2023 - 1C

75.73 - Arquitectura de Software

TP Nº 1 - Grupo DDL

Integrantes:

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

Fecha de entrega: 11/05/23

Índice

Introducción

Para analizar y comparar distintas soluciones informáticas, como arquitectos nos interesará saber qué características tienen y cuáles son las necesidades que logran satisfacer. A estas características de nuestro sistema las llamamos atributos de calidad y deben ser mensurables o testeables. Lamentablemente, favorecer un atributo implica perjudicar otro, por lo cual, es importante conocer las necesidades de nuestros stakeholders para determinar cuáles se deben priorizar.

Los principales atributos que nos interesará evaluar en este trabajo son:

- **Disponibilidad** (*availability*): es la probabilidad de que el sistema esté disponible y su habilidad para reparar fallas de tal manera que el período que el sistema está fuera de servicio sea mínimo.
- **Rendimiento** (*performance*): es la habilidad del sistema para reaccionar ante ciertos eventos en un determinado tiempo. La interacción entre elementos del sistema determina la velocidad y eficiencia en el uso de la red (*network performance*) y el rendimiento percibido por el usuario (*user perceived performance*). También nos interesará evaluar la eficiencia, que es el rendimiento por recurso consumido.
- **Escalabilidad** (*scalability*): es la habilidad para incorporar recursos adicionales de manera efectiva, por ejemplo para soportar un número grande de componentes o de interacciones entre componentes. Una mejora en la escalabilidad produce una mejora mensurable de algún atributo de calidad, no requiere un esfuerzo inapropiado y no interrumpe el funcionamiento del sistema.

Mientras tanto, las **tácticas** son decisiones de diseño o restricciones que nos permitirán promover o inhibir ciertos atributos de calidad. Como arquitectos tenemos que elegir las correctas según los atributos que queremos priorizar por sobre el resto.
Los **estilos arquitectónicos** son un conjunto coordinado de restricciones arquitectónicas que restringen las funciones o características de los componentes y las relaciones permitidas entre esos elementos. En resumen, categorizan arquitecturas y determinan características comunes por medio de restricciones importantes en los elementos y sus relaciones.

Los estilos que estaremos evaluando son:

- Replicated Repository: más de un proceso provee el mismo servicio.
- Cache: replicación del resultado de una solicitud (request) para poder ser utilizado en posteriores solicitudes.
    - Lazy population: el resultado es guardado cuando es solicitado.
    - Active population: el resultado es guardado por adelantado, anticipando que se producirá cierta solicitud.
- Client-Server: el cliente consume los servicios de un servidor.
- Load Balancer: recibe las solicitudes de los clientes y los distribuye entre un grupo de servidores, direccionando luego la respuesta del servidor seleccionado al cliente apropiado.
- Rate-limiting: a partir de cierto número de requests, una porción de estas no son respondidas.

Desarrollo

Nuestro sistema provee diferentes servicios que nos permitirán evaluar el impacto de aplicar diferentes estilos arquitectónicos en tres contextos distintos.

Los servicios a analizar son los siguientes:

- **`/fact`**: retorna un *dato* distinto por cada request.
- **`/metar?station=<code>`**: dado un código de un aeródromo, devuelve un reporte del estado meteorológico del mismo (METAR).
- **`/space_news`**: devuelve los títulos de las últimas 5 noticias sobre actividad espacial.

Los datos los obtenemos aplicando el estilo Client-Server para consumir las siguientes APIs: uselessfacts[1], NOAA Text Data Server API[2] y Spaceflight News API[3].

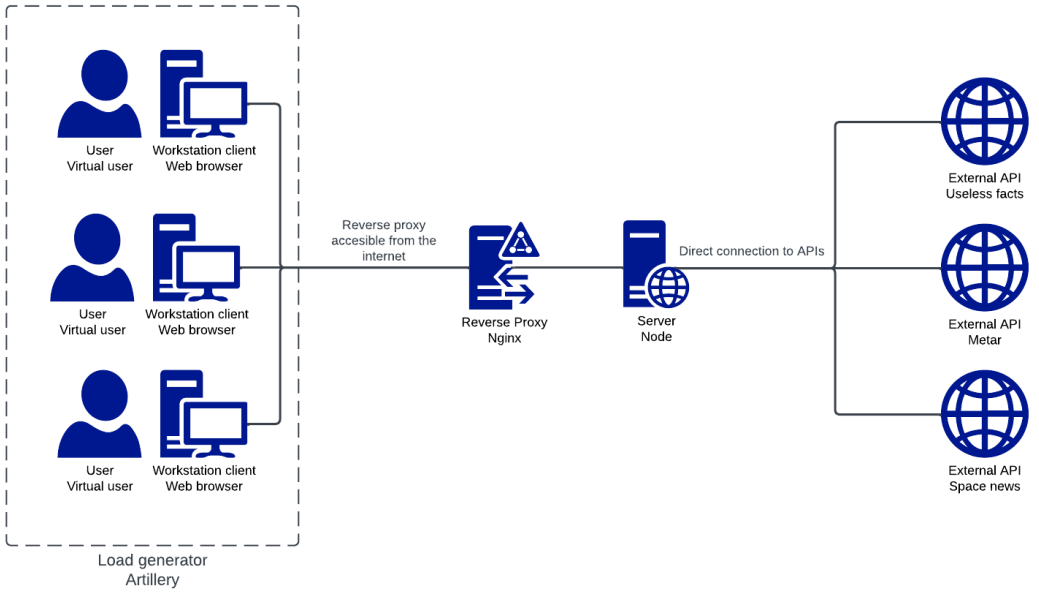
A cada servicio lo vamos a analizar haciendo pruebas de **Load Testing**, con el cual se simula un aumento progresivo de la cantidad de usuarios y sus solicitudes. Para esto utilizaremos la herramienta Artillery, que genera carga de peticiones de usuarios a gran escala.

Para la recolección de los datos utilizaremos la combinación de cAdvisor, Graphite y Grafana.

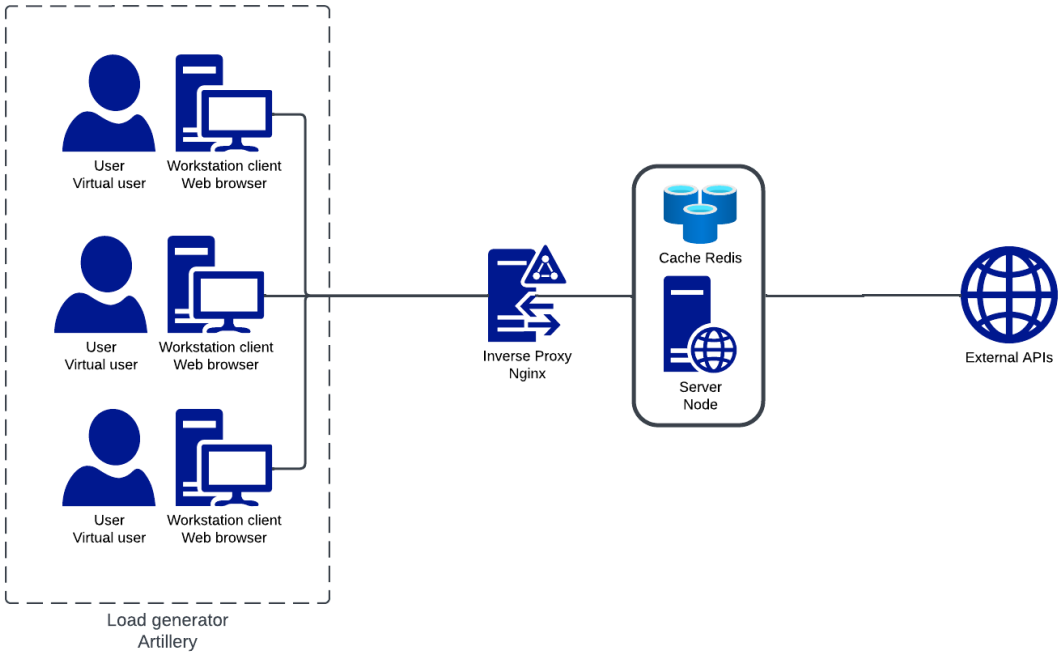
Además, en cada servicio observaremos cómo se comporta el sistema en cuatro situaciones distintas: primero en su caso base y luego con tres estilos arquitectónicos distintos. Las cuatro situaciones están diagramadas a continuación.

[1] Uselessfact API: https://uselessfacts.jsph.pl
[2] NOAA Text Data Server API: https://www.aviationweather.gov/dataserver
[3] Spaceflight NEWS API: https://spaceflightnewsapi.net

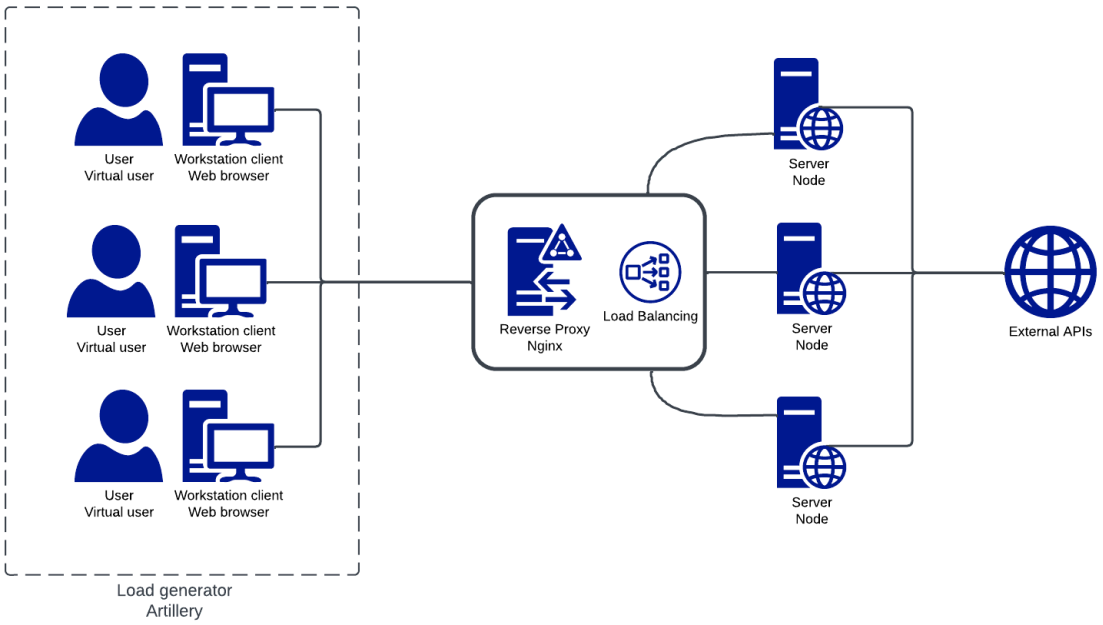
Gráficos de componentes y conectores



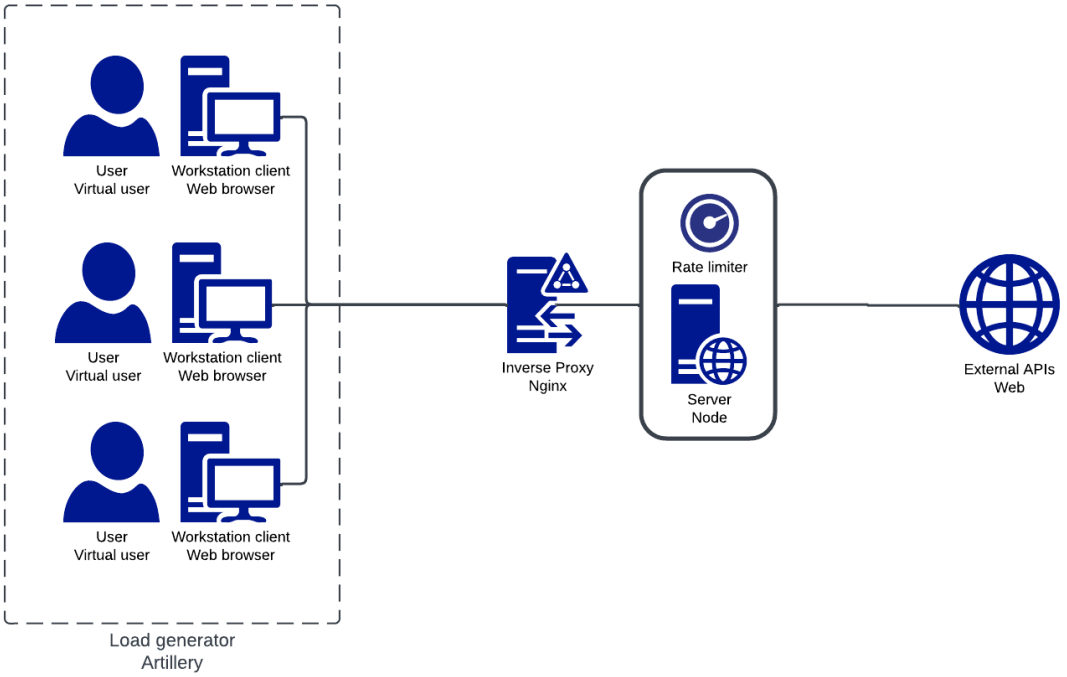
Caso base



Estilo arquitectónico cache



Estilo arquitectónico replication / load balancing



Estilo arquitectónico rate limiting

Useless Facts

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

En los gráficos siguientes (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 500 requests.
- **Plain**: 500 requests por segundo durante 40 segundos.

Useless Facts - Caso base

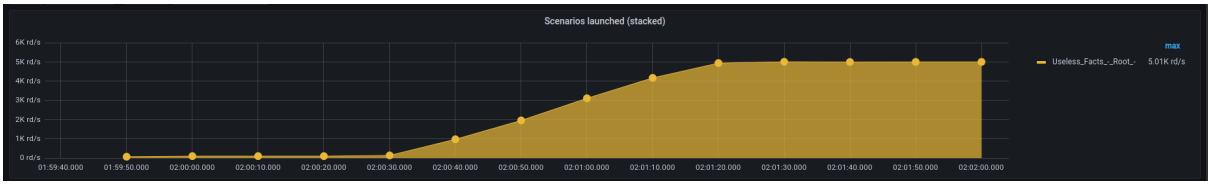


Gráfico 1 - Caso base (escenarios de artillery lanzados)

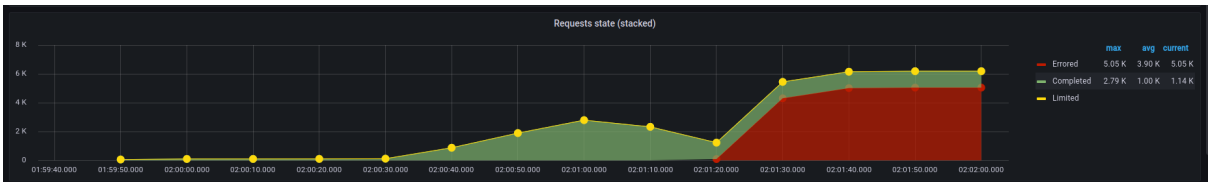


Gráfico 2 - Caso base (estado de los requests)

En el gráfico 1 se pueden notar claramente las etapas: primero el Warm Up, luego el Ramp Up, donde aumenta la cantidad de requests; y por último la meseta donde se mantiene constante la cantidad de requests. Luego de un cierto punto la mayoría de los requests comienzan a fallar, esto es lo que intentaremos solucionar al aplicar las tácticas que siguen.

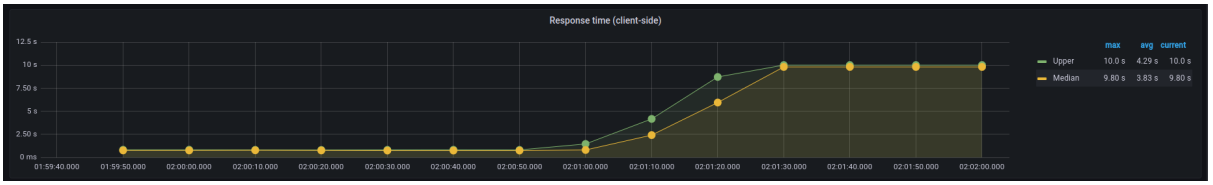


Gráfico 3 - Caso base (tiempo de respuesta lado cliente)

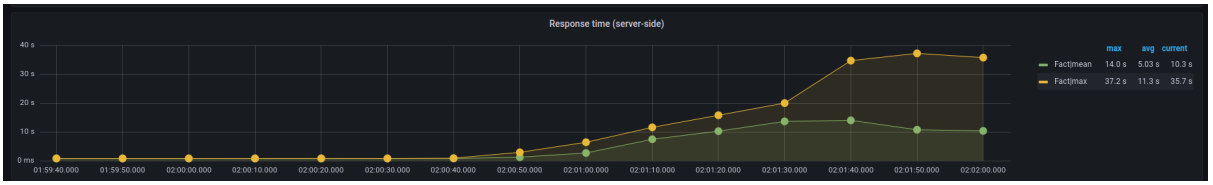


Gráfico 4 - Caso base (tiempo de respuesta lado servidor)

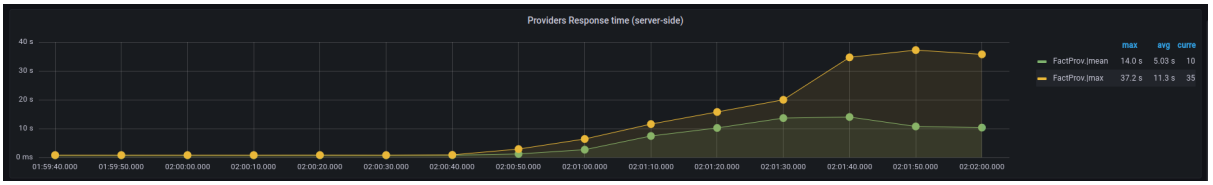


Gráfico 5 - Caso base (tiempo de respuesta de la API externa)

El tiempo de respuesta percibido por el cliente (gráfico 4) es prácticamente el mismo que el del servidor (gráfico 5), esto es porque en este caso base lo único que se hace es llamar a la API externa y retornar la respuesta de la misma directamente hacia el cliente. Además, como era de esperarse, aumenta a medida que crece la carga.

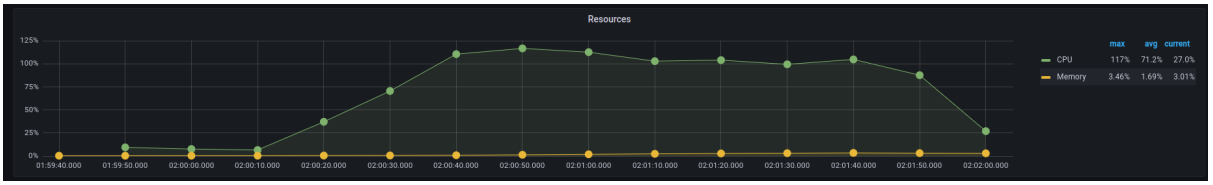


Gráfico 6 - Caso base (recursos utilizados)

En cuanto a los recursos consumidos (gráfico 6), utiliza bastante la CPU y va aumentando a medida que aumenta la cantidad de requests. El uso de memoria no se ve afectado, lo cual es esperable porque no se hacen operaciones que requieran su uso.

Useless Facts - Cache

En principio, la naturaleza del endpoint no parece indicar que el uso de caché pueda ayudar, puesto que en cada request se debería devolver un fact distinto.

Igualmente, podríamos probar aplicar *active population* trayendo y cacheando una cierta cantidad de respuestas de la API provider, luego a nuestros clientes les devolvemos las respuestas directo de el 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.

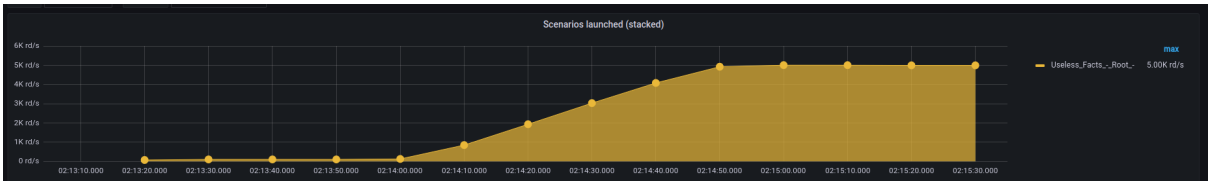


Gráfico 1 - Cache (escenarios de artillery lanzados)

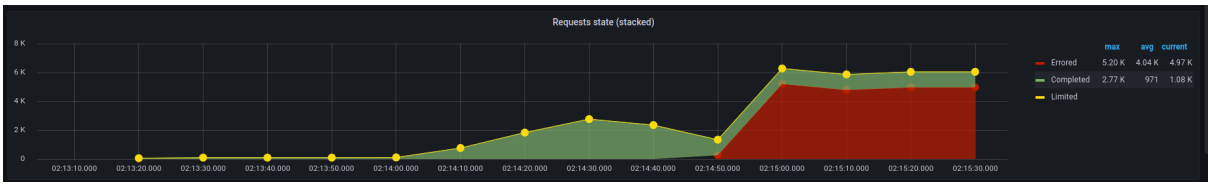


Gráfico 2 - Cache (estado de los requests)

En el gráfico 2 se puede notar algo muy similar al caso base, los requests comienzan a fallar desde el mismo momento, por lo cual, parecería no ayudar a mejorar la disponibilidad.

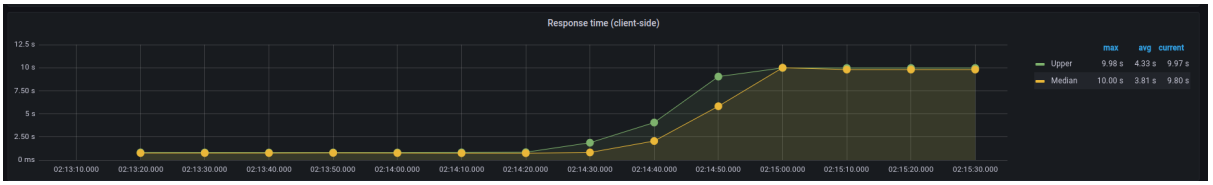


Gráfico 3 - Cache (tiempo de respuesta lado cliente)

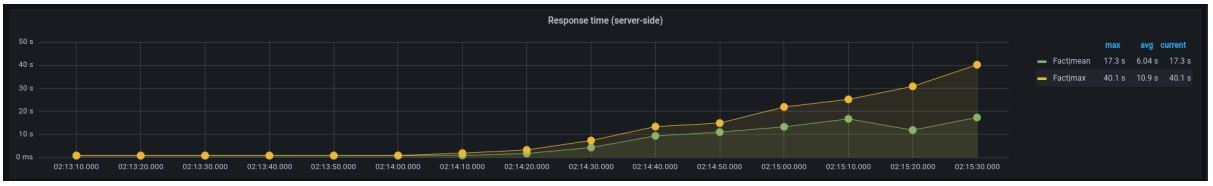


Gráfico 4 - Cache (tiempo de respuesta lado servidor)

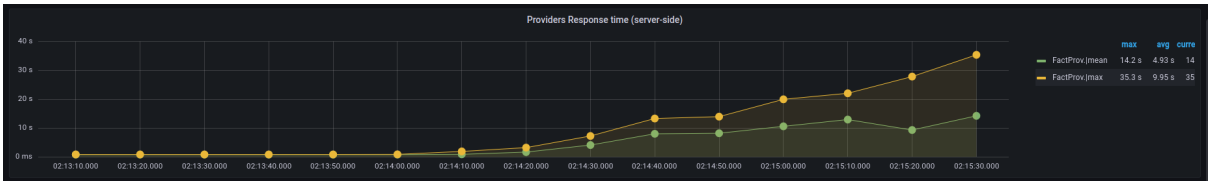


Gráfico 5 - Cache (tiempo de respuesta de la api externa)

De la misma manera, el tiempo de respuesta (gráficos 4, 5 y 6) no se ve afectado usando esta táctica en comparación con el caso base.

Gráfico 6 - Cache (recursos utilizados)

El consumo de recursos también presenta un comportamiento similar al caso base.

Podemos concluir que este estilo no ayuda a mejorar los atributos de calidad antes mencionados, en comparación con el caso base. Incluso se podría decir que agrega complejidad al introducir un componente caché Redis que debe ser tenido en cuenta y mantenido.

Useless Facts - Réplicas

Aquí replicaremos el servidor en tres nodos, en donde Nginx va a funcionar como load balancer delante de ellos.

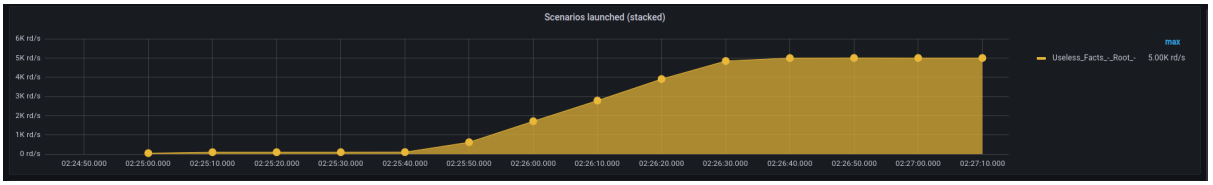


Gráfico 1 - Réplicas (escenarios de Artillery lanzados)

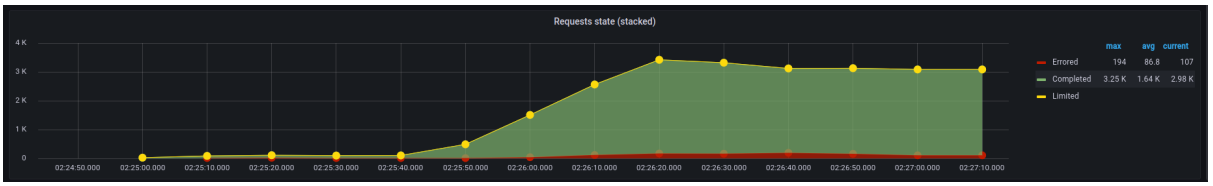


Gráfico 2 - Réplicas (estado de los requests)

Vemos que esta vez la cantidad de requests fallidas disminuye considerablemente, aún cuando la cantidad de requests aumenta.

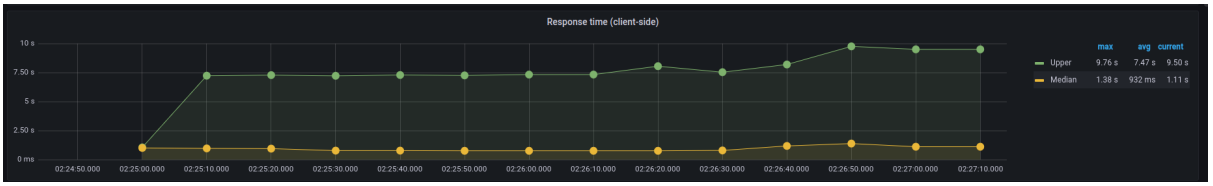


Gráfico 3 - Réplicas (tiempo de respuesta lado cliente)

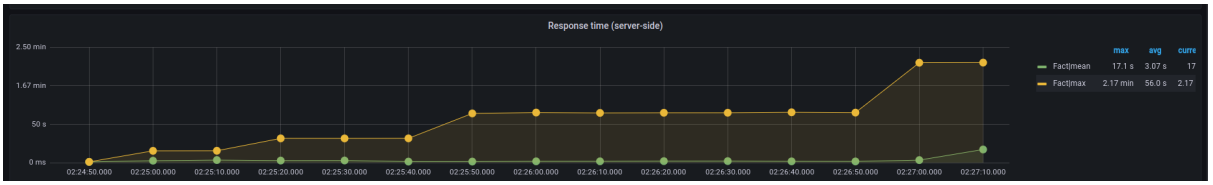


Gráfico 4 - Réplicas (tiempo de respuesta lado servidor)

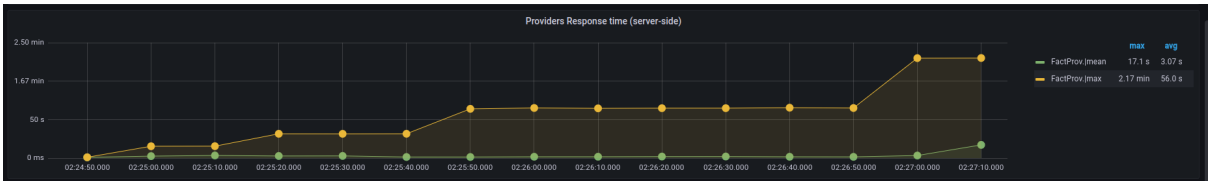


Gráfico 5 - Réplicas (tiempo de respuesta de la API externa)

Se puede notar una mejora en cuanto al tiempo de respuesta percibida por el cliente. En comparación con el caso base, este se mantiene constante por más que aumente la cantidad de requests.

Gráfico 6 - Réplicas (recursos utilizados - nodo 1)

Gráfico 7 - Réplicas (recursos utilizados - nodo 2)

Gráfico 8 - Réplicas (recursos utilizados - nodo 3)

El consumo de recursos se reparte entre los tres nodos como se puede apreciar en los gráficos 6, 7 y 8.

En conclusión, la replicación es una buena idea a tener en cuenta para mejorar la performance de nuestra aplicación.

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 200 requests por segundo. De esta manera, minimizamos la cantidad de requests fallidas y, también, minimizamos la cantidad de requests rechazadas por pasar el rate limit.

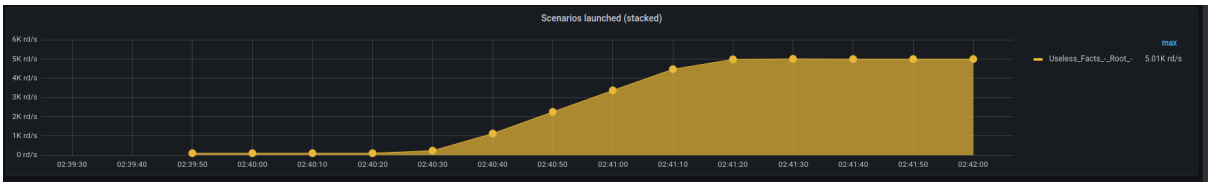


Gráfico 1 - Rate limiting (escenarios de artillery lanzados)

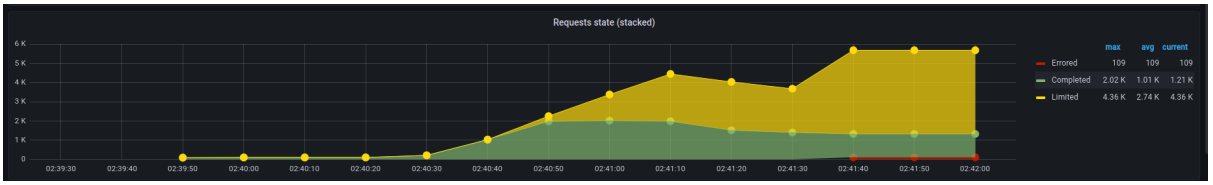


Gráfico 2 - Rate limiting (estado de los requests)

El gráfico 2 muestra como la cantidad de errores es baja, aunque a medida que hay más requests (gráfico 1), también aumenta la cantidad de requests rechazadas por el rate limit.

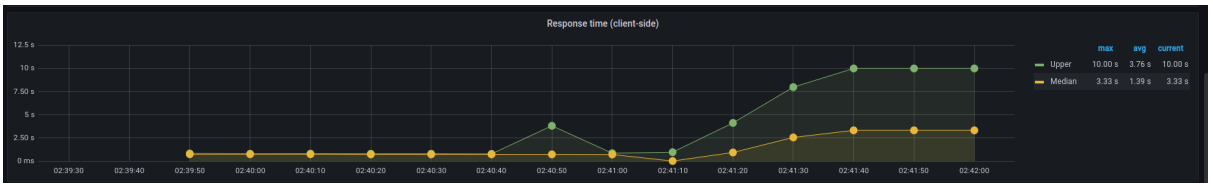


Gráfico 3 - Rate limiting (tiempo de respuesta lado cliente)

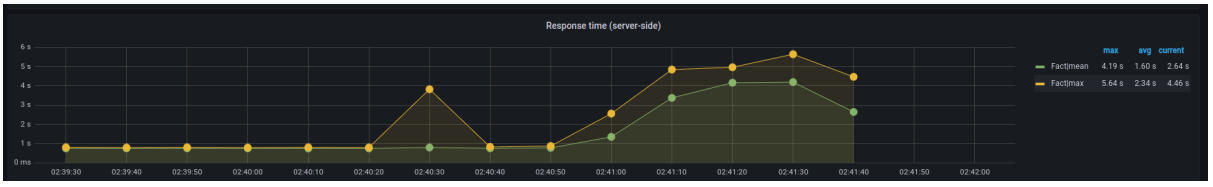


Gráfico 4 - Rate limiting (tiempo de respuesta lado servidor)

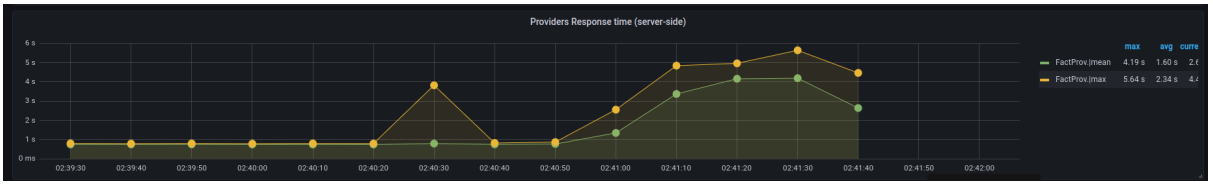


Gráfico 5 - Rate limiting (tiempo de respuesta de la api externa)

El tiempo de respuesta al usar rate limiting es claramente mejor. La mediana de tiempo de respuesta del cliente pasó de 4 segundos en el caso base a 1,4s con rate limiting. Lo mismo en el lado del servidor, donde los tiempos de respuesta pasan de dos dígitos a uno.

Gráfico 6 - Rate limiting (recursos utilizados)

El consumo de recursos es similar al caso base.

Concluyendo, aplicando un rate limit se puede garantizar cierto nivel de disponibilidad cuando el sistema está en riesgo de fallar (por una gran cantidad de requests simultaneas). Además al implementar un rate limit, mejoró sustancialmente el tiempo de respuesta.

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 105 requests.
- **Plain**: 105 requests por segundo durante 45 segundos[4].

Metar - Caso base

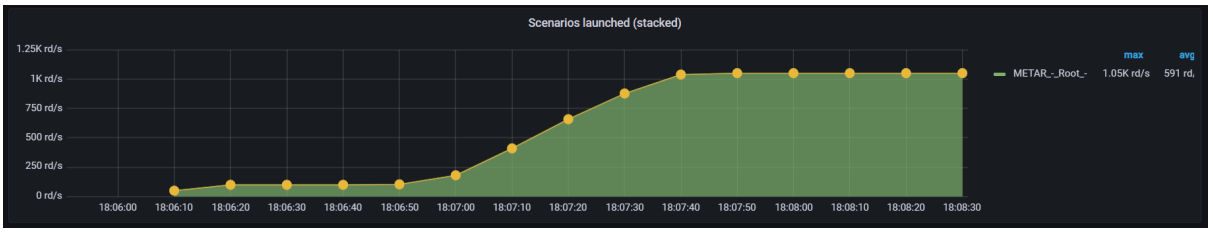


Gráfico 1 - Caso base (escenario)

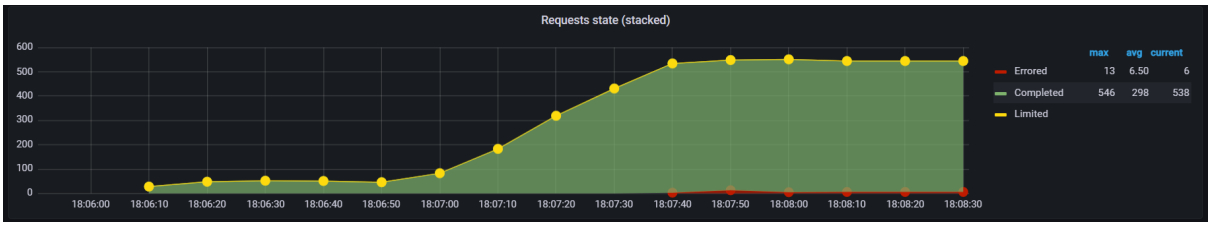


Gráfico 2 - Caso base (estado de las request)

Se puede observar en el gráfico 2 que a partir de la etapa Plain una porción de las request fallan.

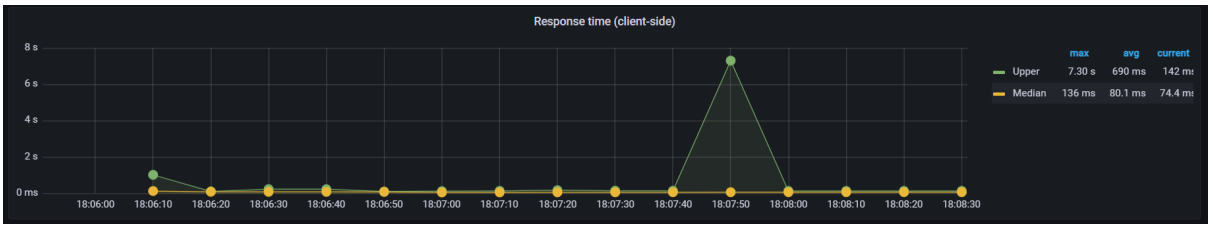


Gráfico 3 - Caso base (tiempo de respuesta del lado cliente)

En el gráfico 3 se puede apreciar que mientras aumenta el número de requests, aparece un pico de tiempo de respuesta. Luego se estabiliza a 80 ms.

[4] La computadora en la que fue testeado no pudo sostener más. En otro contexto se podrían haber generado más carga y obtenido un mayor número de requests fallidas.

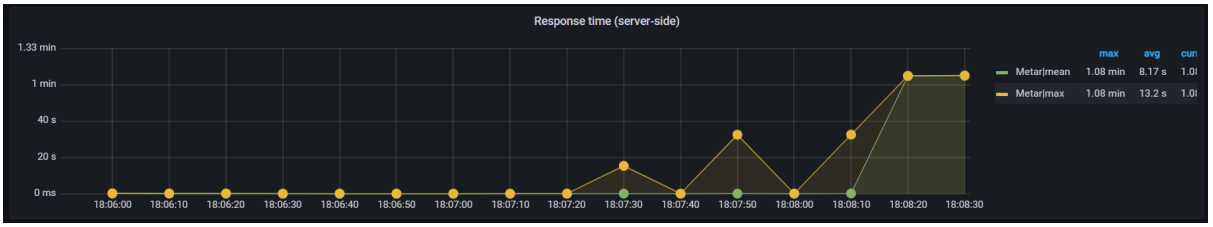


Gráfico 4 - Caso base (tiempo de respuesta del lado servidor)

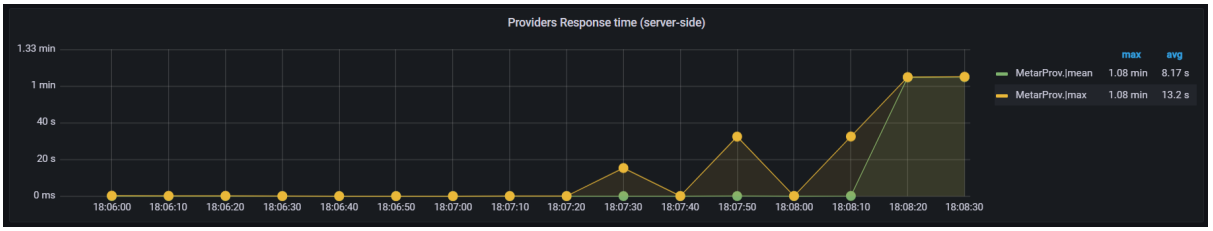


Gráfico 5 - Caso base (tiempo de respuesta de la api externa)

Los tiempos de respuesta del servidor van de la mano con los tiempos que tardan en obtenerse los datos del proveedor. Se observa que finalizando la etapa de Ramp up, los tiempos de respuesta aumentan hasta finalizar las pruebas.

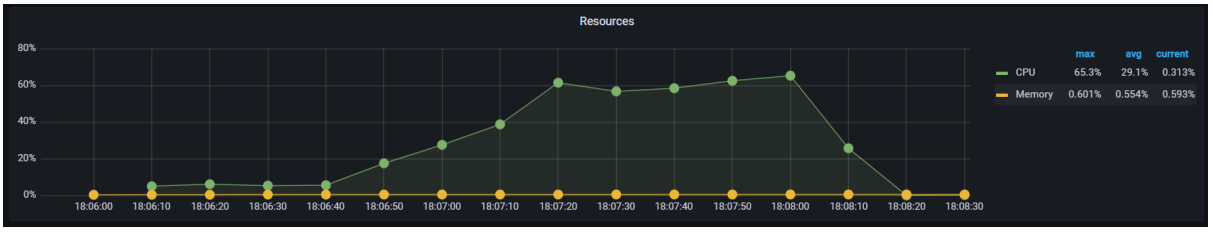


Gráfico 6 - Caso base (recursos utilizados)

El escenario generado llega a consumir un 65% de los recursos.
Además, no se nota un cambio en el consumo de memoria. El cambio de formato de la respuesta al cliente no es una operación intensiva para la memoria.

Metar - Cache

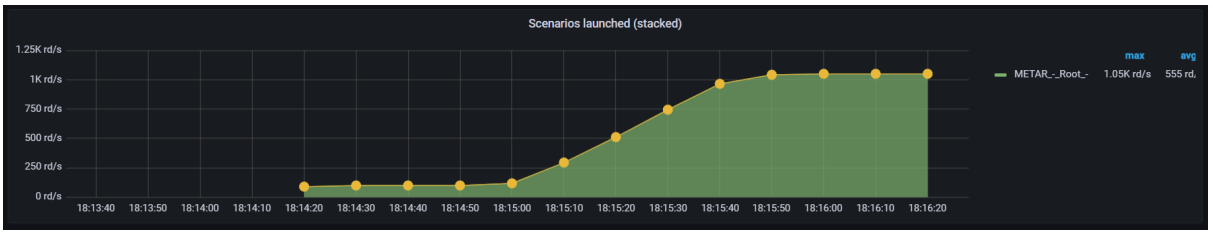


Gráfico 1 - Cache (escenario)

Si bien es posible que en un caso real no nos interese cachear la información de un aeródromo, ya que el uso real podría requerir la información más nueva posible; de todas formas probamos este escenario para ver su impacto en las métricas.
No es posible aplicar *active population*, porque no podemos anticipar la información que va a ser solicitada. Por esta razón aplicamos *lazy population*.

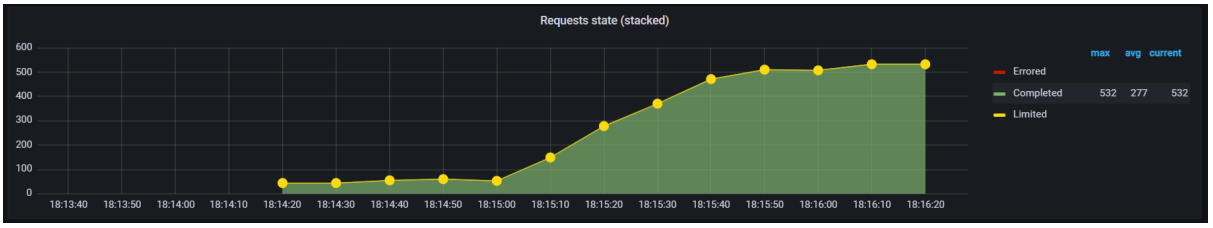


Gráfico 2 - Cache (estado de las requests)

En ese caso, a diferencia del caso base, no hay errores.[5]

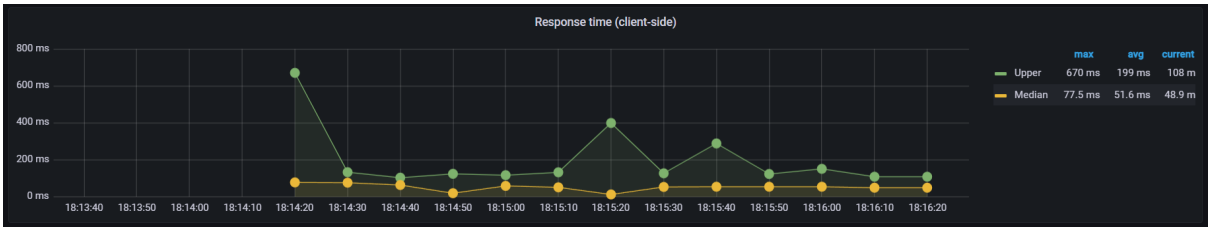
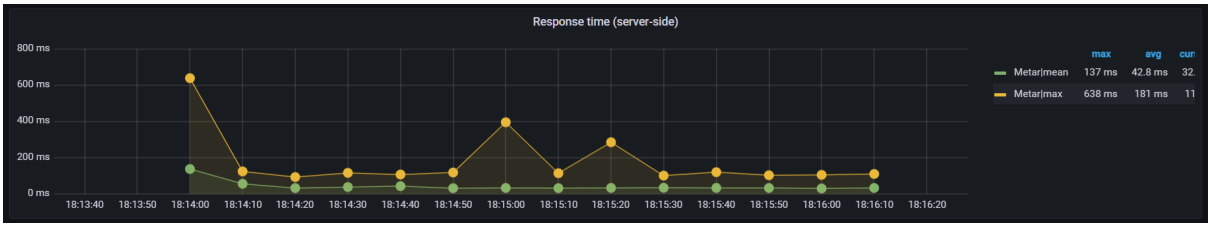


Gráfico 3 - Cache (tiempo de respuesta del lado cliente)

Al igual que en el caso base, hay una subida de tiempo de respuesta durante el Ramp up. Sin embargo, este salto es mucho menor al medido en el caso base.
Además, el tiempo de respuesta promedio de la mediana es 51ms, mientras que en el caso base era de 80 ms.



[5] Si el caso inicial tuviese más requests, es posible que en el caso cache aún apareciesen algunas pocas.

Gráfico 4 - Cache (tiempo de respuesta del lado servidor)

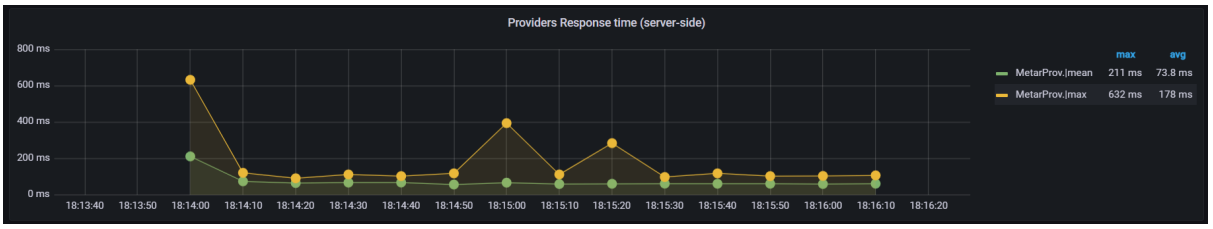


Gráfico 5 - Cache (tiempo de respuesta de la api externa)

El tiempo de respuesta depende del tiempo en que el proveedor responde a nuestras solicitudes. Como se prueba una limitada cantidad de estaciones de aeródromos, se reduce la cantidad de peticiones que haremos a la API externa ya que la información está cacheada. Como la información permanece en nuestro servidor, se reduce la distancia a los datos y podemos seguir respondiendo de manera rápida aunque crezca el número de solicitudes.

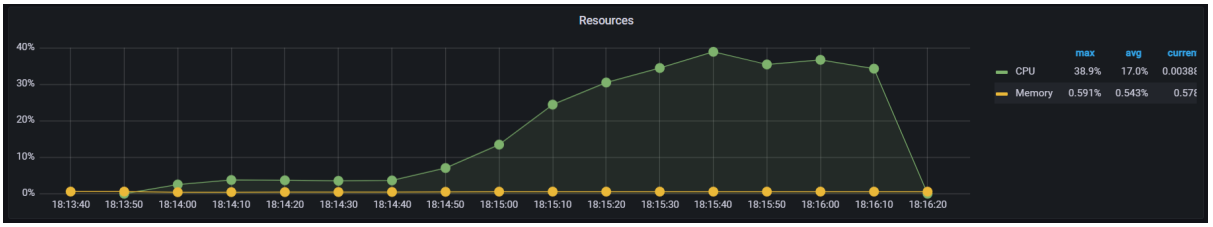


Gráfico 6 - Cache

El consumo de recursos fue del 40%, mientras que en el caso base fue del 65%. Esta baja podría deberse a que no es necesario volver a formatear la información ya que fue cacheada después de ser procesada.
Es importante notar que el consumo de memoria parece ser muy similar, incluso aunque cache dependa del uso de memoria.

En resumen, se notan algunas mejoras al aplicar este estilo, pero no pueden ser concluyentes ya que no pudimos profundizar en los casos en que el servicio falla porque no pudimos obtener mejores mediciones.

Metar - Replication

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

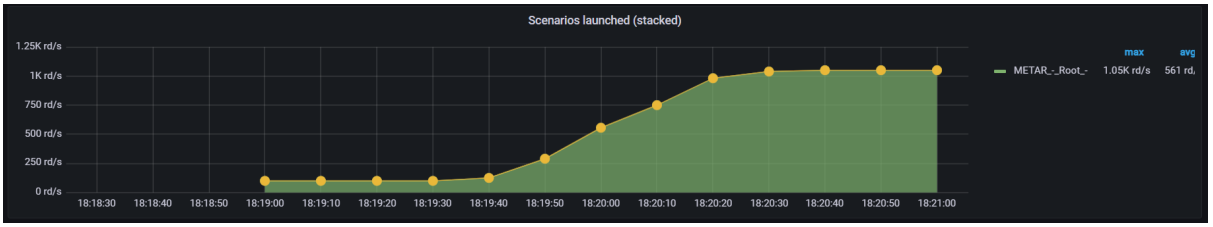


Gráfico 1 - Replication (escenario)

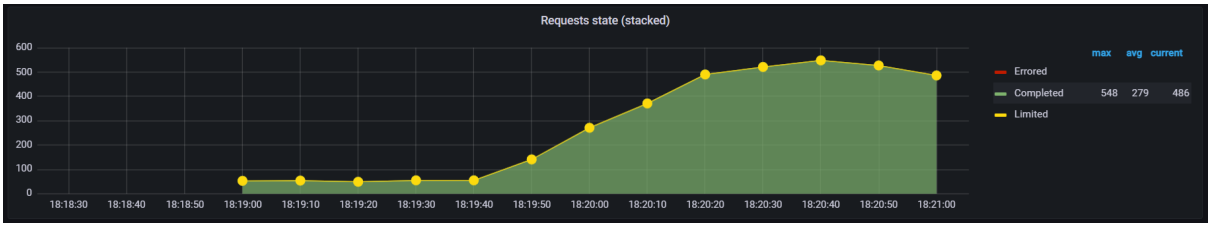


Gráfico 2 - Replication (estado de las requests)

No se observan errores al procesar las solicitudes.

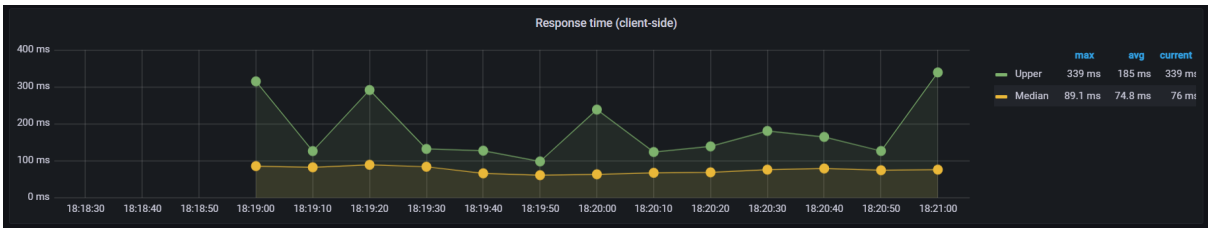


Gráfico 3 - Replication (tiempo de respuesta del lado cliente)

Los tiempos de respuesta del lado cliente se ven más estables que en el caso base donde se había producido un pico.

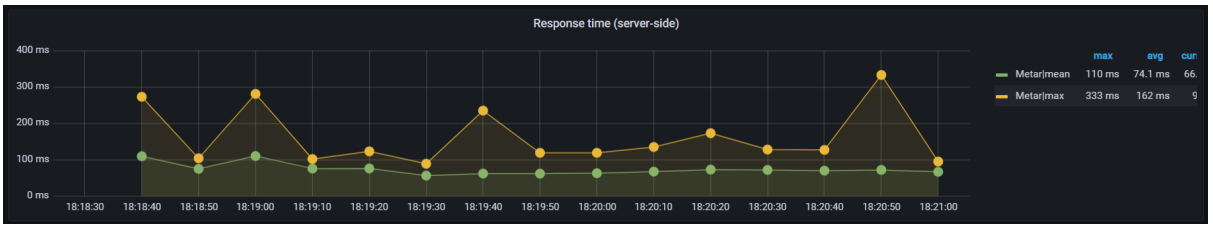


Gráfico 4 - Replication (tiempos de respuesta del lado servidor)

Hay una diferencia notable en los tiempos de respuesta del servidor, ya que por nodo no se supera los 350 ms mientras en el caso base se produjo un aumento significativo hasta superar el minuto.

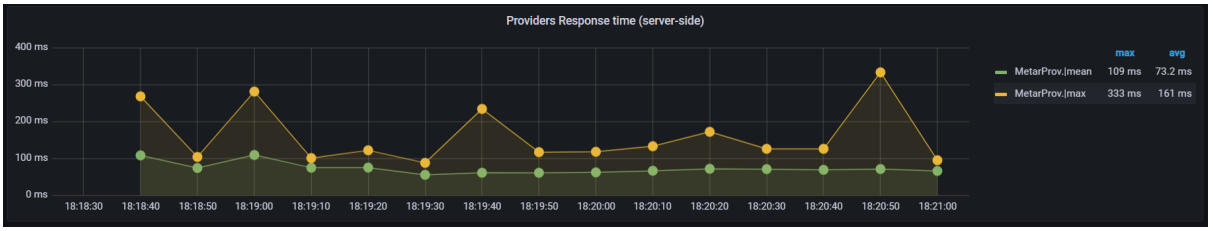


Gráfico 5 - Replication (tiempo de respuesta del lado servidor)

El tiempo de respuesta del servidor sigue acompañando los tiempos del proveedor.

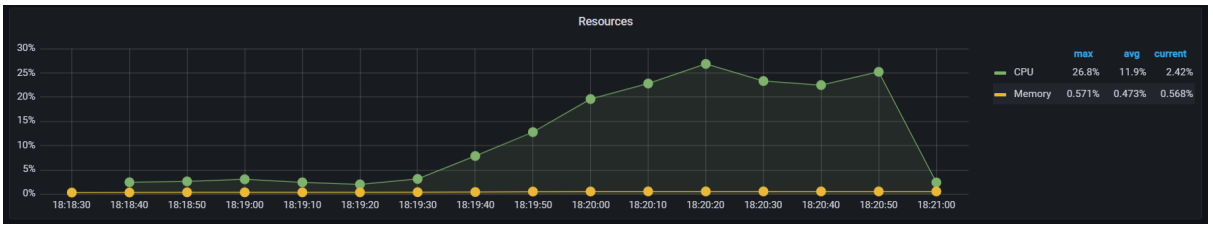


Gráfico 6 - Replication (recursos utilizados por el nodo 1)

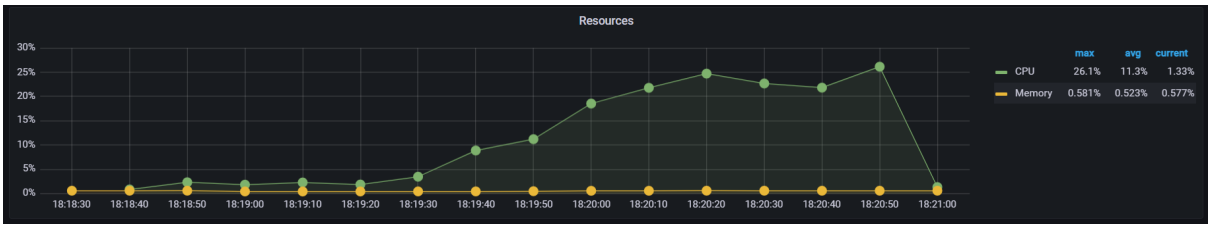


Gráfico 7 - Replication (recursos utilizados por el nodo 2)

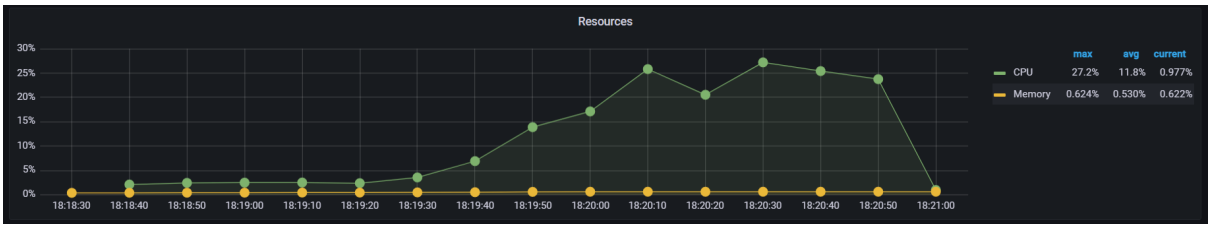


Gráfico 8 - Replication (recursos utilizados por el nodo 3)

Previsiblemente, el consumo por nodo de poder de procesamiento (27% cada uno) fue menor al del caso base (65%) al haberse distribuido la carga entre ellos.

Son notables las mejoras que este estilo logró en los tiempos de respuesta.

Metar - Rate limiting

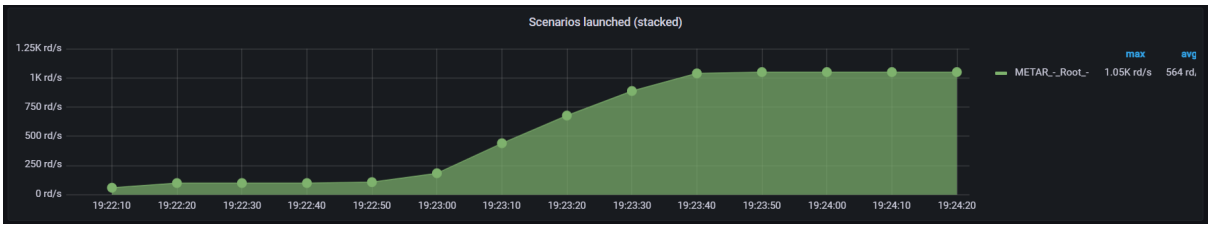


Gráfico 1 - Rate limiting (escenario)

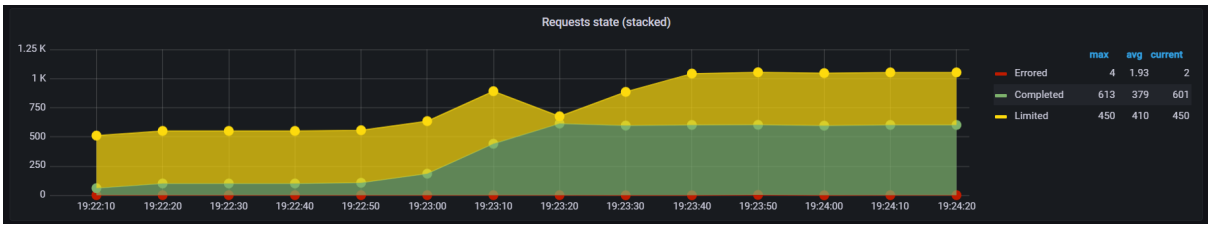


Gráfico 2 - Rate limiting (estado de las requests)

Podemos ver que una porción de las solicitudes fue rechazada, ya que superaban el límite del rate limit.

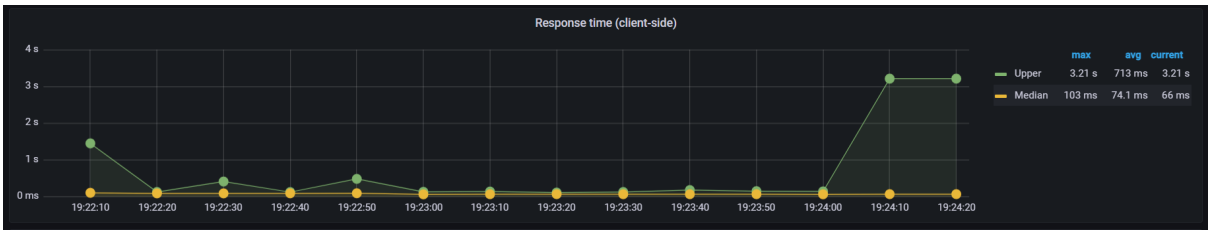


Gráfico 3 - Rate limiting (tiempo de respuesta del lado cliente)

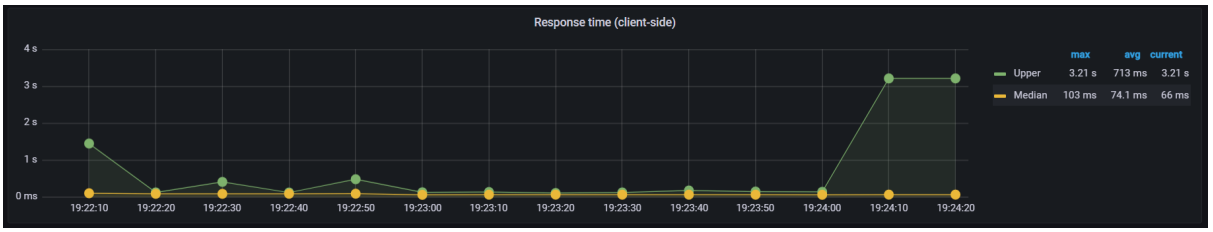


Gráfico 4 - Rate limiting (tiempo de respuesta del lado servidor)

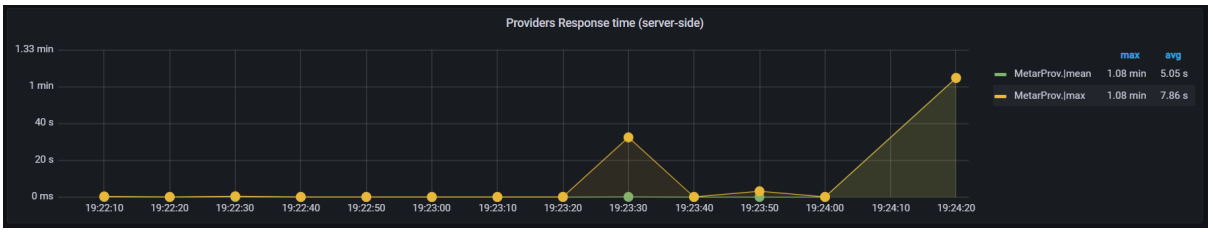


Gráfico 5 - Rate limiting (tiempo de respuesta de la API externa)

Al establecer un límite razonable de requests por unidad de tiempo, las que superan esta cantidad son descartadas con rapidez, mientras las que son aceptadas tienen tiempo de respuesta más rápidos que en el caso base (ya que en este el servidor se veía abrumado).

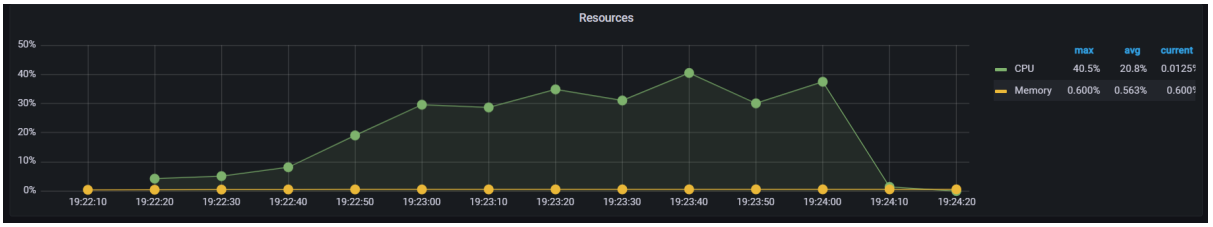


Gráfico 6 - Rate limiting (recursos utilizados)

Al igual que en el estilo cache, el consumo de procesador disminuye de 60% al 40% y el uso de memoria es casi despreciable.

Para finalizar, concluimos que aplicar este estilo permite mejorar la performance y el consumo de recursos a cambio de rechazar una porción de las solicitudes.

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. Veremos cómo impactan los 4 estilos arquitectónicos en este caso:

Space news - Caso base

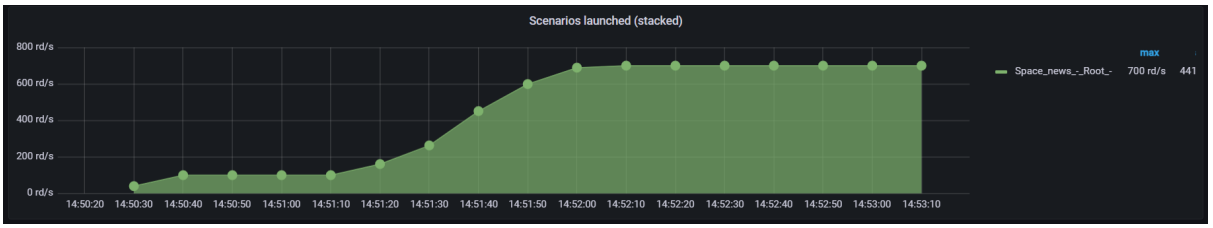


Gráfico 1 - Caso base (escenario)

Para la API de Space News también creamos un escenario de carga con un warm up ligero, un ramp up hasta que comiencen a saltar errores y un plain para ver cómo se estabiliza (véase gráfico 1).

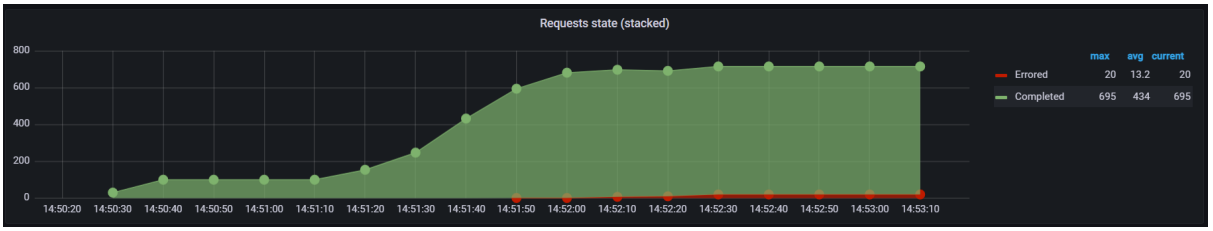


Gráfico 2 - Caso base (estado de los request)

A partir de cierta intensidad durante el ramp up, comenzaron a aparecer errores.

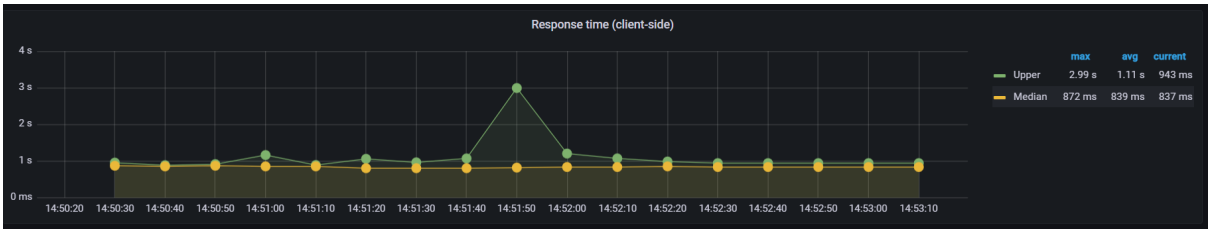


Gráfico 3 - Caso base (tiempo de respuesta del lado cliente)

Mientras aparecían estos errores del ramp up algunas requests tuvieron tiempos de respuesta largos (hasta 3 segundos) como se puede apreciar en el gráfico 3.

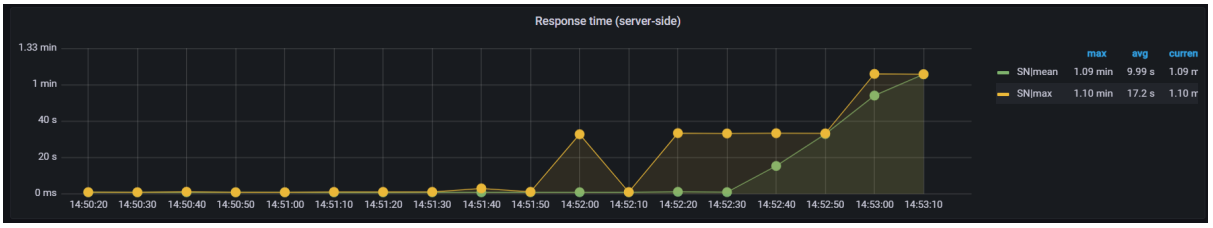


Gráfico 4 - Caso base (tiempo de respuesta del lado servidor)

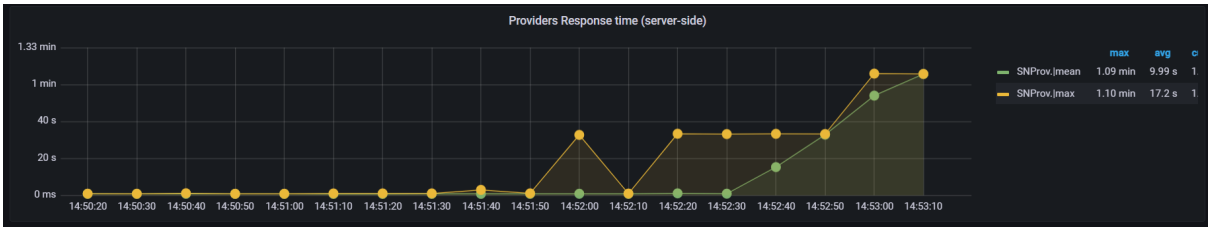


Gráfico 5 - Caso base (tiempo de respuesta de la API externa)

Es importante notar en los gráficos 5 y 6 cómo ciertas requests rezagadas son respondidas recién unos segundos después de terminado el escenario.

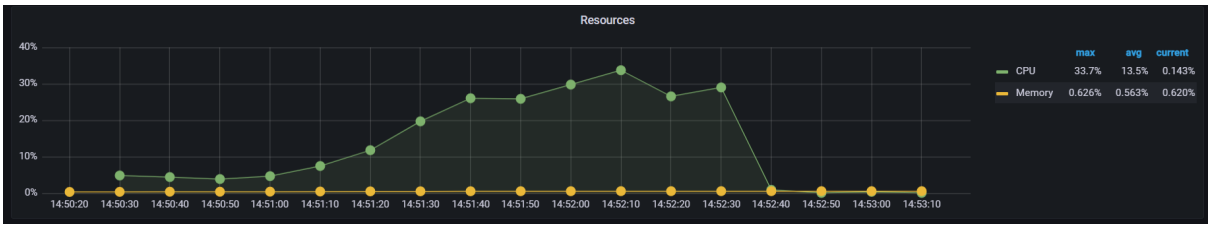


Gráfico 6 - Caso base (recursos utilizados)

En cuanto al consumo de recursos, se requiere más poder de procesamiento mientras más requests hay. Deja de haber consumo una vez termina el escenario. Como en las APIs anteriores, el consumo de memoria es bajo y varía mínimamente.

Estas mediciones del caso base serán útiles para ver qué impacto tienen el resto de estilos en este mismo escenario.

Space news - Cache

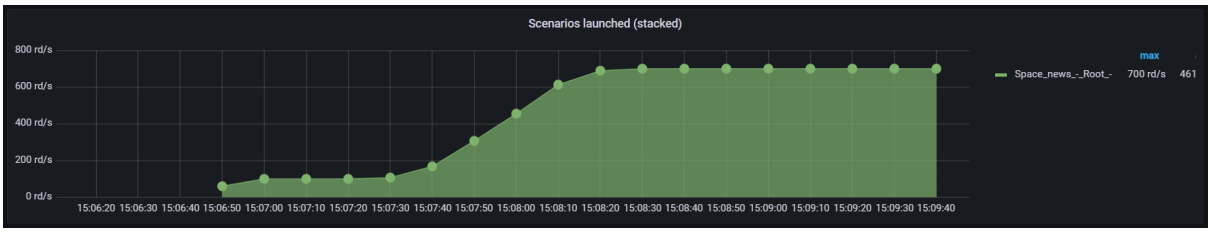


Gráfico 1 - Cache (escenario)

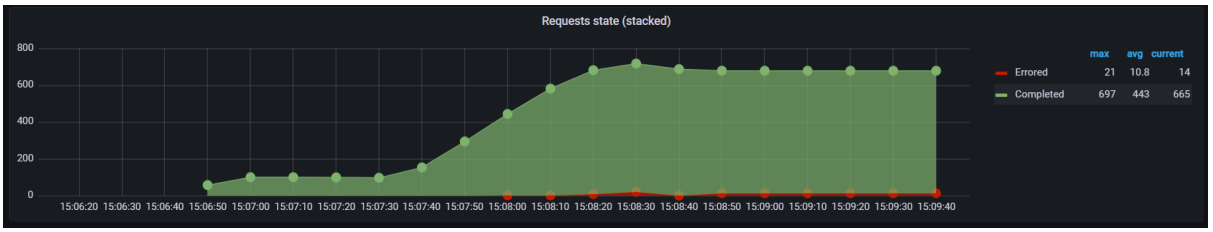


Gráfico 2 - Cache (estado de los request)

Al utilizar cache, no se notan cambios en el número de errores.

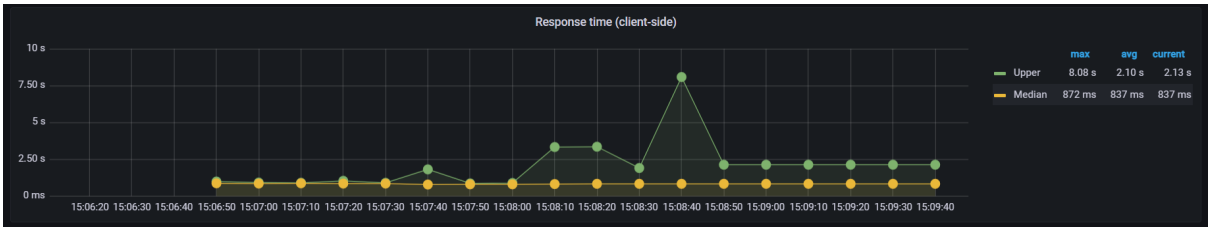


Gráfico 3 - Cache (tiempo de respuesta del lado cliente)

En cuanto al tiempo de respuesta del lado del cliente, la respuesta mediana tarda lo mismo, pero algunas requests en particular tienen duraciones mucho más largas (hasta 8 segundos - gráfico 3 de cache) que en el caso base (donde la peor fue de 3 segundos - gráfico 3 del caso base).

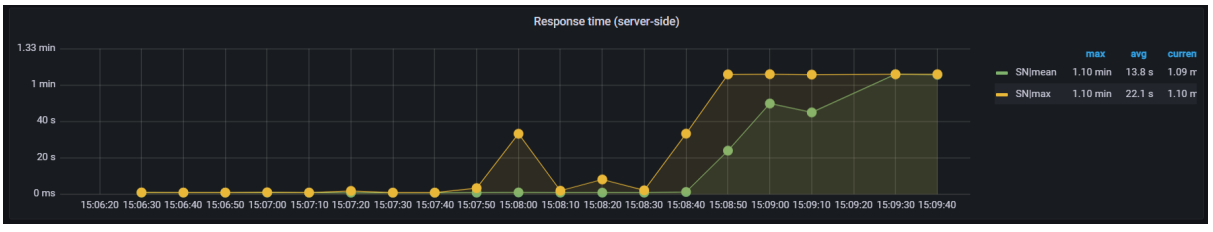


Gráfico 4 - Cache (tiempo de respuesta del lado servidor)

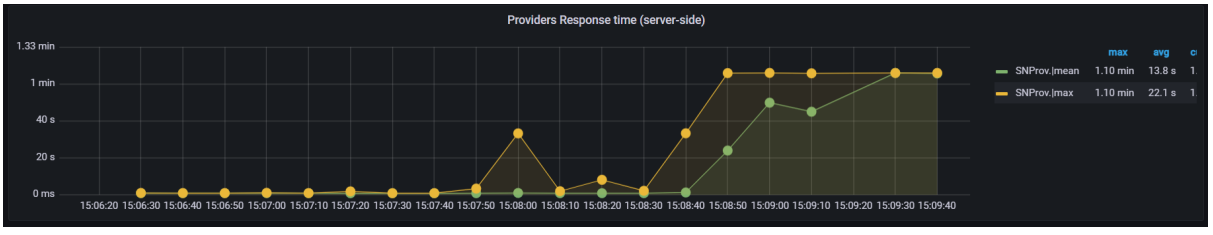


Gráfico 5 - Cache (tiempo de respuesta de la API externa)

El server response time también fue peor en este caso. El tiempo medio de respuesta con cache fue de 14 segundos, mientras que en el caso base fue de 10 segundos.

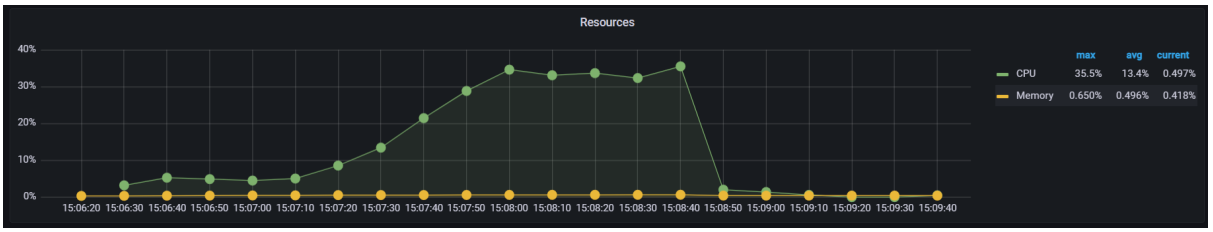


Gráfico 6 - Cache (recursos utilizados)

El consumo de poder de procesamiento es muy similar al caso base. La implementación de un cache no marcó diferencias en este aspecto.

El cache no parece haber traído ninguna ventaja en este caso.

Space news - Réplicas

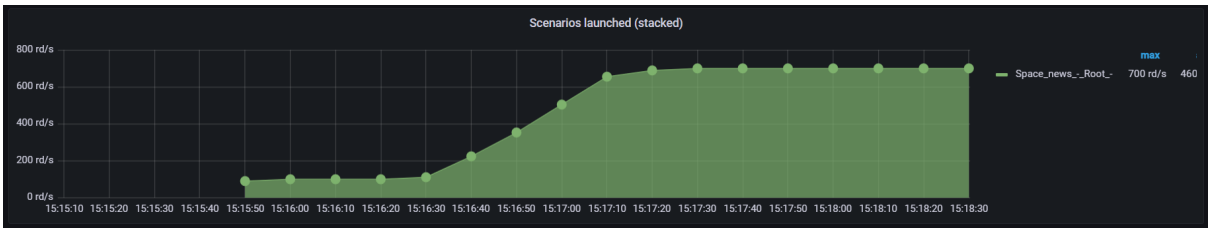


Gráfico 1 - Réplicas (escenario)

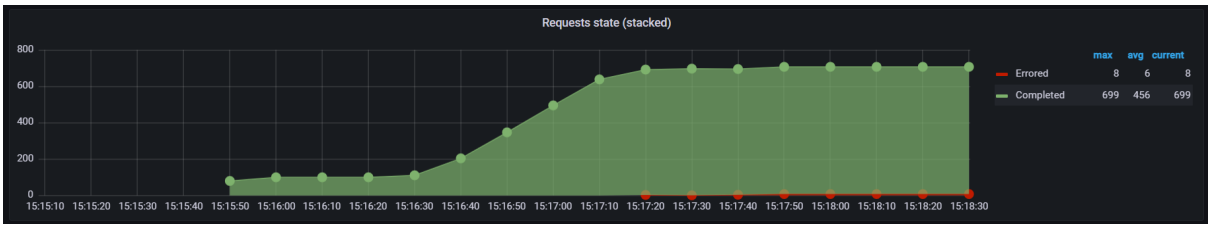


Gráfico 2 - Réplicas (estado de los request)

Implementar réplicas fue muy efectivo para disminuir el número de errores. Pasamos de un máximo de 20 errores, en el caso base; a solo 8 errores.

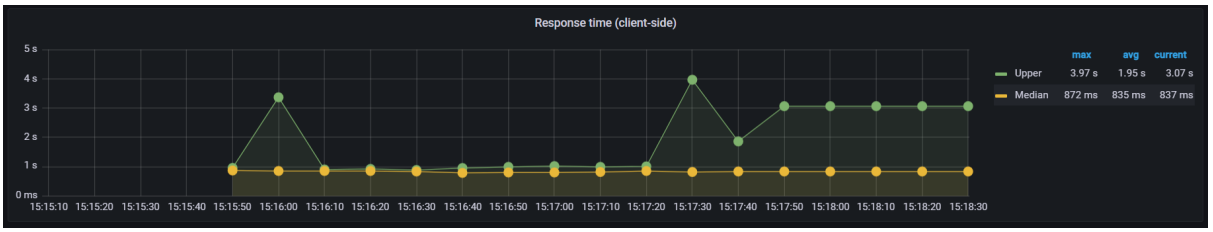


Gráfico 3 - Réplicas (tiempo de respuesta del lado cliente)

Al igual que en todos los casos anteriores, la mediana de tiempo de respuesta fue siempre el mismo, pero los picos máximos de respuesta siguen siendo peores (4 segundos) que el caso base de 3 segundos (pero mucho mejores que el máximo de 8 segundos en cache).

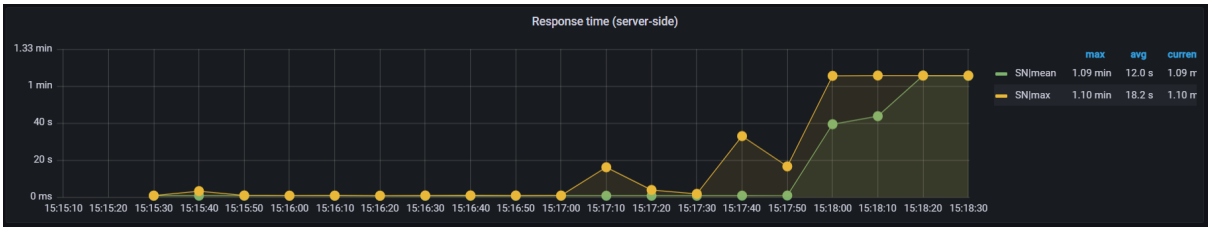


Gráfico 4 - Réplicas (tiempo de respuesta del lado servidor)

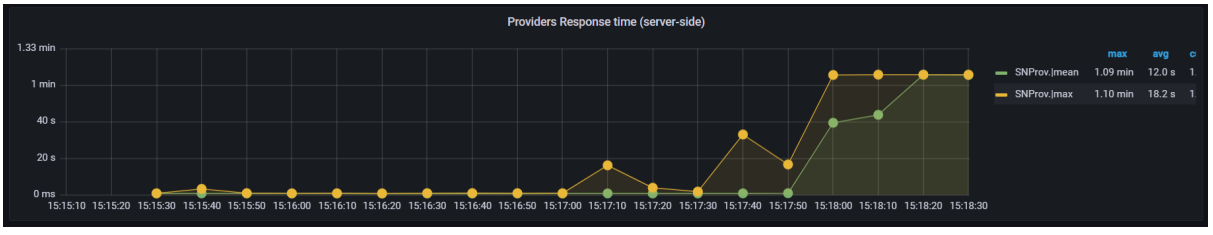


Gráfico 5 - Réplicas (tiempo de respuesta de la API externa)

El tiempo de respuesta del lado del servidor (gráficos 4 y 5) sigue siendo peor que en el caso base.

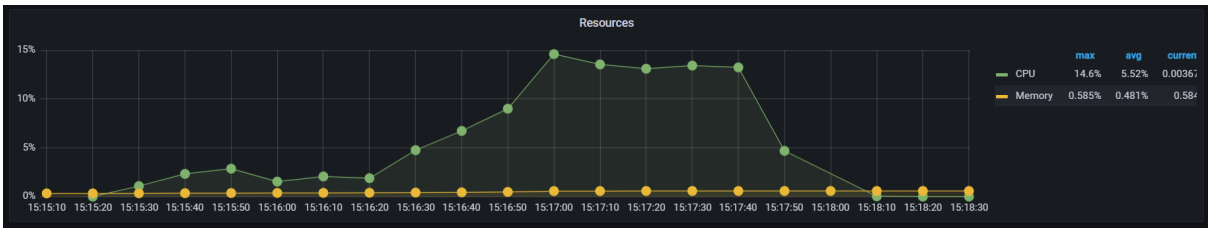


Gráfico 6 - Réplicas (nodo 1 - recursos utilizados)

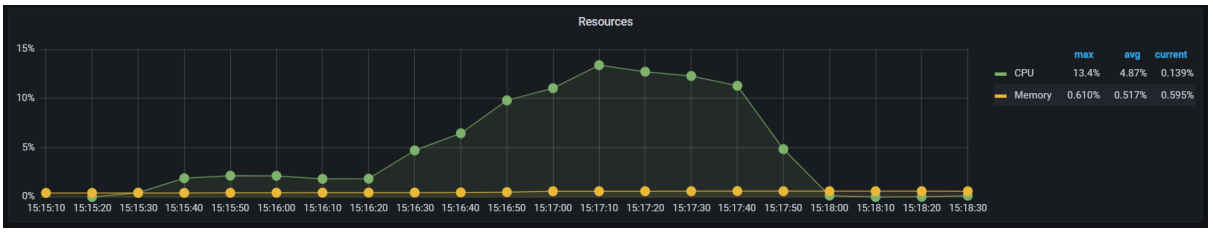


Gráfico 7 - Réplicas (nodo 2- recursos utilizados)

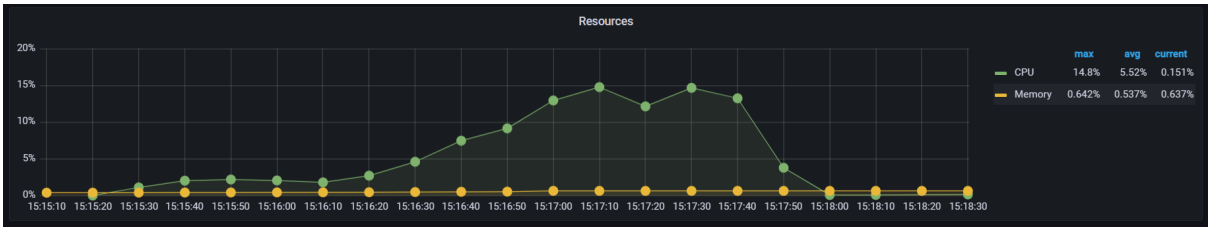


Gráfico 8 - Réplicas (nodo 3 - recursos utilizados)

En los gráficos 6, 7 y 8 se puede notar la distribución de carga. Cada gráfico corresponde al consumo de recursos de cada nodo.
Al usar replicación, la carga de cada nodo es menor. Cada uno tiene entre 13% y 14%, mientras que cuando se usa uno solo llegaba hasta 33%.

La replicación trajo buenos resultados en general, con el costo de requerir más nodos.

Space news - Rate limiting

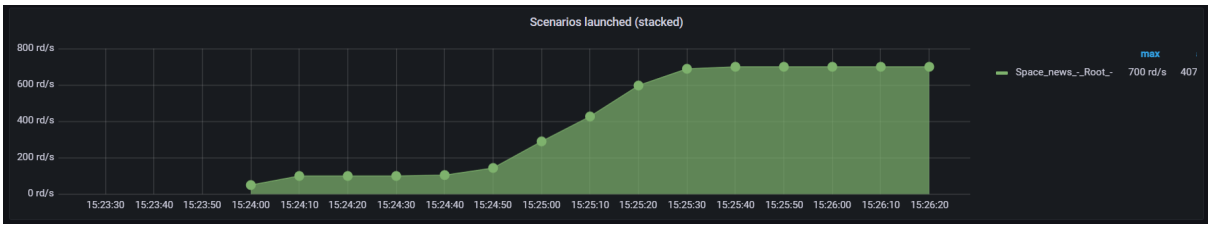


Gráfico 1 - Rate limiting (escenario)

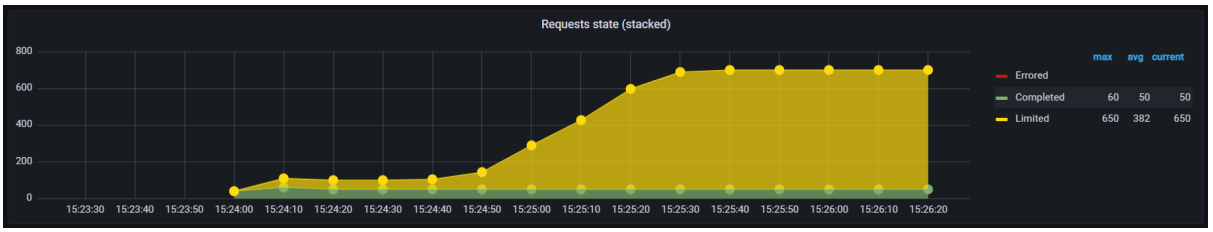


Gráfico 2 - Rate limiting (estado de los requests)

En el gráfico 2 de estado de requests se puede observar cómo cada vez más requests son limitadas (la porción en amarillo del gráfico 2). La mayoría de estas no se ejecutan ya que se supera el límite.

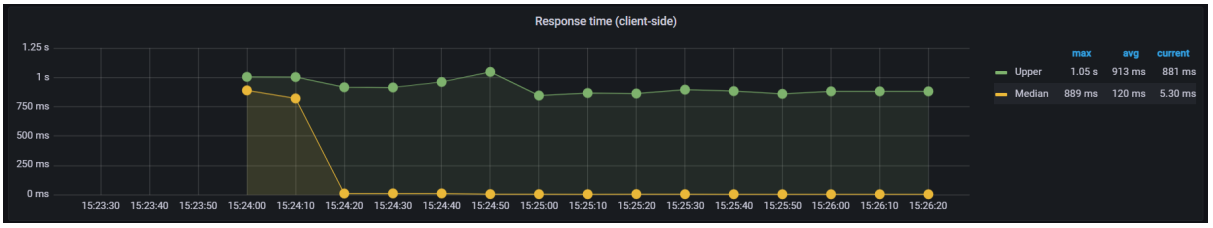


Gráfico 3 - Rate limiting (tiempo de respuesta del lado cliente)

Si bien la mayoría de las requests son limitadas, aquellas que se ejecutan tienen un rendimiento muy superior al resto de los casos, rondando los 5ms.
Sin embargo, aún persisten algunas con tiempos de respuesta similares a casos anteriores (véase el upper promedio de 913 ms).

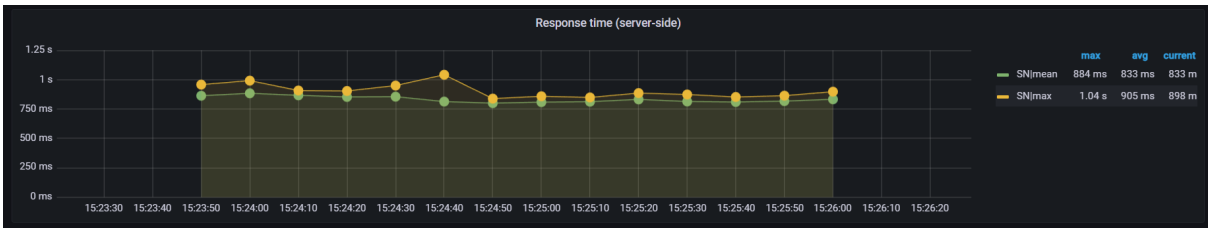


Gráfico 4 - Rate limiting (tiempo de respuesta del lado servidor)

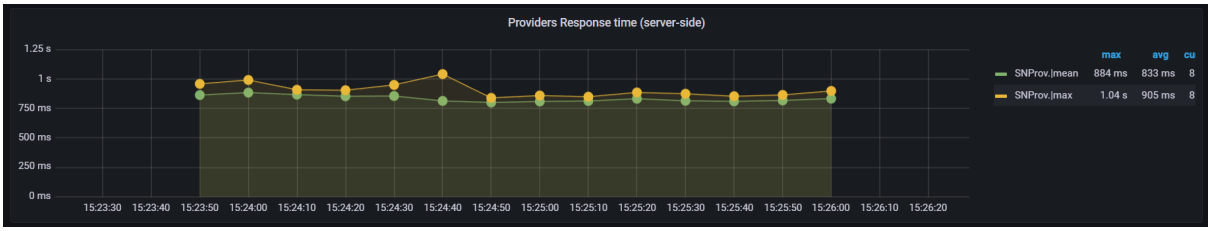


Gráfico 5 - Rate limiting (tiempo de respuesta de la API externa)

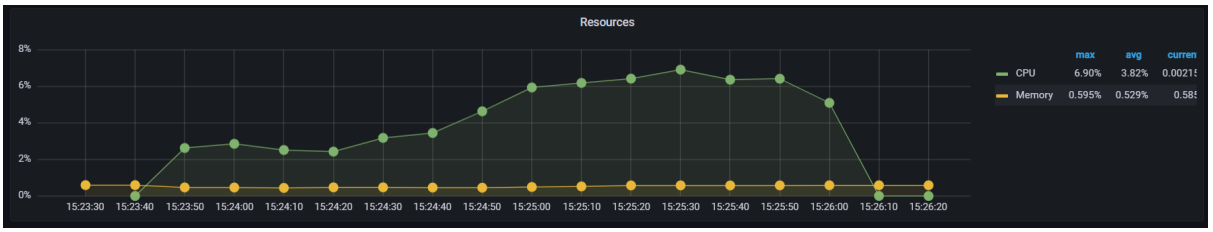


Gráfico 6 - Rate limiting (recursos utilizados)

El consumo de recursos también es mucho menor. Con un máximo de 7%, se utiliza mucho menos poder de procesamiento que el caso base (que tenía 33%).

En este apartado de Space News, rate limiting fue muy efectivo para mejorar los tiempos de respuesta y disminuir la demanda de recursos. Sin embargo, excluyó a una porción de los clientes. Debería elegirse este estilo solo cuando se esté dispuesto a rechazar muchas requests a cambio de tener un bajo tiempo de respuesta para las restantes.

Conclusiones

Estas mediciones nos permitieron observar el impacto real de todos los estilos arquitectónicos que veníamos estudiando. Recorramos una por una:

Primero, el estilo **Client-Server** nos permite extender el sistema agregando nuevos servicios con facilidad. No fue necesario popular una base de datos con *useless facts*, tampoco tuvimos que scrapear una web para obtener los títulos de noticias aeroespaciales, y no necesitamos comunicarnos con diferentes aeródromos para conocer sus condiciones meteorológicas. Estas responsabilidades son delegadas a servidores externos que nuestro servidor, como cliente, puede consumir. Lamentablemente, esto también causa una dependencia a estos servicios por lo que la disponibilidad y performance depende del estado de ellos.

En los casos que probamos en este trabajo práctico, el estilo **cache** no fue muy efectivo. En la mayoría de los casos no trajo ventajas sino que solo sumó complejidad. Es posible que en otros casos, con otras APIs (distintas a las elegidas en este TP) sea de mayor utilidad. Creemos que esto se debe a que los tests los hicimos en nuestras computadoras que no están preparadas para ser usadas como servidores, por lo que el sistema se ve comprometido ante menores números de requests. Esto mismo podemos observar cuando comparamos los resultados que tuvimos cada uno de los integrantes. En cambio, si pudiéramos notar que los proveedores están limitando la comunicación con nuestro servidor o vemos que su disponibilidad o performance se ven comprometidas, podría sernos de utilidad este estilo.

En cuanto a **replication**, creemos que fue el estilo más efectivo para mejorar los atributos de disponibilidad, rendimiento y escalabilidad. Repartir las requests entre varios nodos permitió tener menos errores y mejores tiempos de respuesta, en comparación con el caso base de un simple client-server.

Por último veamos **rate limiting**: Este estilo impacta la disponibilidad de una porción de los clientes (ya que los limitados no tendrán acceso al servicio) pero demuestra claras ventajas en el atributo rendimiento. Al utilizar rate limiting, los tiempos de respuesta fueron claramente menores. Además, asegura que el servicio no se caiga por completo, ya que evita la saturación de nuestros recursos y replicación de fallas.

Referencias

- https://lucid.app/
- https://spaceflightnewsapi.net
- https://www.aviationweather.gov/dataserver
- https://uselessfacts.jsph.pl
- https://www.softwaretestingclass.com/what-is-performance-testing/