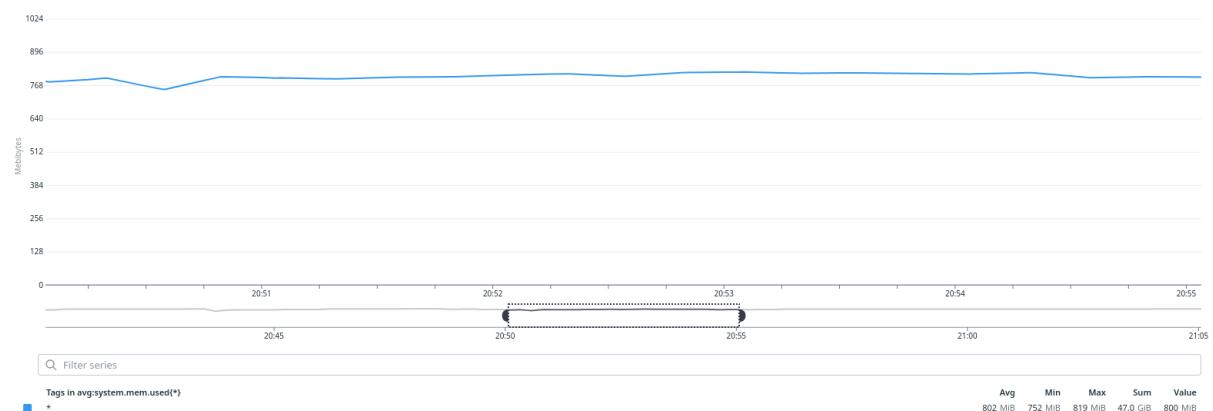
[Save to Dashboard](#) More...

Gráfico 5 de Facts réplicas - Uso de memoria

Response Time

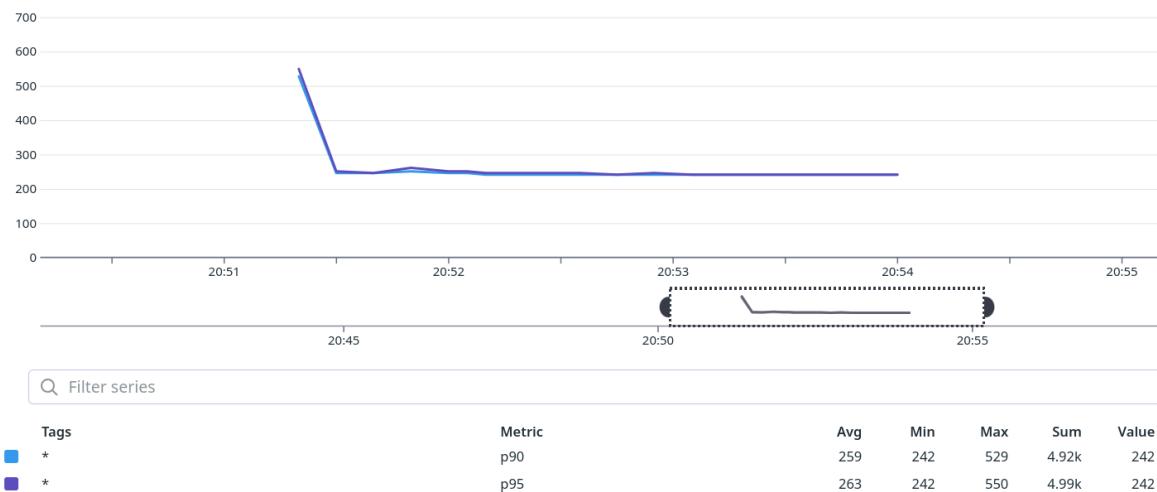
[Save to Dashboard](#) [More...](#)


Gráfico 6 de Facts réplicas - Tiempo de respuesta

Max Latency (fact_provider)

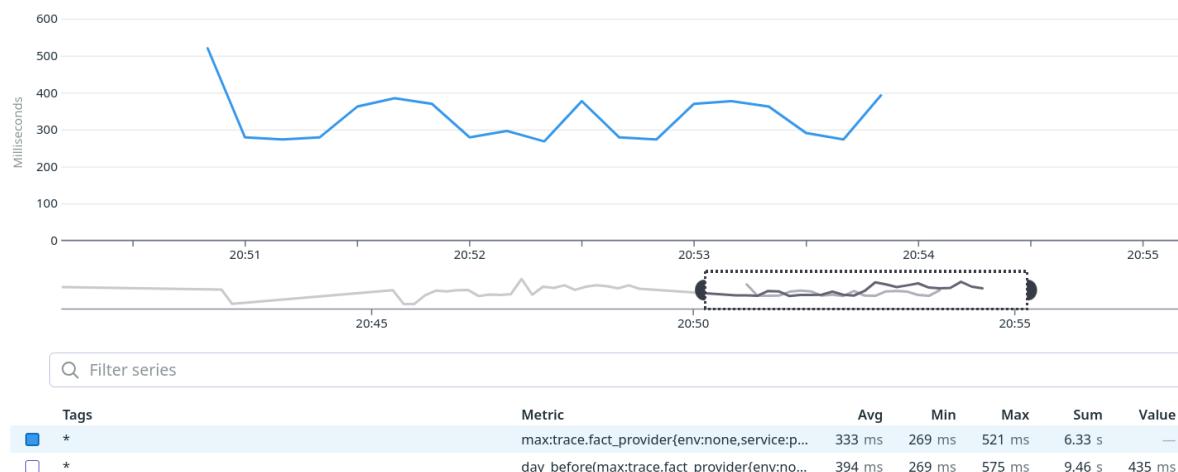
[Save to Dashboard](#) [More...](#)


Gráfico 7 de Facts réplicas - Latencia (fact provider)

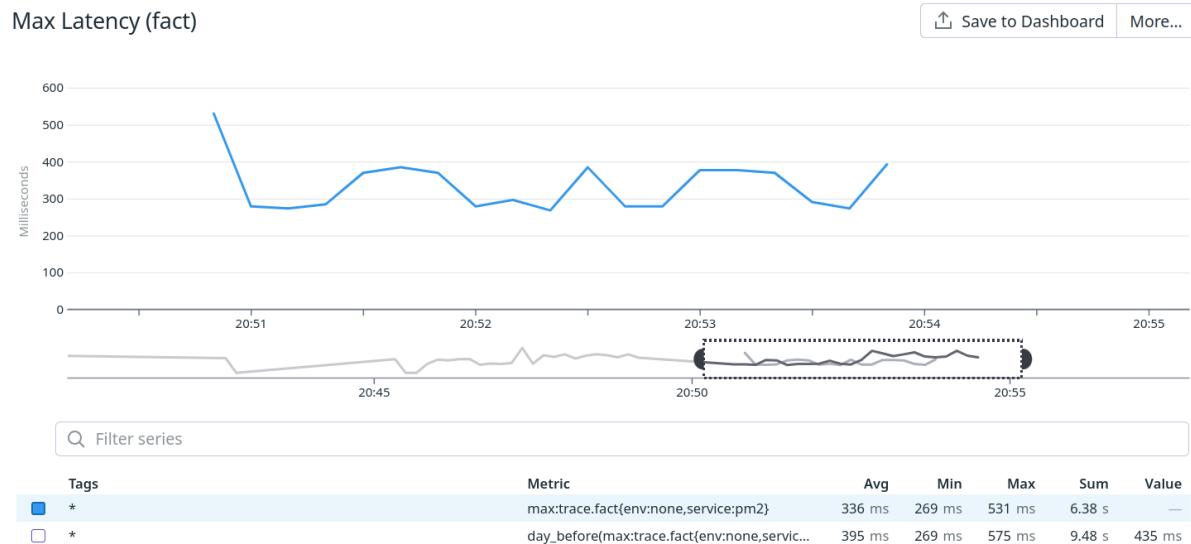


Gráfico 8 de Facts réplicas - Latencia (fact)

En este caso, las réplicas principalmente trajeron la ventaja de bajar el consumo de recursos (véase 38% de CPU).

El response time es similar al caso base, mientras que la latencia no sufrió del gran pico que tuvo el caso base.

Useless Facts - Rate limiting

Para evitar que una avalancha de clientes nos genere una falta de disponibilidad del servicio, podemos aplicar un rate limit, es decir, limitar la cantidad de requests que puede hacer un cliente en una determinada ventana de tiempo.

En nuestro caso, encontramos que el límite más conveniente se encuentra en el rango de 700 requests cada 10 segundos. De esta manera, minimizamos la cantidad de requests fallidas y, también, minimizamos la cantidad de requests rechazadas por pasar el rate limit.

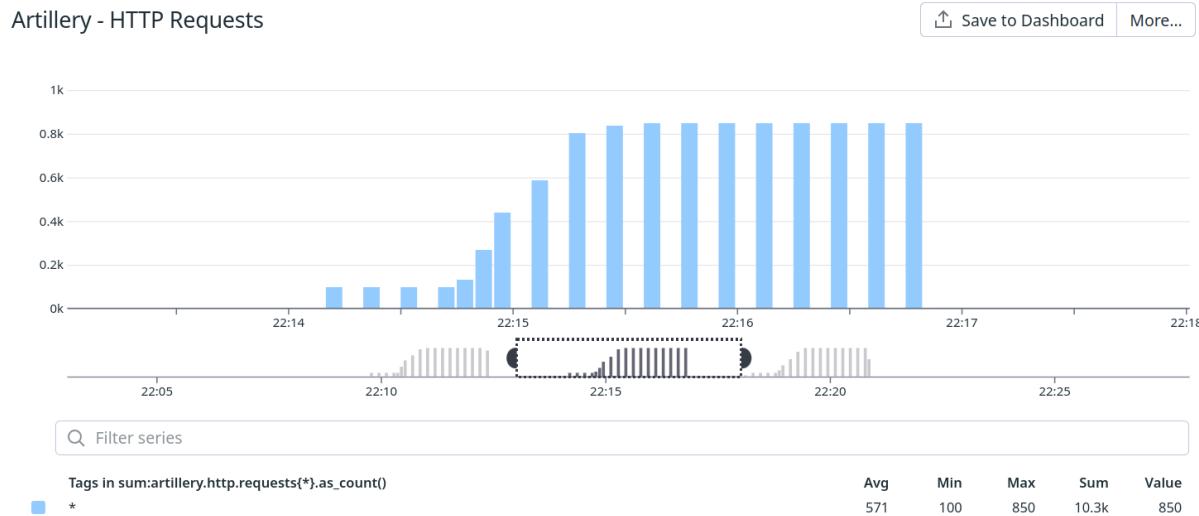


Gráfico 1 de Facts Rate limiting - Número de requests



Gráfico 2 de Facts Rate limiting - Códigos de respuesta de requests

Artillery - Error Type

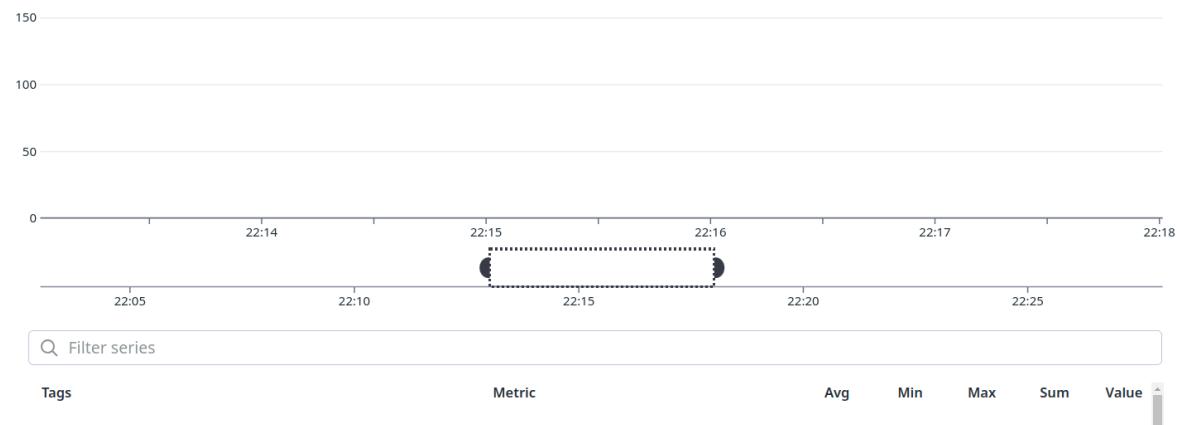
[Save to Dashboard](#) More...

Gráfico 3 de Facts Rate limiting - Tipos de error

System CPU Usage

[Save to Dashboard](#) More...

Gráfico 4 de Facts Rate limiting - Uso de CPU

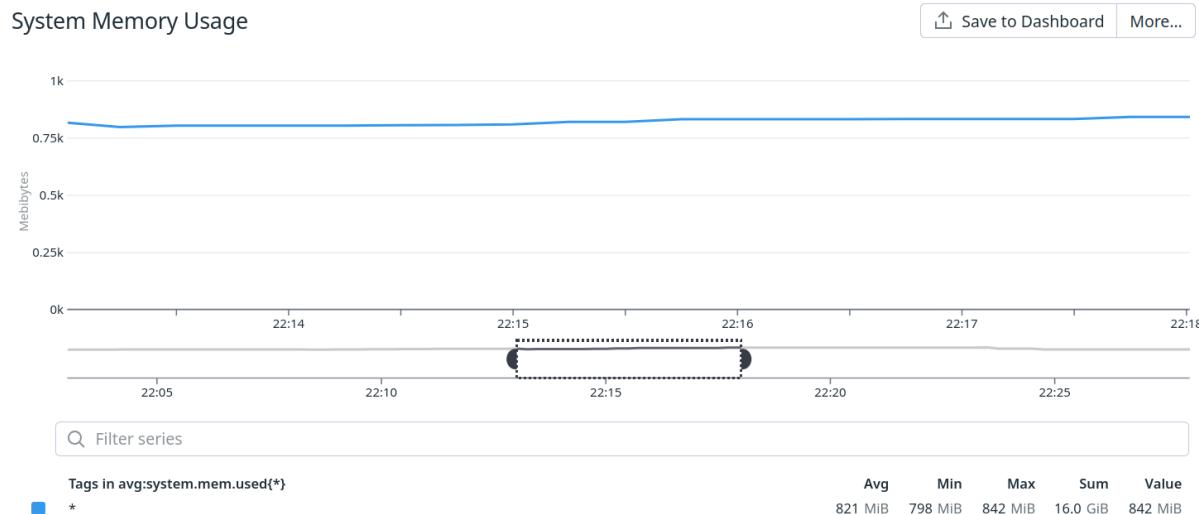
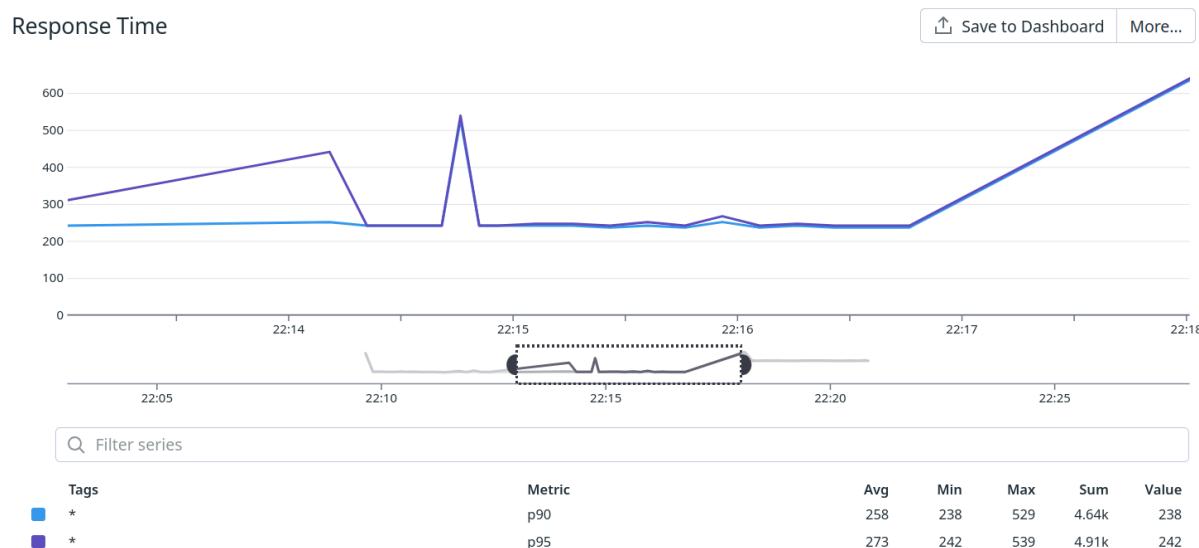


Gráfico 5 de Facts Rate limiting - Uso de memoria

Gráfico 6 de Facts Rate limiting - Tiempo de respuesta⁴

⁴ La última pendiente positiva es falsa. Es un error del gráfico, contaminado por una prueba posterior.

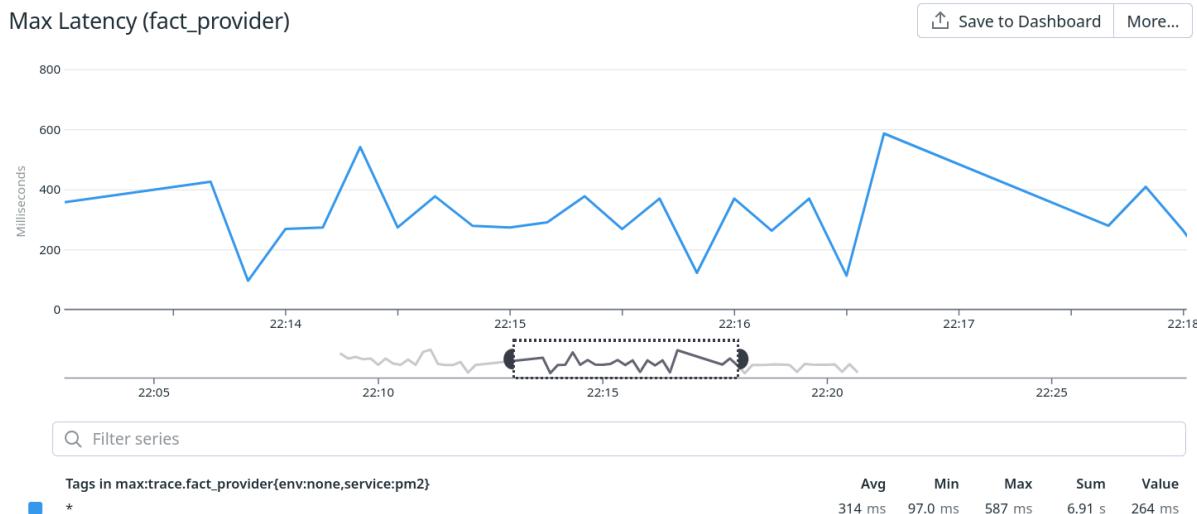


Gráfico 7 de Facts Rate limiting - Latencia (facts provider)

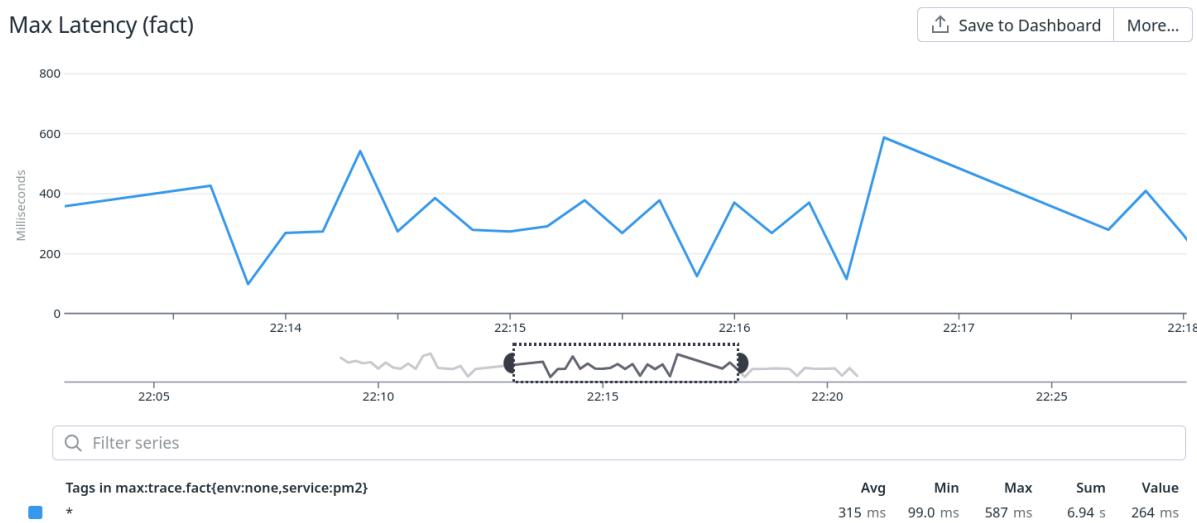


Gráfico 7 de Facts Rate limiting - Latencia (facts)

En este caso, el rate limiting ayudó a evitar los timeouts generados por el aumento de carga. Se puede notar que no hay timeouts pero, a modo de *tradeoff*, alrededor de un 10% de los requests fueron limitados y deberán intentar de nuevo más tarde.

Al igual que cache y réplicas, la táctica rate limiting solucionó el pico de latencia del caso base y mejoró la latencia a un promedio de 320 ms.

Useless Facts - Análisis

Conociendo las métricas de los cuatro casos, podemos sacar conclusiones para el endpoint de Useless Fact:

Sólo caché y réplicas lograron solucionar el problema de las requests fallidas. Por su naturaleza, el escenario de rate limiting tuvo que bloquear cerca de un 10% de las requests, perjudicando la **disponibilidad** del endpoint.

Además, todos los casos lograron solucionar el problema de latencia del caso base, donde al final del escenario saltaba drásticamente la latencia máxima. Esto es una gran mejora para el **performance**.

Más allá de la mejora de **disponibilidad**, el estilo réplicas no trajo otras ventajas. No presentó grandes mejoras de **performance**, a diferencia de cache. Este último estilo parecería ser el que mejor funcionó para Useless Facts, ya que no solo solucionó el problema de las requests fallidas (al igual que réplicas) sino que mejoró los tiempos de respuesta de las requests (mientras quedaban respuestas almacenadas).

El único atractivo único de réplicas fue su menor uso de CPU, lo que permitiría que **escale** a más requests en el futuro. Su latencia es similar a la alcanzada con rate limiting.

Metar

Enviamos un código OACI de un aeródromo para delegar la consulta a la NOAA Text Data Server API. Esta información es procesada por nuestro sistema para luego devolverla en formato JSON a nuestro cliente.

Nuevamente volvemos a crear un escenario que nos genere un Warm up, un Ramp up y un Plain para probar el servicio METAR. A continuación se observarán los resultados de probar los siguientes valores:

- **Warm up:** 10 requests por segundo durante 45 segundos.
- **Ramp up:** aumenta la cantidad de requests progresivamente hasta llegar a 30 requests.
- **Plain:** 30 requests por segundo durante 90 segundos.

Metar - Caso base

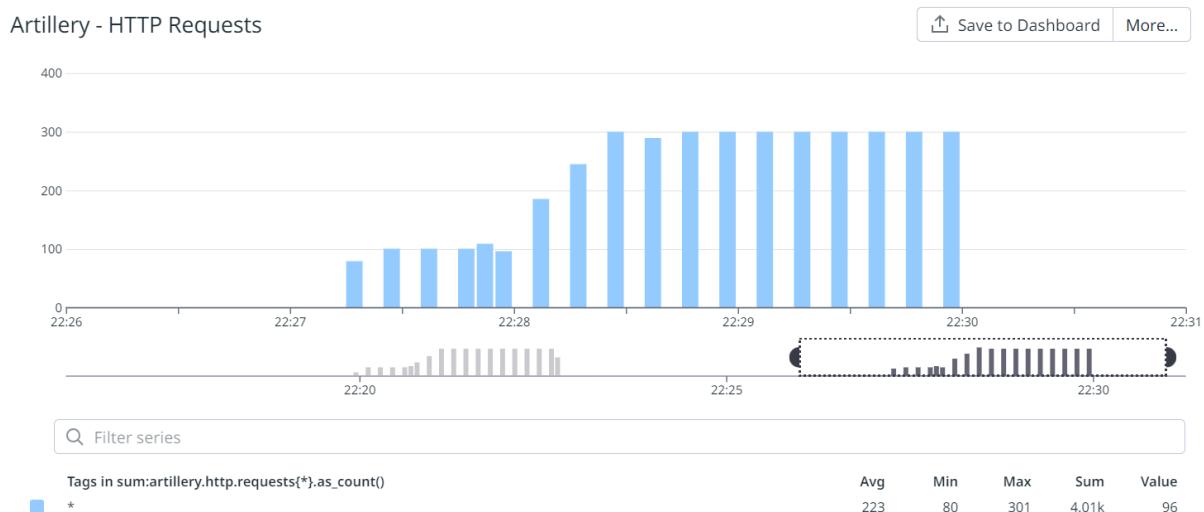


Gráfico 1 de Metar caso base - Número de requests



Gráfico 2 de Metar caso base - Códigos de respuesta de requests

Artillery - Error Type

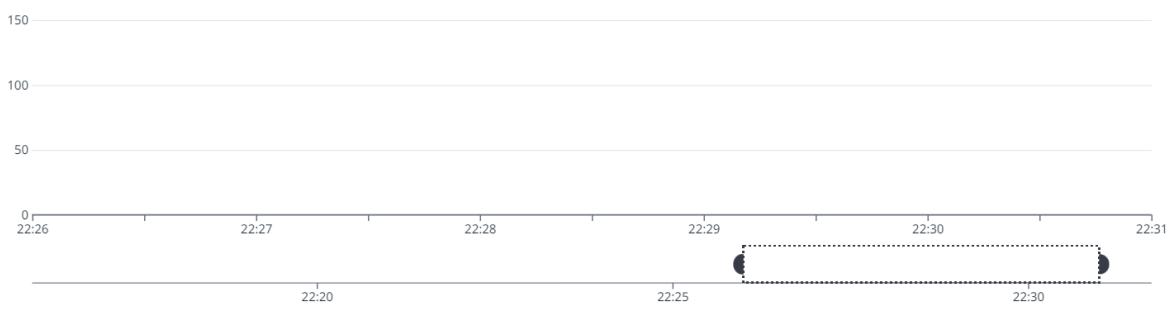
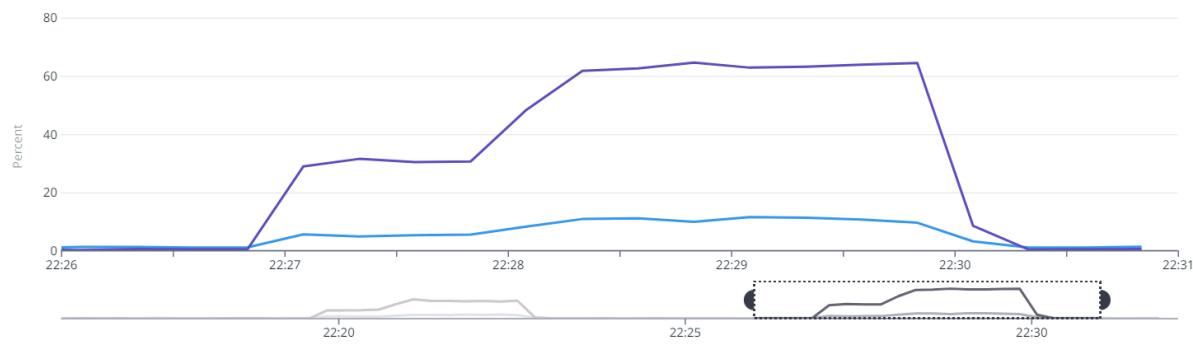
[Save to Dashboard](#) [More...](#)

Gráfico 3 de Metar caso base - Tipos de error

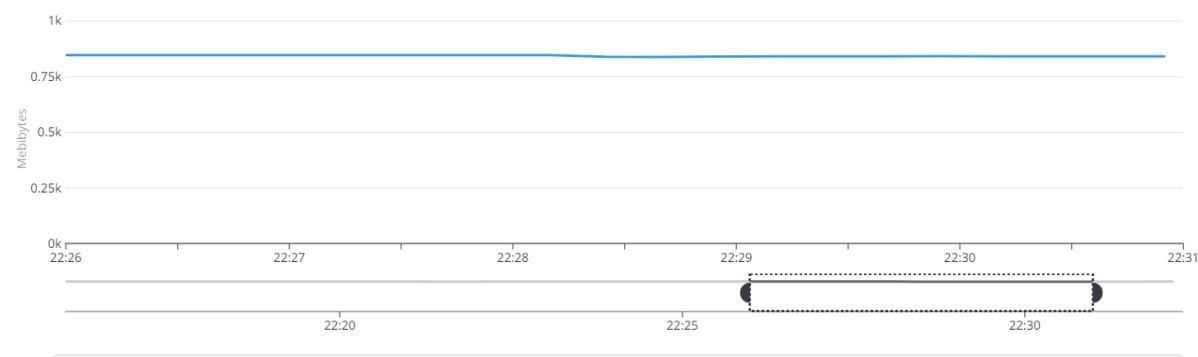
System CPU Usage

[Save to Dashboard](#) [More...](#)[Filter series](#)

Tags	Metric	Avg	Min	Max	Sum	Value
■ *	avg:system.cpu.user{*}	5.9 %	1.14 %	11.6 %	117 %	1.4 %
■ *	avg:system.cpu.system{*}	31.3 %	0.27 %	64.7 %	626 %	0.6 %

Gráfico 5 de Metar caso base - Uso de CPU

System Memory Usage

[Save to Dashboard](#) [More...](#)[Filter series](#)

Tags in avg:system.mem.used{*}	Avg	Min	Max	Sum	Value
■ *	842 MiB	837 MiB	846 MiB	16.5 GiB	840 MiB

Gráfico 6 de Metar caso base - Uso de memoria

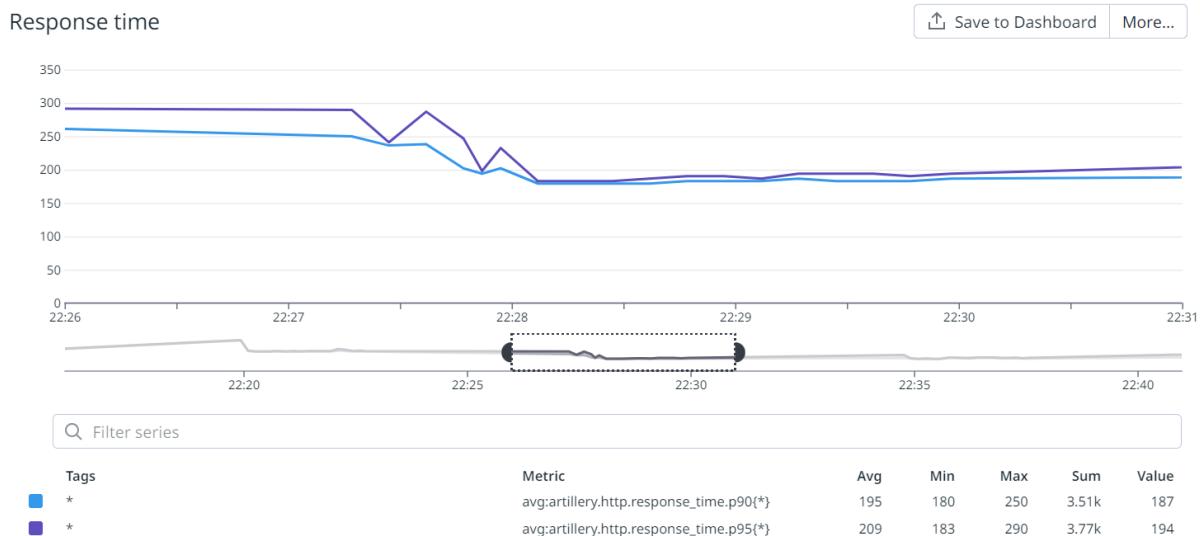


Gráfico 7 de Metar caso base - Tiempo de respuesta

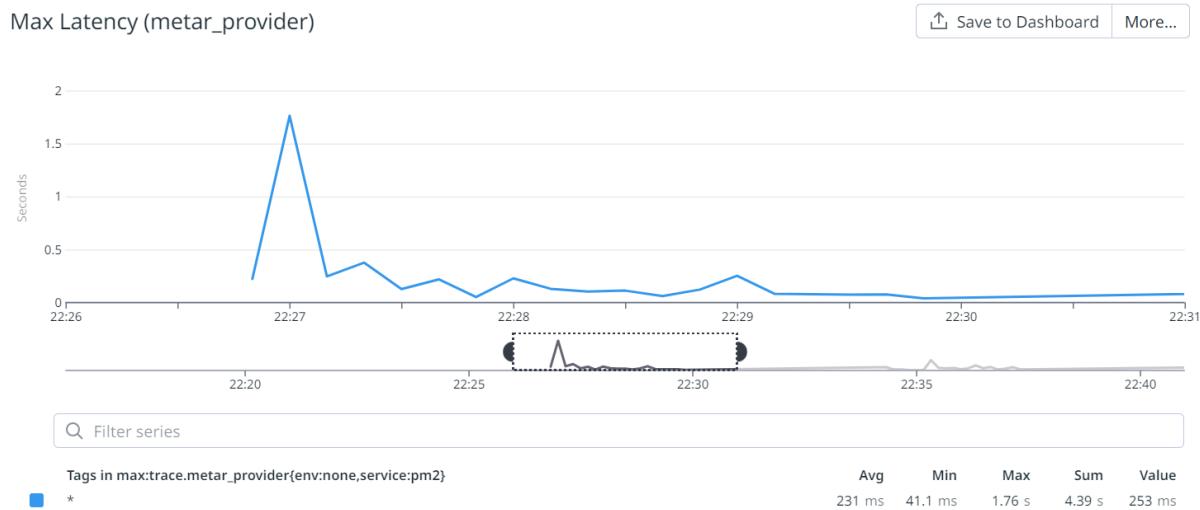


Gráfico 8 de Metar caso base - Latencia (metar provider)

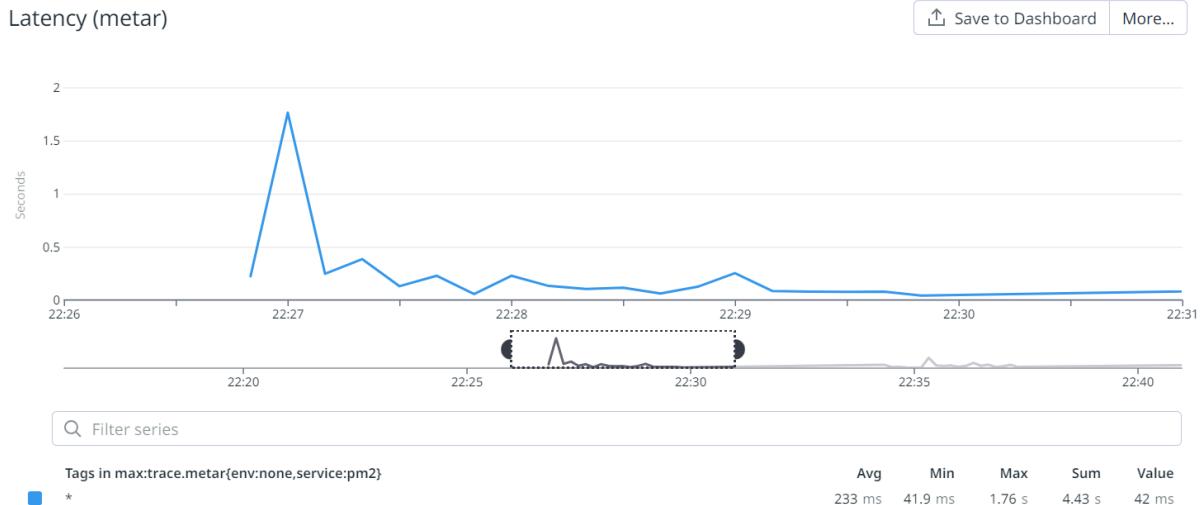


Gráfico 9 de Metar caso base - Latencia (metar)

Analicemos el caso base de Metar:

El escenario genera aproximadamente 4000 requests (gráfico 1), de las cuales 670 fallan (gráfico 2).

En cuanto al consumo de recursos (gráficos 5 y 6) el uso de CPU crece junto a la aparición de las requests, hasta llegar al 65%. El consumo vuelve a niveles de reposo una vez terminado el escenario. La memoria no varía en ningún momento, manteniéndose en 840mb.

El tiempo de respuesta de las requests empieza en 250ms, y llega a bajar hasta 180 ms. La latencia comenzó con un pico de 1,7 segundos, para luego bajar a un promedio de 230 ms. En las respuestas finales, llegó a los 50ms. La latencia fue la misma entre metar y metar provider, lo cual tiene sentido considerando que el endpoint llama a la API externa, realiza una leve transformación de los datos y retorna el resultado.

Metar - Cache

Para esta implementación de cache utilizamos lazy population, guardando la respuesta de la primera request para las siguientes.

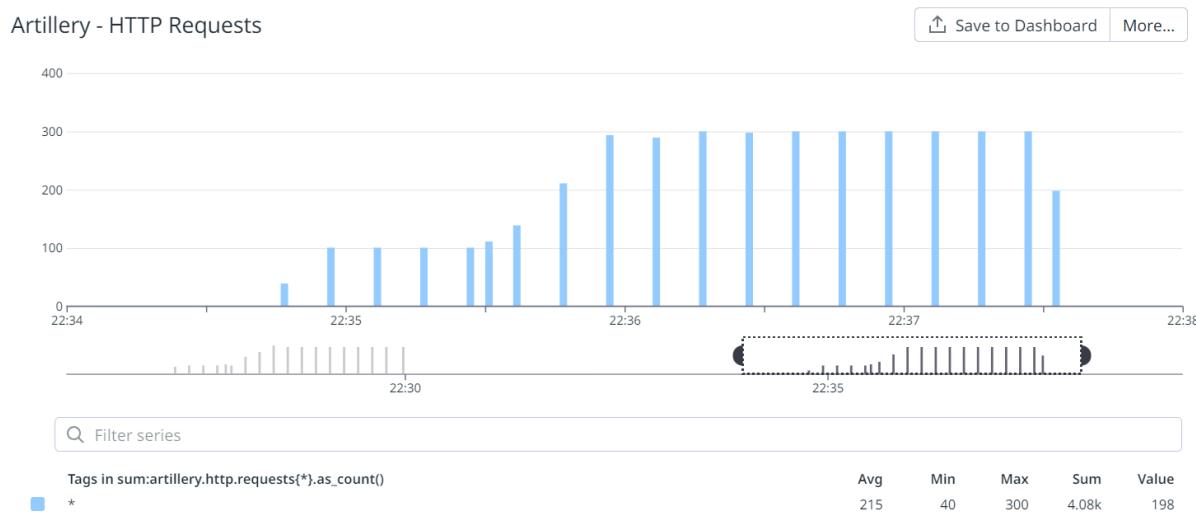


Gráfico 1 de Metar cache - Número de requests



Gráfico 2 de Metar cache - Códigos de respuesta de requests

Artillery - Error Type

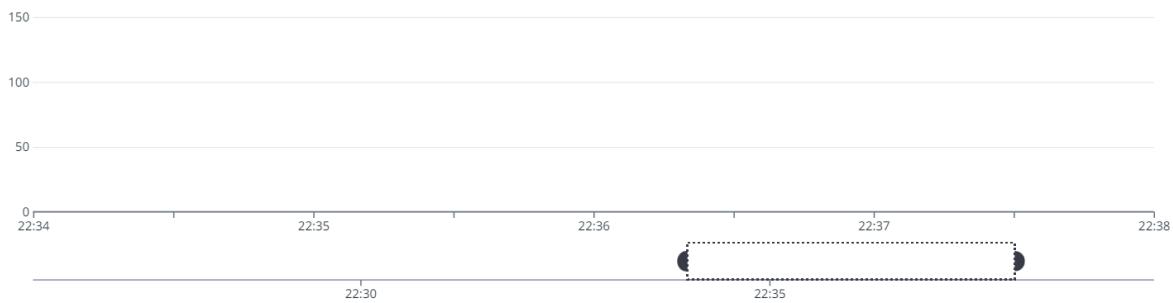
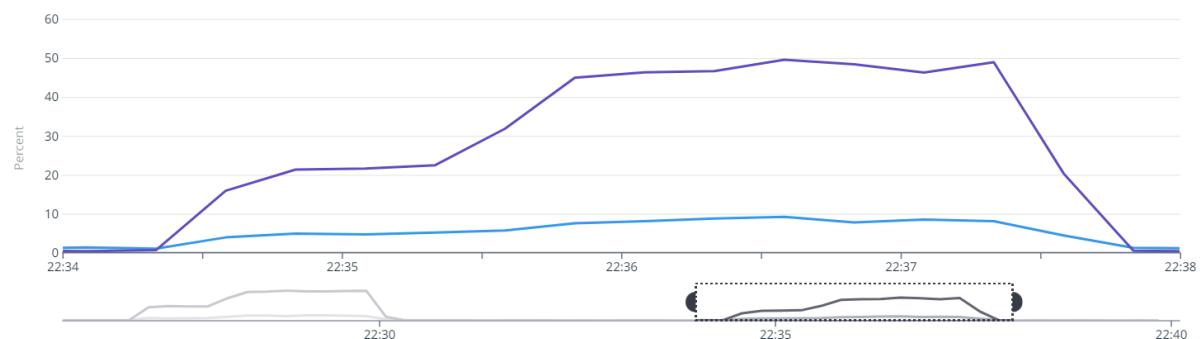
[Save to Dashboard](#) [More...](#)

Gráfico 3 de Metar cache - Tipos de error

System CPU Usage

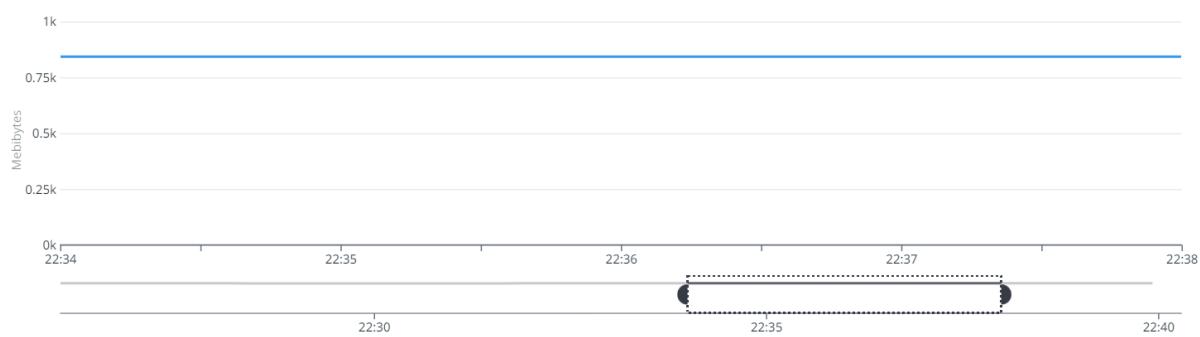
[Save to Dashboard](#) [More...](#)[Filter series](#)

Tags
█ *
█ *

Metric	Avg	Min	Max	Sum	Value
avg:system.cpu.user(*)	5.7 %	1.07 %	9.3 %	91 %	1.3 %
avg:system.cpu.system(*)	29.2 %	0.40 %	49.6 %	467 %	0.54 %

Gráfico 5 de Metar cache - Uso de CPU

System Memory Usage

[Save to Dashboard](#) [More...](#)[Filter series](#)

Tags in avg:system.mem.used(*)
█ *

Avg	Min	Max	Sum	Value
844 MiB	844 MiB	845 MiB	13.2 GiB	845 MiB

Gráfico 6 de Metar cache - Uso de memoria



Gráfico 7 de Metar cache - Tiempo de respuesta

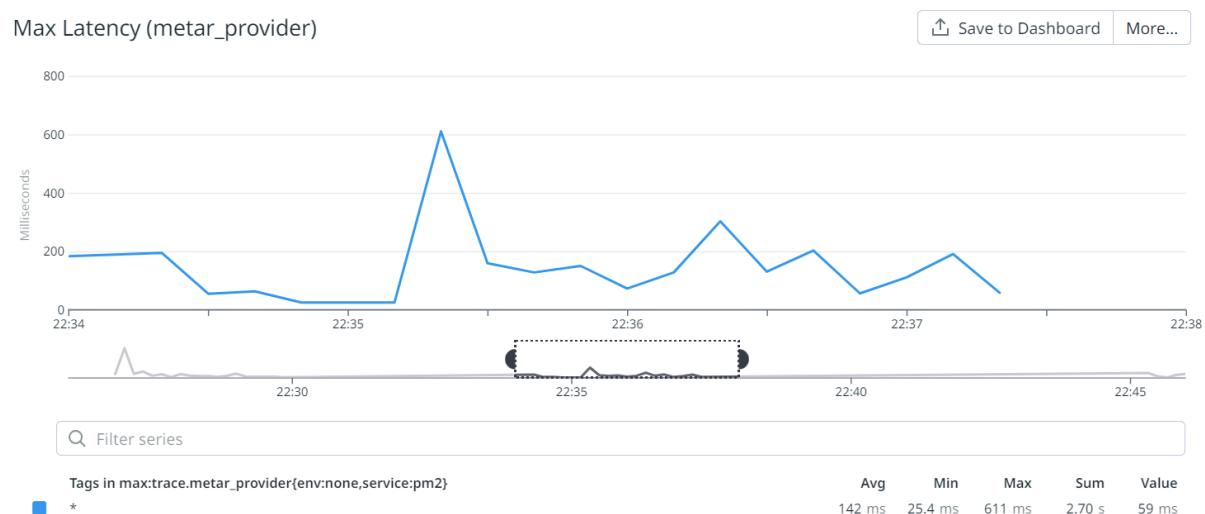


Gráfico 8 de Metar cache - Latencia (metar provider)

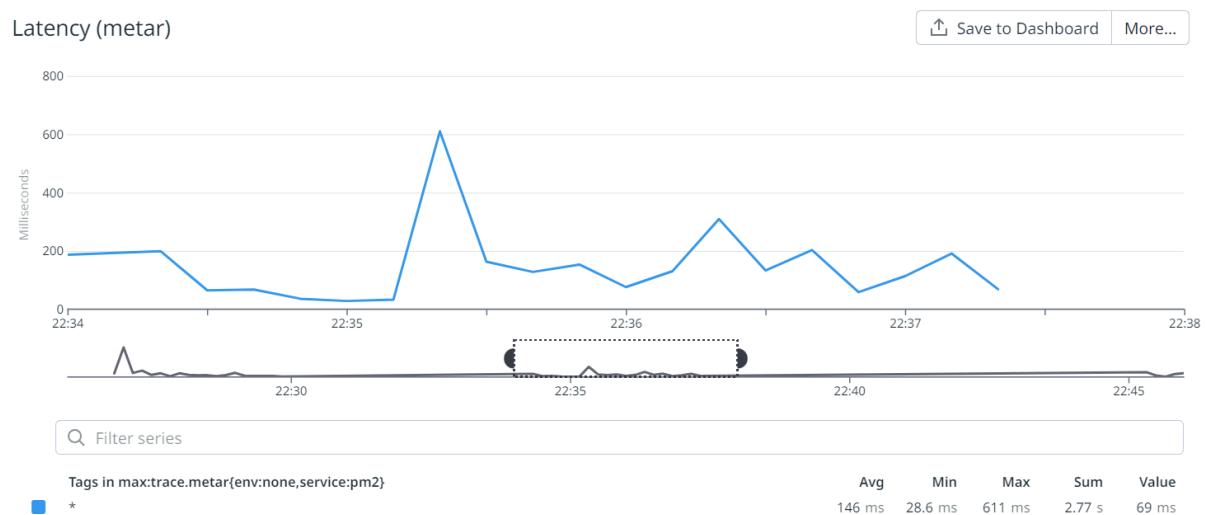


Gráfico 9 de Metar cache - Latencia (metar)

Comparemos ahora el mismo escenario de Artillery, pero con cache en funcionamiento: El número de requests es el mismo que en el caso base, 4000. Pero fallan 840 (**25% más que el caso base**).

Al igual que en el caso base, el consumo de CPU aumenta durante la carga, y baja cuando terminan las solicitudes. En este caso de cache, el uso llegó a 50% (**15 puntos menos que el caso base**). El uso de memoria parece ser aproximadamente **el mismo, 840 mb**.

Veamos ahora el gráfico 7. El tiempo de respuesta empieza en 200 ms y baja hasta 180 ms. Comparado con los 250 ms y 180 ms del caso base, son valores entre un **20% menores**.

Según los gráficos 8 y 9, la latencia del caso caché promedió los 140 ms (-**60%**), con un piso de 30 ms y un techo de 600 ms. Al igual que en el caso base, la latencia fue la misma entre metar y metar_provider.

Según estos datos, en el caso de metar podemos concluir que cache es una clara mejora para el **rendimiento**. Tanto el tiempo de respuesta como de latencia mejoraron claramente. La **escalabilidad** también fue beneficiada, pues el uso de CPU fue más eficiente y el uso de memoria no varió en absoluto (el cache tiene un impacto nulo en la memoria ya que este se almacena en Redis).

Sin embargo, parece impactar negativamente la **disponibilidad** pues el número de errores fue un 25% mayor que el caso base.

Metar - Réplicas

Se vuelve a replicar el servidor en tres nodos y a utilizar un load balancer para distribuir las cargas.

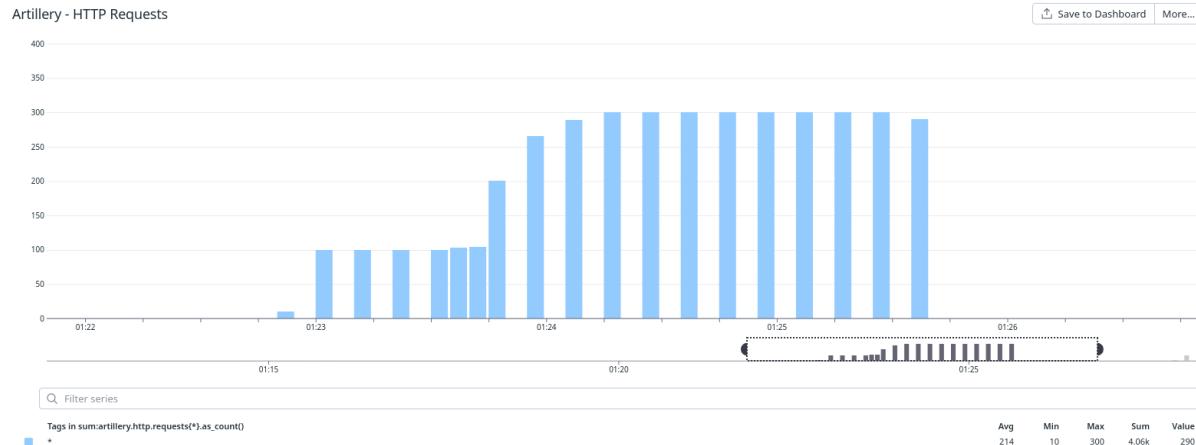


Gráfico 1 de Metar réplicas - Número de requests



Gráfico 2 de Metar réplicas - Códigos de respuesta de requests

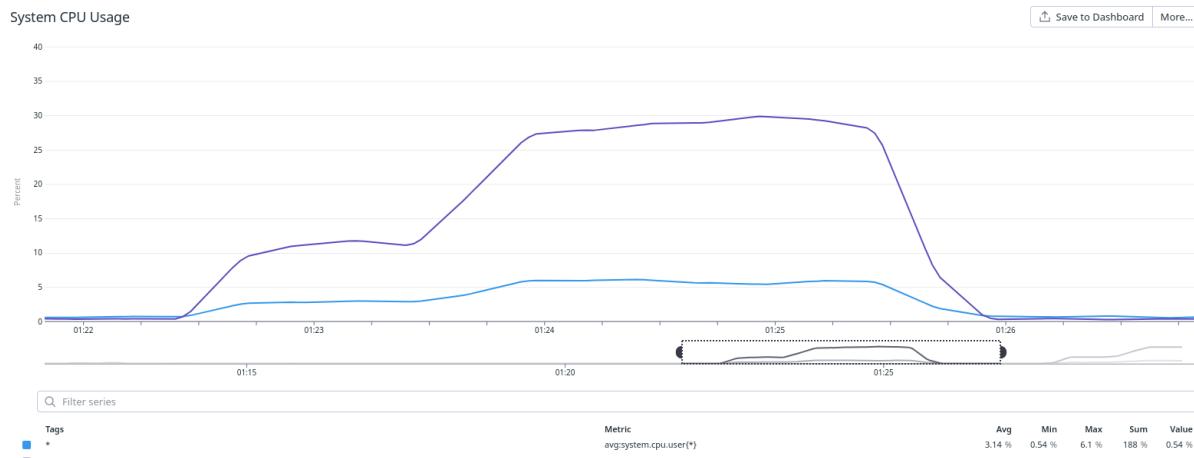


Gráfico 5 de Metar réplicas - Uso de CPU

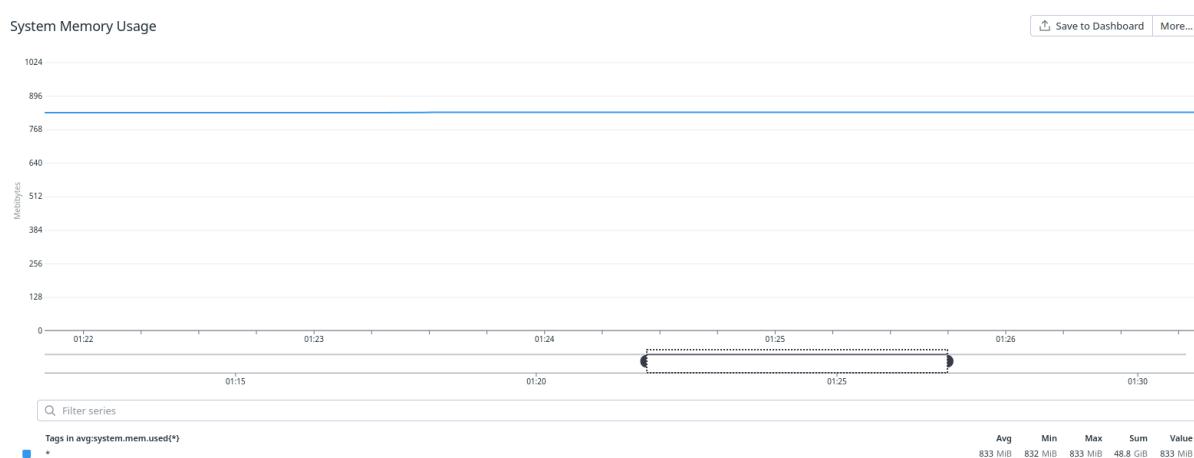


Gráfico 6 de Metar réplicas - Uso de memoria

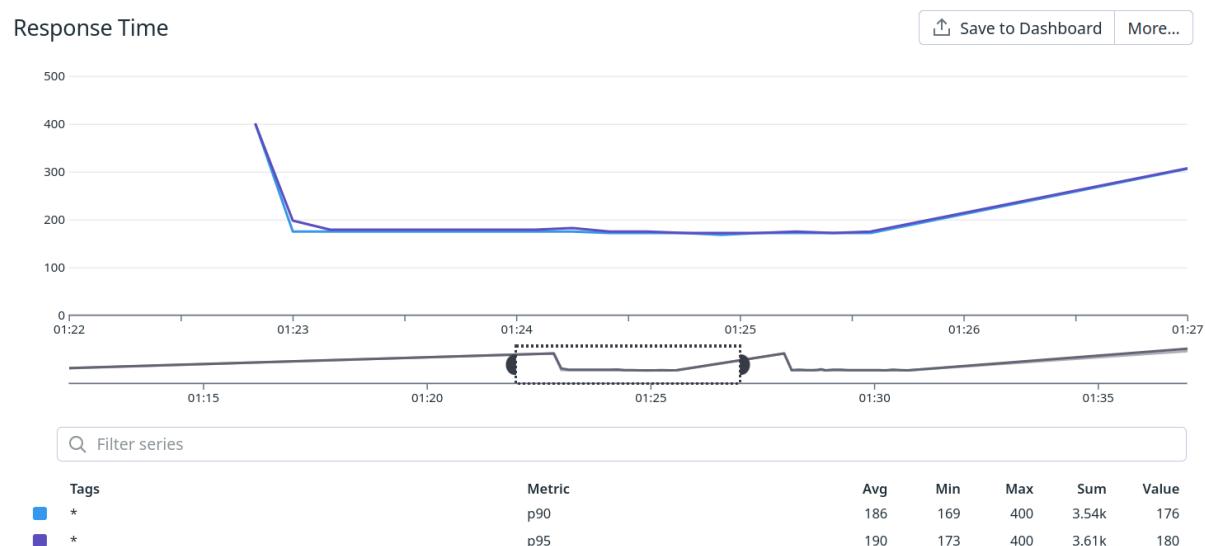


Gráfico 7 de Metar réplicas - Tiempo de respuesta

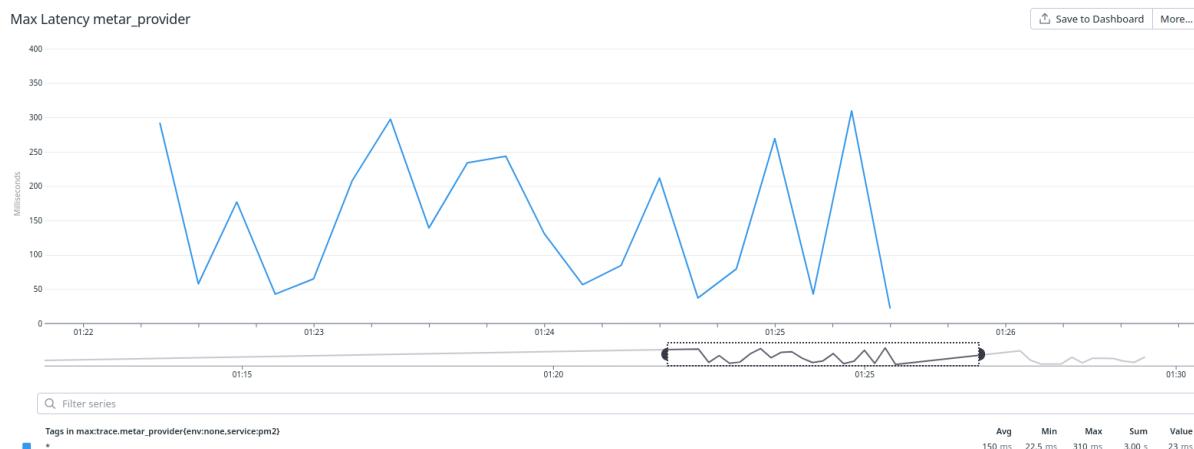


Gráfico 8 de Metar réplicas - Latencia (metar provider)

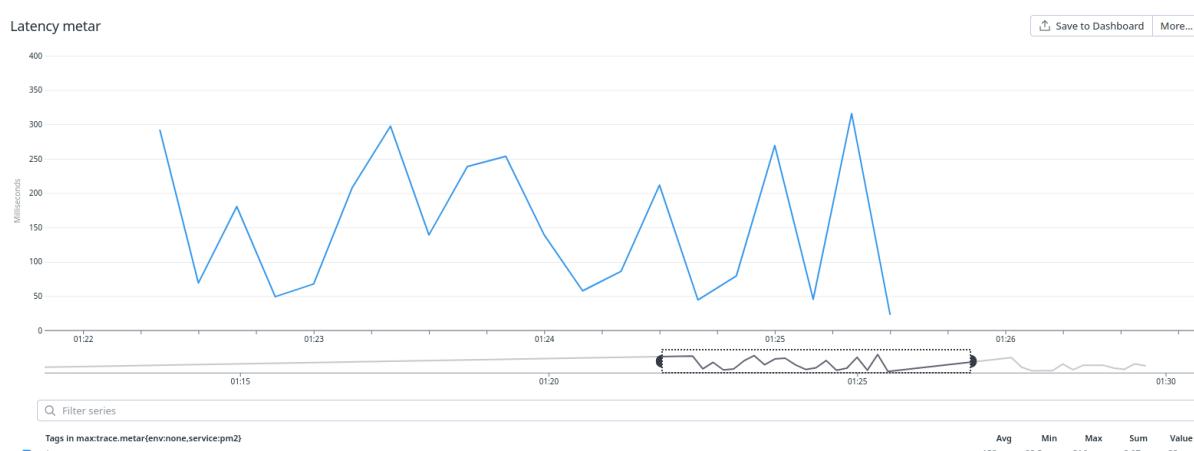


Gráfico 9 de Metar réplicas - Latencia (metar)

Para este escenario implementamos replicación con 3 VMs conectada al Load Balancer provisto por Azure que regulaba las solicitudes mediante round-robin.

También con 4000 requests, en este caso fallaron 330 (la **mitad** que el caso base).

Al igual que en los casos anteriores, el consumo de CPU crece y decrece con la demanda del escenario. En este caso, llega hasta el 30%. Esto es **menos de la mitad** del caso base (y también menor que el caso cache). Al igual que en todas las situaciones anteriores, el consumo de memoria se mantiene **constante en 840 mb**.

Salvando un pico de 400 ms al comienzo, durante todo el escenario el tiempo de respuesta se mantuvo en 170 ms (**igual que el piso del caso base**).

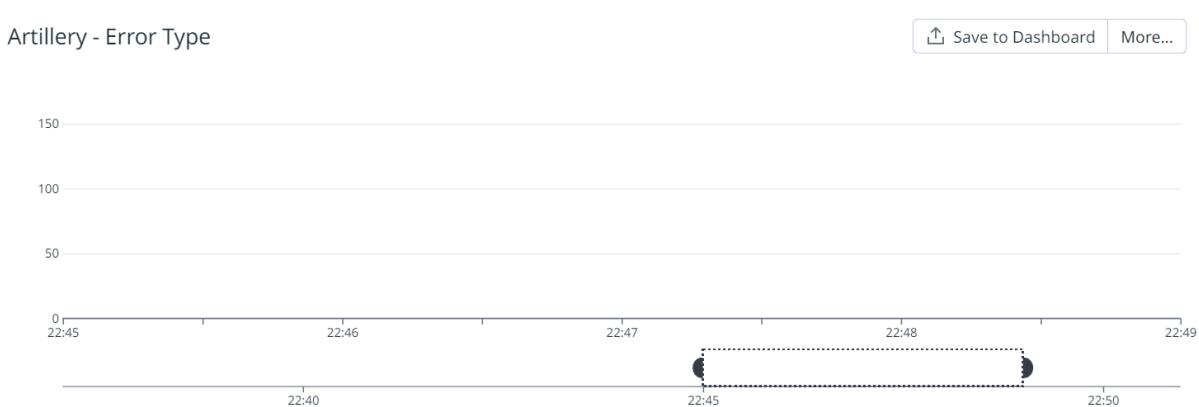
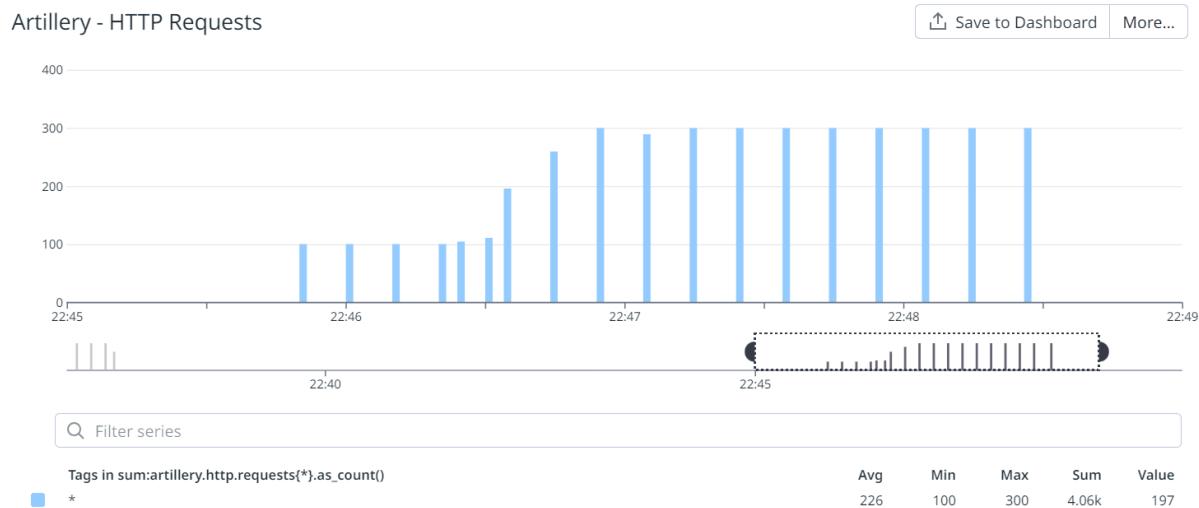
En cuanto a latencia, esta estuvo compuesta exclusivamente de picos y valles⁵, desde 25 ms hasta 310 ms. Este mínimo fue el **50% del caso base**, y el **máximo 5 veces mejor**⁶

⁵ No comprendemos por qué la latencia sube y baja tanto a lo largo del escenario (gráfico 8 y 9).

⁶ Si no se tiene en cuenta el único pico del caso base, la replicación tuvo un pico de latencia 50% peor que el caso base.

La baja del número de fallos y el consumo de CPU son notables. También son una gran mejora el tiempo de respuesta y la latencia. El estilo replication parecería no tener ninguna desventaja, salvando el aumento de complejidad que implica tener tres instancias del nodo.

Metar - Rate limiting



System CPU Usage

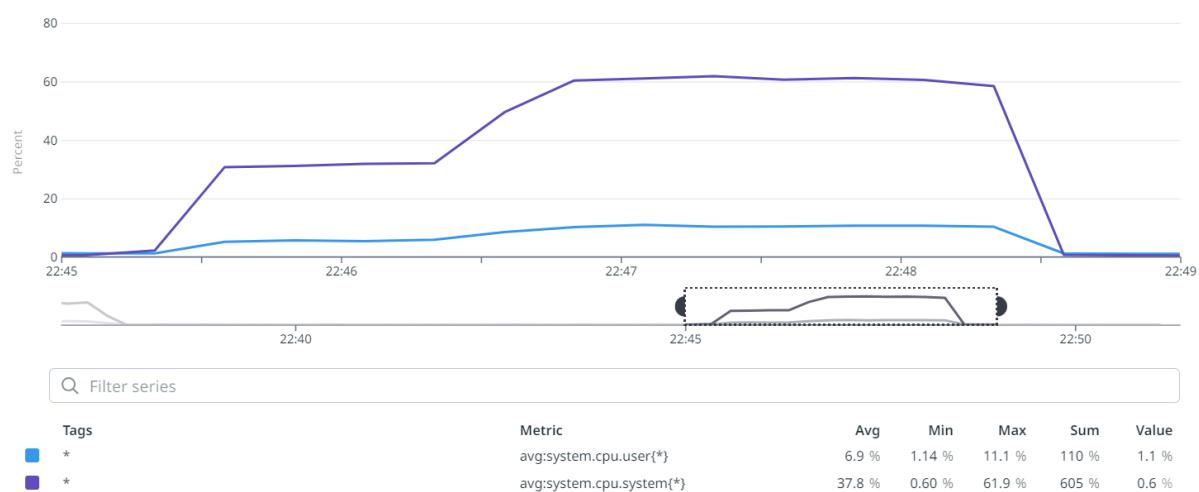


Gráfico 5 de Metar rate limiting - Uso de CPU

System Memory Usage

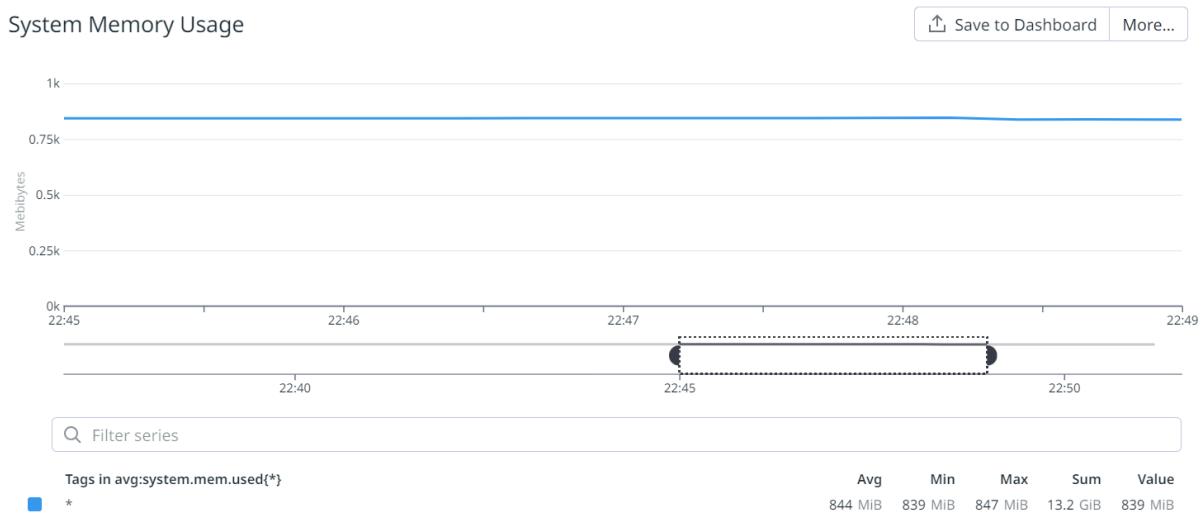


Gráfico 6 de Metar rate limiting - Uso de memoria



Gráfico 7 de Metar rate limiting - Tiempo de respuesta

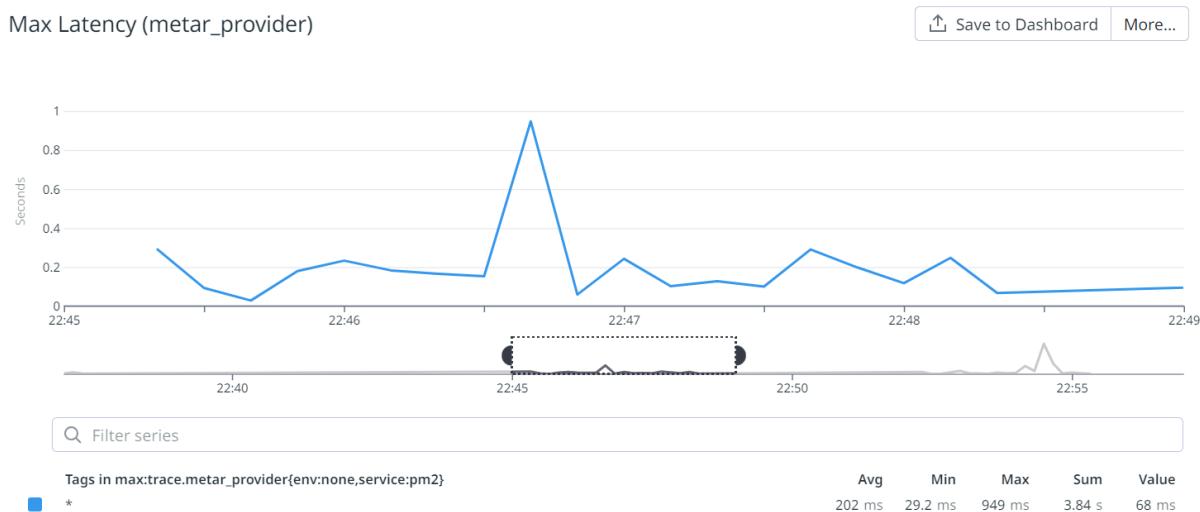


Gráfico 8 de Metar rate limiting - Latencia (metar provider)

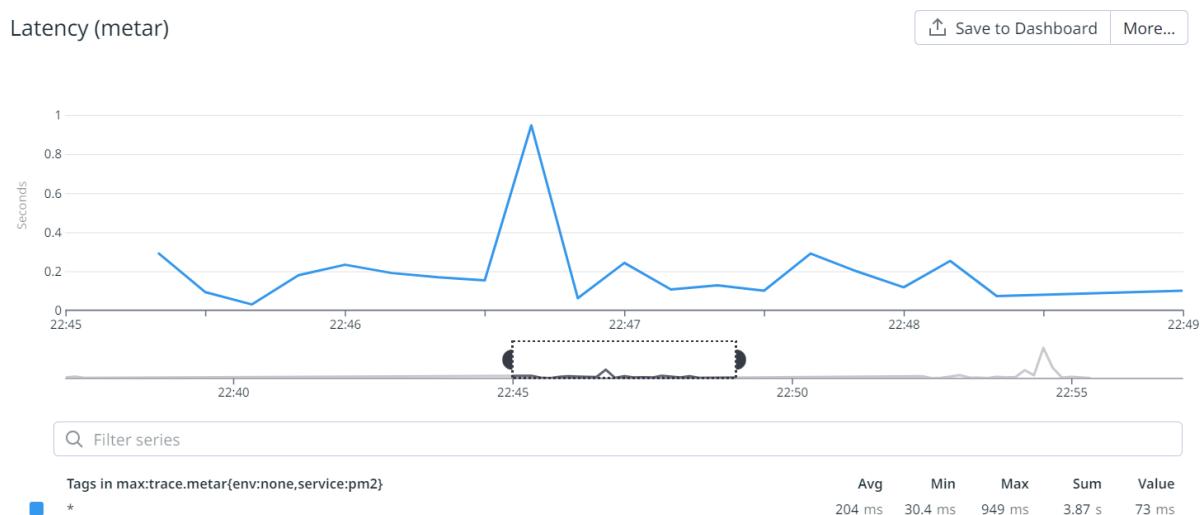


Gráfico 9 de Metar rate limiting - Latencia (metar)

Analicemos los datos tomados del mismo escenario de los últimos 3 casos, pero aplicando la táctica rate limiting, permitiendo a cada cliente realizar 50 requests por cada 10 segundos transcurridos.

De las 4000 requests, 2400 fallan. Esto es **4 veces peor** que el caso base.

El uso del CPU fue del 62% (**casi igual al caso base**), y el de memoria se mantuvo **constante en 840 mb** (igual que todos los casos anteriores).

El tiempo de respuesta es 180 ms, **similar al caso base**. La latencia promedió los 200 ms y solamente tuvo un pico de 900 ms. Estos valores también son **similares al caso base**.

Al igual que en todos los casos anteriores, metar y metar provider tienen la misma latencia.

El impacto de rate limiting en este caso parecería desastroso. El uso de CPU, tiempo de respuesta, duraciones y latencias fueron similares al caso base, pero la **disponibilidad** de la aplicación empeoró drásticamente (**el 60% de las requests fallaron**).

Metar - Análisis

Habiendo visto el impacto de las decisiones tomadas en el mismo escenario de METAR, veamos qué conclusiones podemos sacar:

Parecería que los mejores estilos a elegir en este caso, son **replication** y **cache**. Ambos mejoraron el tiempo de respuesta y latencia (favoreciendo el **rendimiento**) y disminuyeron el consumo de recursos (mejor **escalabilidad**). Es particularmente notable el bajo número de requests fallidas en replication.

La táctica rate limiting trajo muy pocas ventajas a cambio de sumar complejidad y perjudicar la **disponibilidad**. Todos sus atributos fueron iguales o peores que el caso base, con la gran desventaja de dejar sin servicio a la mayoría de los clientes del escenario.

Es interesante ver además que nuestra aplicación, cuando se usa el endpoint */metar*, en ningún caso varía su consumo de memoria. Podría concluirse que nuestra aplicación casi no usa memoria (lo cual tiene sentido teniendo en cuenta nuestra implementación. Ninguna parte de la aplicación requiere un gran uso de la memoria).

Space news

En esta parte de la API se obtienen las últimas 5 noticias sobre actividad espacial obtenidas desde la Spaceflight News API y se devuelven sus títulos.

El escenario de carga fue el mismo que Metar: **warm-up** de 45 segundos con 10 requests/s, **ramp-up** de 45 segundos hasta 30 rqts/s y **plain** de 30 requests durante 90 segundos.

Space news - Caso base

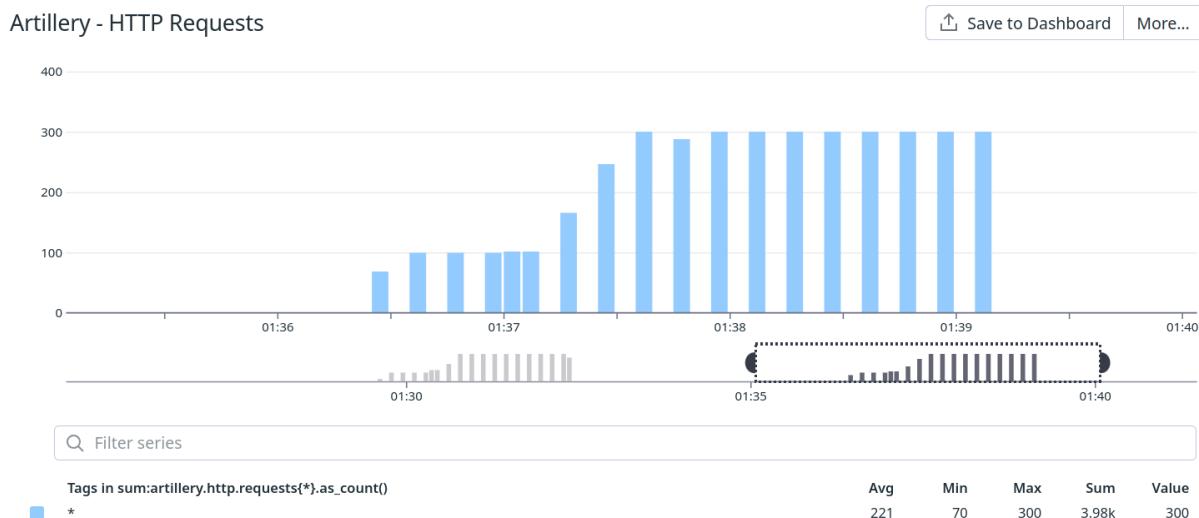


Gráfico 1 de Space news caso base - Número de requests

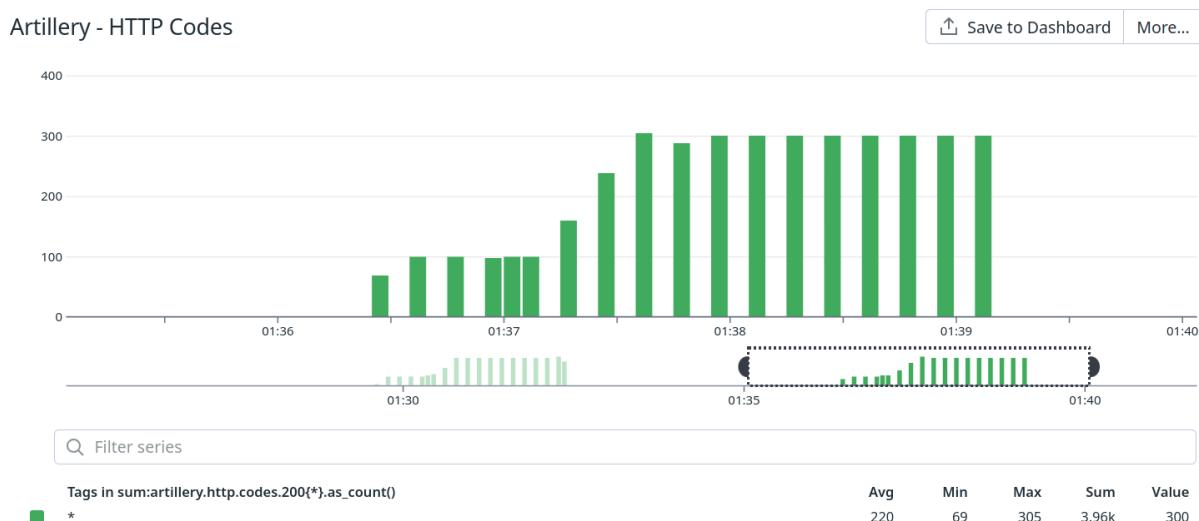


Gráfico 2 de Space news caso base - Códigos de respuesta de requests

Artillery - Error Type

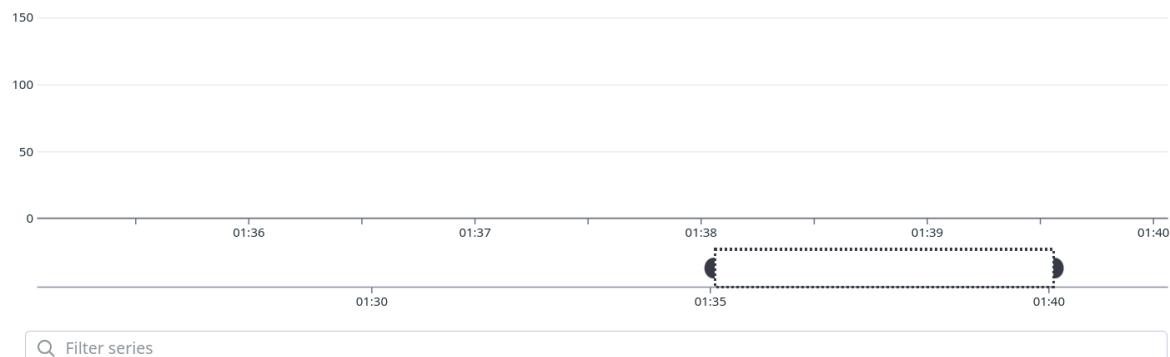
[Save to Dashboard](#) More...

Gráfico 3 de Space news caso base - Tipos de error

System CPU Usage

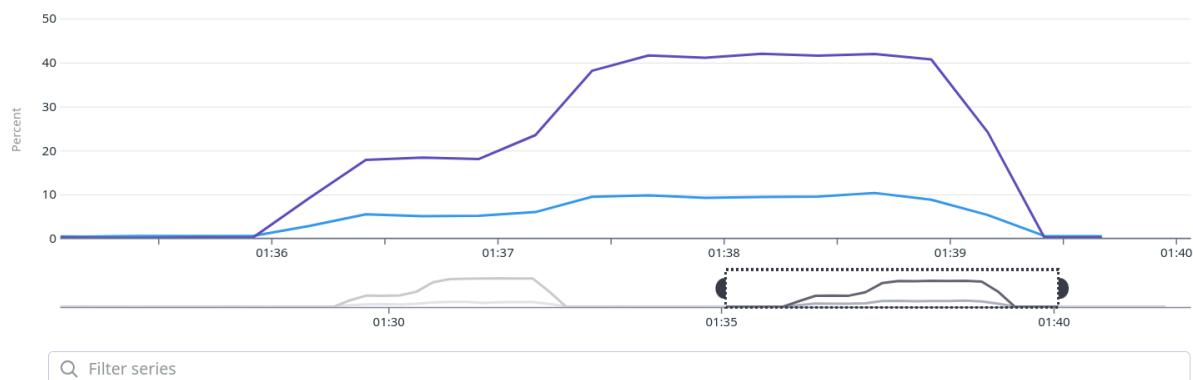
[Save to Dashboard](#) More...

Gráfico 5 de Space news caso base - Uso de CPU

Tags	Metric	Avg	Min	Max	Sum	Value
■ *	avg:system.cpu.user{*}	5.3 %	0.54 %	10.4 %	102 %	0.6 %
■ *	avg:system.cpu.system{*}	21.1 %	0.20 %	42.0 %	401 %	0.33 %

System Memory Usage

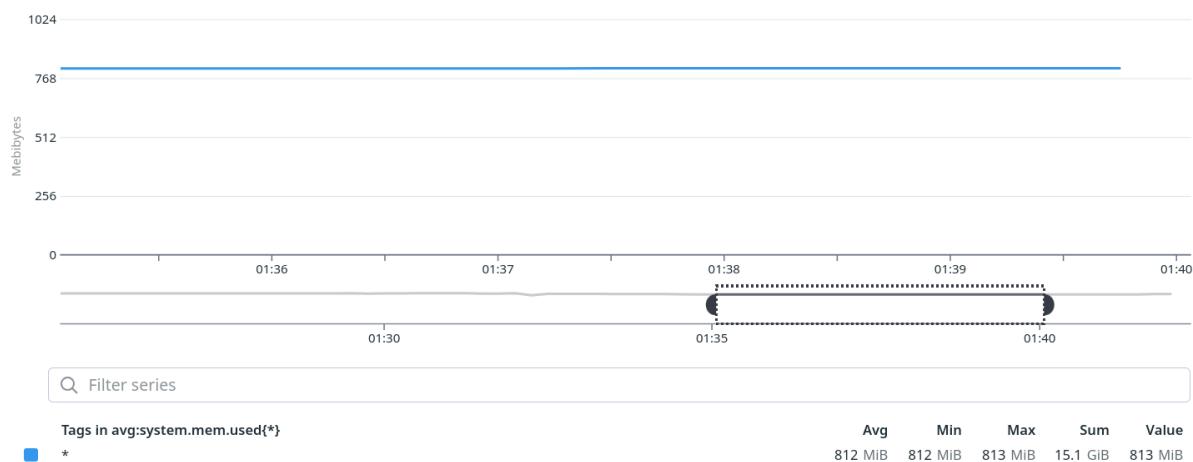
[Save to Dashboard](#) [More...](#)


Gráfico 6 de Space news caso base - Uso de memoria

Response Time

[Save to Dashboard](#) [More...](#)


Gráfico 7 de Space news caso base - Tiempo de respuesta P90 y P95

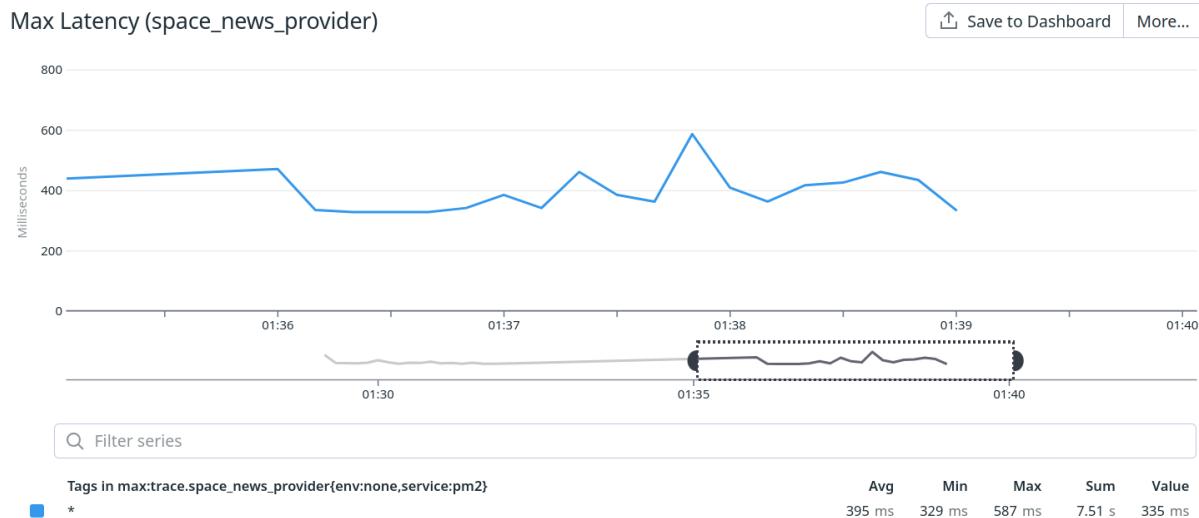


Gráfico 8 de Space news caso base - Latencia (space_news provider)

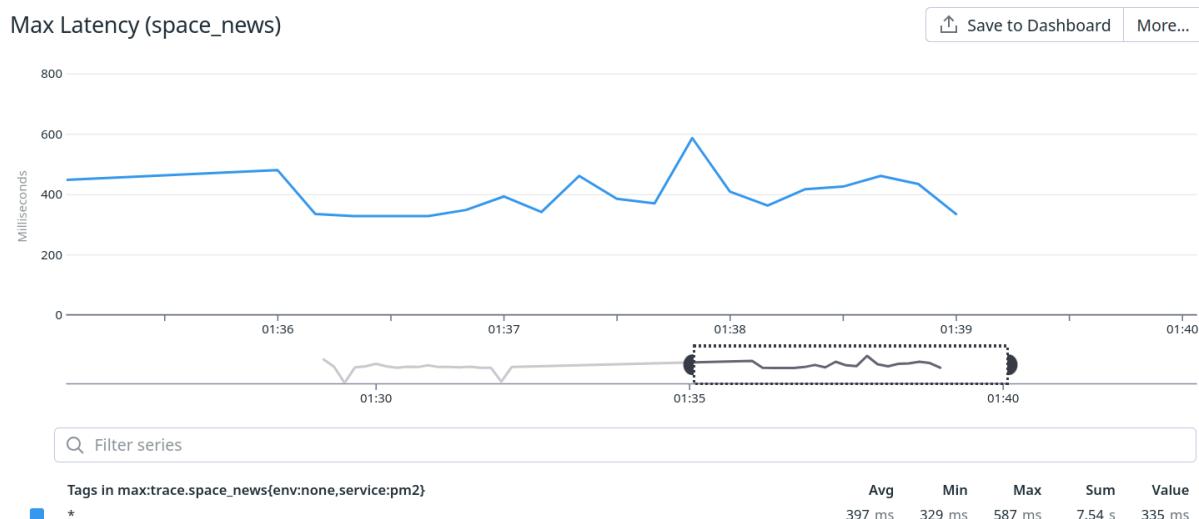


Gráfico 9 de Space news caso base - Latencia (space_news)

Aplicando la misma carga que en los endpoint Metar, space news no tiene fallos (de las 4000 requests realizadas).

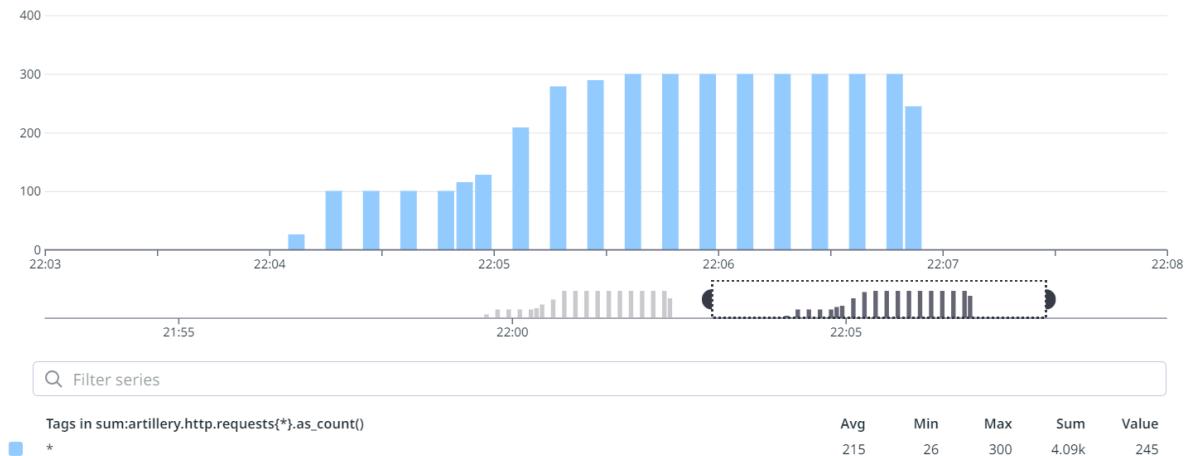
El uso de CPU acompaña a la carga del escenario hasta llegar al 42%, mientras que la memoria se mantiene constante en 810 mb.

El tiempo de respuestas fue bajando a lo largo del escenario, con tres llanuras. La primera de 600 ms (P90) y 750 ms (P95), la segunda de 420 ms (tanto en P90 como P95) y la tercera de 300 ms (también tanto P90 como P95).

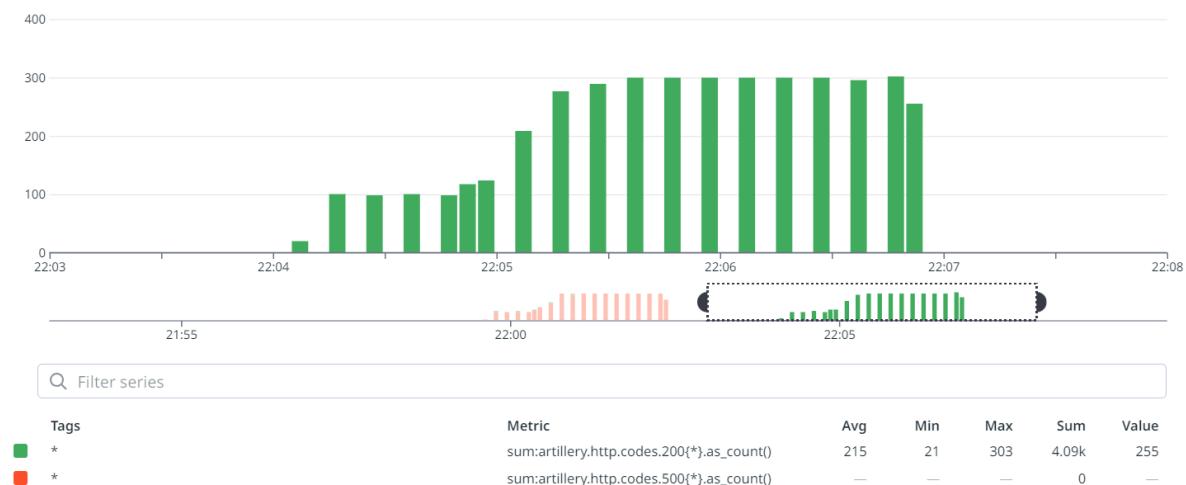
La latencia promedió 400 ms, con mínimos en 330 ms y máximos en 590 ms.

Space news - Cache

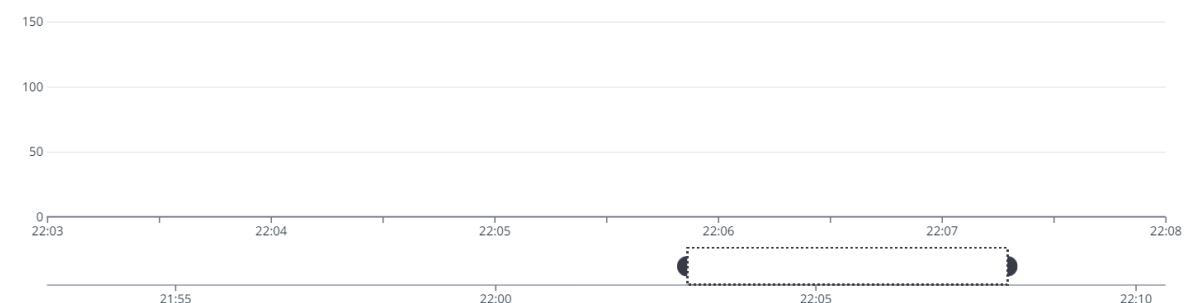
Artillery - HTTP Requests

**Gráfico 1 de Space news cache - Número de requests**

Artillery - HTTP Codes

**Gráfico 2 de Space news cache - Códigos de respuesta de requests**

Artillery - Error Type

**Gráfico 3 de Space news cache - Tipos de error**

System CPU Usage

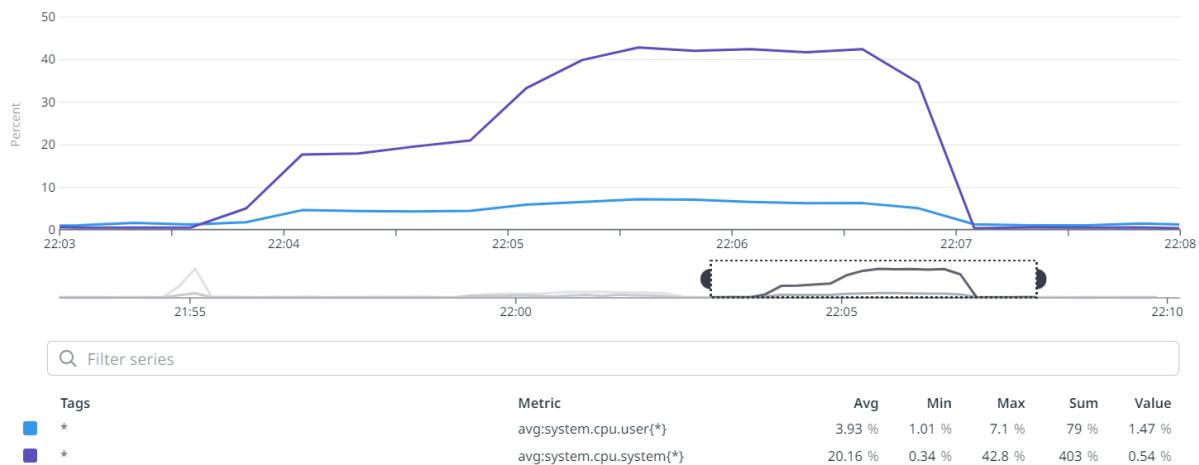
[Save to Dashboard](#) [More...](#)


Gráfico 5 de Space news cache - Uso de CPU

System Memory Usage

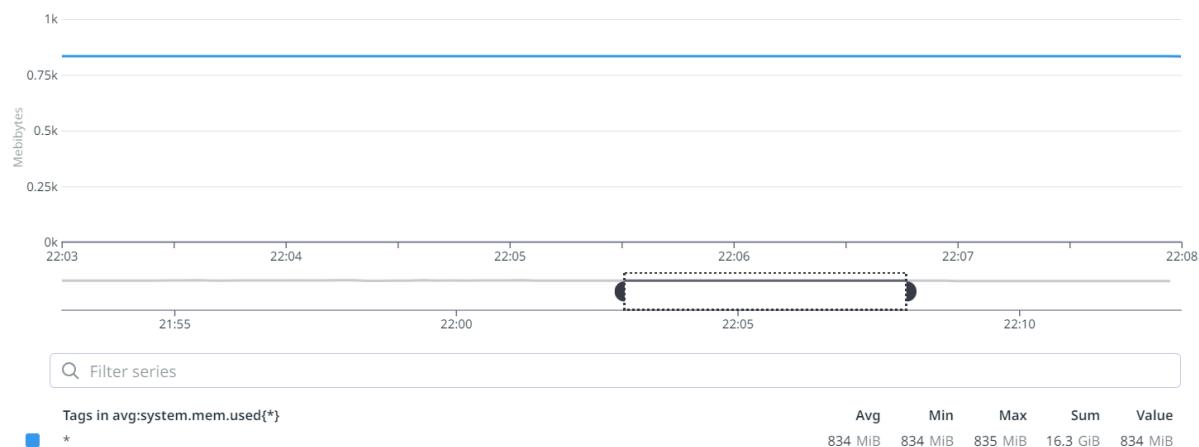
[Save to Dashboard](#) [More...](#)


Gráfico 6 de Space news cache - Uso de memoria

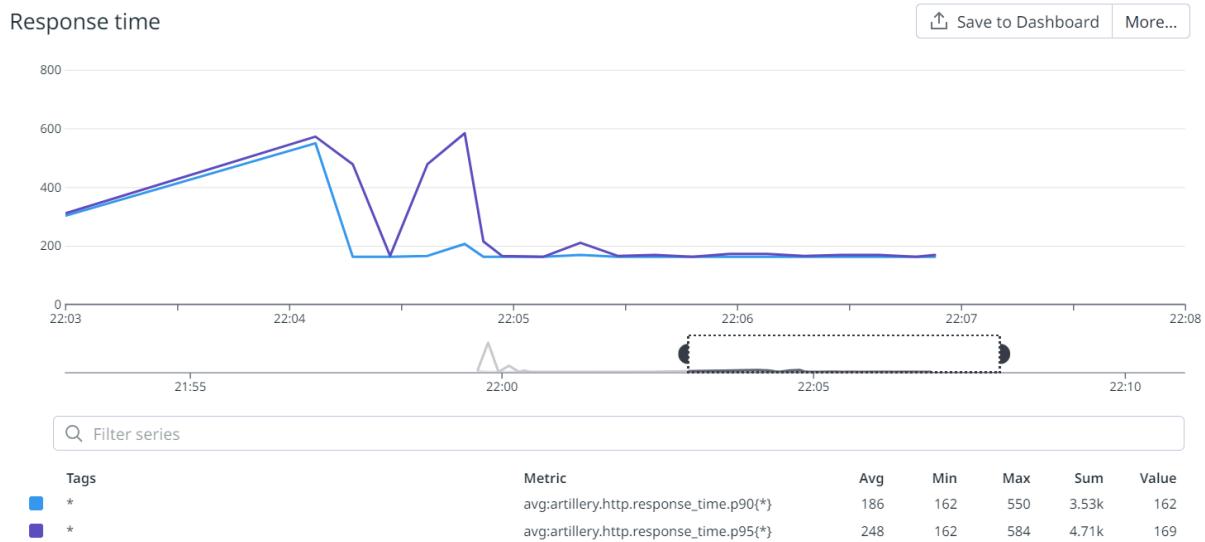


Gráfico 7 de Space news cache - Tiempo de respuesta P90 y P95

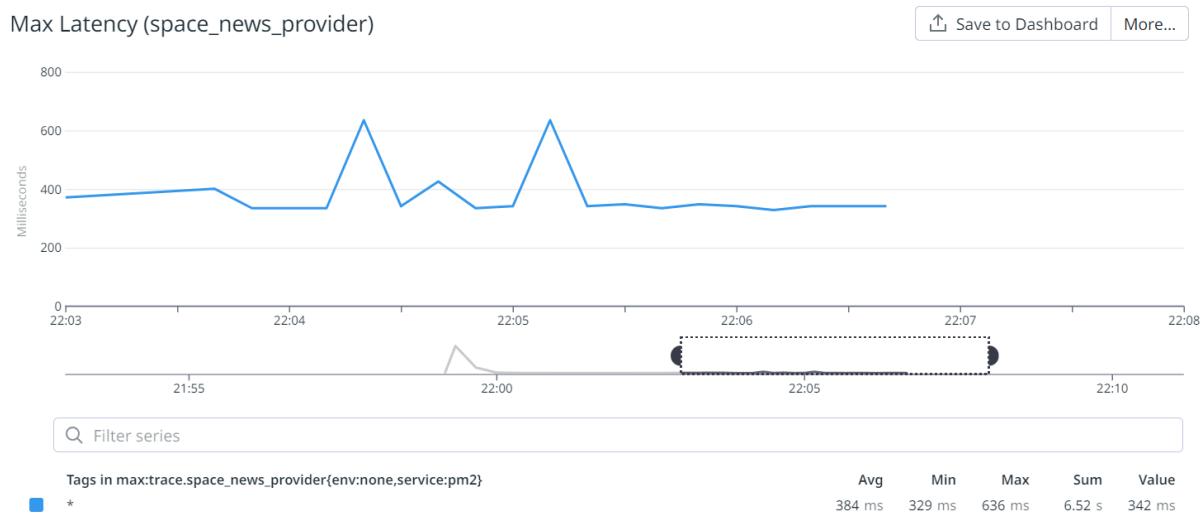


Gráfico 8 de Space news cache - Latencia (space_news_provider)

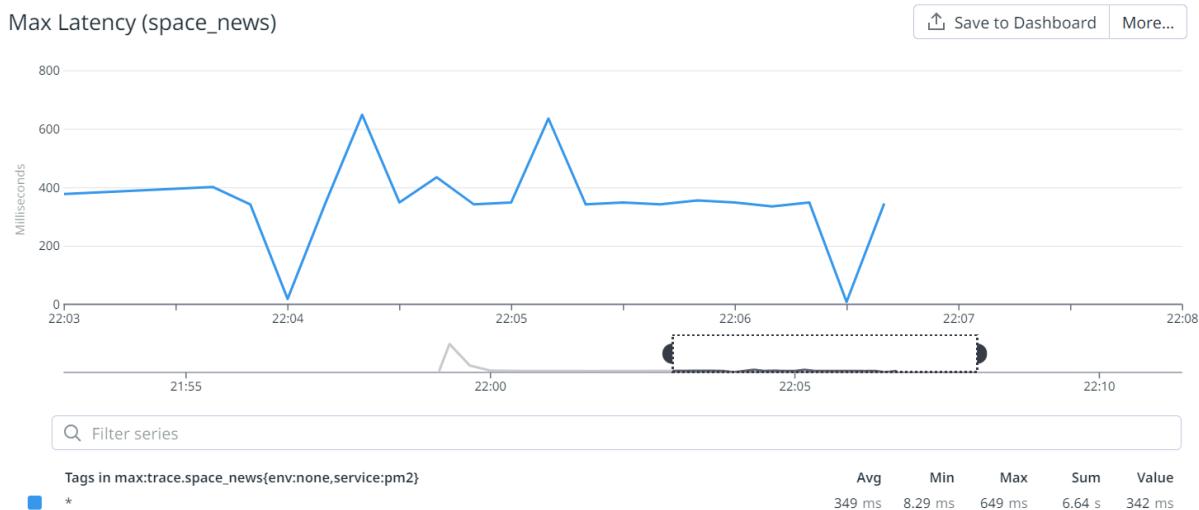


Gráfico 9 de Space news cache - Latencia (space_news)

Al implementar cache, tampoco hay errores de las 4000 requests.

El consumo de CPU también acompañó la carga llegando hasta el 42% (**igual que el caso base**) y el uso de memoria se mantuvo constante en 830 mb (**3% más que el caso base**).

El tiempo de respuesta de P90 fue muy superior al caso base. Comenzando con un máximo de 600 ms, la mayoría del escenario el tiempo de respuesta se mantuvo en 150 ms (como mínimo el **doble** de rápido). Sin embargo, el último 5% de las requests tuvieron picos de latencia de hasta 500 ms.

Por último la latencia. El estilo cache fue el único donde space_news_provider y space_news tuvieron gráficos **distintos**. space_news llegó a tener mínimos de 10 ms, mientras que en space_news_provider los mínimos fueron de 330 ms.

El uso de recursos es similar al caso base. Sin embargo, es notable el ligeramente mayor uso de memoria de cache. Esta diferencia del 4% podría ser por margen de error, ya que el cache implementado con Redis no consume memoria de nuestra VM.

El impacto en el tiempo de respuesta y latencia es notable. Estos mejoraron sustancialmente.

Es interesante notar el tiempo de respuesta del P95 de las requests. El pico de 500ms podría denotar que fue necesario volver a llamar al servicio proveedor, a diferencia de la mayoría de las requests donde el cache logró mejorar su performance.

Space news - Réplicas

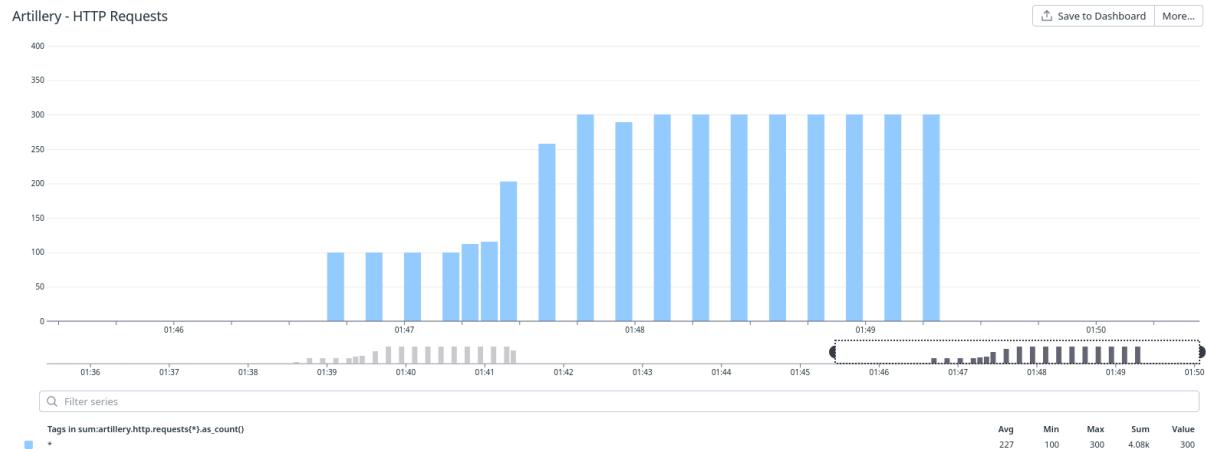


Gráfico 1 de Space news réplicas - Número de requests

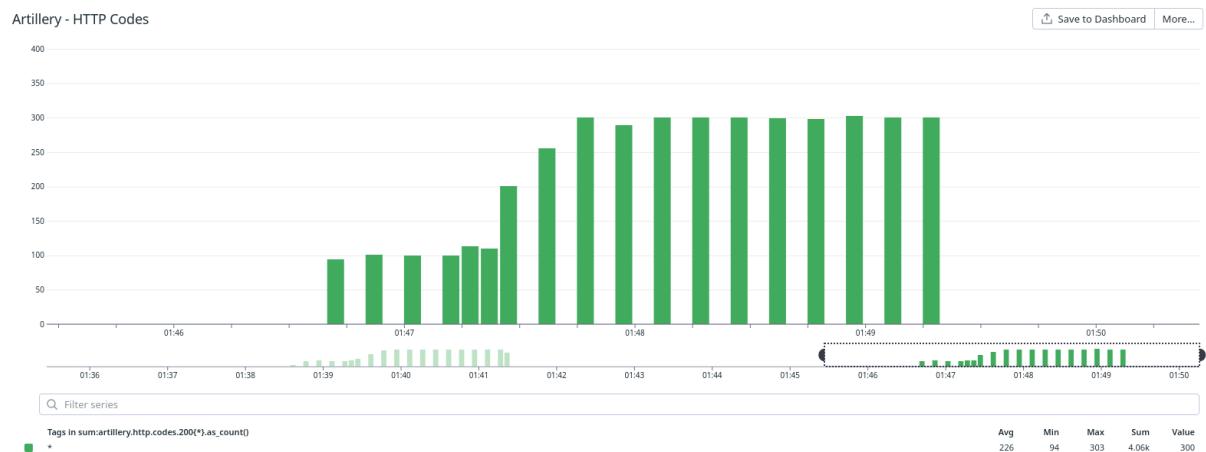


Gráfico 2 de Space news réplicas - Códigos de respuesta de requests

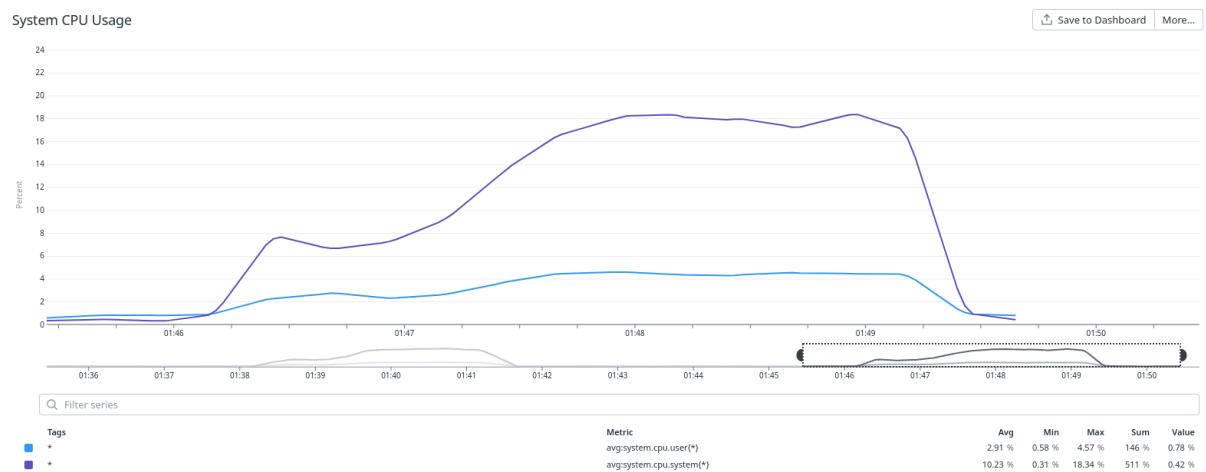


Gráfico 5 de Space news réplicas - Uso de CPU

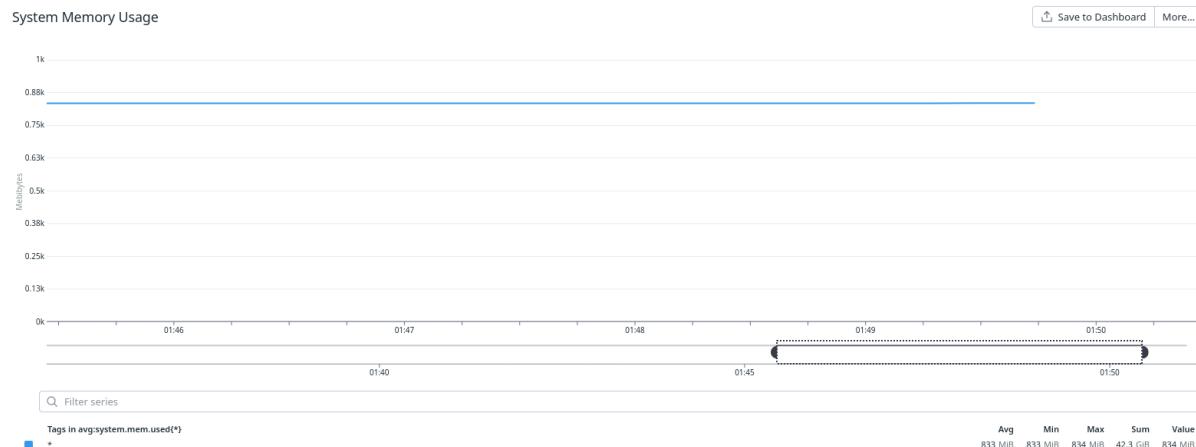


Gráfico 6 de Space news réplicas - Uso de memoria

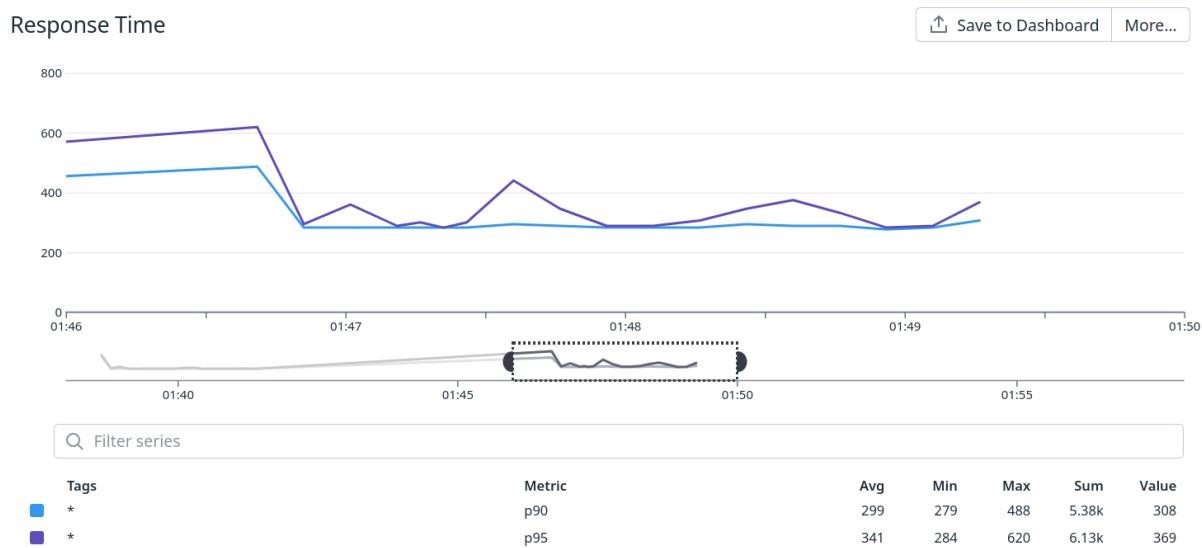


Gráfico 7 de Space news réplicas - Tiempo de respuesta P90 y P95

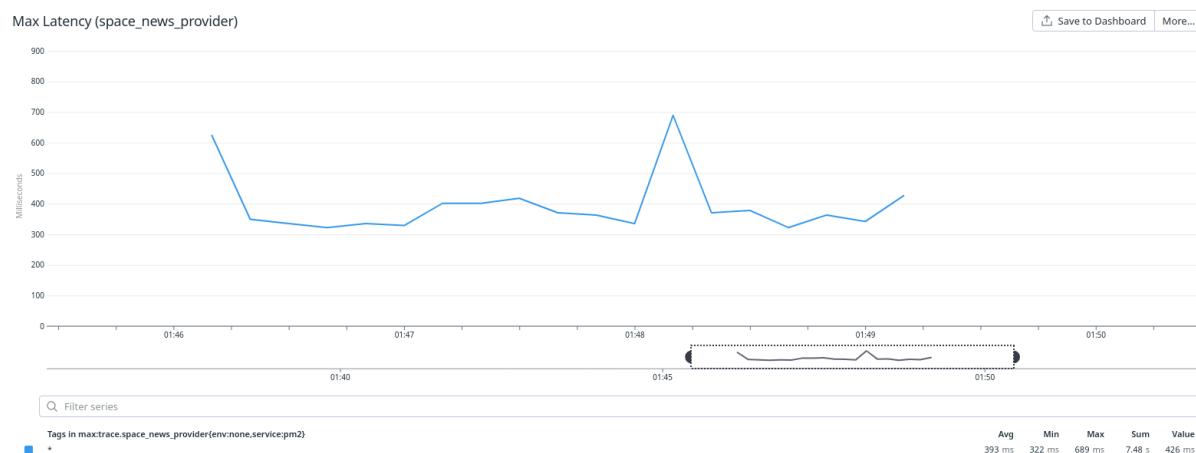


Gráfico 8 de Space news réplicas - Latencia (space_news_provider)

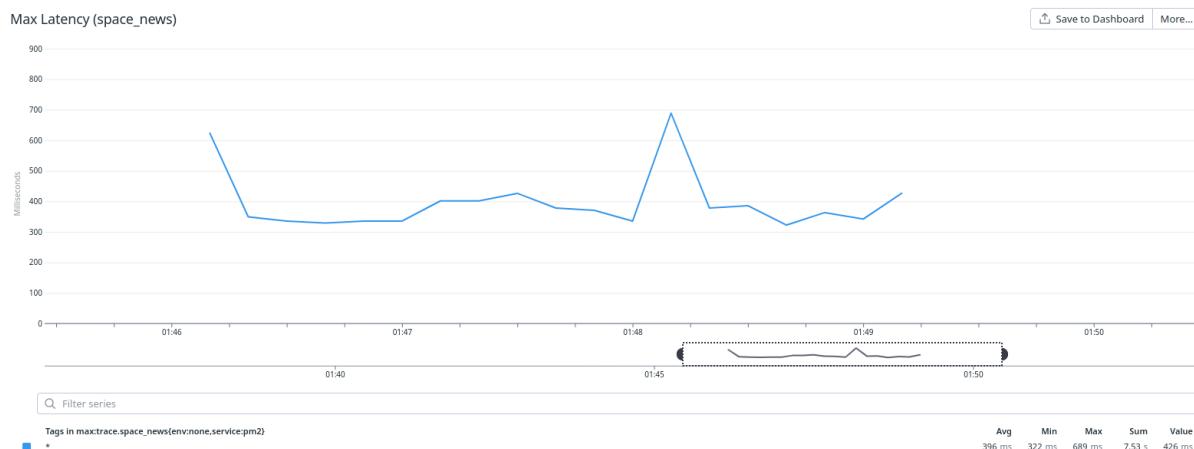


Gráfico 9 de Space news réplicas - Latencia (space_news)

Al igual que en los casos anteriores, con réplicas no hubo fallos (de las mismas 4000 requests).

Sin embargo, es notable la menor carga de CPU. Este llegó hasta el 18% (**24 puntos menos que el caso base**). El uso de memoria fue **2% mayor** que el caso base, 830 mb.⁷

El tiempo de respuesta mejoró en todos los sentidos al implementar réplicas. El P90 comenzó con un pico de 500 ms (**15% menos que el caso base**) y se estabilizó en 280 ms (**-7%**). El P95 comenzó con un pico de 600 ms (**-20% que el P95 del caso base**), tuvo dos picos de 400 ms y luego también se estabilizó en 280 ms.

La latencia comenzó en un pico de 600 ms para luego bajar y mantenerse cerca de 350 ms (salvando un segundo pico de 700 ms). Este mínimo de 350 ms es **10% mejor** que el promedio de 400 ms y **similar** mínimo del caso base (330 ms).

La implementación de réplicas trajo beneficios claros. El consumo de recursos fue menor y el rendimiento del tiempo de respuesta y latencia también mejoraron. La implementación de réplicas en el caso Space news mantiene la **disponibilidad** (ya que no excluye a ninguna request) y mejora el **rendimiento y escalabilidad**.

⁷ Al igual que en cache, podría interpretarse que consumen la misma memoria, ya que este 2% podría estar adentro del margen de error de la medición.

Space news - Rate limiting

Para este escenario fijamos un límite de 50 requests cada 10 segundos:

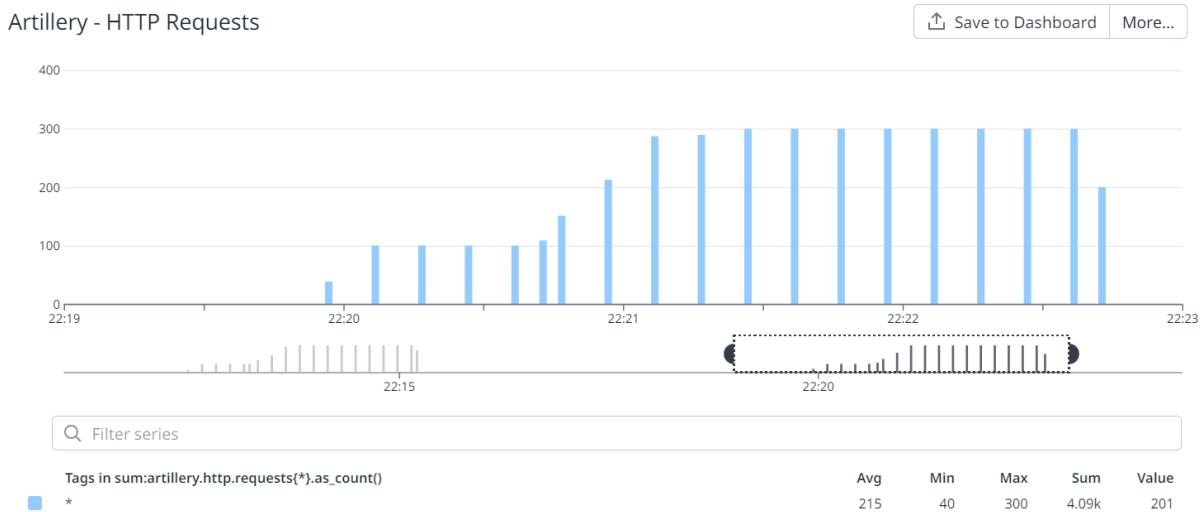


Gráfico 1 de Space news rate limiting - Número de requests

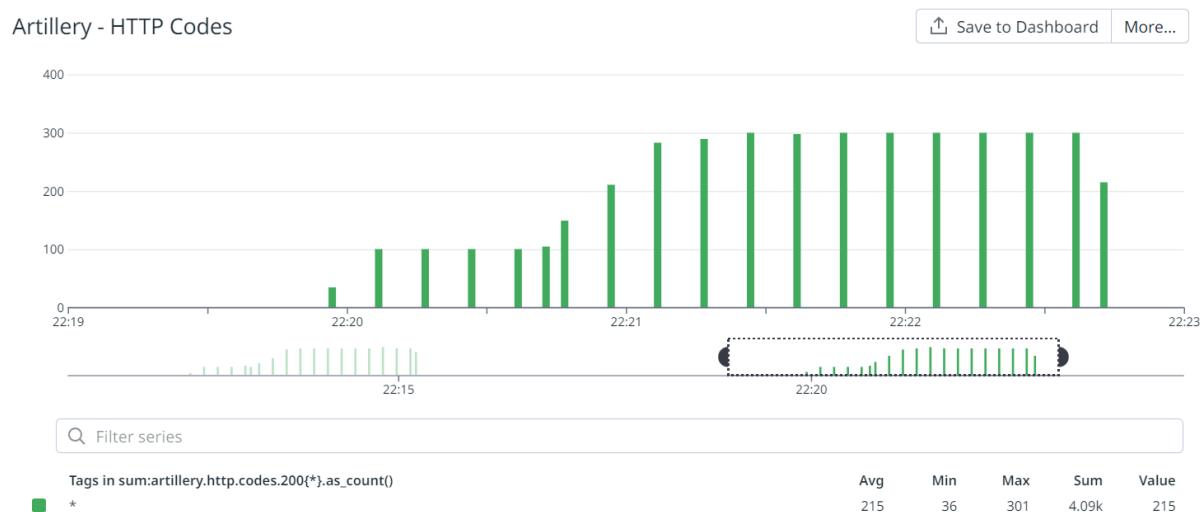


Gráfico 2 de Space news rate limiting - Códigos de respuesta de requests

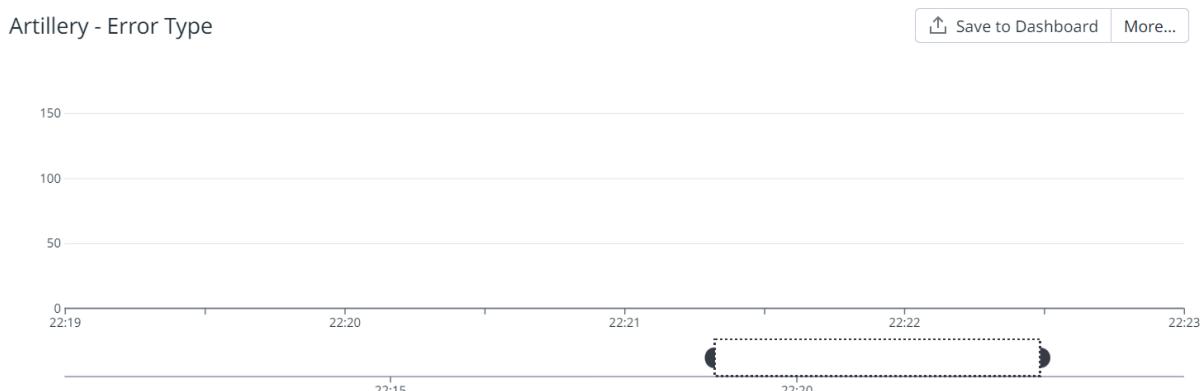


Gráfico 3 de Space news rate limiting - Tipos de error

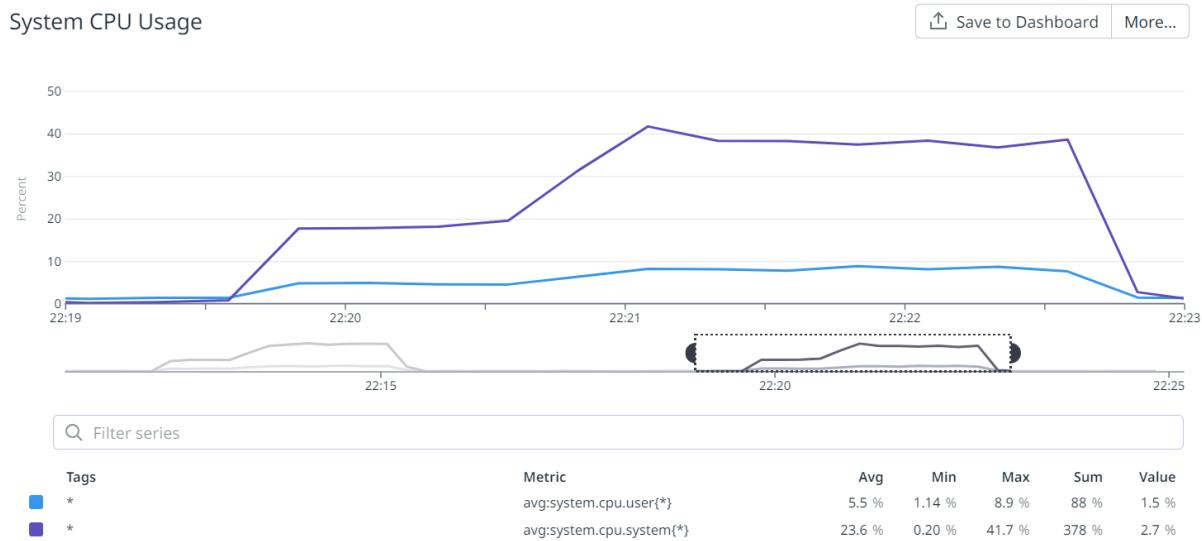


Gráfico 5 de Space news rate limiting - Uso de CPU

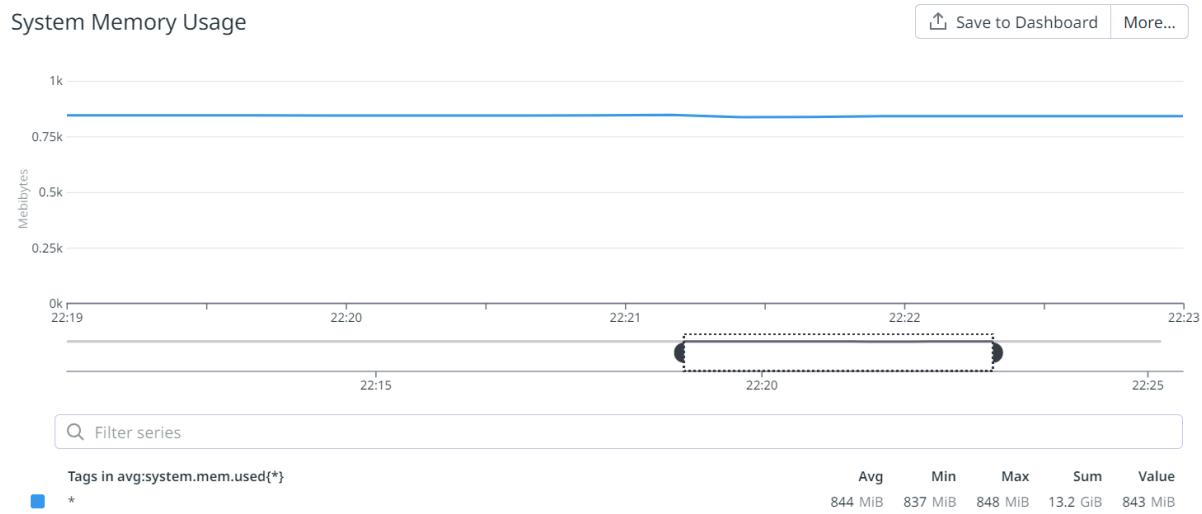


Gráfico 6 de Space news rate limiting - Uso de memoria

Response time

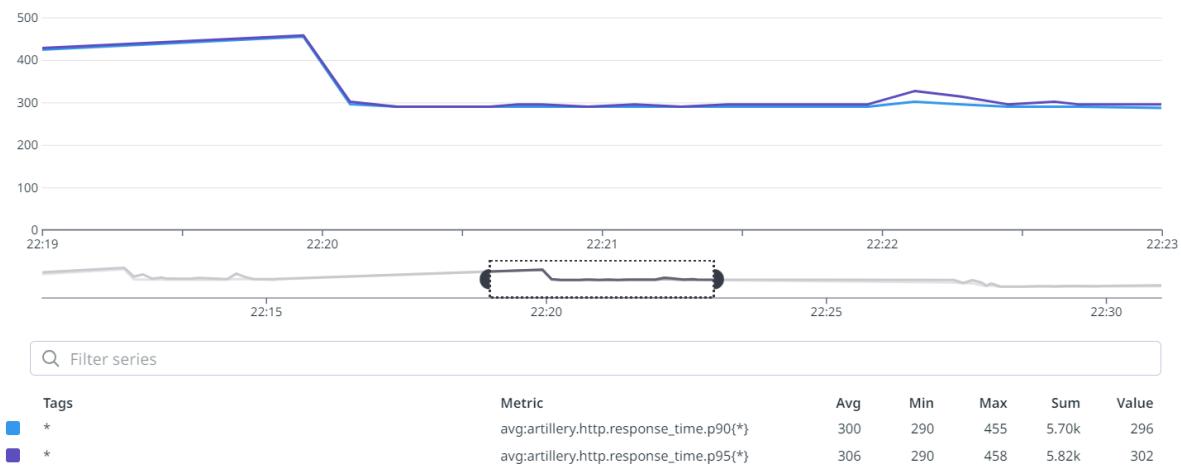
[Save to Dashboard](#) More...

Gráfico 7 de Space news rate limiting - Tiempo de respuesta P90 y P95

Max Latency (space_news_provider)

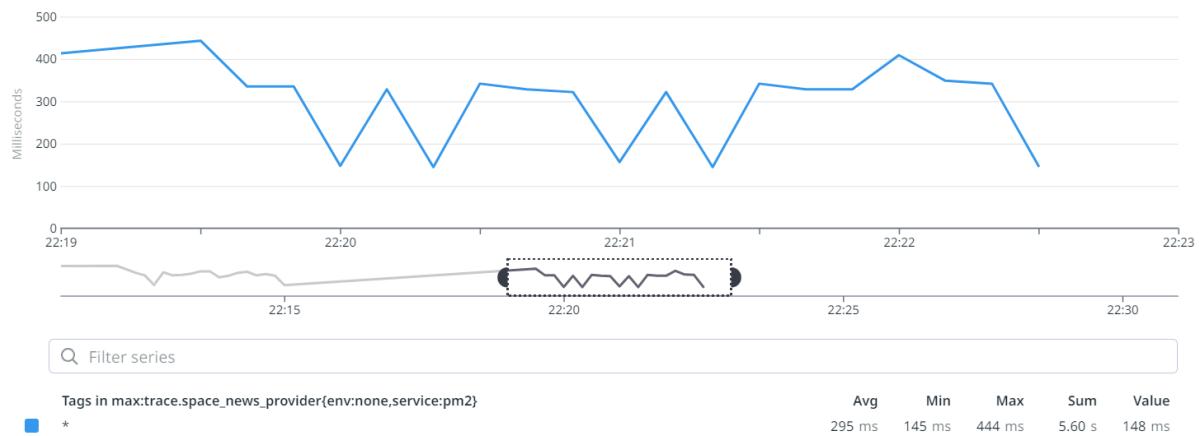
[Save to Dashboard](#) More...

Gráfico 8 de Space news rate limiting - Latencia (space_news_provider)

Max Latency (space_news)

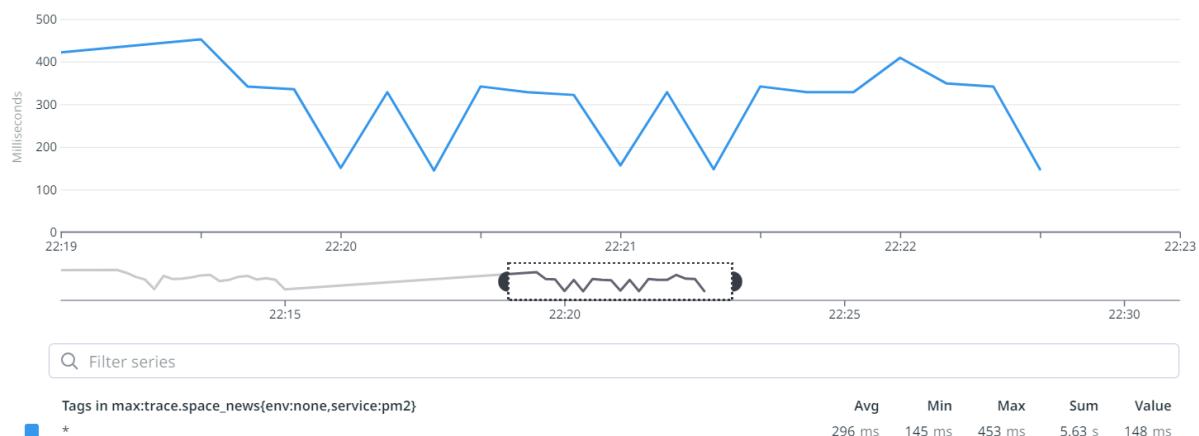
[Save to Dashboard](#) More...

Gráfico 9 de Space news rate limiting - Latencia (space_news)

Por último veamos cómo se comporta este escenario con rate limiting. De las 4000 requests, ninguna falla (al igual que el caso base).

El uso de CPU fue igual que el caso base, y el de la memoria fue **4% mayor**.

El tiempo de respuesta de este caso fue muy similar al caso base, excepto que su pico máximo fue **30% menor** (450 ms vs 600 ms del caso base).

La latencia en rate limiting fue muy irregular. Sus mínimos llegaron a 150 ms y los máximos a 450 ms. Sin embargo ambos valores son menores que el caso base (330 ms y 600 ms). También se notan mejoras si se tiene en cuenta el promedio de los tiempos de respuesta, siendo un **30% mejores** las de rate limiting (300 ms contra 400 ms).

Rate limiting trajo mejoras al tiempo de respuesta y latencia sin perjudicar la disponibilidad. Es notable en este caso que ninguna request fue rechazada (gráfico 3). Interpretamos que esto es así ya que el servidor dio a basto para responder los pedidos aplazados inicialmente, ya que la carga de todo el escenario es baja y cercana al límite del rate limiter.

Space news - Análisis

El escenario elegido para este endpoint tuvo una carga baja y no hizo aparecer errores en el caso base. El análisis se centrará entonces en los atributos de rendimiento y escalabilidad.

En casi todos los casos el consumo de recursos fue muy similar. Existen pequeñas diferencias (como variaciones de 4% de consumo de memoria) que creemos que están dentro del margen de error.

El único caso donde es notable la **mejora de consumo de CPU es en replicación**, donde el máximo de uso de procesador fue 24 puntos menos que el caso base (18% vs 42%). En cuanto a la nula variación de memoria, esto se correlaciona con nuestra aplicación, pues la misma no requiere memoria para su funcionamiento.

Cache fue, con diferencia, el mejor estilo para mejorar el tiempo de respuesta. Mientras el resto de casos llegaban hasta los 280 ms de tiempo de respuesta, la mayoría de las requests en caché se respondieron en 150 ms.

Rate limiting no trajo ventajas por sobre replicación y cache. Su mejora en consumo de recursos o tiempo de respuesta son inferiores a los dos estilos.

Conclusiones

Previo a explicar detalladamente las conclusiones derivadas del análisis individual de cada uno de los casos, es importante mencionar que no todos los escenarios presentados en el informe fueron ejecutados en la misma cuenta de Azure, si no que se han ejecutado en por lo menos 2 cuentas distintas. En todas las oportunidades, obtuvimos resultados muy similares, donde las variaciones eran despreciables a los fines del trabajo práctico. Por lo tanto, podemos considerar como definitivas las mediciones expuestas en el presente escrito.

Además, probar los escenarios con varias máquinas virtuales con la misma especificación nos permite afirmar que estas son efectivamente iguales, como promete proveer Azure con su listado de máquinas disponibles.

Dichas mediciones nos permitieron observar el impacto real de todos los estilos arquitectónicos que veníamos estudiando. Estas son nuestras conclusiones:

Con estas métricas podemos notar que se deben elegir distintos estilos arquitectónicos según qué atributo se desea priorizar.

- Notamos que los mejores tiempos de respuesta (y por ende performance) venían con el estilo **cache**, pero este podía tener más fallos o picos de ralentización (por tener que actualizar su cache, por ejemplo), perjudicando la disponibilidad.
- Contrariamente, la replicación beneficiaba la disponibilidad (por minimizar las requests fallidas) y la escalabilidad (por consumir menos recursos del procesador), pero no mejoraba sustancialmente la performance (a diferencia del cache).
- La táctica **rate limiting** no fue tan atractiva como en el TP1. No trajo grandes mejoras de performance por sobre el cache, ni disminuyó drásticamente el uso de recursos como la replicación. Notamos que solo logró perjudicar la disponibilidad de nuestro servicio, ya que bloqueó a una porción de las requests. Sin embargo este estilo puede seguir siendo útil en otros contextos externos a los estudiados en este informe.

Únicamente en los escenarios cache, *Max latency (endpoint)* y *Max latency (endpoint_provider)* fueron distintos. Esto se debe a que la implementación con cache no necesita esperar a la respuesta del provider para responder la request, ya que nuestra aplicación ya tiene nuestra respuesta guardada. El resto de implementaciones dependen del provider externo para dar su propia respuesta.

Nos pareció muy provechoso usar dos escenarios de carga distintos para los tres endpoints distintos. Fue interesante notar cómo, bajo la misma carga, Metar tuvo algunas requests fallidas en su caso base mientras que Space News no presentó ningún problema de disponibilidad. A su vez, Useless Facts requirió aumentar la carga del escenario para hacer fallar a una porción de sus requests.

Entendemos que los endpoints fallan en distintos puntos por la disponibilidad de sus proveedores, donde Metar fue el más fácil de hacer fallar.

Además, notamos que nunca varió sustancialmente el uso de memoria en nuestra aplicación, bajo ningún escenario o estilo. Concluimos es así porque nuestra aplicación no

hace un gran uso de memoria. El único estilo que consume memoria, cache, utiliza Redis, que es externo a la VM estudiada.

Es muy interesante notar que nuestras conclusiones en este informe son diferentes al TP1, incluso aunque se tratase de la misma aplicación. En este trabajo práctico encontramos que cache traía ventajas si se priorizaba la performance, mientras que en el TP1 esto no se pudo ver en las métricas. Replicación, en cambio, también fue útil en este TP, pero en este informe se vio superada en el ámbito performance por el cache (a diferencia del TP1, donde replicación fue la clara ganadora en todos los atributos).

Rate limiting, al igual que en el TP1, solo es una opción atractiva si se está dispuesto a sacrificar disponibilidad y rechazar a una porción de los clientes del servicio.

Podríamos concluir que ningún estilo es universalmente superior a otro, sino que debemos estudiar las ventajas y desventajas de cada uno en cada aplicación que realicemos. Solo con métricas podemos tomar con confianza ciertas decisiones de diseño.

Referencias

- <https://learn.microsoft.com/>
- <https://docs.ansible.com/>
- <https://developer.hashicorp.com/terraform/docs>
- <https://pm2.io/docs/runtime/overview/>
- <https://docs.datadoghq.com/>
- <https://www.artillery.io/docs>
- <https://medium.com/@djsmith42/how-to-metric-edafaf959fc7>