



UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

75.73 - ARQUITECTURA DEL SOFTWARE

Segundo cuatrimestre 2023

Trabajo Práctico N°1

Grupo 'Arquitortugas del Software'

Nombre	Padrón	Mail
Barreneche, Franco	102205	fbarreneche@fi.uba.ar
Bauni, Chiara	102981	cbauni@fi.uba.ar
Carol Lugones, Ignacio	100073	icarol@fi.uba.ar
Leloutre, Daniela	96783	dleloutre@fi.uba.ar

Índice

1. Introducción	2
2. Definición de tácticas	4
2.1. Caso Base	4
2.2. Caché	4
2.3. Replicación	5
2.4. Rate Limiting	6
3. Stress Testing	8
3.1. Ping	8
3.1.1. Caso base	8
3.2. Random Quote	9
3.2.1. Caso base	9
3.2.2. Rate limiting	11
3.2.3. Replicación	12
3.3. Metar	14
3.3.1. Caso base	14
3.3.2. Caché	15
3.3.3. Rate limiting	17
3.3.4. Replicación	19
3.4. Spaceflight News	20
3.4.1. Caso base	20
3.4.2. Caché	22
3.4.3. Rate limiting	23
3.4.4. Replicación	25
4. Endurance Testing	27
4.1. Ping	27
4.1.1. Caso base	27
4.2. Random Quote	28
4.2.1. Caso base	28
4.2.2. Rate limiting	29
4.2.3. Replicación	31
4.3. Metar	32
4.3.1. Caso base	32
4.3.2. Caché	33
4.3.3. Rate limiting	35
4.3.4. Replicación	36
4.4. Spaceflight News	38
4.4.1. Caso base	38
4.4.2. Caché	39
4.4.3. Rate limiting	41
4.4.4. Replicación	42
5. Conclusión	44

1. Introducción

El presente informe tiene como objetivo analizar cómo se pueden ver afectados los atributos de calidad de una aplicación ante variados escenarios. Asimismo, se analizan distintas tácticas para mejorar dichos atributos.

Para el desarrollo del trabajo se utilizan las siguientes tecnologías:

- Node.js + Express
- Docker + Docker compose
- Artillery
- Nginx
- Redis
- Grafana + Graphite + cAdvisor

El trabajo consiste en implementar una API que realice llamados a distintas APIs externas y luego tomar métricas del uso de recursos y tiempos de respuesta. Dicha API cuenta con los siguientes endpoints:

- /ping: Devuelve un valor constante "pong"
- /metar: Devuelve un reporte del estado metereológico de un determinado aeródromo
- /spaceflight_news: Devuelve el título de las últimas 5 noticias sobre actividad espacial
- /quote: Devuelve una cita famosa y su autor, al azar

Para realizar las métricas se proponen dos escenarios de carga:

- *Endurance testing*: es un tipo de load testing donde se evalúa si la aplicación es capaz de soportar una determinada cantidad de solicitudes durante un período dado de tiempo.
- *Stress testing*: el propósito es identificar el punto de interrupción de la aplicación, para ello se incrementa significativa y continuamente el número de usuarios durante un determinado período de tiempo.

Para cada tipo de carga se analiza el comportamiento de la API en su **caso base** y luego aplicando distintas tácticas que puedan mejorarlo según las particularidades de cada endpoint.

Las métricas se analizan a partir de los gráficos generados en Grafana, en particular:

- *Scenarios launched*: Cantidad de solicitudes en 10 segundos
- *Requests State*: Cantidad de cada tipo de respuesta de la API (Completed, Errorred o Limited)
- *Response time*: Tiempo de respuesta de la solicitudes, desde la perspectiva del cliente

- *Resources*: Consumo de CPU y memoria RAM
- *All*: Demora de cada endpoint en responder, teniendo en cuenta el llamado a la API externa sumado al procesamiento interno
- *All internal*: Demora de cada API externa en responder

2. Definición de tácticas

2.1. Caso Base

Como primer paso y a fin de tener una base contra la cual comparar las tácticas, evaluamos las distintas cargas para los endpoints originales.

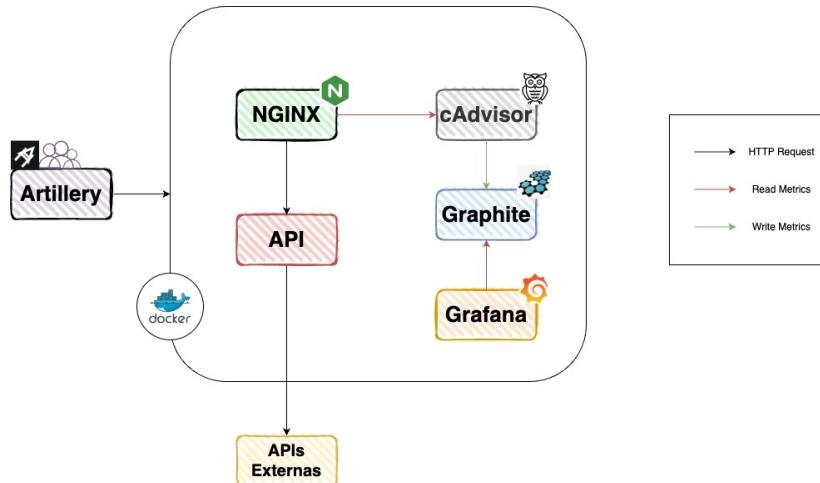


Figura 1: Vista de Componentes y Conectores del caso base

Tal como describe el diagrama de la arquitectura base, con el uso de Artillery se simula el arribo de usuarios que realizarán peticiones HTTP a la aplicación. Las solicitudes las recibe Nginx y las redirige a la API implementada con NodeJS y Express. Por otro lado, la herramienta cAdvisor se encarga de almacenar datos sobre el uso y rendimiento de los contenedores; estos datos se almacenan en Graphite y luego mediante Grafana es posible acceder a una interfaz que permite visualizarlos.

2.2. Caché

Para la implementación de caché utilizamos un contenedor con Redis y el cliente *node-redis* para comunicarnos con él. La motivación principal del caché es mejorar los tiempos de respuesta y la performance para solicitudes frecuentes dentro de un tiempo determinado. Analizamos para cada endpoint la conveniencia y características del uso de la caché:

- /quote: El objetivo del endpoint es brindar una cita al azar cada vez que es consultado. Aunque podría argumentarse a favor de incorporar una caché con tiempos de expiración de entradas muy pequeños con tal de servir la misma cita a clientes que se conectan en tiempos similares, consideramos que esta situación serviría a casos demasiado particulares, mientras que en general incorporar caché interferiría con el propósito de la API.
- /metar: Este servicio recolecta información nueva cada 5 minutos aproximadamente según su documentación. Es razonable incorporar una memoria caché con tiempo de expiración similar por entrada, aunque como la información varía según el código de aeropuerto consultado, almacenamos una entrada por código. Además, al desconocer

cuántos de los códigos de aeropuerto serían realmente consultados y con qué frecuencia, nos pareció adecuado optar por *lazy population*. En vistas de lo anteriormente descrito, una entrada en caché puede ser retornada a varios clientes (que consulten por el mismo código de aeropuerto) hasta su vencimiento.

- /spaceflight_news: Según la FAQ de la API, los artículos se refrescan cada 10 minutos y por la naturaleza de la información no esperaríamos que se consulten con mayor frecuencia. Tal y como lo describimos, el caso de estudio se prestaría para un tipo de caché con tiempo de expiración de alrededor de 10 minutos, lazy population, sin vaciado y en donde se almacenarían los primeros 5 títulos de las noticias cada vez. Sin embargo nos pareció una buena oportunidad para evaluar active population, aprovechando un sistema de eventos/notificaciones de expiración de entradas que Redis provee. El sistema podría realizar una query de refresco de información cada 10 minutos (o menos tiempo si la API demorase mucho en responder) y almacenar el resultado en caché con el fin de que la amplia mayoría de los clientes percibiesen un tiempo de respuesta comparable al insumido en la consulta del caché.

Nota: Para realmente probar la eficacia de la caché, el tiempo de prueba debería contemplar al menos 5 refrescos. Esto insumiría 25 minutos para *metar* y 50 minutos para *spaceflight_news*, dificultando además la interpretación del gráfico resultante. Por ello las pruebas utilizan un tiempo de caché de un minuto.

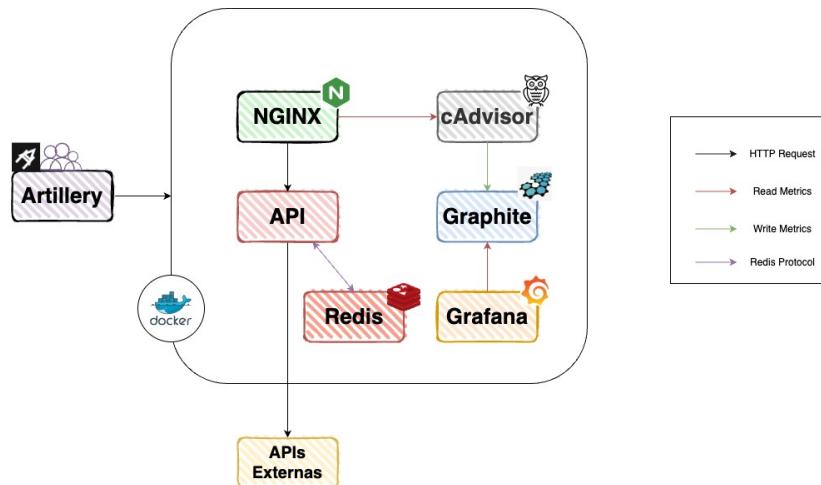


Figura 2: Vista de Componentes y Conectores utilizando caché

2.3. Replicación

Esta táctica consiste en hacer tres copias del servicio implementado en NodeJS, convirtiendo a Nginx en un balanceador de carga. De esta forma, se aumenta la capacidad para procesar solicitudes entrantes.

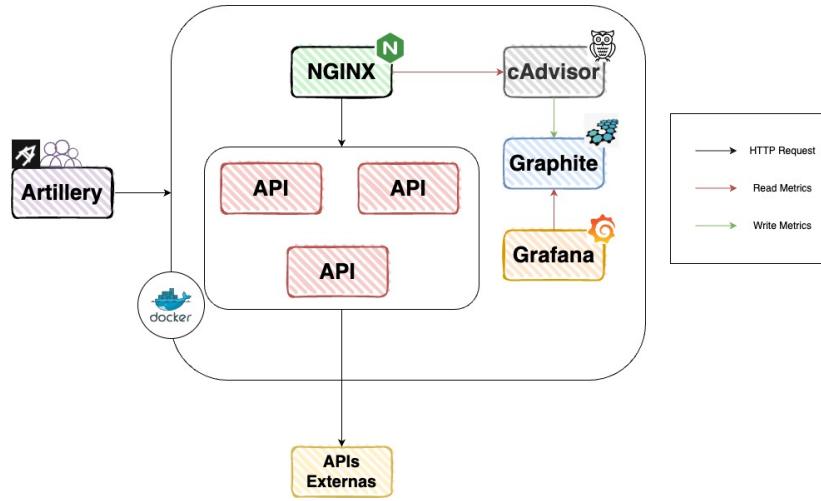


Figura 3: Vista de Componentes y Conectores utilizando replicación

2.4. Rate Limiting

Desde la configuración de Nginx, agregamos una instrucción para limitar el tráfico del servicio. Esto previene que el servicio se suspenda ya sea por alta demanda o por ataques malintencionados.

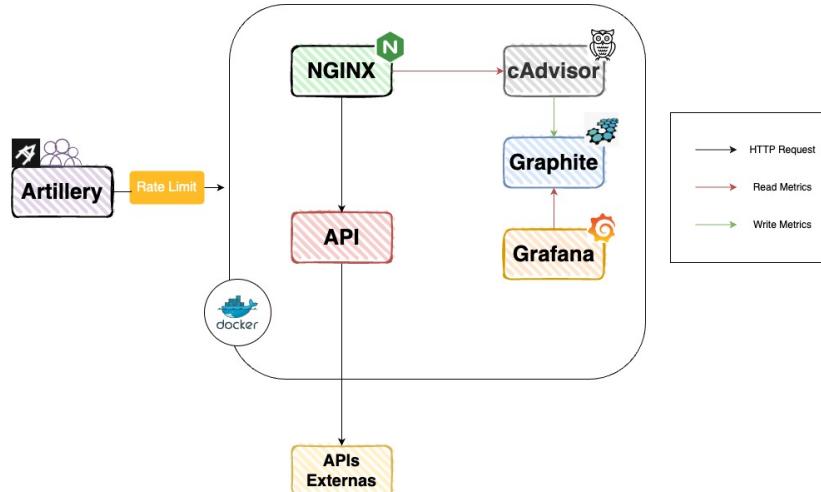


Figura 4: Vista de Componentes y Conectores utilizando rate limiting

Para configurarlo se debe definir la cantidad de solicitudes que se permitirán en una determinada ventana de tiempo. Definimos los siguientes límites:

- /quote: 5 req/s
- /metar: 70 req/s
- /spaceflight_news: 70 req/s

Cuando se exceda el límite, la API devolverá el código de error **429 Too Many Requests**.

3. Stress Testing

3.1. Ping

3.1.1. Caso base

El endpoint de *ping* no depende de una API externa y por ello nos permite observar las características inherentes a nuestro sistema.

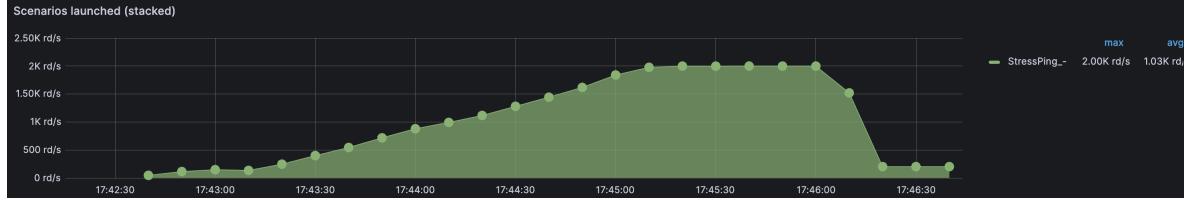


Figura 5: Cantidad de escenarios de carga lanzados

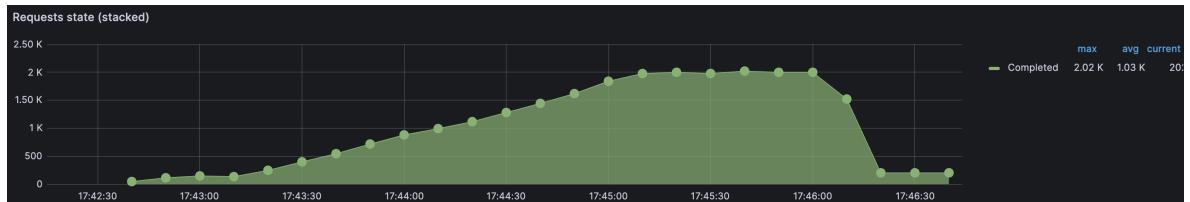


Figura 6: Tipo de respuestas obtenidas

Vemos que el patrón de carga comienza con una función lineal con una pendiente pronunciada hasta llegar al máximo. Este valor se mantiene por un corto periodo para luego caer con una pendiente pronunciada. Este patrón se va a replicar en los casos a analizar.

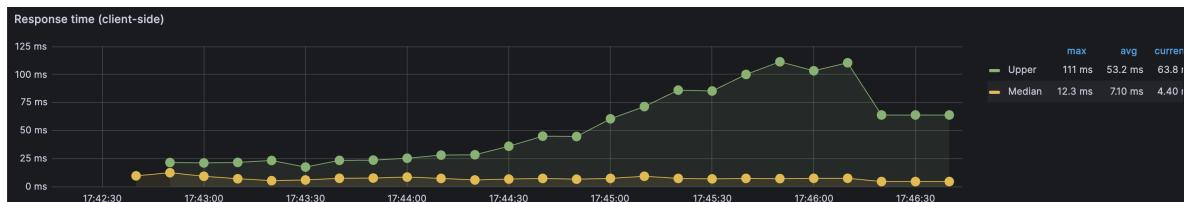


Figura 7: Demora en respuestas

Se ve el mismo patrón en los tiempos de respuestas, lo cual tiene sentido ya que a medida que pasa el tiempo se aumenta la cantidad de requests por segundo hasta alcanzar el máximo. Esto se ve de misma manera reflejado en el uso de recursos aunque el mismo no es significativo.

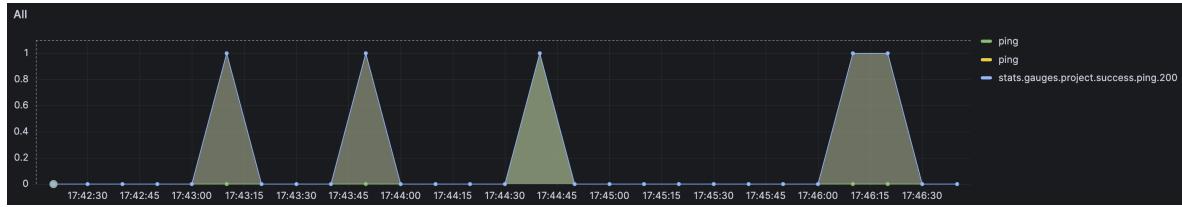


Figura 8: Demora del endpoint

El tiempo de demora del endpoint es bajo, dado que se resuelven en el mismo equipo. Se observan picos ocasionales en la demora que es posible que se deban a la misma carga del sistema ya sea por interferencias de otros procesos o algún otro motivo. Esto puede generar que del lado del cliente se perciban demoras en las respuestas.

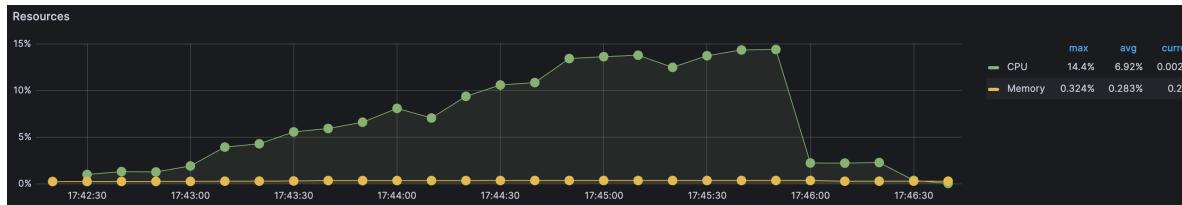


Figura 9: Consumo de recursos

Es curioso que las estadísticas de consumo de recursos que se reportan son muy bajas, lo cual coincide con la sencillez de la consulta, pero veremos que contrastará con la ejecución del test de endurance. Sospechamos que el reporte de uso de recursos puede haberse visto afectado por la carga.

3.2. Random Quote

3.2.1. Caso base

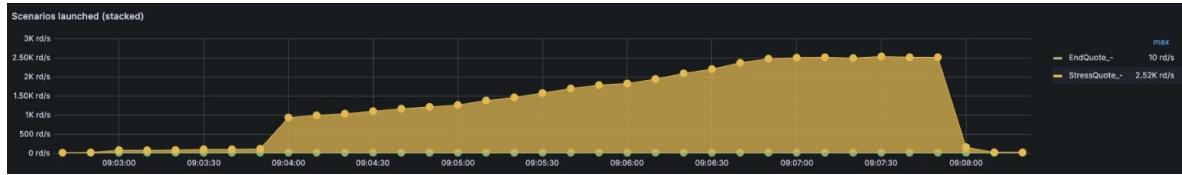


Figura 10: Cantidad de escenarios de carga lanzados

Para la prueba de carga configuramos Artillery de forma tal que la cantidad de request por segundo aumente conforme avance el tiempo. En el gráfico observamos que la cantidad de escenarios lanzados es una función lineal a trozos, en donde las pendientes crecen a medida que incrementa t . También podemos detectar las etapas de Warm Up y Cool Down en los extremos de la figura.



Figura 11: Tipo de respuestas obtenidas

La documentación oficial de la API *quote* indica un rate limit de 180 requests por minuto. Decidimos acercarnos lentamente a dicho valor e incluso excederlo para ponerlo a prueba. La incertidumbre en el tiempo, inherente a la transmisión de mensajes por internet y de los tiempos de procesamiento distorsionan estas pruebas. No obstante, podemos confirmar que la tasa de request permitida corresponde a la informada por dos vías: Primero observamos que se produce un error aislado a los tres minutos, justo al alcanzar las 180 queries. Además, si comparamos el momento en que comienzan a rechazarse las solicitudes en gran cantidad con la cantidad de solicitudes esperadas según la configuración de Artillery para ese instante de tiempo, veremos que coinciden.

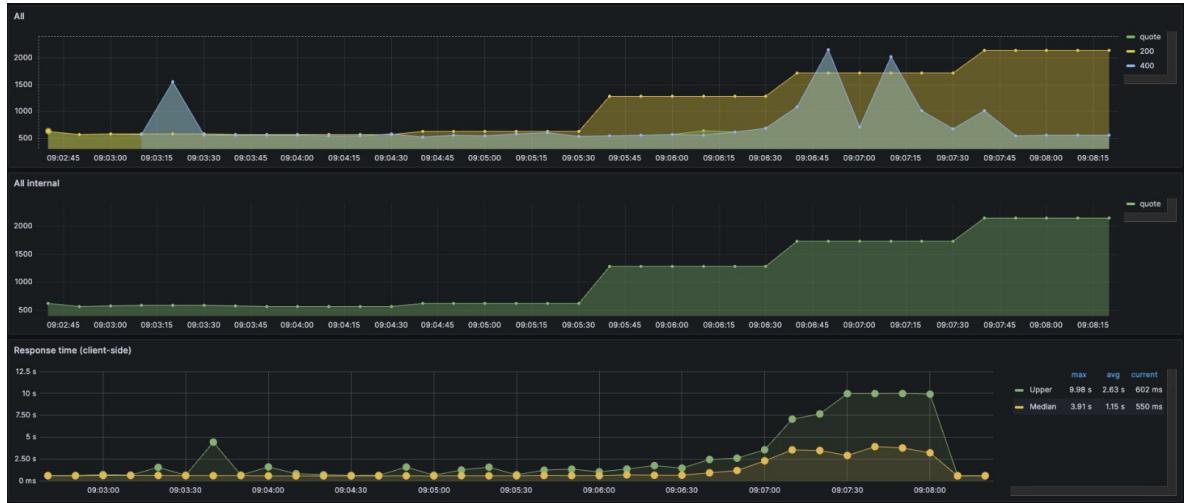


Figura 12: Demora en respuestas

Al observar los tiempos de respuesta encontramos que estos crecen incluso antes de que la API externa comience a rechazar las requests debido a la carga del sistema. Es interesante notar los picos en el tiempo de respuesta de las request con resultados fallidos. Estos denotan periodos en donde la API externa comienza a rechazar pedidos, pero aún se encuentra procesando request anteriores y esto lleva a que incluso las solicitudes rechazadas se demoren. Tras este lapso todas las request se rechazan velozmente, reduciendo el tiempo medio de respuesta.



Figura 13: Consumo de recursos

La evolución en el uso de recursos no presenta sorpresa alguna, debe considerarse que nuestro sistema involucra muy poco procesamiento de las queries y las acciones necesarias, al menos en el caso base, no dependen del estado de la query. Por ello solo podemos observar cómo el uso de recursos se incrementa a medida que lo hacen la cantidad de escenarios lanzados.

3.2.2. Rate limiting

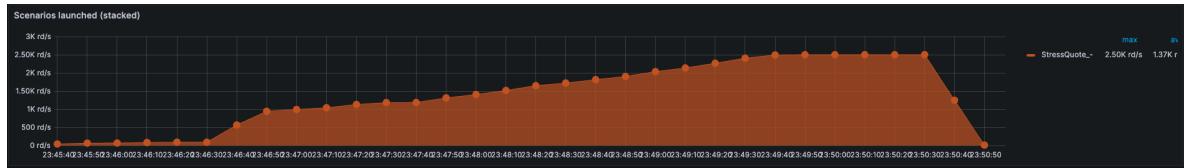


Figura 14: Cantidad de escenarios de carga lanzados

Al igual que el caso anterior, como se ve en la figura 14, aumentamos la carga de solicitudes durante un período de tiempo de aproximadamente 5 minutos.

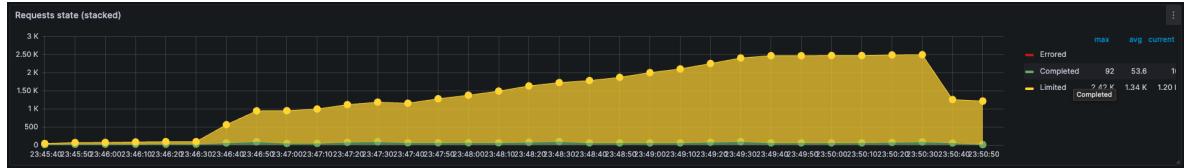


Figura 15: Tipo de respuestas obtenidas

En la figura 15 se observa cómo desde el inicio aumenta continuamente la cantidad de solicitudes limitadas, adoptando una forma similar a la curva descripta en la figura 14; mientras que la curva verde, correspondiente a las solicitudes completadas, se mantiene siempre por debajo del mismo valor, lo cual es consistente con el límite elegido.

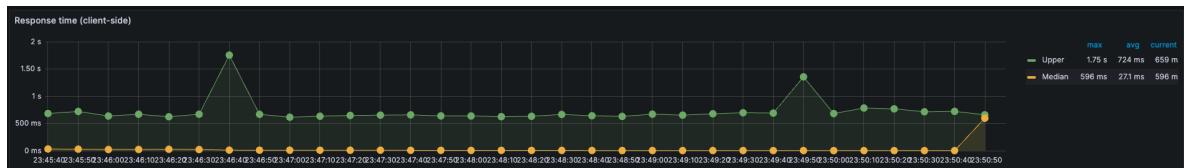


Figura 16: Demora de respuestas (client-side)

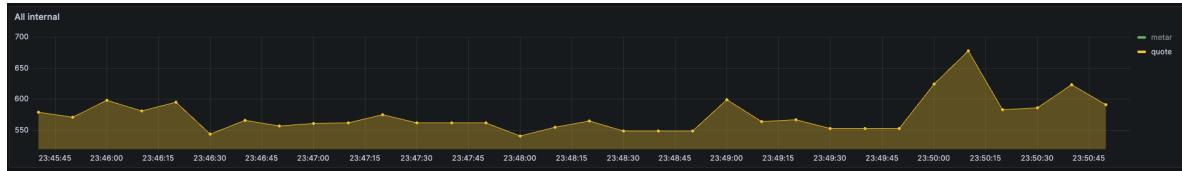


Figura 17: Demora de API externa

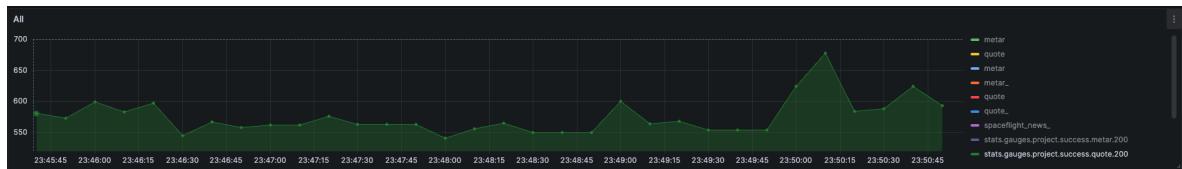


Figura 18: Demora del endpoint en respuestas con código 200

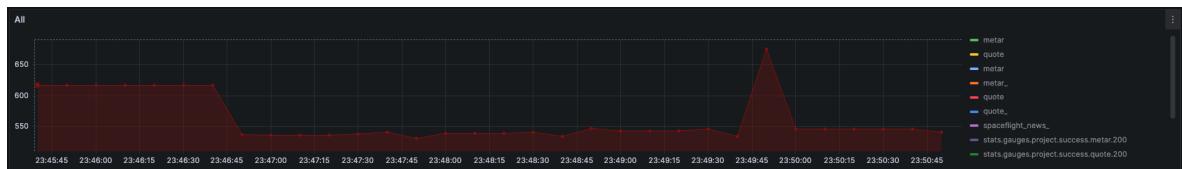


Figura 19: Demora del endpoint en respuestas con código 400

Comparando los gráficos, se puede ver que alrededor del primer minuto hubo un pico en el tiempo de respuesta del lado del cliente, esto se puede explicar por una demora sostenida en el tiempo de respuesta para solicitudes que devolvieron código 400 (ver figura 19). A los 4 minutos, aproximadamente se ve en las figuras 19 y 122 otro pico en el tiempo de respuesta de errores, que también impacta en los tiempos de respuesta que percibe el cliente.

Comparando esta táctica con el caso base, observamos que la limitación de solicitudes por segundo evita el pico de errores que se observó en el caso base (y, en consecuencia, las demoras), pero también disminuye significativamente (aproximadamente un 50 %) la cantidad de solicitudes completadas.

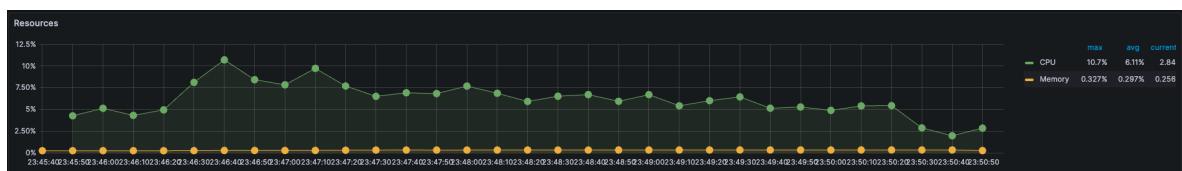


Figura 20: Consumo de recursos

3.2.3. Replicación

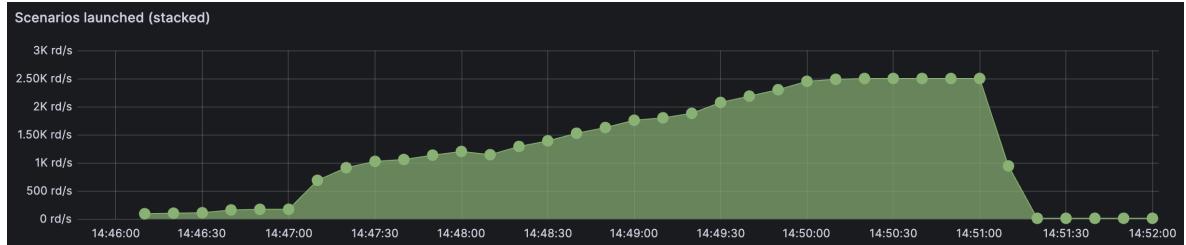


Figura 21: Cantidad de escenarios de carga lanzados

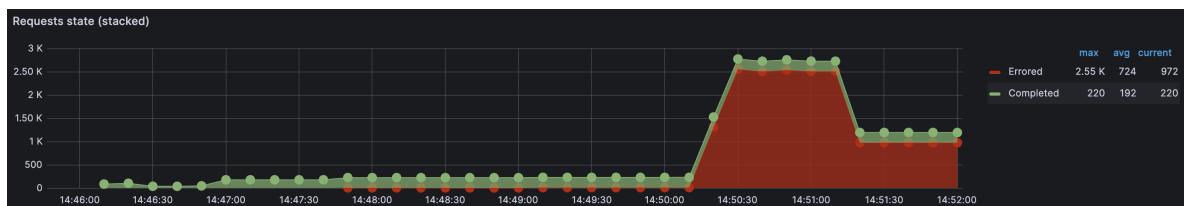


Figura 22: Tipo de respuestas obtenidas

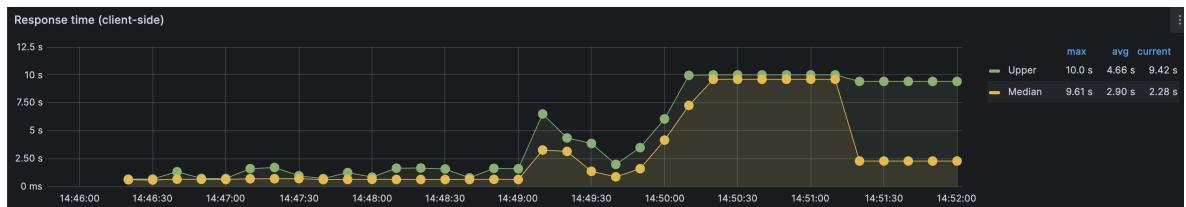


Figura 23: Demora de respuestas (client-side)

En este caso, se puede observar en la figura 23 como a los 4 minutos, coincidiendo con el punto de carga más alto, la API comienza a arrojar mayor cantidad de respuestas con errores. Esto es consistente con los máximos que se observan en las figuras relativas a la demora del endpoint (figuras 23, 24 y 25), particularmente podemos ver que la demora de la API externa es la causante luego de un incremento continuo en respuestas de tipo 400.

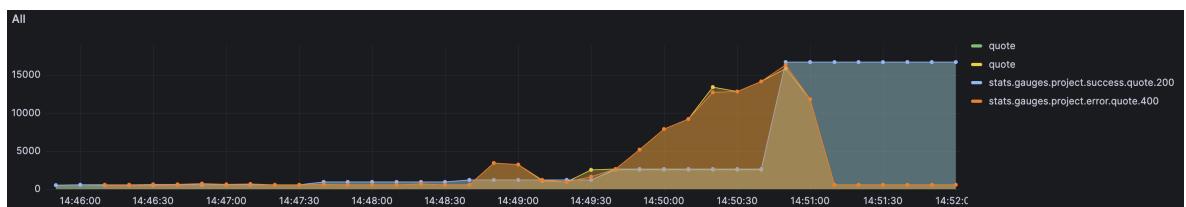


Figura 24: Demora del endpoint



Figura 25: Demora de la API externa

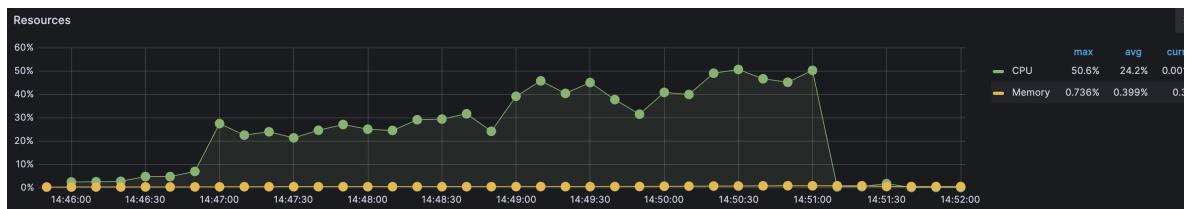


Figura 26: Consumo de recursos

En el caso base, la demora de respuesta de la API externa comienza a aumentar de forma escalonada a partir de los 3 minutos aproximadamente; en este caso aumenta significativamente a los 4 minutos aumentando el tiempo de respuesta general y los errores. No encontramos una mejora con esta táctica.

3.3. Metar

3.3.1. Caso base

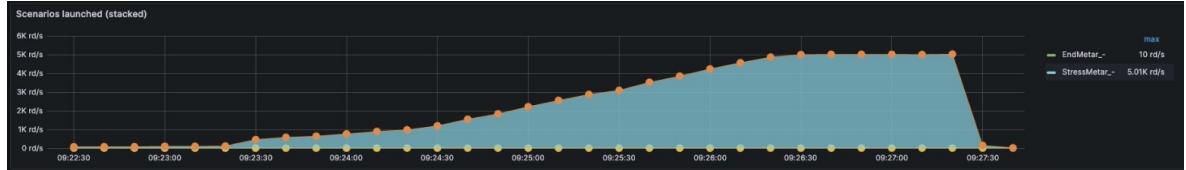


Figura 27: Cantidad de escenarios de carga lanzados

No encontramos listado un límite de request para el endpoint de *metar*, por lo que nos vimos forzados a inferirlo a través del test de estrés. Configuramos Artillery de modo tal que la cantidad de request aumente exponencialmente con el paso del tiempo.

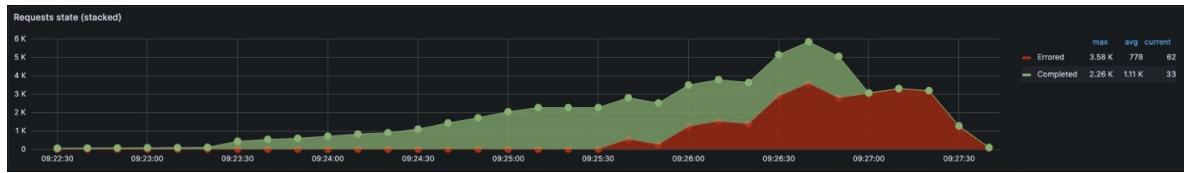


Figura 28: Tipo de respuestas obtenidas

Descubrimos que, a diferencia de *quote*, *metar* falla una fracción creciente de request antes de dejar de responder por completo. Para guiar el endurance testing estimamos la cantidad de request por segundo hasta que ocurrió el primer error en unos 60 request por segundo.

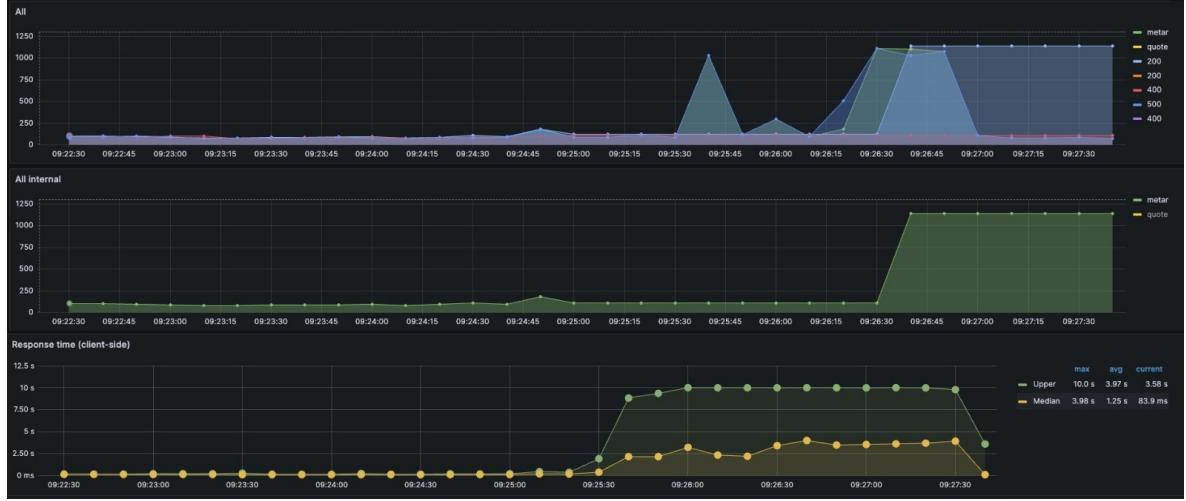


Figura 29: Demora en respuestas

Los tiempos de respuesta en el caso de los 200 son comparables a los de *quote*, pero las request fallidas tardan tanto en ser respondidas que afectan la escala total del gráfico. En consecuencia se distingue desde el cliente la espera promedio cuando el pedido es exitoso y la espera máxima cuando éste falla. A medida que una proporción de request mayor fracasa, más se acercan ambas medidas, lo cuál nos recuerda que la comparación entre casos extremos y el promedio solo nos brinda una noción general de que la experiencia de usuario es similar, pero no nos dice nada de la naturaleza de dicha experiencia.



Figura 30: Consumo de recursos

Una vez más el uso de CPU crece junto con la tasa de request, pero la memoria se mantiene relativamente constante.

3.3.2. Caché



Figura 31: Cantidad de escenarios de carga lanzados

Podemos ver que de este no se obtiene nada relevante sobre el gráfico del caso base, ya que se asumió el mismo caso de prueba.



Figura 32: Cantidad de escenarios de carga lanzados

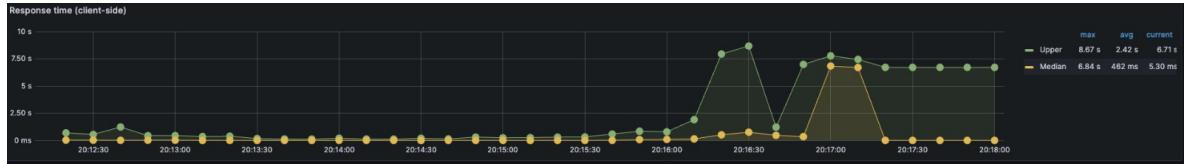


Figura 33: Tipo de respuestas obtenidas

El segundo gráfico demuestra que los tiempos de respuesta en promedio bajan, con unos pocos outliers provocados por la alta cantidad de requests. Lo que se ve reflejado en el primer gráfico por disminución en la cantidad de request erróneas.

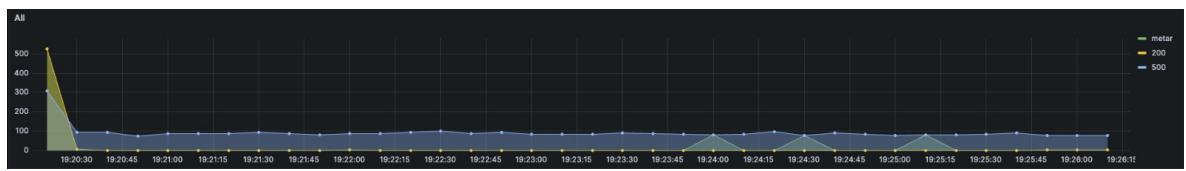


Figura 34: Demora de respuestas (client-side)

El tiempo de respuesta reportado para el servidor también disminuye, ya que no debe esperar la respuesta de la API externa, sino la de la base de datos de Redis, que se encuentra en la misma computadora y presenta una demora menor.

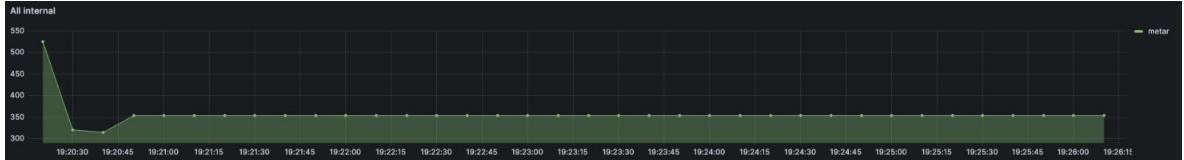


Figura 35: Demora de API externa

Es importante notar que no se actualizan los tiempos que se muestran, ya que no hay más requests a API externas para actualizar este gráfico, por lo que parece que el tiempo es casi constante.

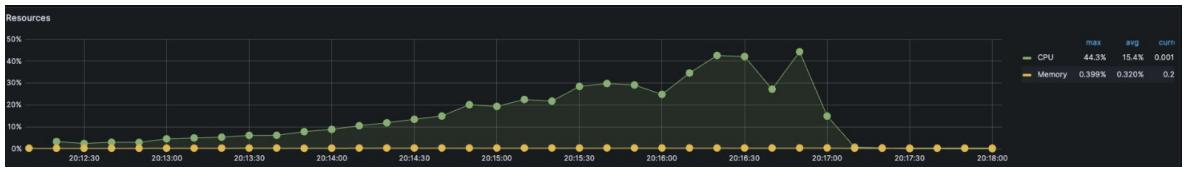


Figura 36: Consumo de recursos

De los recursos lo que se puede observar es que se usan significativamente menos que en el caso base. Esto se debe, al igual que en todos los casos de cache, a que Redis al estar en un servicio separado la cpu no tiene que handlear el esfuerzo que conlleva levantarla. Además de que el uso de Redis implica que no se utilizará cpu en pegadas a APIs externas innecesariamente ya que ya estarán cacheadas.

3.3.3. Rate limiting

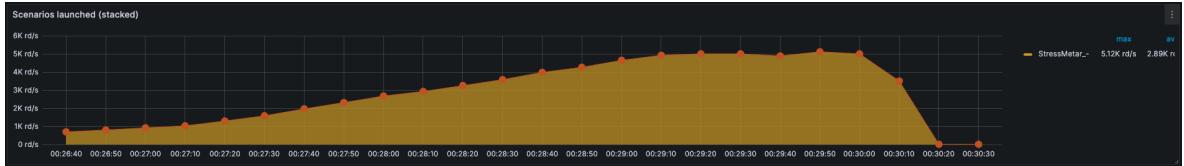


Figura 37: Cantidad de escenarios de carga lanzados

En la figura 38 se ve cómo aumenta continuamente la cantidad de solicitudes limitadas con respecto a las completadas. Al igual que para el caso base, a los 3 minutos comienzan a observarse errores, pero a menor escala.

Además, se puede observar en la figura 40 que luego de unos 3 minutos hay un pico en la demora de la API externa, esto se da después de un pico de demora en respuestas de tipo 400 (ver figura 42).

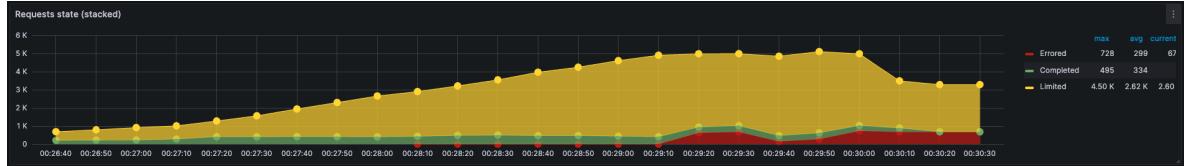


Figura 38: Tipo de respuestas obtenidas

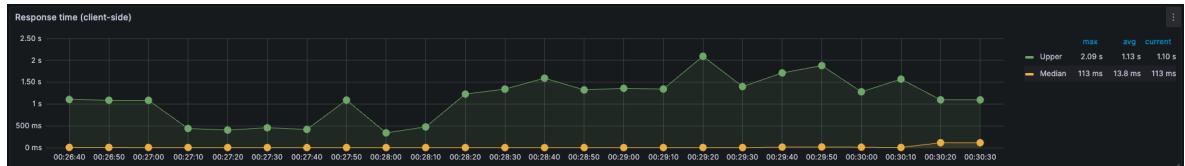


Figura 39: Demora de respuestas (client-side)



Figura 40: Demora de API externa



Figura 41: Demora del endpoint en respuestas con código 200



Figura 42: Demora del endpoint en respuestas con código 400

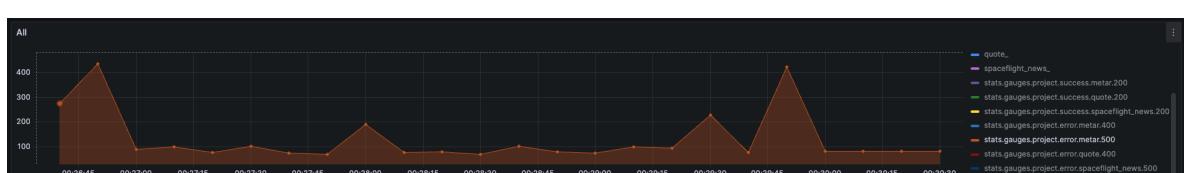


Figura 43: Demora del endpoint en respuestas con código 500

A diferencia del caso base, en este caso observamos que los tiempos de demora para el cliente son más inestables durante los primeros minutos. Esto se puede explicar por las demoras en respuestas con errores 400 y 500 que se ven al comienzo de los gráficos 42 y 43.

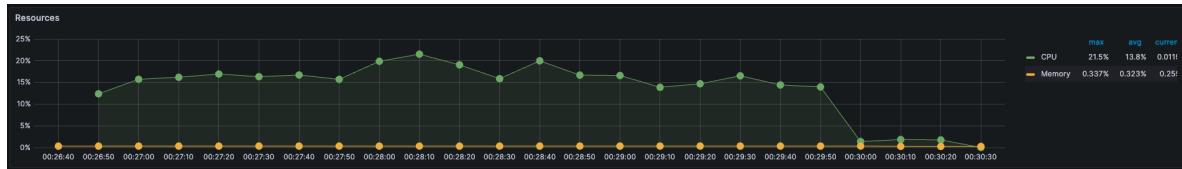


Figura 44: Consumo de recursos

3.3.4. Replicación

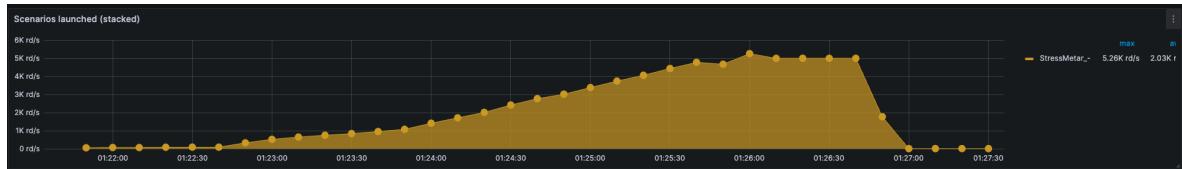


Figura 45: Cantidad de escenarios de carga lanzados

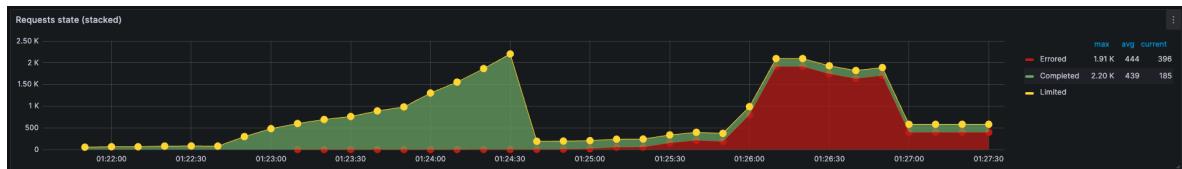


Figura 46: Tipo de respuestas obtenidas

En el gráfico 46 se observa que a los 4 minutos comienza a aumentar significativamente la cantidad de errores, si bien el pico es menor al alcanzado con el caso base, la cantidad de solicitudes exitosas también es menor. Estos errores coinciden con una demora en los tiempos de respuesta percibidos por el cliente, que a su vez coinciden con un aumento en la demora de respuesta de errores 500.

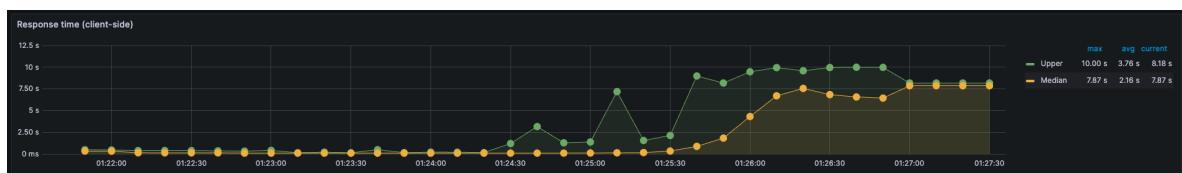


Figura 47: Demora de respuestas (client-side)

Comparando las figuras 46 y 47 se puede observar como durante los picos de demora de respuestas, disminuye considerablemente la cantidad de respuestas obtenidas.



Figura 48: Demora de API externa

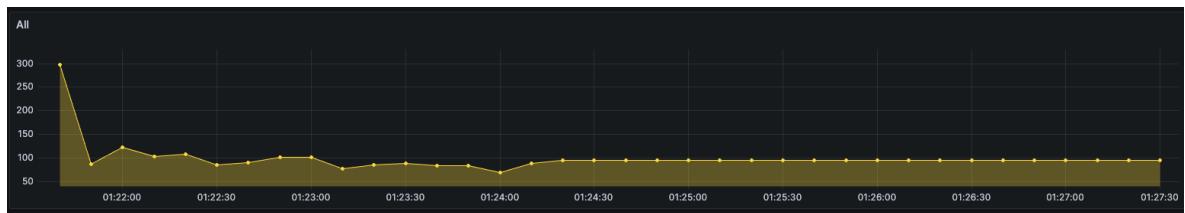


Figura 49: Demora del endpoint en respuestas con código 200



Figura 50: Demora del endpoint en respuestas con código 400



Figura 51: Demora del endpoint en respuestas con código 500

Además, en este caso se ve una reducción significativa en el uso de la CPU, con algunos incrementos hacia al final de la ejecución. Esto puede estar ligado a la velocidad en las respuestas durante los primeros 2 minutos.



Figura 52: Consumo de recursos

3.4. Spaceflight News

3.4.1. Caso base

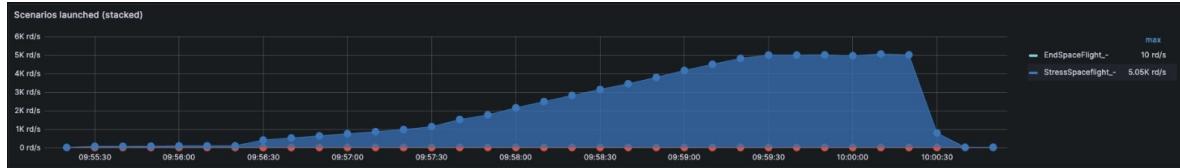


Figura 53: Cantidad de escenarios de carga lanzados

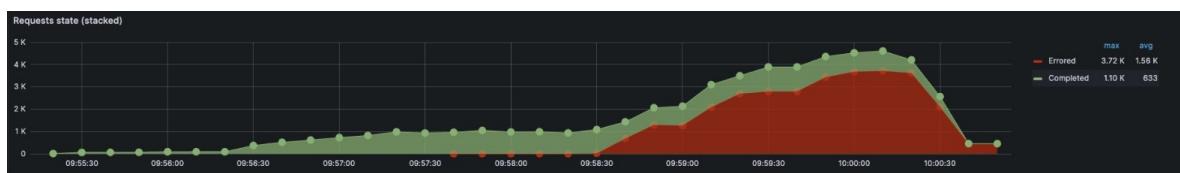


Figura 54: Tipo de respuestas obtenidas

Para *spaceflight_news* tampoco contábamos con información sobre límites impuestos sobre las queries. Una vez más aprovechamos el test de estrés para estimar el límite por nuestra cuenta. Aquí decidimos ignorar el primer error, puesto que existe un intervalo de tiempo muy grande desde que ocurre hasta que la cantidad de request erróneas comienzan a crecer.

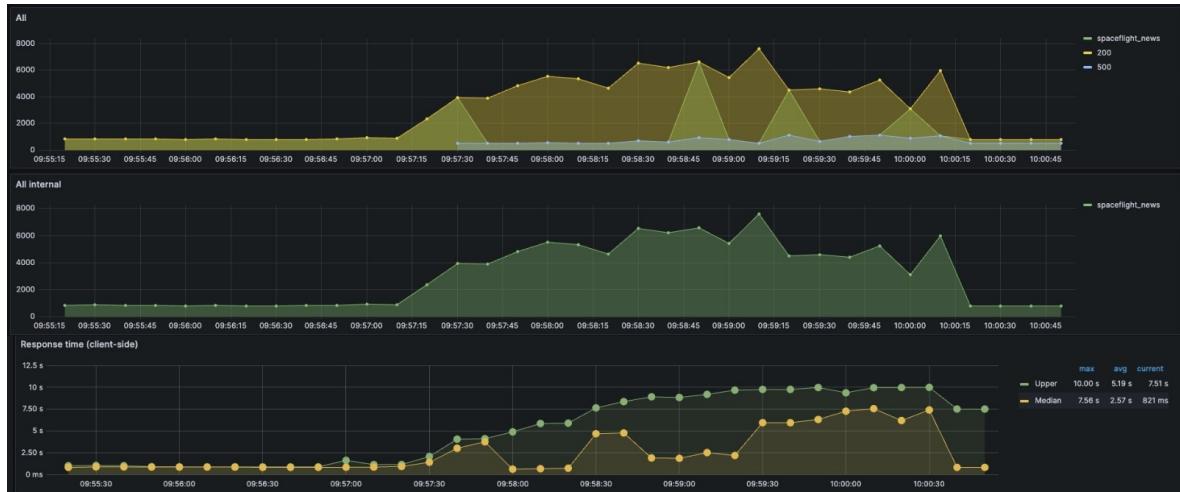


Figura 55: Demora en respuestas

Al igual que *metar*, *spaceflight_news* no interrumpe la respuesta a los request súbitamente, sino que rechaza una fracción de consultas hasta finalmente rechazarlas todas. Se refleja un incremento constante en los tiempos de respuesta que solo se alivia cuando se comienza a rechazar solicitudes. Desde el cliente observamos que la máxima y la mediana de los tiempos de respuesta se encuentran próximas cuando se completan o se rechazan todas las request, pero se aparta en el intervalo de transición.



Figura 56: Consumo de recursos

3.4.2. Caché

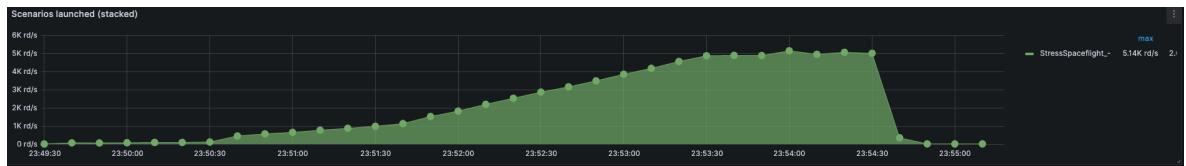


Figura 57: Cantidad de escenarios de carga lanzados

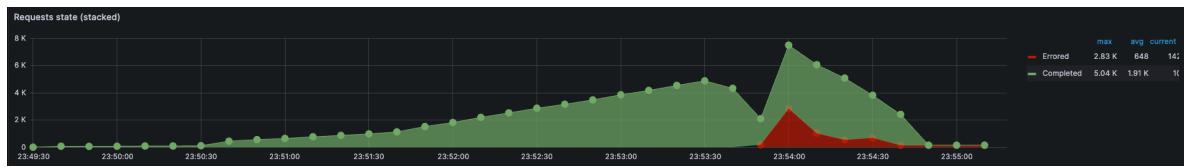


Figura 58: Tipo de respuestas obtenidas

En base a lo observado en *metar*, esperamos que al incorporar cache mejoren los tiempos de respuesta de spaceflightnews.

De aqui podemos ver que la cantidad de errores recibidos por fallas en la conexión se redujo enormemente con respecto al caso base. Esto se debe, como veremos en el siguiente grafico, a tiempos de respuesta mucho menores desde el lado del server.

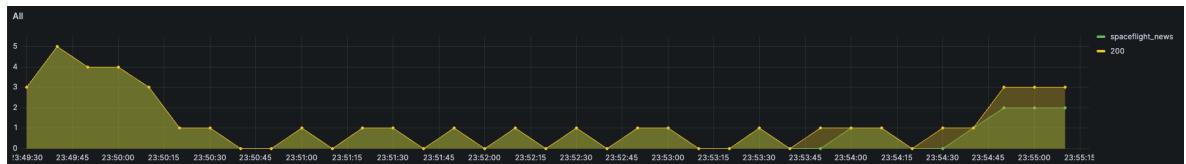


Figura 59: Demora de respuestas (client-side)

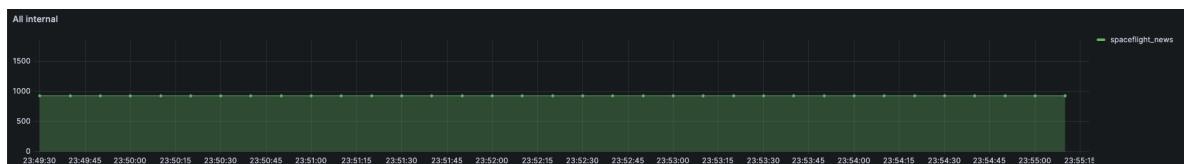


Figura 60: Demora de API externa

Notamos que la cantidad de request a la API externa una vez más disminuyó y casi no se perciben saltos en los tiempos de delay a APIs externas.

Al usar active population con la base de datos de redis, nunca hubo necesidades de demorar enormemente las devoluciones de respuesta por pegadas a la API externa, por ello mejoran enormemente los tiempos de respuesta de cara al cliente.



Figura 61: Tiempo de respuesta

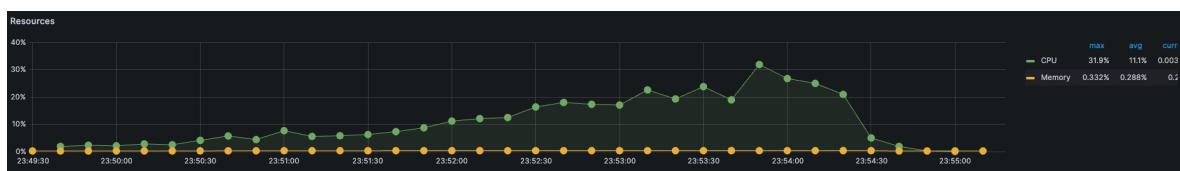


Figura 62: Consumo de recursos

Se puede destacar que se emplean mucho menos recursos. No se trata solo del ahorro del CPU, sino de la liberación temprana de recursos asociados a conexiones, que genera una disminución en el uso de memoria.

3.4.3. Rate limiting

En este caso, puede verse como disminuye la cantidad de errores de un máximo de 3700 a uno de 859.

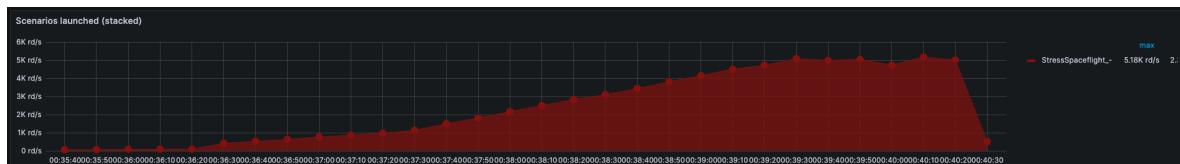


Figura 63: Cantidad de escenarios de carga lanzados

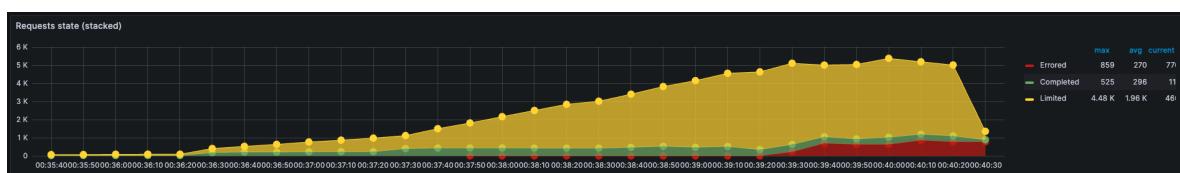


Figura 64: Tipo de respuestas obtenidas

Comparando los tiempos de respuesta, se ve que la API de Spaceflight comienza a tener demoras a partir de los 3 minutos; esto es un minuto más tarde que en el caso base.

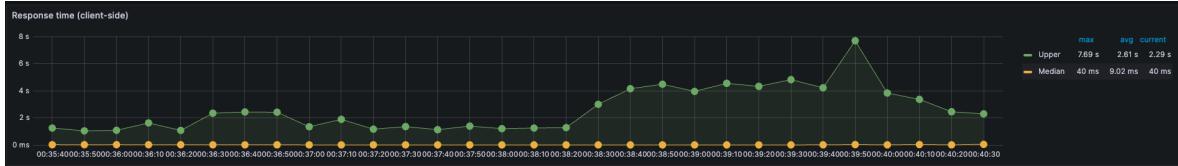


Figura 65: Demora de respuestas (client-side)

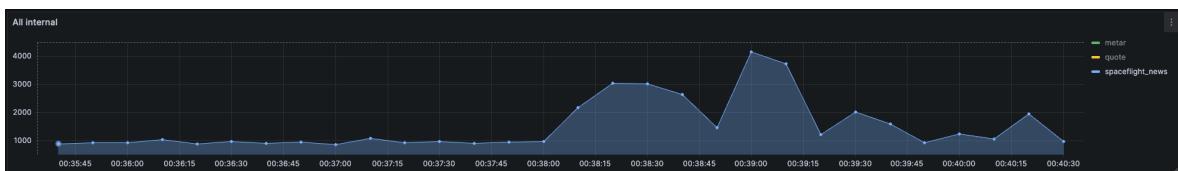


Figura 66: Demora de API externa



Figura 67: Demora del endpoint en respuestas con código 200

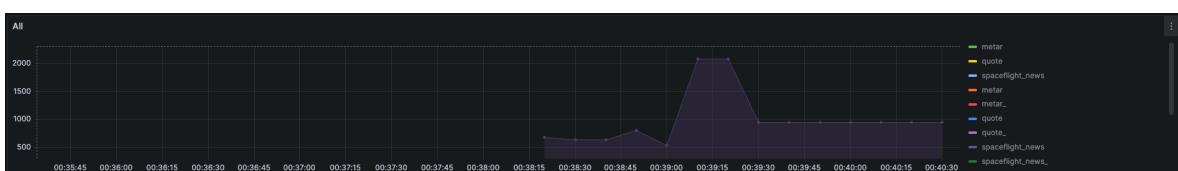


Figura 68: Demora del endpoint en respuestas con código 500



Figura 69: Consumo de recursos

3.4.4. Replicación

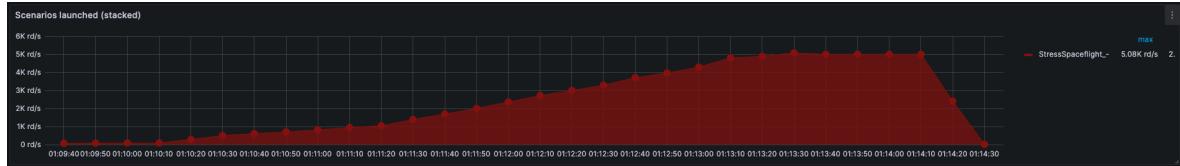


Figura 70: Cantidad de escenarios de carga lanzados

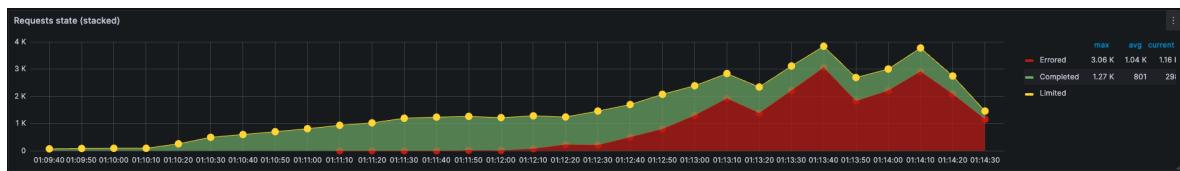


Figura 71: Tipo de respuestas obtenidas

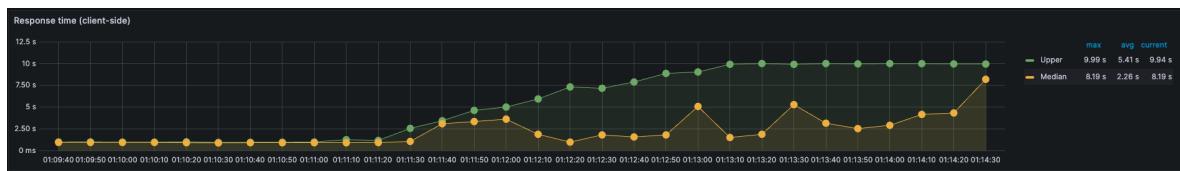


Figura 72: Demora de respuestas (client-side)

En la figura 73 se ven tres picos donde aumenta considerablemente la demora de respuesta, estos podrían asociarse a las tres replicas. En consecuencia, en el gráfico 71 se observa el mismo comportamiento acompañado de la aparición de errores. De forma similar a los casos anteriores, los errores comienzan luego de los 3 minutos de iniciada la ejecución, esto ocurre cuando la carga llega a su punto máximo.

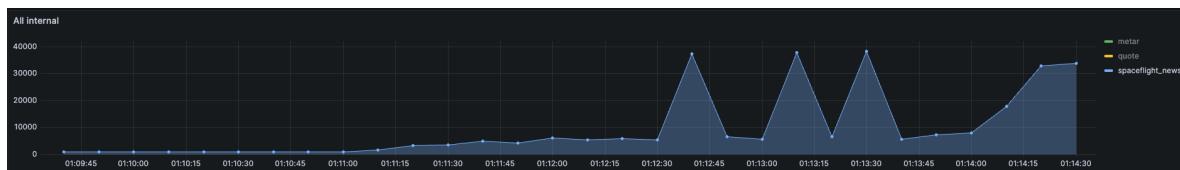


Figura 73: Demora de API externa

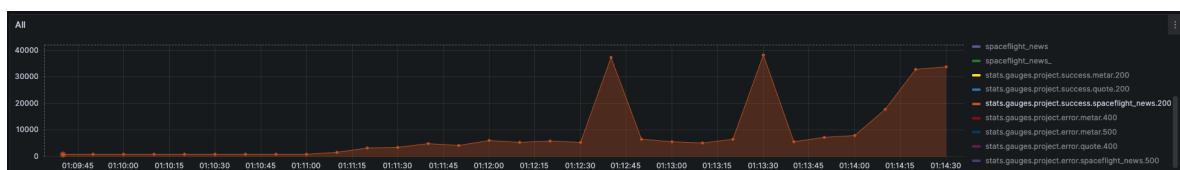


Figura 74: Demora del endpoint en respuestas con código 200

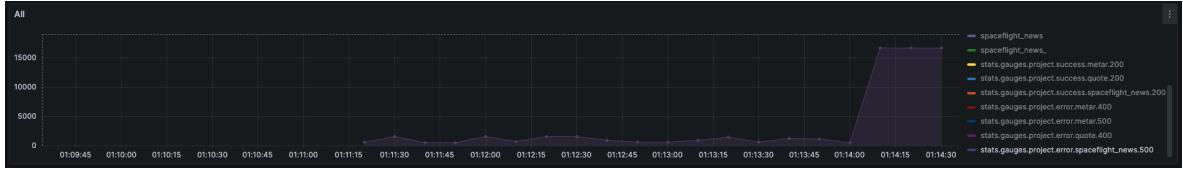


Figura 75: Demora del endpoint en respuestas con código 500

Notamos que las demoras ocurren con respuestas éxitosas, por lo que inferimos que se originan por la alta carga de solicitudes.

Al igual que sucedió con la API de *metar*, el consumo de CPU disminuyó notablemente.

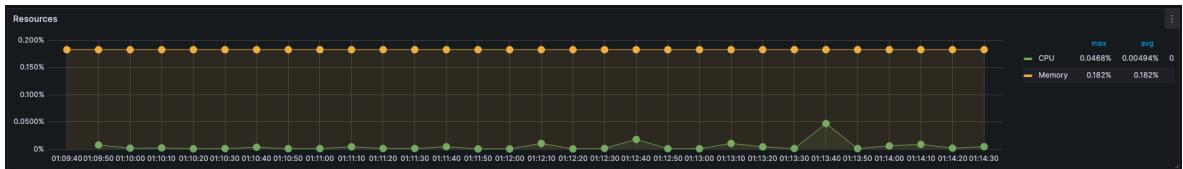


Figura 76: Consumo de recursos

4. Endurance Testing

4.1. Ping

4.1.1. Caso base

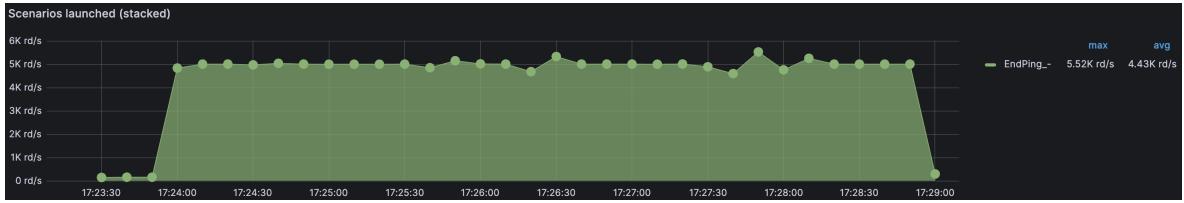


Figura 77: Cantidad de escenarios de carga lanzados

La tasa de lanzado de escenarios se mantiene constante, patrón que se repetirá en el resto de los casos.

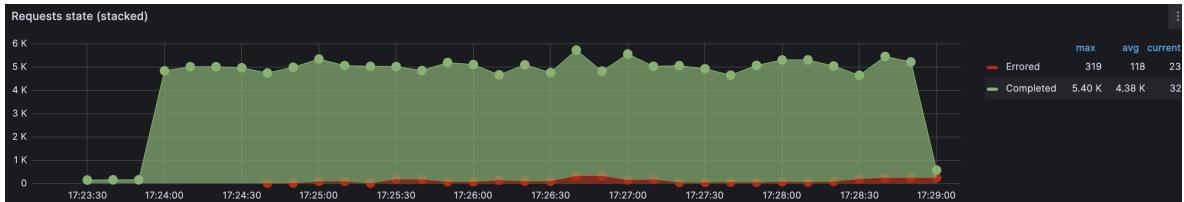


Figura 78: Tipo de respuestas obtenidas

Nos sorprendió ver errores en un caso tan sencillo, pero luego comprendimos que al encontrarse la carga del lanzado de escenarios en el mismo equipo donde se están procesando y encontrarse la implementación en node, con un solo hilo de ejecución, es más probable colapsar.

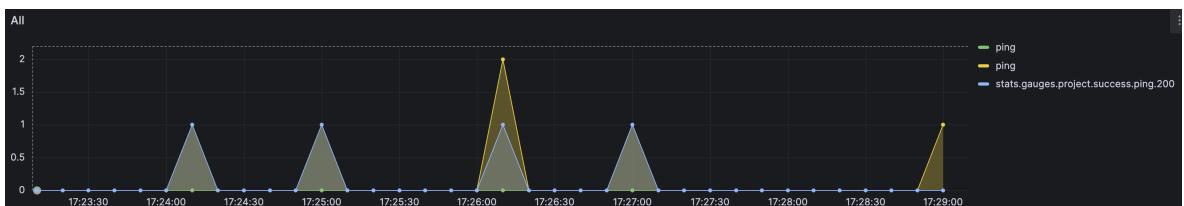


Figura 79: Demora del endpoint

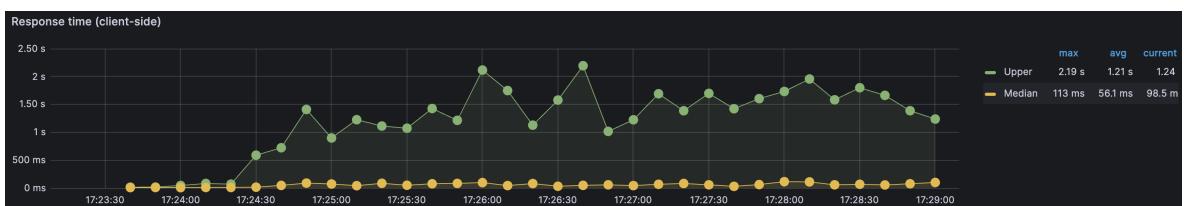


Figura 80: Demora de respuesta (client-side)

Los tiempos de demora son extremadamente bajos, puesto que se resuelven en el mismo equipo. Existen picos ocasionales que devienen de la sola carga del sistema y posiblemente de interferencias de otros procesos externos, lo que provoca que algunos clientes perciban demoras altísimas respecto a la media.

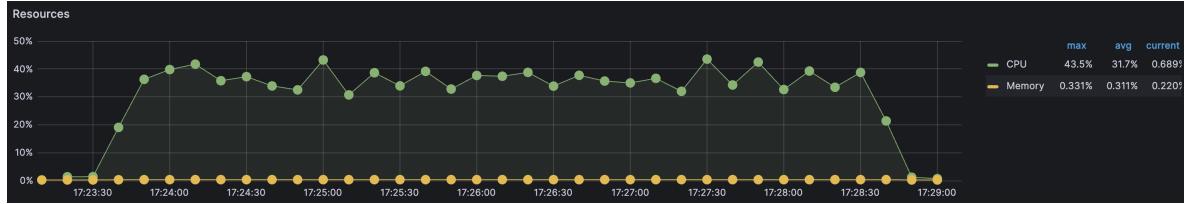


Figura 81: Consumo de recursos

El uso de recursos es mucho más alto que el resto de los casos, al no haber dependencias de apis externas logramos bajar mucho los tiempos de respuesta, pero hemos incrementado la carga por unidad de tiempo.

4.2. Random Quote

4.2.1. Caso base



Figura 82: Cantidad de escenarios de carga lanzados



Figura 83: Tipo de respuestas obtenidas

Para las pruebas de endurance se estableció la tasa de arribo en 2 request por segundo (120 request por minuto), ya que con 3 request por segundo ya se alcanzaban 180 por minuto y se observaban errores frecuentes debido a que la última request del minuto se demoraba y contaba para el límite del siguiente minuto. En el propio gráfico se puede contrastar que la tasa de solicitudes se mantuvo durante todo el test y que no se encontraron errores que devengan del uso continuado de la API.



Figura 84: Demora en respuestas

Pese al ritmo constante de las request los tiempos de respuesta oscilan significativamente debido a demoras en la red, con una media de 600 ms. Nuestro sistema depende de dicha respuesta y por ello sufre los mismos retrasos. El cliente sufre la demora antes mencionada sumada al tiempo insumido en la comunicación con nuestro sistema y en el caso más extremo llega a percibir una demora de 1,5 segundos.



Figura 85: Consumo de recursos

El uso de CPU y memoria es relativamente constante, en consonancia con lo anteriormente descrito.

4.2.2. Rate limiting

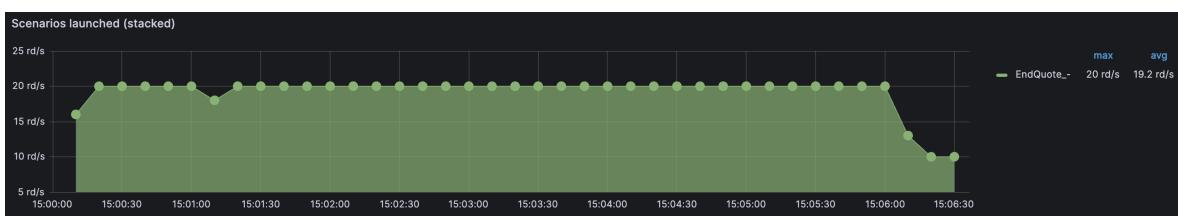


Figura 86: Cantidad de escenarios de carga lanzados

Por ser que la cantidad de solicitudes simuladas está levemente por sobre lo soportado por la API externa, podemos ver que en el gráfico 87 se limitan algunas de ellas en forma

contante, de acuerdo a la carga enviada.

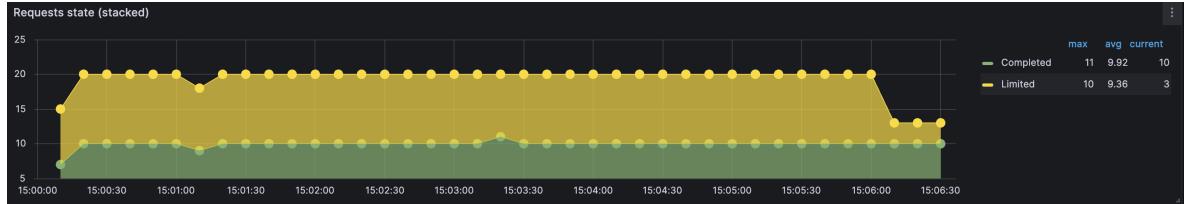


Figura 87: Tipo de respuestas obtenidas

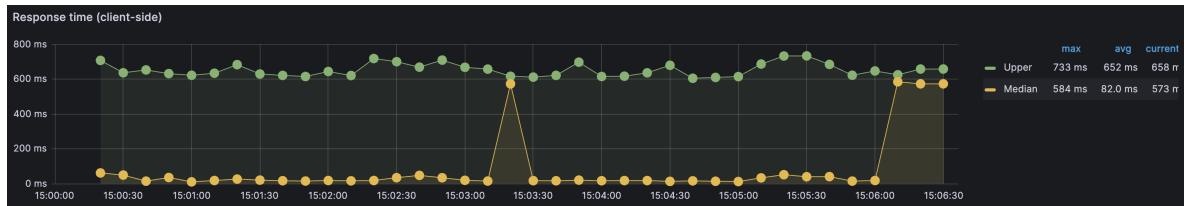


Figura 88: Demora de respuestas (client-side)

Los tiempos de demora se mantienen entre los 600 y 800ms, con dos picos en el valor de la mediana. Notamos que el comportamiento de la demora del endpoint es el mismo que el de la demora de la API externa, por lo que las demoras que sufre el cliente provienen de los tiempos de respuesta de la API de Random Quotes.

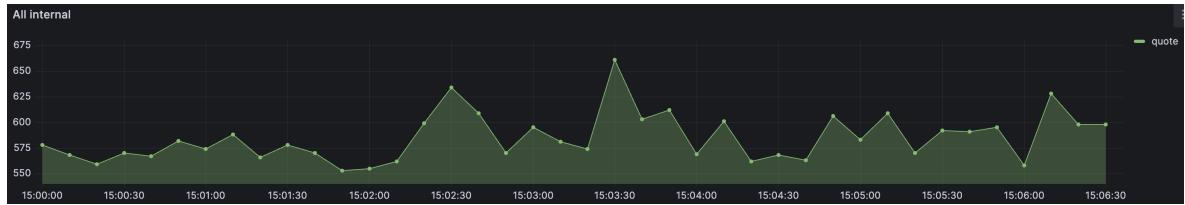


Figura 89: Demora de API externa

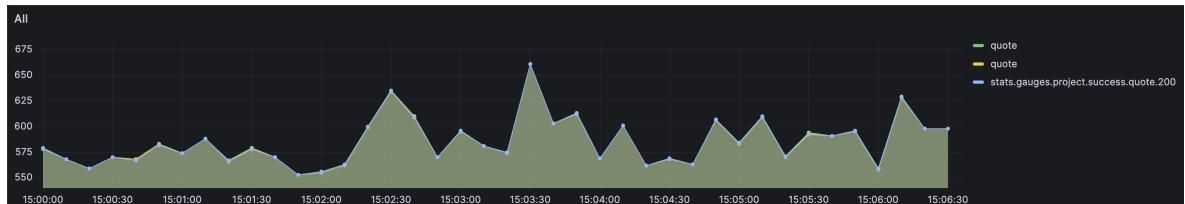


Figura 90: Demora del endpoint

Notamos que el consumo de CPU tiene condice con las variaciones en los tiempos de respuesta de la API.

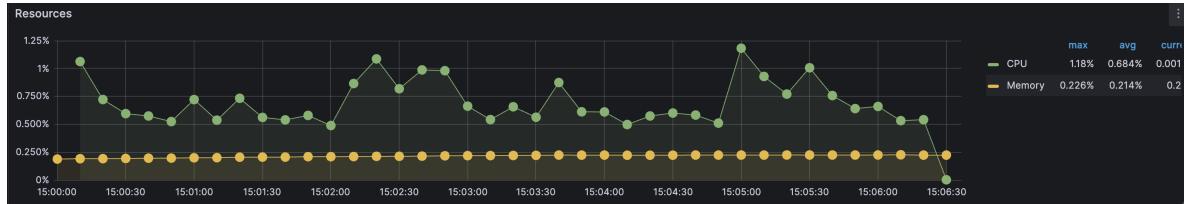


Figura 91: Consumo de recursos

4.2.3. Replicación

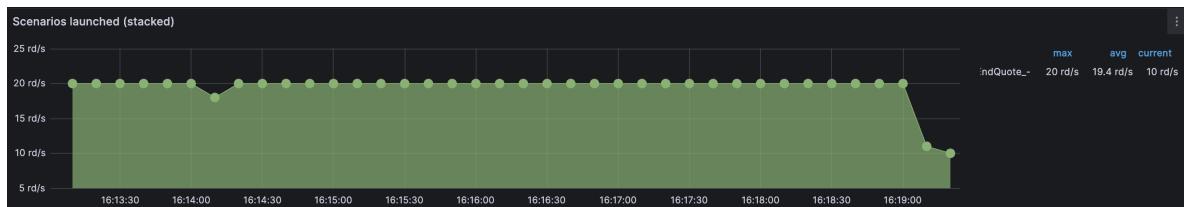


Figura 92: Cantidad de escenarios de carga lanzados

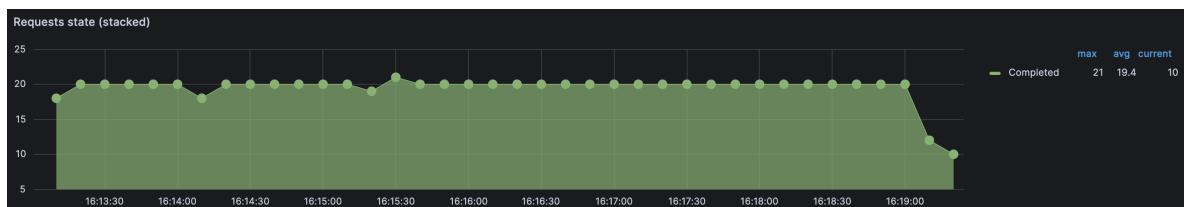


Figura 93: Tipo de respuestas obtenidas

Con esta táctica se completan exitosamente todas las solicitudes. Las demoras hacia el cliente tienen algunos picos, pero su mediana se mantiene alrededor de los 500 ms en lo que dura la ejecución. Además, la API externa no presenta grandes demoras que afecten la disponibilidad del servicio.

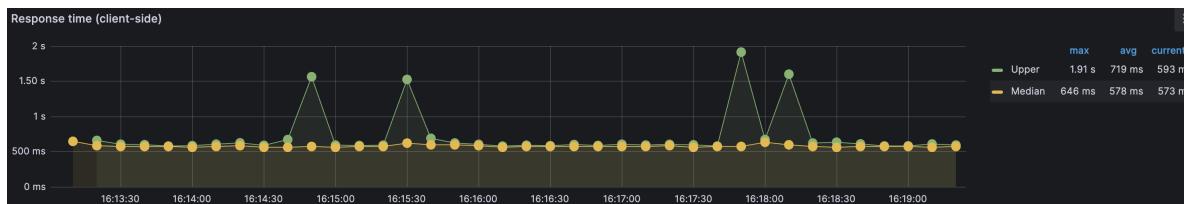
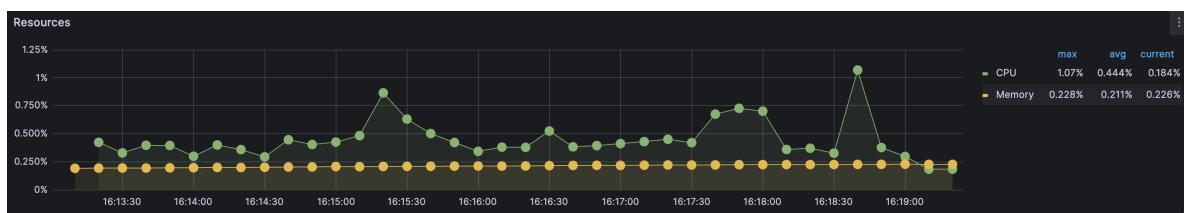
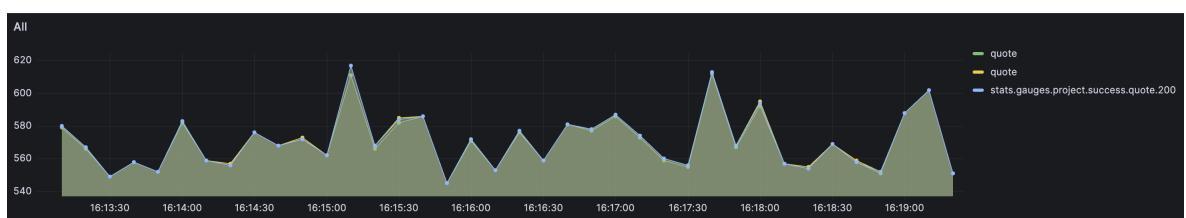
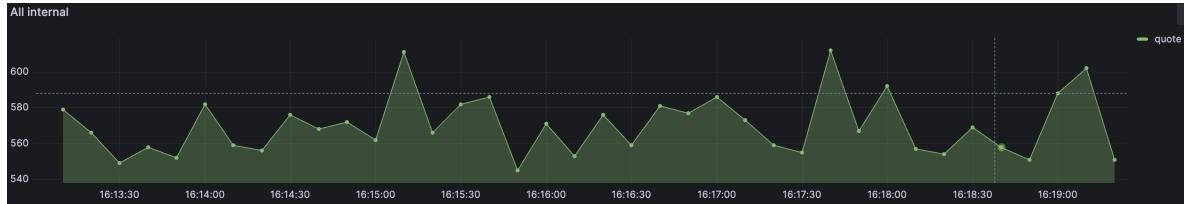
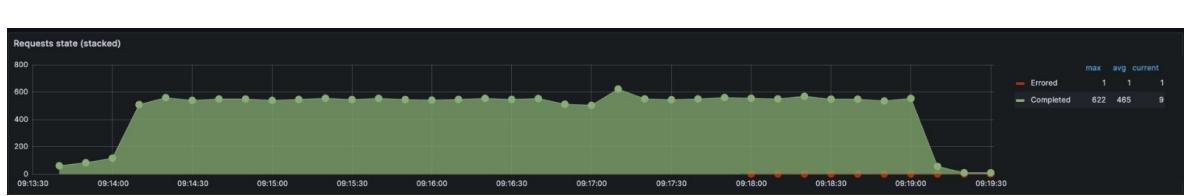
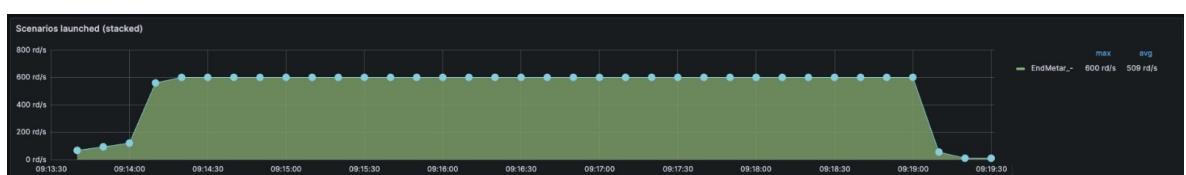


Figura 94: Demora de respuestas (client-side)



4.3. Metar

4.3.1. Caso base



Como se adelantó en la sección de stress, para la prueba de endurance establecimos la tasa de request en 60 por segundo. Detectamos un solo error hacia el final de la prueba que solo puede haber venido del uso continuado del endpoint, ya que filtramos los códigos de aeropuerto con una respuesta exitosa anteriormente.



Figura 100

Los tiempos de respuesta se mantienen relativamente constantes, a excepción de un pico súbito de más de 1,5 segundos en medio de la ejecución. En el gráfico del estado de las request se observa como los resultados se demoran en un primer momento y luego se compensan en el siguiente.

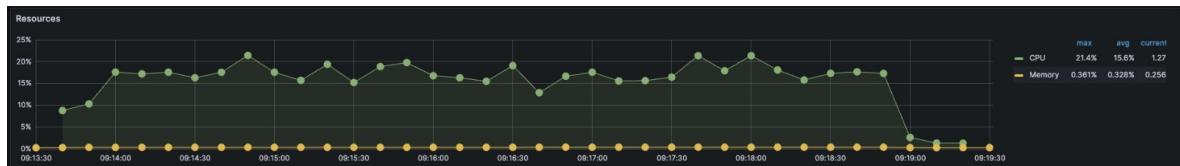


Figura 101: Consumo de recursos

El nivel de uso de memoria se mantiene, mientras que el de CPU oscila sobre un valor constante. El pico en los tiempos de respuesta no se ve reflejado significativamente en nuestro servicio.

4.3.2. Caché



Figura 102: Cantidad de escenarios de carga lanzados

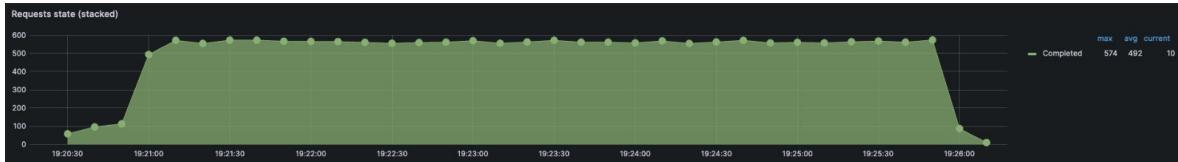


Figura 103: Tipo de respuestas obtenidas

Vemos un estado muy similar al del base case, con la excepción de que no ocurre ningún spike inesperado en el estado de request, esto es esperable ya que todas las respuestas por cache deberían ser medianamente similares.

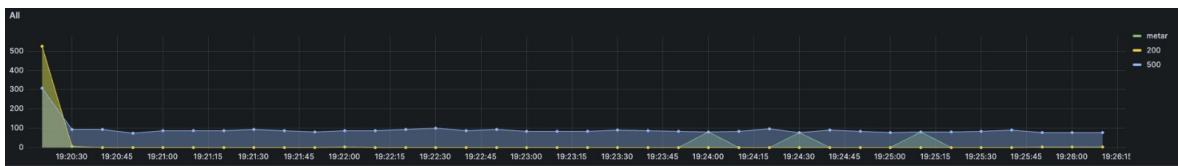


Figura 104: Demora en tiempo de respuesta del server

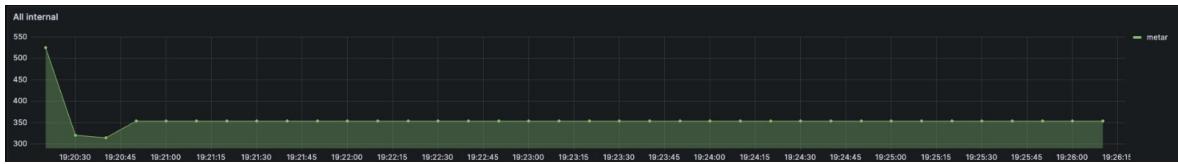


Figura 105: Demora de la API externa

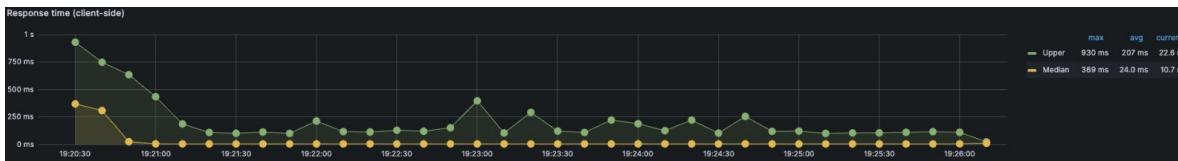


Figura 106: Demora en tiempo de respuesta del server (Client-side)

Se obtiene lo que se esperaba que es un uso constante del server luego de la primera request, lo que significa que la cache mejoró enormemente los tiempos promedio y máximos de respuesta con respecto a los escenarios que no la incluyen. Al mismo tiempo, no se observa un pico como en el caso anterior a mitad de la request, creemos que el poblamiento activo del cache permite homogeneizar los tiempos de respuesta lo suficiente para contrarrestar estos picos, sea cual sea su causa última.

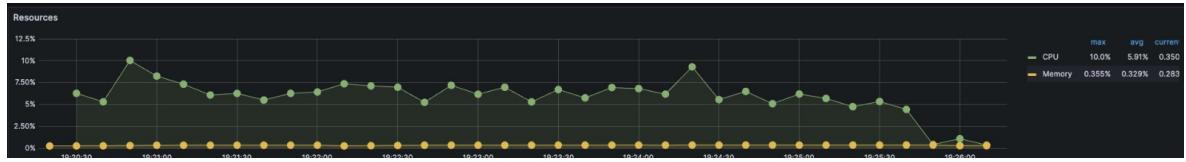


Figura 107: Consumo de recursos

A nivel de recursos, al igual que con stress testing, vemos una mejora. Y las razones se mantienen ya que se le exige menos al server porque no tiene que mantener una constante conexión con las APIs externas.

4.3.3. Rate limiting

Con el uso de esta táctica, se observa que la mayoría de las solicitudes son limitadas por Nginx. Este caso no presenta ninguna mejora respecto al caso base ya que comparando sus máximos alcanzados, esta limitación no parecería necesaria.

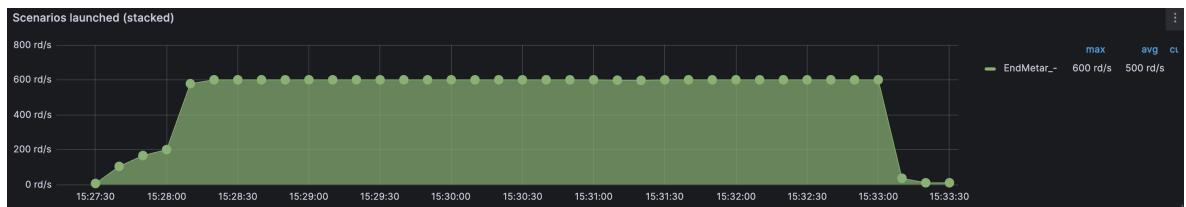


Figura 108: Cantidad de escenarios de carga lanzados

Alrededor de los 5 minutos de la ejecución, aparece un pico de 4s en la demora de respuesta, esto coincide con la aparición del único error que se registró.

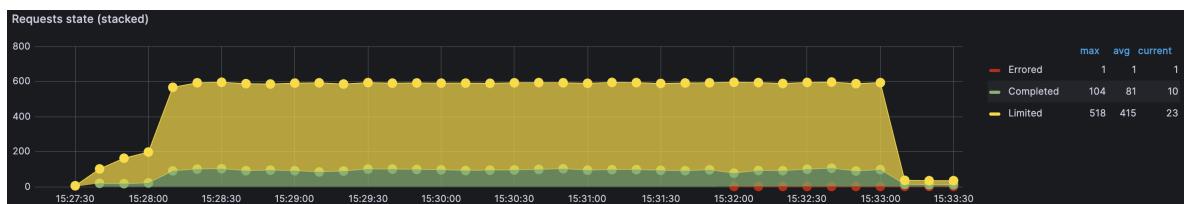


Figura 109: Tipo de respuestas obtenidas

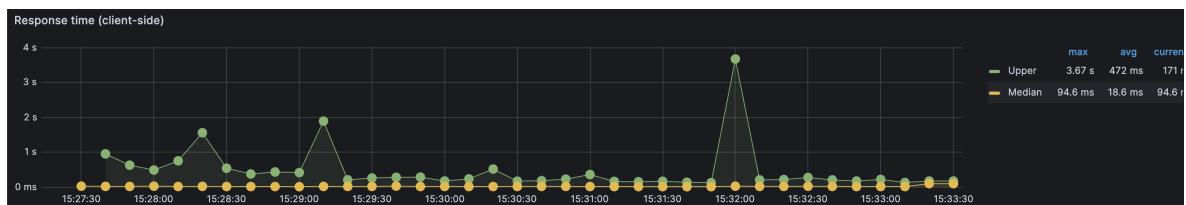


Figura 110: Demora de respuestas (client-side)

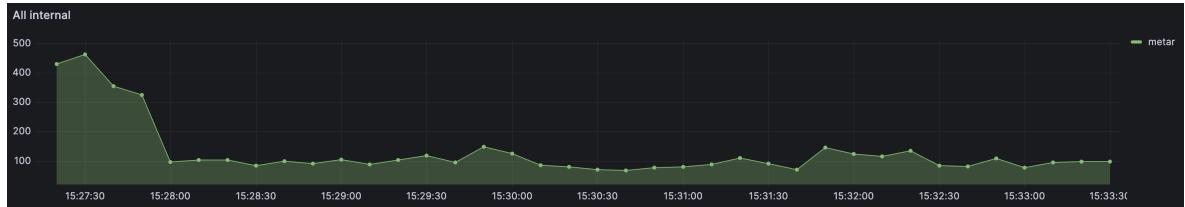


Figura 111: Demora de API externa

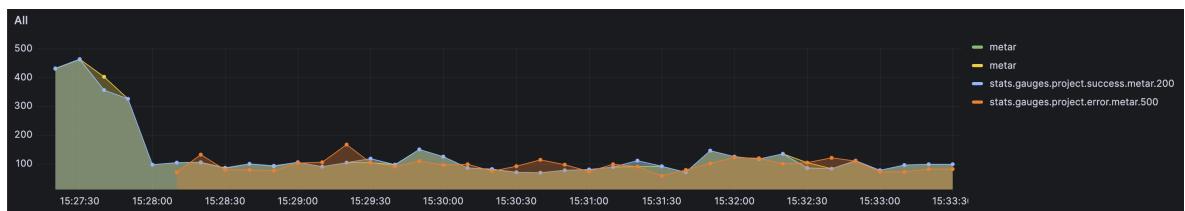


Figura 112: Demora del endpoint

Observamos que inicialmente, la demora de la API externa se encontraba en valores altos, provocando una demora del endpoint; sin embargo, esto no afectó significativamente en la respuesta al cliente ni generó errores.

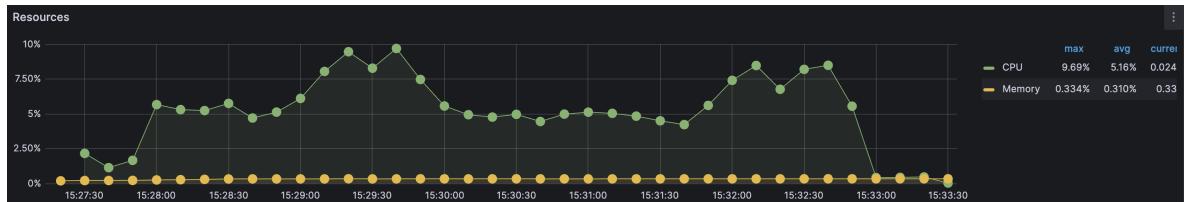


Figura 113: Consumo de recursos

4.3.4. Replicación

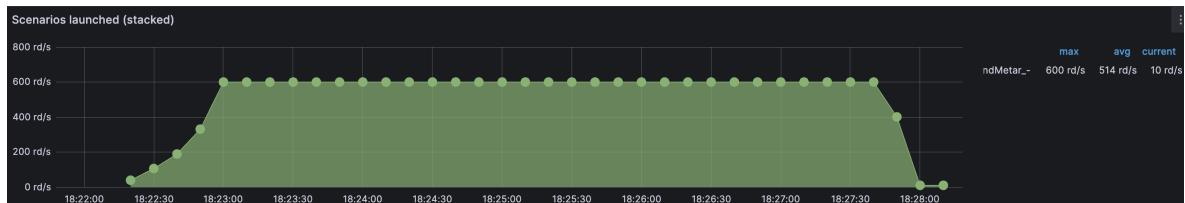


Figura 114: Cantidad de escenarios de carga lanzados

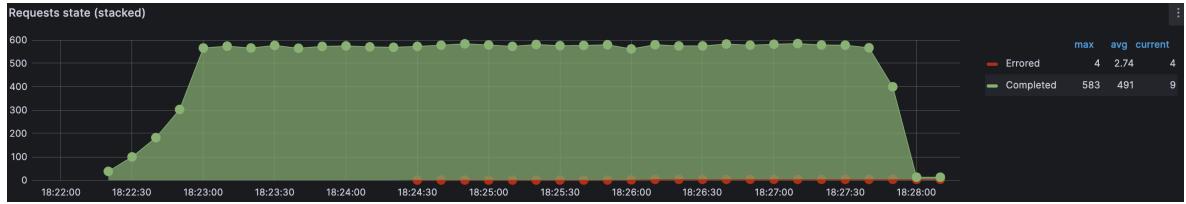


Figura 115: Tipo de respuestas obtenidas

De forma similar al caso anterior, se encuentran algunos picos en la demora de respuesta al cliente. Sin embargo, esta demora no parecería afectar significativamente al servicio ya que sigue funcionando sin problemas. La demora de la API externa y, en consecuencia, del endpoint, se mantiene relativamente baja oscilando alrededor de los 100 ms.

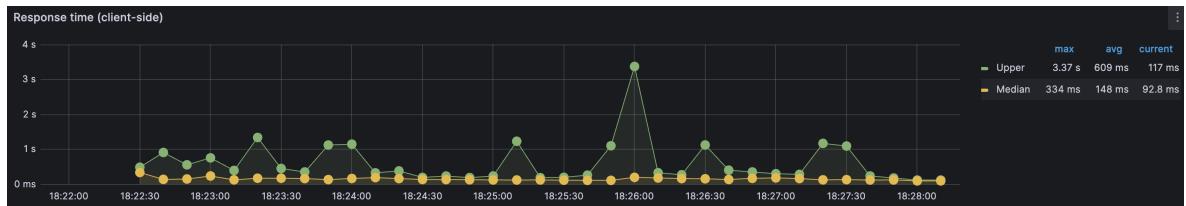


Figura 116: Demora de respuestas (client-side)

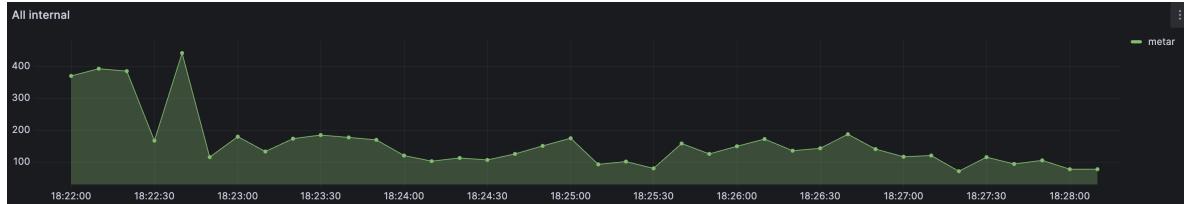


Figura 117: Demora de API externa

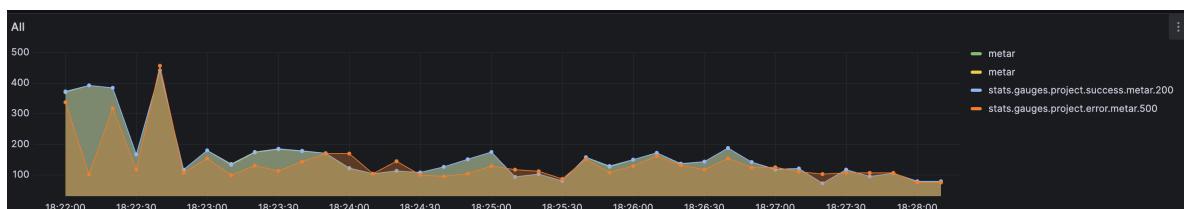


Figura 118: Demora del endpoint

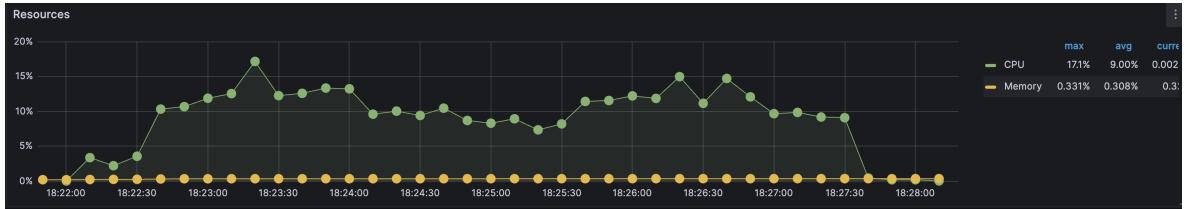


Figura 119: Consumo de recursos

4.4. Spaceflight News

4.4.1. Caso base

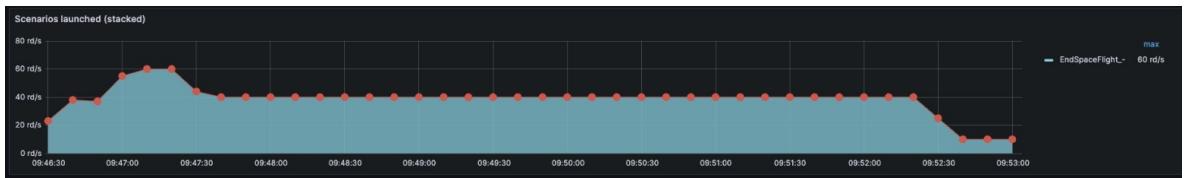


Figura 120: Cantidad de escenarios de carga lanzados



Figura 121: Tipo de respuestas obtenidas

Para el endurance test se estableció la tasa de arribos en dos por segundo. Observamos en los gráficos que no ocurren errores y que los escenarios se lanzan a un ritmo muy estable en el tiempo.



Figura 122: Demora en respuestas

Es muy llamativo notar la variabilidad en los tiempos de respuesta de la API externa, que se encuentra entre 800 y 1000 ms. Desde la perspectiva del cliente dicha variabilidad se suma a la de los mensajes hacia el sistema que desarrollamos, provocando que el máximo alcance los 1250 ms ocasionalmente, pero que en general la media y el máximo se encuentren alrededor de 900 ms.



Figura 123: Consumo de recursos

Los picos en el uso de CPU parecen correlacionarse con los picos en los tiempos de respuesta. Lo cual es razonable, ya que como se envían solicitudes a un ritmo constante, si ocurre un retraso en la obtención de la respuesta desde la API externa, se recibirá muy próxima a la respuesta de otra solicitud, lo que incrementará el uso de CPU aparente en ese lapso de tiempo.

4.4.2. Caché

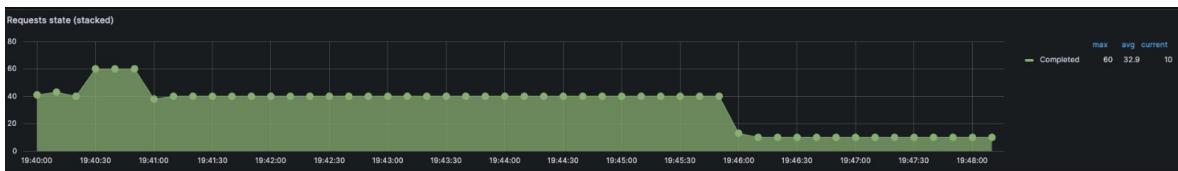


Figura 124: Cantidad de escenarios de carga lanzados



Figura 125: Tipo de respuestas obtenidas

Siguiendo las configuraciones de Artillery antes descritas, obtenemos un caso muy similar al caso sin cache. Lo cual tiene sentido ya que para endurance los atributos que mejora cache ya eran muy buenos.

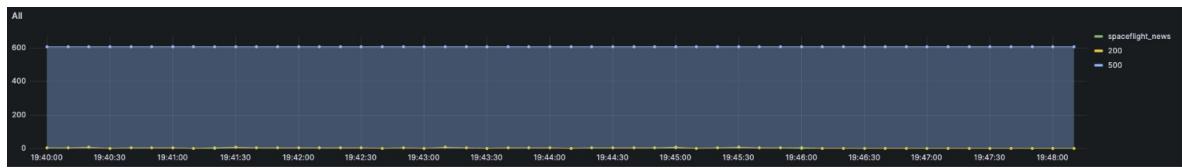


Figura 126: Tiempo de respuesta del lado del server



Figura 127: Tiempo de demora al responder al client-side

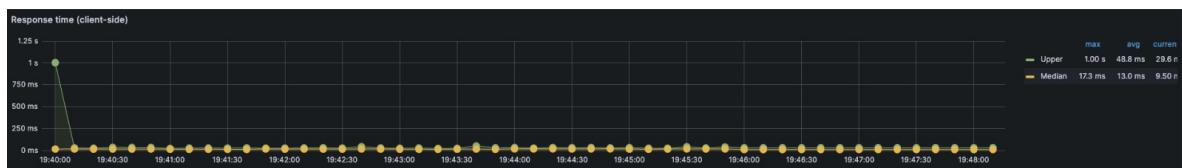


Figura 128: Demora de respuestas (client-side)

De lo cual, al igual que con *metar*, vemos significantes mejores. Y las razones de ellos son las mismas que con *metar*. Tal vez vale la pena notar que requests mas significativamente pesadas como las de *spaceflight_news* justifican aun mas el cache, ya que se reduce enormemente el tiempo de ejecución, y así reduciendo el tiempo que se usa en el server o manteniendo la conexión activa que con *metar*.

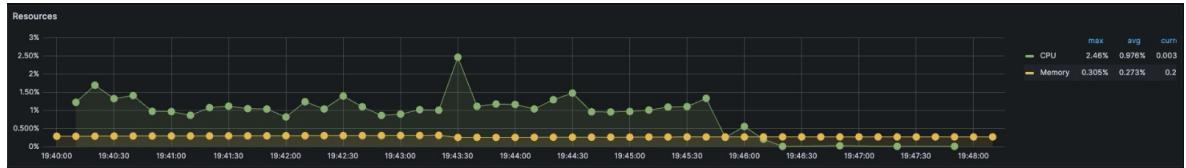


Figura 129: Consumo de recursos

Así como en *metar*, vemos mejoras a nivel de uso cpu por razones ya descritas anteriormente.

4.4.3. Rate limiting

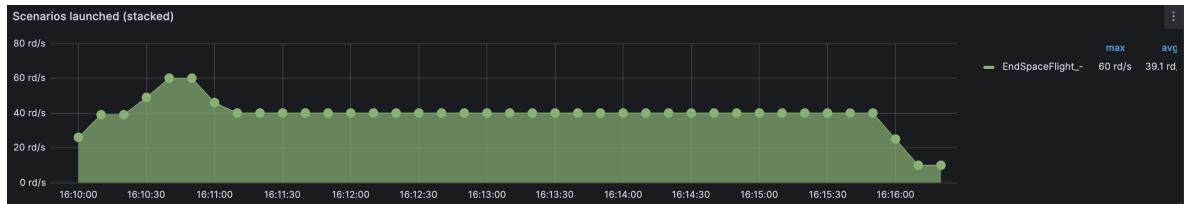


Figura 130: Cantidad de escenarios de carga lanzados

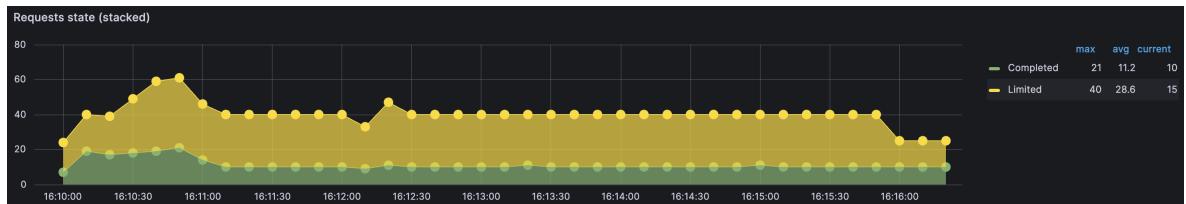


Figura 131: Tipo de respuestas obtenidas

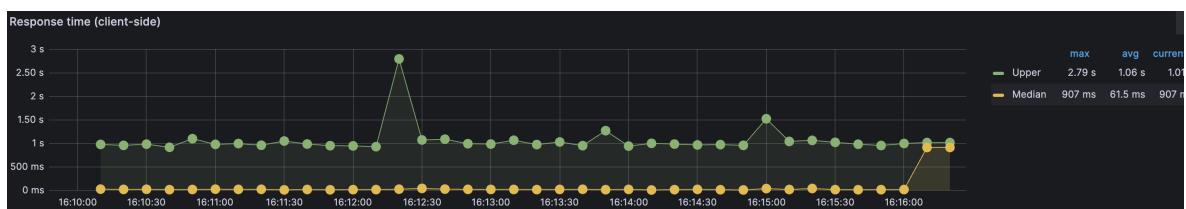


Figura 132: Demora de respuestas (client-side)

Las demoras provienen de la API externa, oscilando continuamente entre los 850 y los 1000 ms. Los tiempos de respuesta son relativamente bajos y constantes, con un pico notorio alrededor de los 2 minutos de iniciada la ejecución.

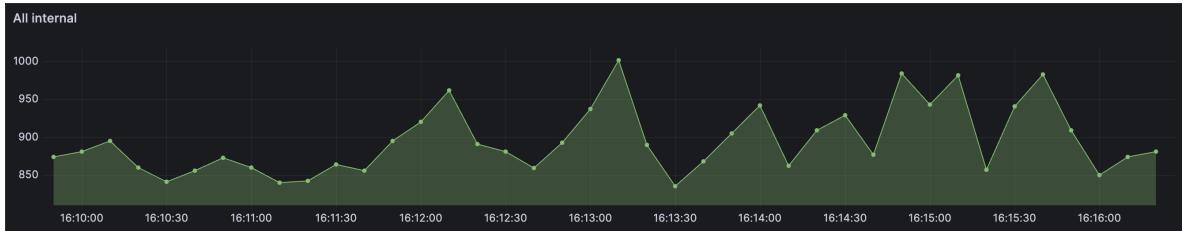


Figura 133: Demora de API externa

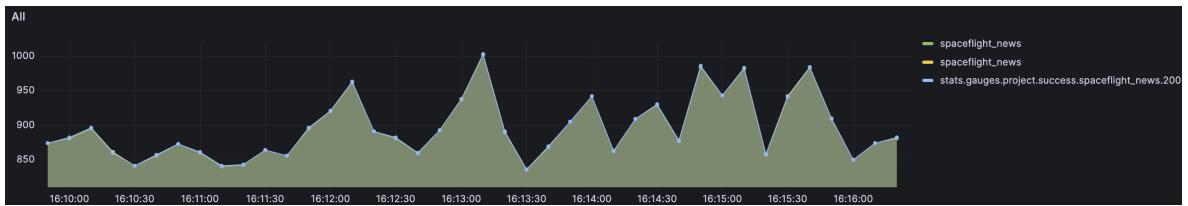


Figura 134: Demora del endpoint

El consumo de CPU mantiene un comportamiento similar al de las demoras.

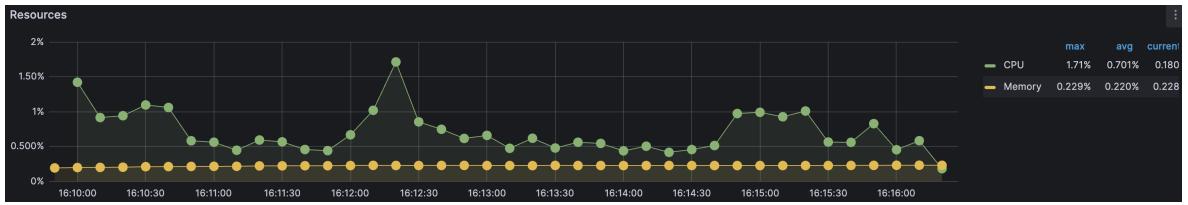


Figura 135: Consumo de recursos

4.4.4. Replicación

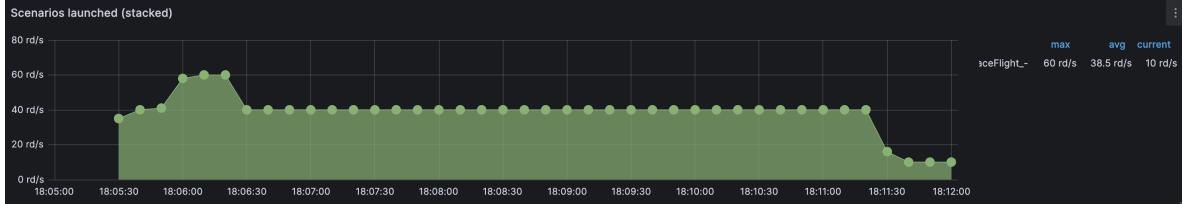


Figura 136: Cantidad de escenarios de carga lanzados

Con esta táctica se completan todas las solicitudes exitosamente y los tiempos de demora rondan alrededor de 1 s (levemente superior al caso base).

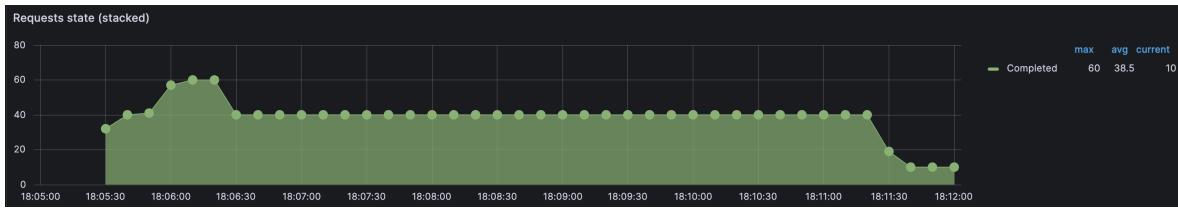


Figura 137: Tipo de respuestas obtenidas

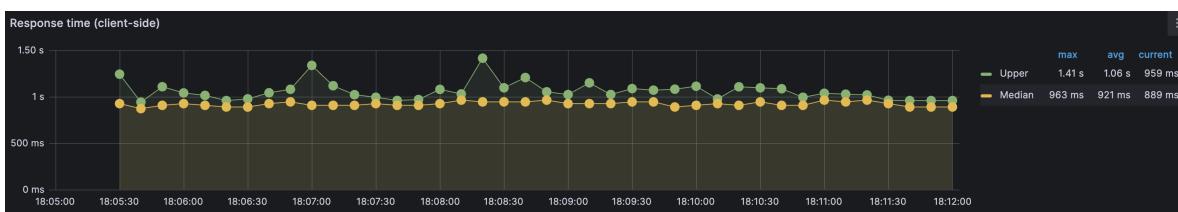


Figura 138: Demora de respuestas (client-side)

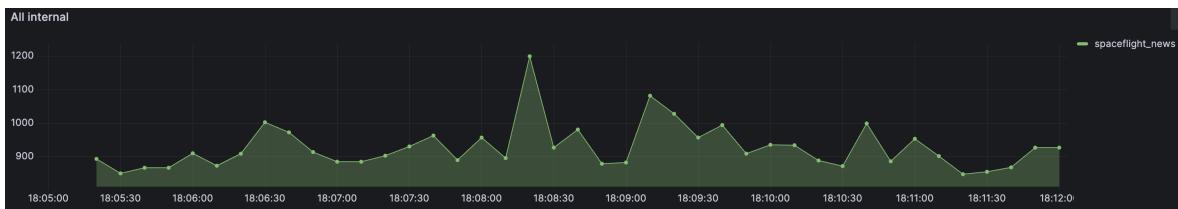


Figura 139: Demora de API externa

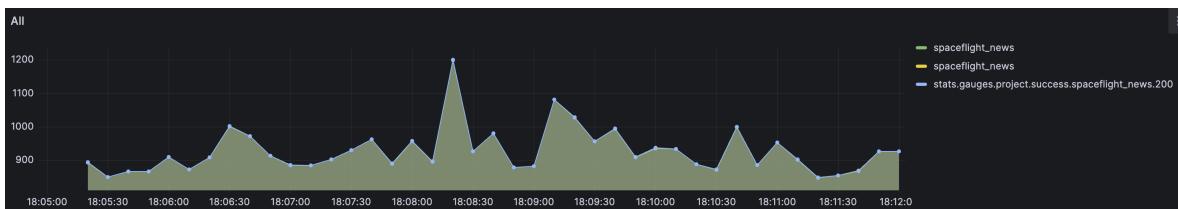


Figura 140: Demora del endpoint

Las demoras del endpoint se atribuyen a las demoras de la API externa, estas no son significativas.

Nuevamente notamos una disminución considerable en el uso de la CPU.

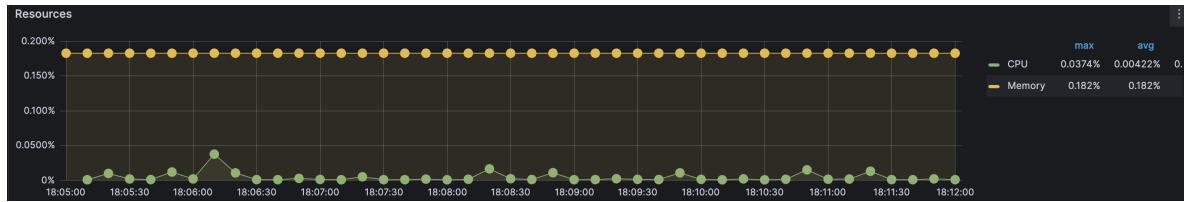


Figura 141: Consumo de recursos

5. Conclusión

Hemos evaluado el modo en que afectan un conjunto de tácticas a casos seleccionados e interpretado sus resultados. Aunque las características que las distintas tácticas impactan es previsible, depende mucho del caso de estudio el grado en el que lo hacen. Hemos observado ejemplos donde aplicar una táctica ni siquiera logra una mejora e incluso empeora los resultados. Más aún, una aplicación descuidada puede actuar completamente en contra del propósito de la API, como hubiese ocurrido al agregar cache a *quote*. Las herramientas brindadas han resultado ser de gran ayuda para experimentar con los cambios y valoramos la realización de este tipo de pruebas antes de aplicar soluciones en un caso futuro.