



Universidad de Buenos Aires
Facultad de Ingeniería

Trabajo Práctico 1

[75.73]
Arquitectura de Software

Grupo ArkiTechs:

Bucca, Martín (109161)

Civini, Diego (107662)

Danko, Tomás (107431)

Nasif, Francisco José (101044)

Indice

Indice.....	2
Introducción.....	3
Ramas Repositorio Github.....	3
Cómo correr escenarios y aplicación.....	3
Escenarios desplegados.....	4
Aclaraciones sobre los Gráficos.....	5
Tácticas.....	6
Caso base.....	6
Atributos de Calidad.....	7
Análisis por Endpoint.....	7
ping.....	7
dictionary.....	10
spaceflight_news.....	14
quote.....	17
Caché.....	20
Atributos de Calidad.....	21
Análisis por Endpoint.....	21
dictionary.....	21
spaceflight_news.....	26
Replicación.....	30
Atributos de Calidad.....	30
Análisis por Endpoint.....	30
ping.....	30
dictionary.....	33
spaceflight_news.....	35
quote.....	37
Rate limiting.....	40
Atributos de Calidad.....	41
Análisis por Endpoint.....	42
ping.....	42
dictionary.....	44
spaceflight_news.....	48
quote.....	51
Conclusiones.....	55
Comparaciones.....	55
Ping.....	55
Dictionary.....	55
Spaceflight News.....	56
Quote.....	56
Lecciones aprendidas.....	57
Referencias.....	58

Introducción

El objetivo de este trabajo práctico fue el de implementar un servicio HTTP en Node.js-Express que represente una API que consume otras APIs para dar información a sus usuarios, similar a lo que brindaría una API para una página de inicio personalizada. Luego, se pidió someter sus endpoints a diversas intensidades/escenarios de carga en algunas configuraciones de deployment, tomar mediciones y analizar resultados. Los endpoints a analizar son:

- /ping: devuelve correctamente con el mensaje “pong”.
- /dictionary?word={word}: devuelve las fonéticas y significados de la palabra dada como parámetro, consultando a la Free Dictionary API.
- /spaceflight news: devuelve los últimos 5 títulos de noticias sobre actividad espacial, consultando a la Spaceflight News API.
- /quote: devuelve una frase aleatoria sobre tecnología, consultando a la Techy API.

A lo largo del desarrollo de este trabajo, pudimos poner en práctica y ganar experiencia en tecnologías como NodeJs, Express, Docker, Docker Compose, Nginx, Redis, Artillery, Cadvisor, Graphite y Grafana.

Ramas Repositorio Github

En el [repositorio de github](#) se encuentra la rama main y 4 ramas que representan cada una de las tácticas implementadas en el trabajo:

- Rama “base”: Implementación del caso base.
- Rama “redis”: Implementación de la táctica “cache”.
- Rama “rate-limiting”: Implementación de la táctica “Rate Limiting”.
- Rama “replication”: Implementación de la táctica “replicación”.

Cómo correr escenarios y aplicación

Correr con docker

Una vez ubicado en la rama en la que se quiere trabajar, existen dos comandos para levantar y borrar toda la infraestructura con Docker compose:

- ./run.sh: Hace `docker compose -f docker-compose.yml up -build`
- ./stop.sh: Hace
 - `docker compose -f docker-compose.yml stop -t 5`
 - `docker compose -f docker-compose.yml down`

O también se puede `correr docker compose up -build` para levantar todo y `docker compose stop` para frenar todo sin que se borren los contenedores.

Correr pruebas de carga con artillery

Para correr las pruebas de carga para cada uno de los endpoints se deberá primero tener corriendo el servidor y luego correr ***dentro de la carpeta perf***

`./run-scenario.sh <nombre de la prueba> api`

Para ping: `./run-scenario.sh ping api`

Para dictionary*: `./run-scenario.sh dict api`

Para spaceflight_news: `./run-scenario.sh space api`

Para quote: `./run-scenario.sh quote api`

* En caso de querer correr las pruebas para dictionary con un archivo .csv de palabras, se deberá descargar el [archivo](#) (editar según cantidad de palabras deseadas) y colocar en /perf/store/static/files.

Escenarios desplegados

Propiedad	Configuración de las fases					
	1ra Plain	2da Ramp	3ra Plain 2	4ta Slide	5ta Spike	6ta Plain 3
<i>duration</i> (s)	80	20	180	80	30	90
<i>arrivalRate</i> (req / s)	3	3	25	25	40	3
<i>rampTo</i> (req)	-	25	-	2	-	-

Para las pruebas de carga, decidimos usar la misma configuración para todas los endpoints ya que nos permite poder comparar cómo se comportan los distintos endpoints de nuestra aplicación a los mismos escenarios de carga.

Para cada escenario fuimos probando distintos valores de carga y llegamos a lo siguiente:

1. La primera fase “Plain”, se buscó someter al endpoint a una carga constante de 3 requests por segundos durante un tiempo de 20 segundos.
2. En la segunda fase “Ramp”, se buscó incrementar linealmente la carga durante 80 segundos, llevándola de 3 request a 25 requests por segundos.

3. En la tercer fase “Plain 2”, se buscó mantener una carga constante alta de 25 requests por segundo durante un tiempo largo de 180 segundos para ver cómo se comportan los endpoints ante esta carga durante un tiempo considerable que nos permita hacer buenos análisis.
4. En la cuarta fase “Slide”, se buscó disminuir la carga linealmente a lo largo de 80 segundos para pasar de 25 a 2 requests por segundo.
5. En esta fase “Spike”, se busca generar un “pico” en la carga. Se pasa abruptamente la carga de 2 a 40 requests por segundo y se mantiene enviando esa carga por 30 segundos.
6. En la última etapa “Plain 3”, se vuelve a disminuir la carga a 3 requests por segundos y se somete el endpoint por 90 segundos más.

En total entonces nos quedaría un escenario de 8 minutos en total con 6 fases con diferentes configuraciones.

En el caso particular de dictionary, la carga generada envía diferentes palabras obtenidas de forma aleatoria de este [archivo csv](#) que contiene diversas palabras aceptadas por la API externa.

Aclaraciones sobre los Gráficos

Para cada una de las tácticas probadas en este Trabajo Práctico y cada uno de los endpoints implementados, mostramos y analizamos 6 gráficos con métricas sobre distintos aspectos a tener en cuenta:

- **Escenarios Desplegados:** Muestra la cantidad de requests enviados en función del tiempo. El gráfico muestra cada 10 segundos, es por eso que “acumula” y multiplica por 10 ya que está mostrando las requests de los últimos 10 segundos.
- **Estado de las Requests:** Este gráfico muestra el estado de las requests en función del tiempo. Para un determinado momento, se puede observar en qué estado resultaron las requests. Los posibles estados son *Errored*, *Completed*, *Limited* y *API-Limit*.
- **Tiempo de Respuesta (percibido por el cliente):** Muestra el tiempo de respuesta percibido por el cliente en función del tiempo. Muestra cuál fue el tiempo máximo de todas las requests enviadas en un instante dado del gráfico y la mediana. Se muestra el tiempo en milisegundos (ms) o segundos (s) dependiendo el endpoint y táctica.
- **Recursos Utilizados:** Se muestra el porcentaje de memoria y CPU utilizado en función del tiempo. Se muestra el porcentaje en función a los recursos totales que fueron asignados al contender corriendo la aplicación.
- **Latencia del Endpoint:** Mide la demora de cada endpoint en responder (API remota + procesamiento propio) en función del tiempo. Muestra la mediana y el máximo tiempo en cada instante determinado. Se muestra en milisegundos (ms).

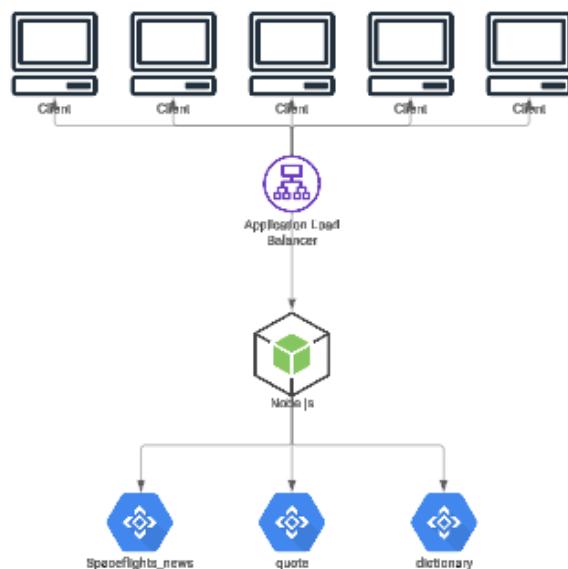
- **Latencia de la API externa***: Mide la demora de cada API externa en responder en función del tiempo. Muestra la mediana y el máximo tiempo en cada instante determinado. Se muestra en milisegundos (ms).

* El endpoint /ping no tiene API externa.

Tácticas

Caso base

En el caso base tenemos una sola instancia de la aplicación node corriendo y un nodo de nginx funcionando como load balancer. Hay 3 endpoints que, a su vez, envían peticiones a APIs externas para generar la respuesta. Por último tendremos un conjunto de clientes haciendo requests. Vamos a usar esta táctica sólo para tomar como referencia y poder verificar si hay mejoras en comparación con el resto de las tácticas.



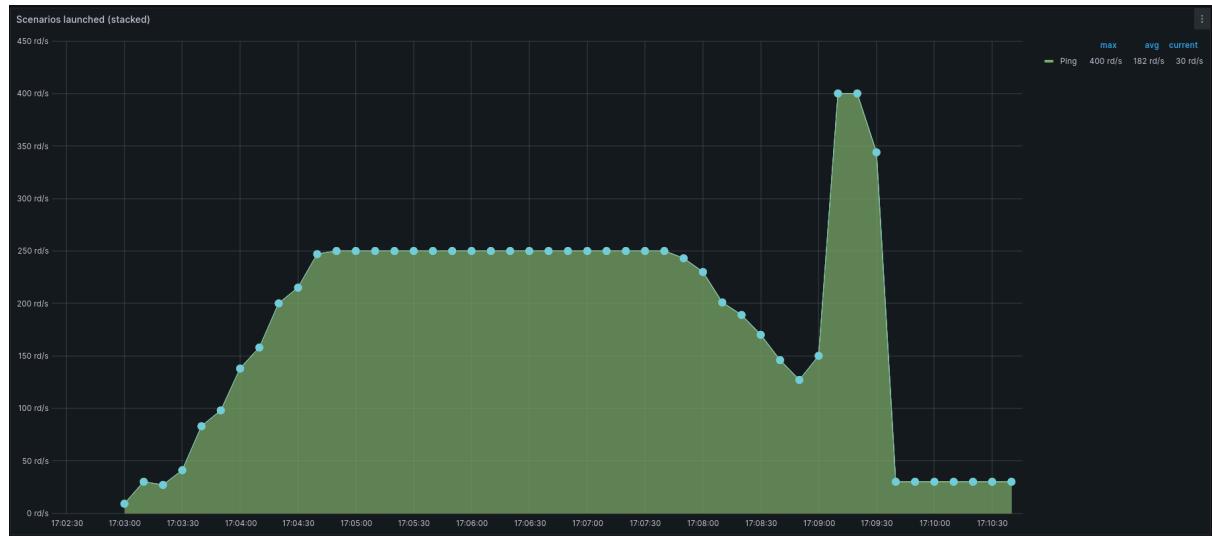
Atributos de Calidad

- **Performance**: En el caso base es importante tener en cuenta el rendimiento ya que sirve para usar como “baseline” y comparar el impacto del resto de las tácticas.
- **Reliability**: La aplicación debería funcionar sin errores en el caso base. Se tiene que asegurar que funcione en el caso más simple de todos.

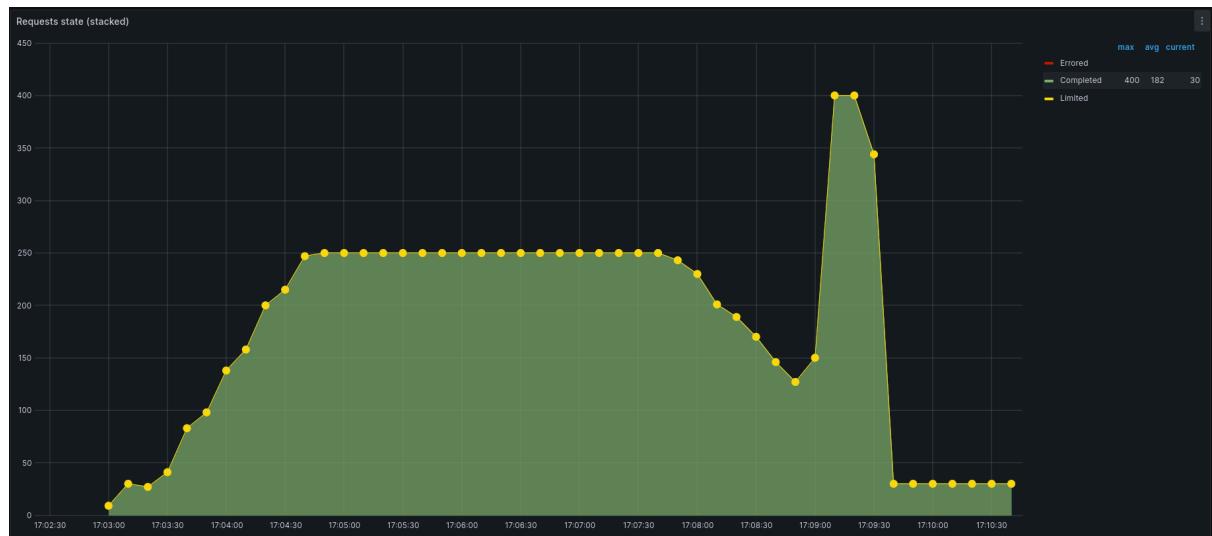
Análisis por Endpoint

ping

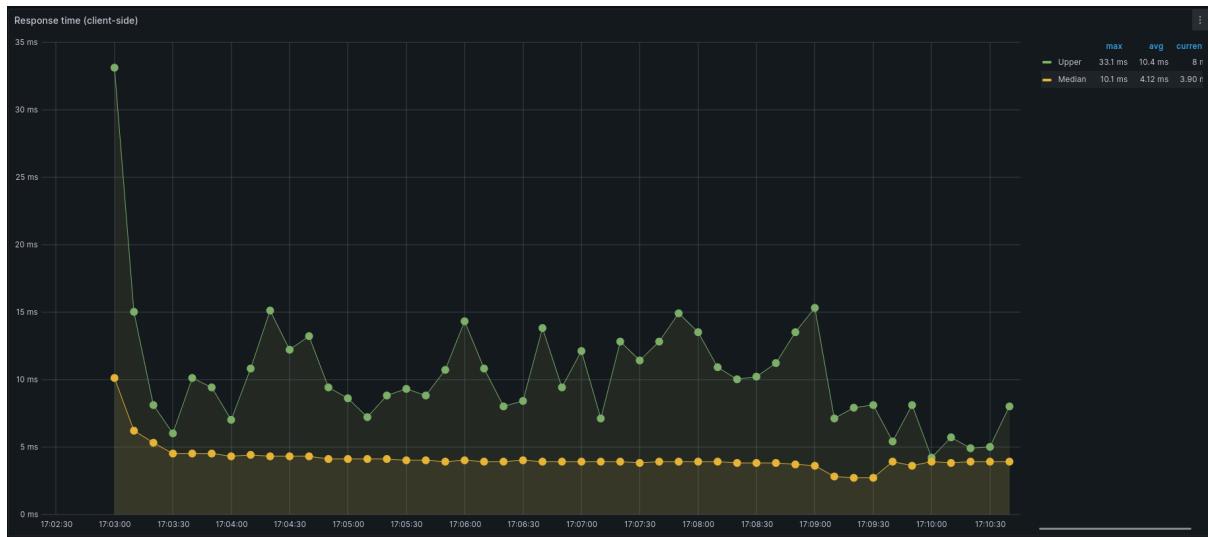
Escenarios desplegados



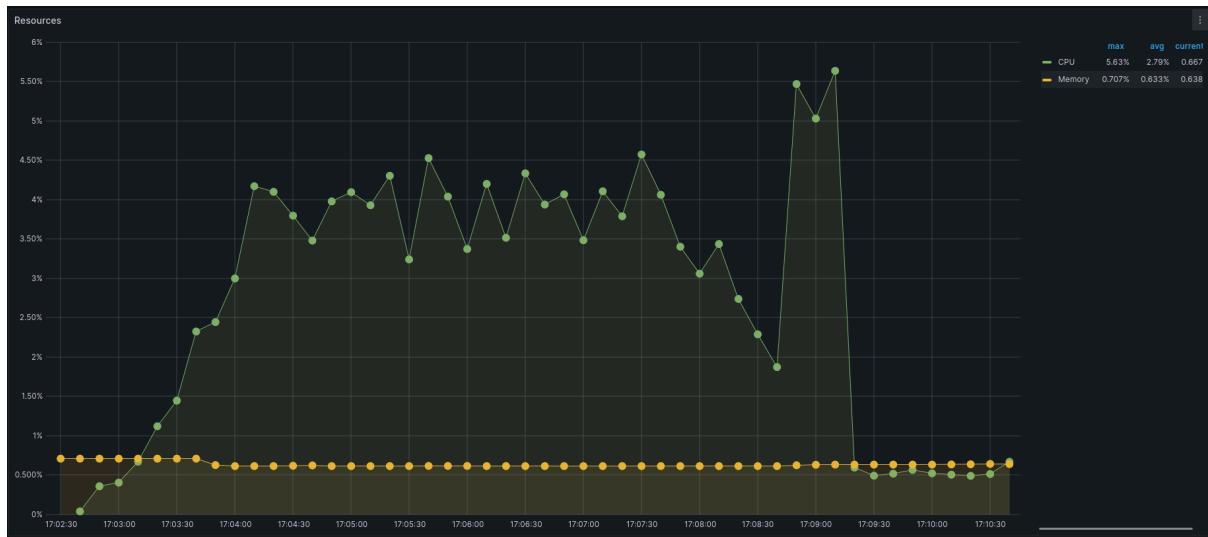
Estado de las requests



Tiempo de respuesta (lado del cliente)



Recursos utilizados



Latencia del Endpoint

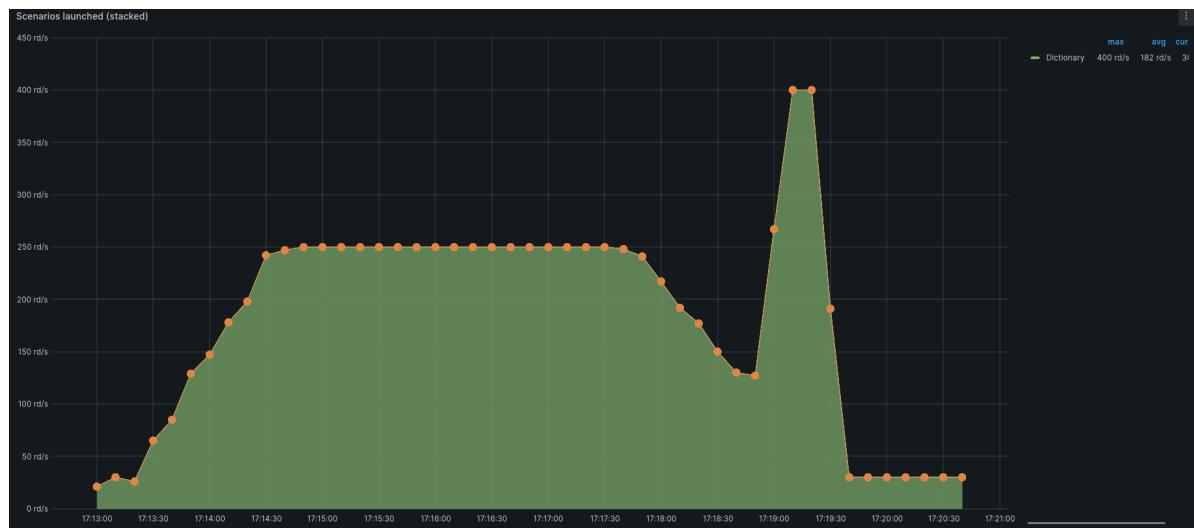


Comentarios

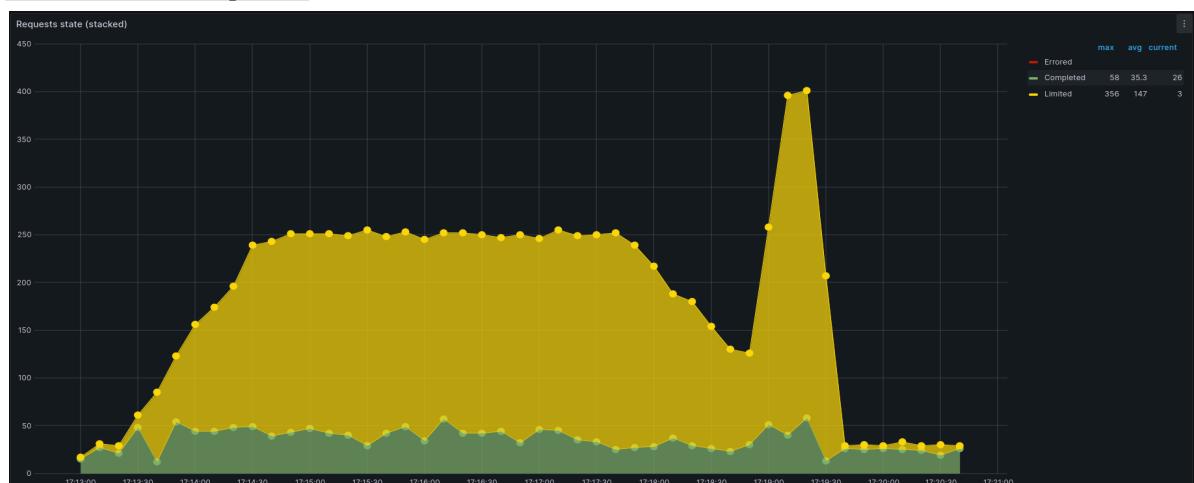
Podemos ver que el gráfico de requests enviadas se condice con el gráfico de requests respondidas, siendo que en este caso todas las requests pudieron responderse correctamente. Esto no es de extrañar, pues el procesamiento requerido para replicar a una request de este tipo es mínimo. Respecto al tiempo de respuesta percibido por el cliente y la latencia del Endpoint del lado del servidor, vemos que nuevamente se condice con los gráficos anteriores. Los **tiempos son muy pequeños, pues el procesamiento es mínimo**. Sobre el consumo de recursos, vemos que respecto al CPU el consumo respeta un patrón similar a los escenarios propuestos. Por otro lado, la memoria se mantiene constante; esto no es de extrañar, pues el procesamiento de este Endpoint no requiere de ningún almacenamiento en memoria considerable, pero sí requiere un procesamiento de CPU, por más pequeño que sea.

dictionary

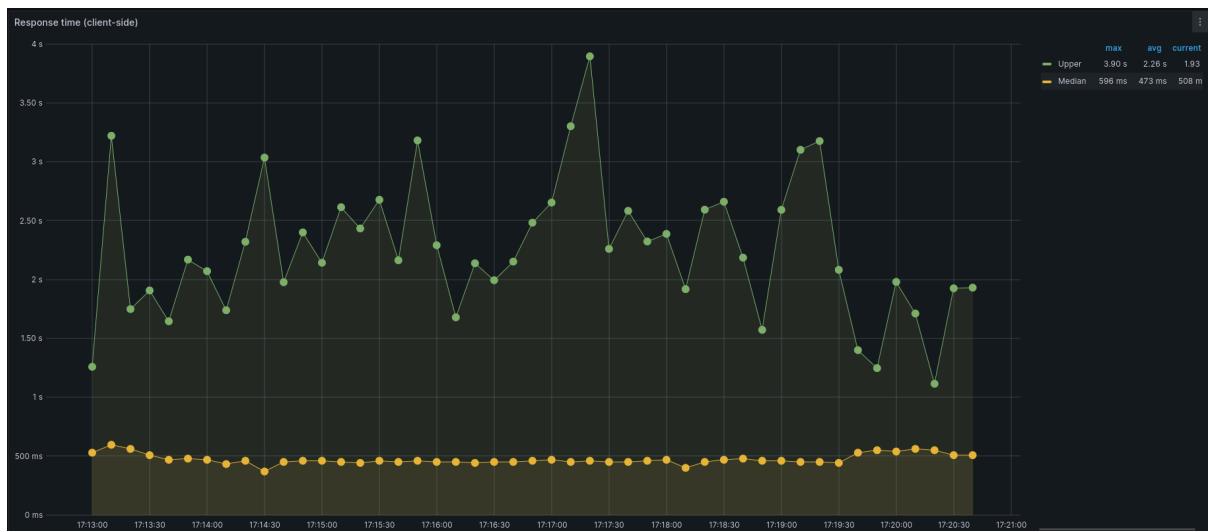
Escenarios desplegados



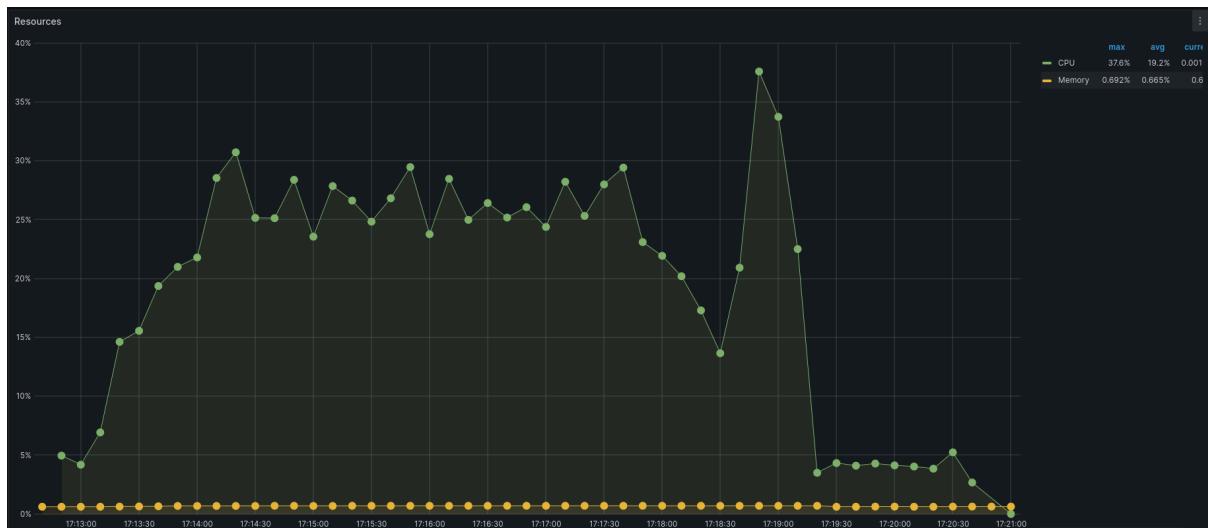
Estado de las requests



Tiempo de respuesta (lado del cliente)



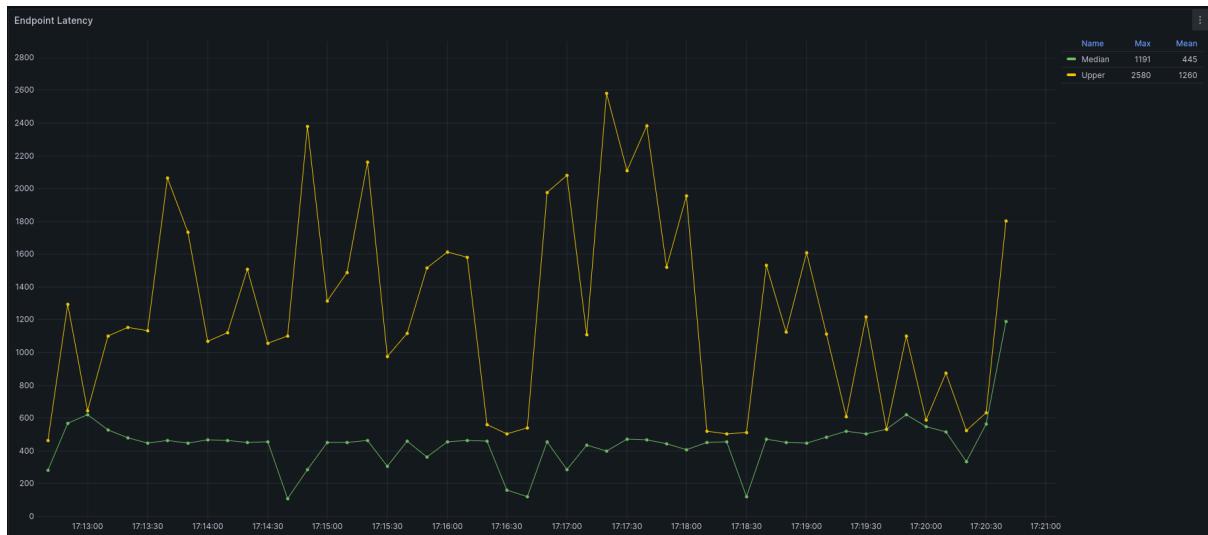
Recursos utilizados



Latencia de la API externa



Latencia del Endpoint



Reporte de Artillery

```
Summary report @ 17:20:45(-0300)
-----
http.codes.200: ..... 1701
http.codes.500: ..... 6911
http.request_rate: ..... 18/sec
http.requests: ..... 8612
http.response_time:
  min: ..... 32
  max: ..... 3893
  median: ..... 459.5
  p95: ..... 1107.9
  p99: ..... 1790.4
http.responses: ..... 8612
vusers.completed: ..... 8612
vusers.created: ..... 8612
vusers.created_by_name.Dictionary: ..... 8612
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 34.5
  max: ..... 3895.6
  median: ..... 459.5
  p95: ..... 1107.9
  p99: ..... 1826.6
```

Comentarios

Para este Endpoint, los resultados son considerablemente distintos a **/ping**. Respecto al estado de las requests, vemos que una gran parte fueron limitadas por parte de la API externa utilizada, que ya impone una frontera desde el primer escenario, en el que la tasa de llegada de requests es de **3 por segundo**. Esto nos indica que la misma tiene una implementación de Rate Limiting incorporada.

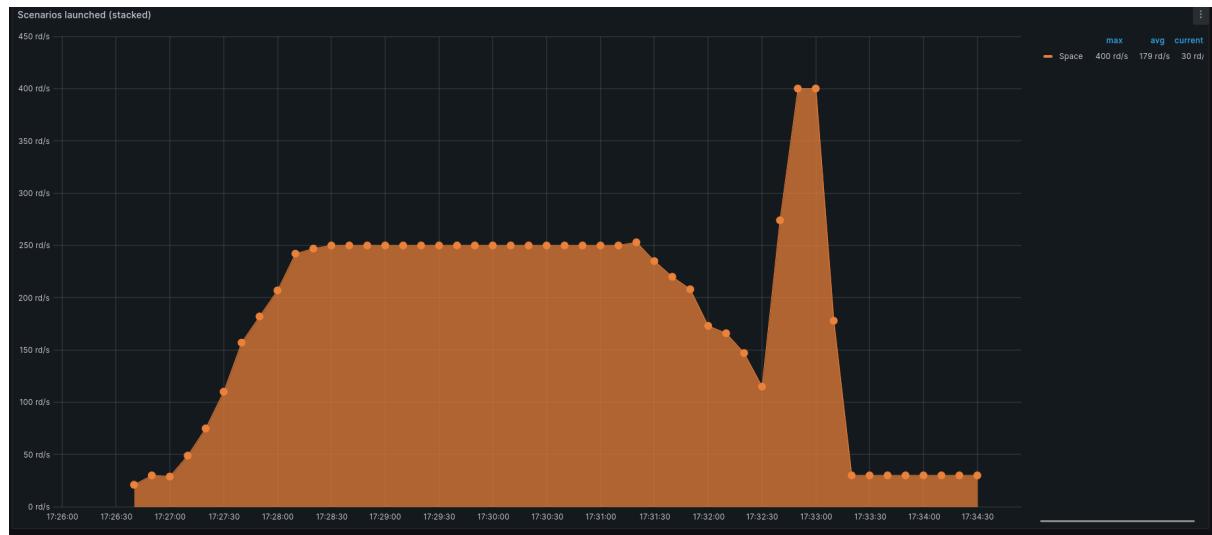
Respecto al tiempo de respuesta percibido por el cliente y la latencia del Endpoint del lado del servidor, vemos que llegamos a tiempos del orden de los segundos para varias requests, algo que no es de extrañar teniendo en cuenta que estamos recurriendo a una API externa. Además, las gráficas de latencia de la consulta a la API externa y de respuesta de nuestro Endpoint son muy similares, lo cual se explica si tenemos en

cuenta que el procesamiento del servidor es despreciable comparado a la consulta en términos temporales.

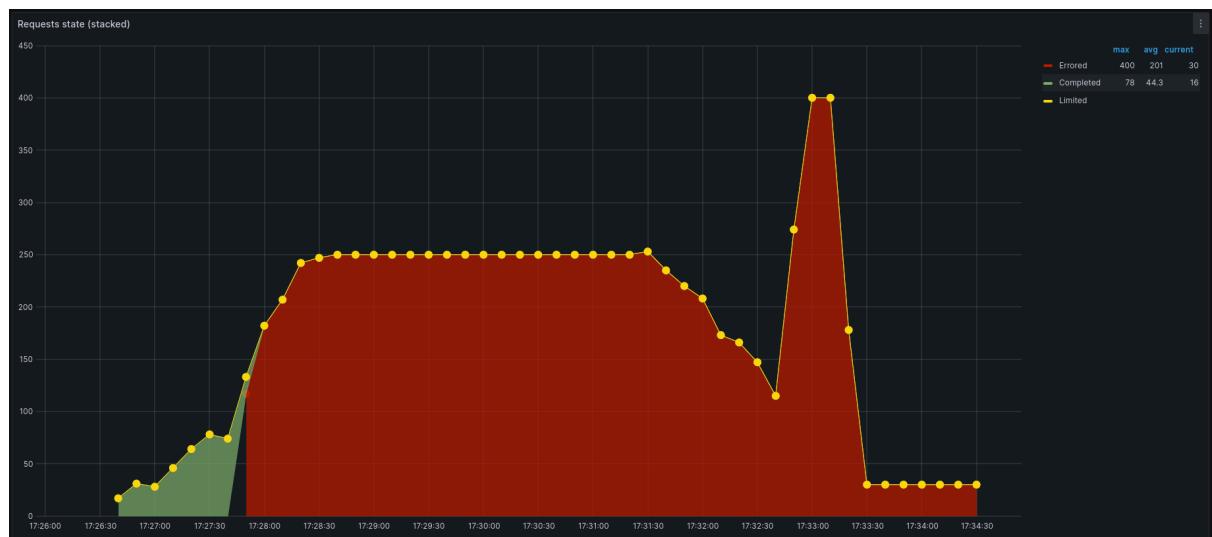
Sobre el consumo de recursos, tenemos un caso similar al de **/ping**: el CPU respeta un patrón similar a los escenarios propuestos, pero el uso de memoria se mantiene constante.

spaceflight_news

Escenarios desplegados



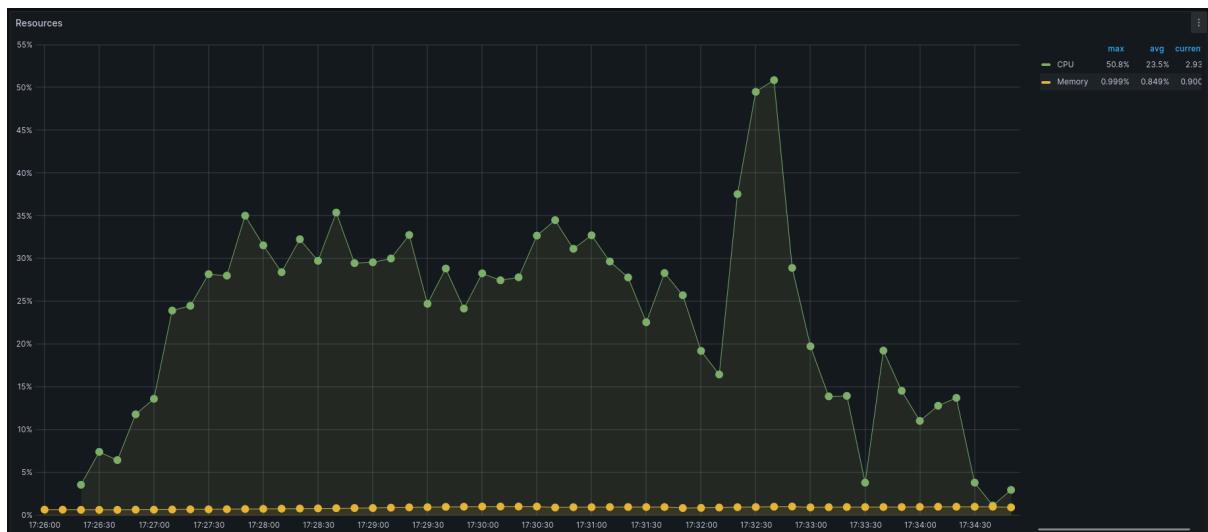
Estado de las requests



Tiempo de respuesta (lado del cliente)



Recursos utilizados



Latencia de la API externa



Latencia del Endpoint



Reporte de Artillery

```
Summary report @ 17:34:34(-0300)

errors.ETIMEDOUT: ..... 8268
http.codes.200: ..... 354
http.request_rate: ..... 18/sec
http.requests: ..... 8622
http.response_time:
  min: ..... 514
  max: ..... 9890
  median: ..... 2369
  p95: ..... 8186.6
  p99: ..... 9607.1
http.responses: ..... 354
vusers.completed: ..... 354
vusers.created: ..... 8622
vusers.created_by_name.Space: ..... 8622
vusers.failed: ..... 8268
vusers.session_length:
  min: ..... 516
  max: ..... 9891.4
  median: ..... 2369
  p95: ..... 8186.6
  p99: ..... 9607.1
```

Comentarios

Vemos algo muy interesante sobre el estado de las requests. A partir de la mitad de la fase de 'Ramp', todas las requests enviadas son rechazadas. Si hilamos un poco más fino y prestamos atención al código de error recibido, vemos lo siguiente:

```
nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)" "nginx-1 | 172.18.0.1 - - [01/Oct/2024:21:15:33 +0000] "GET /spaceflight_news HTTP/1.1" 499 0 "-" "Artillery (https://artillery.io)"
```

Como podemos ver, obtenemos un error de **código 499**, que es típico en Nginx cuando las conexiones de varios clientes tardan mucho tiempo en procesarse y empiezan a

acumularse los hilos, razón por la cual llegado un punto Nginx comienza a cerrar dichas conexiones para no ralentizar al resto de clientes.

Esto cobra más sentido una vez analizados los gráficos de tiempo de respuesta percibido por el cliente y la latencia del Endpoint del lado del servidor. Por el lado del cliente, vemos que para las requests completadas se llega a tiempos de respuesta de hasta **10 segundos**, tiempo considerablemente grande, hasta que se dejan de recibir a causa de la administración de Nginx. Por el lado de nuestro servidor, la latencia del Endpoint llega hasta los **200 segundos**, al seguir intentando procesar estas requests enviadas. Es interesante notar que el gráfico de la latencia de la API externa no es continuo y se deja de almacenar estos valores durante gran parte de la simulación, pues se llega a un punto en el que el propio axios devuelve error en la request:

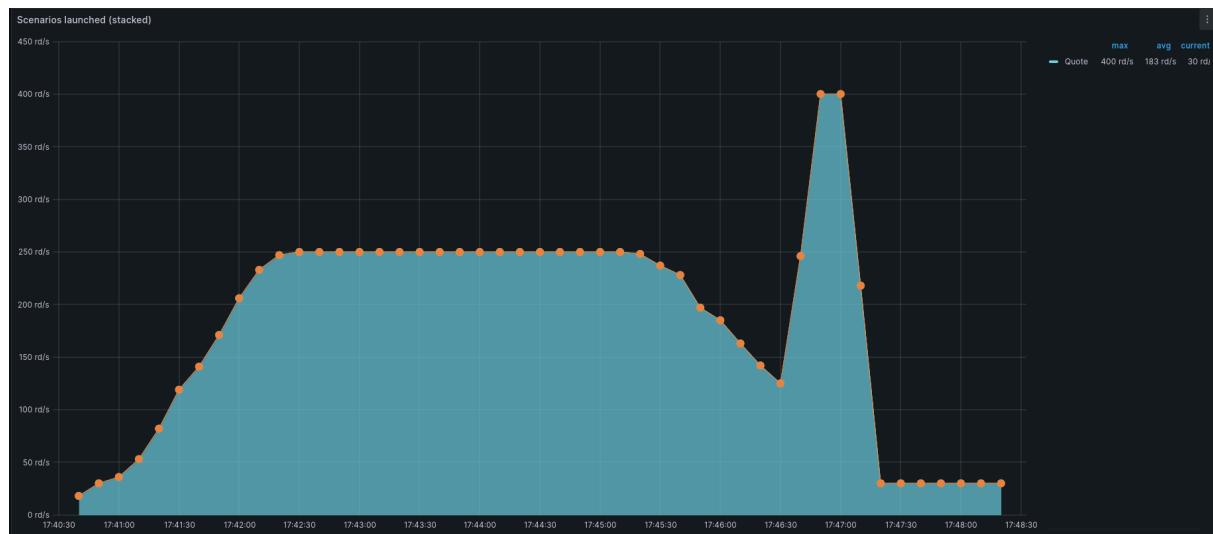
```
app |   response: {  
app |     status: 504,  
app |     statusText: 'Gateway Time-out',
```

Este error, de **código 504**, nos indica un problema de conexión: axios deja de procesar dicha request pues se alcanza un **time-out** y no se puede alcanzar la API externa.

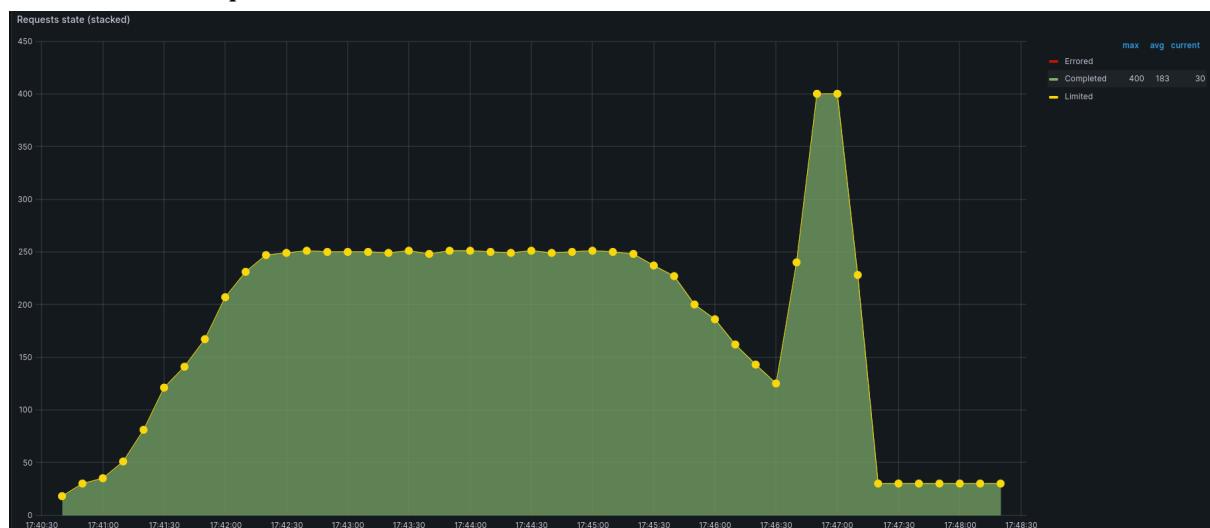
Sobre el consumo de recursos, se repiten los patrones anteriores: el CPU respeta un patrón similar a los escenarios propuestos, pero el uso de memoria se mantiene constante.

quote

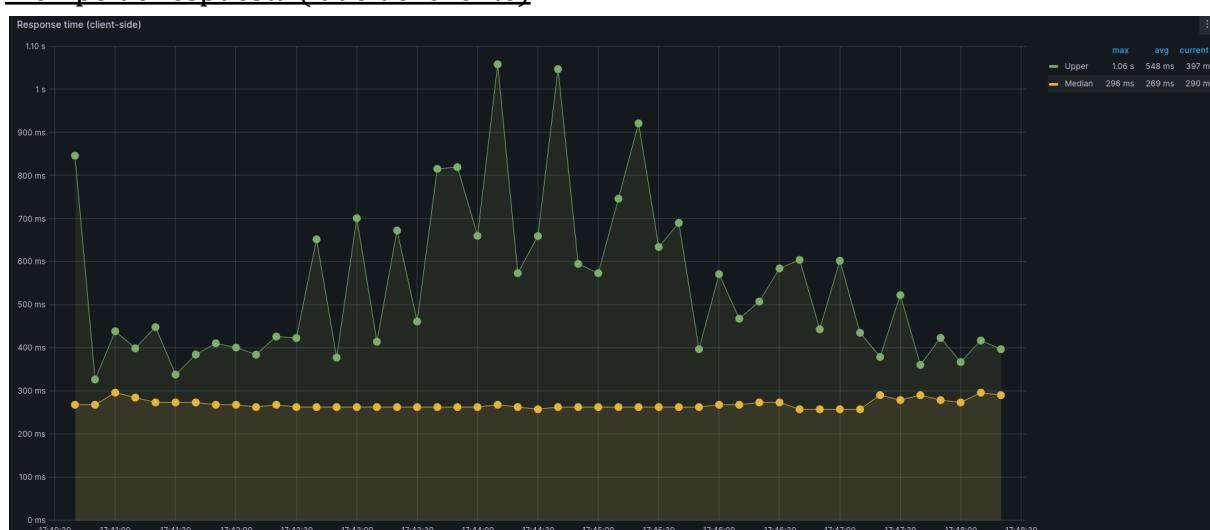
Escenarios desplegados



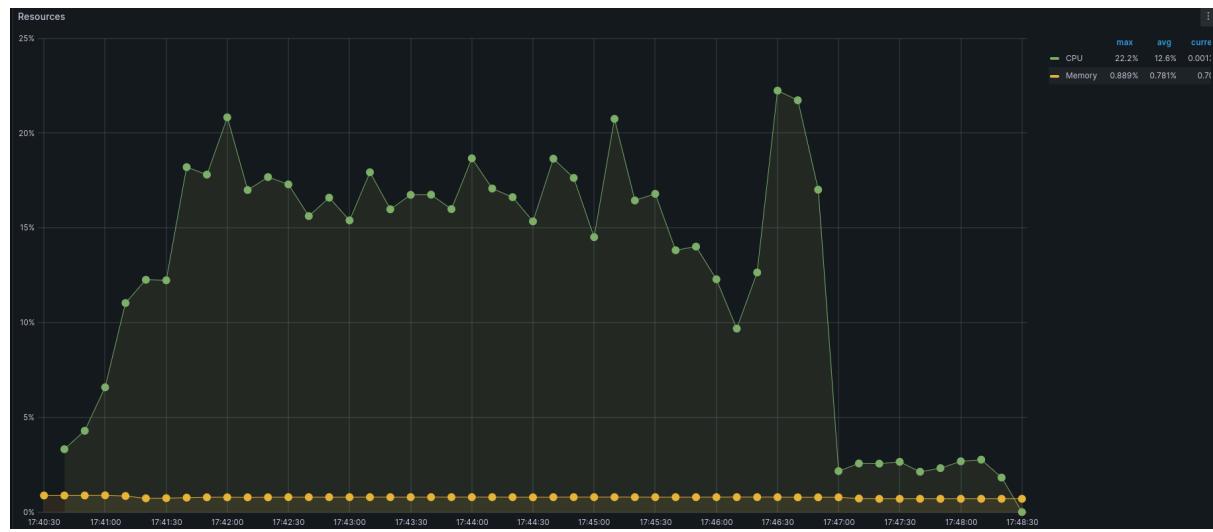
Estado de las requests



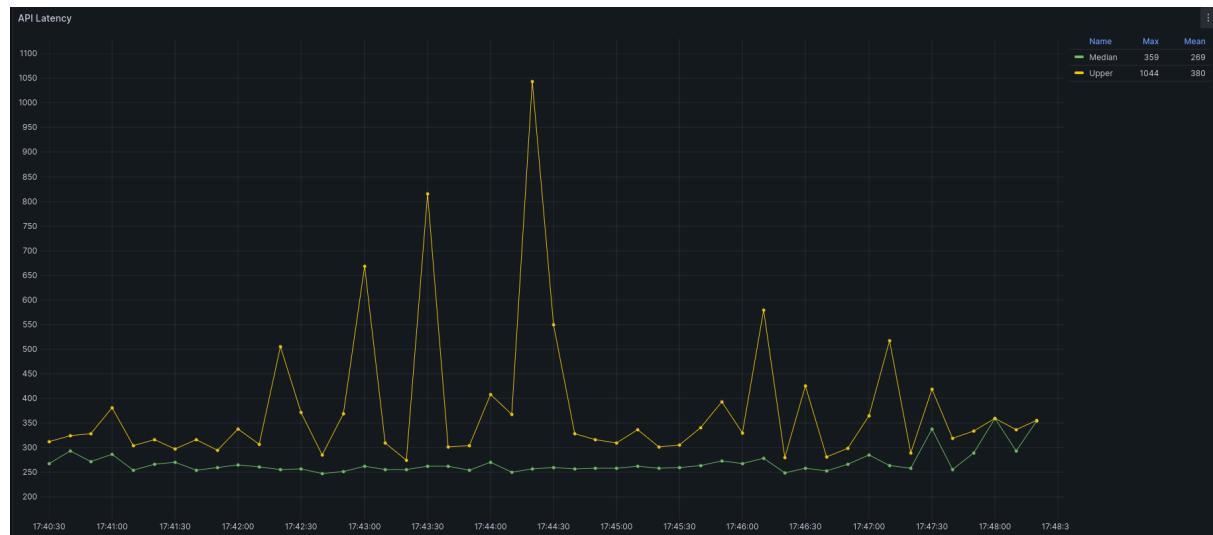
Tiempo de respuesta (lado del cliente)



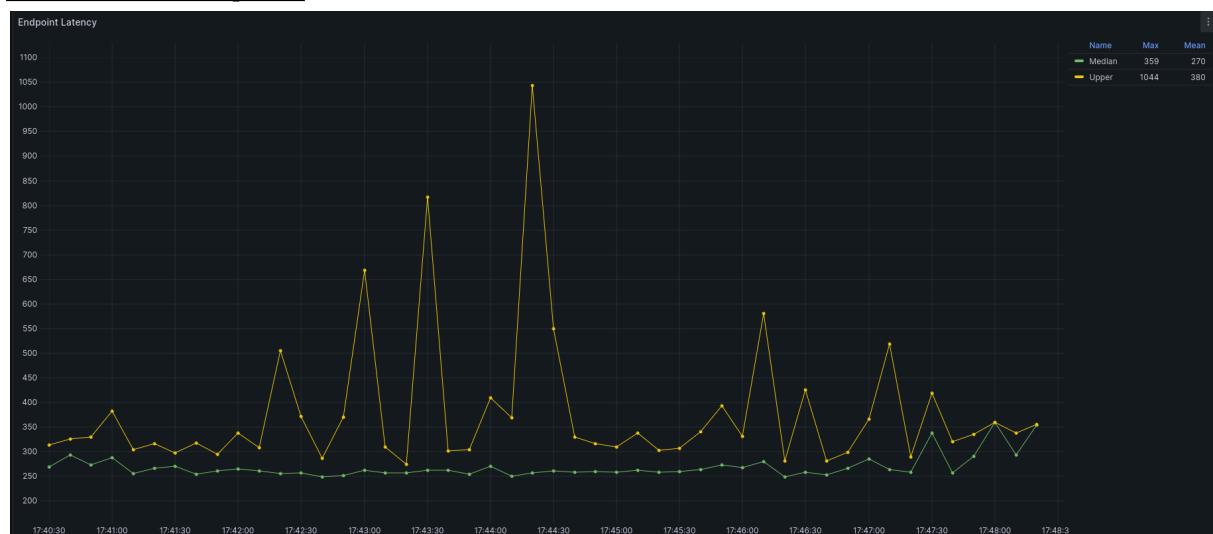
Recursos utilizados



Latencia de la API externa



Latencia del Endpoint



Comentarios

Para este Endpoint, respecto al estado de las requests, podemos ver que todas las requests enviadas pudieron ser respondidas con éxito, lo cual nos indica que la API externa no tiene ningún tipo de Rate Limiting como ocurrió en [/dictionary](#), ni una demora excesiva como en el caso de [/spaceflight_news](#).

Respecto al tiempo de respuesta percibido por el cliente, vemos que la mediana se mantiene por debajo de los **300 milisegundos**, y se llegan a alcanzar valores de hasta poco más de un segundo para algunas requests, que coinciden con la fase 'Plain 2' y tienen sentido si consideramos que se debe procesar una cantidad alta y constante de requests. En cuanto a la latencia de la API externa y nuestro Endpoint, vemos que las diferencias son prácticamente nulas, dado que el aporte del procesamiento de nuestro servidor a la latencia es despreciable. Es interesante destacar que los picos en estos gráficos se correlacionan con los picos en el gráfico de tiempo de respuesta percibido por el cliente, y nos dan a entender que dicho tiempo está dado principalmente por el tiempo que se tarda en consultar a la API externa.

Sobre el consumo de recursos, tenemos un caso similar a los anteriores: el CPU respeta un patrón similar a los escenarios propuestos, pero el uso de memoria se mantiene constante.

Caché

Se agrega un caché redis para poder almacenar temporalmente los datos y accederlos más rápido en caso de que los mismos recursos quieran ser accedidos en un corto periodo de tiempo. Esto es especialmente útil en tanto queramos consultar cierta información que no suele cambiar de manera frecuente. De tal forma, podemos mejorar la performance y eficiencia de nuestra aplicación, al no tener que recurrir constantemente a APIs externas. Para cada uno de los endpoints se utilizó un criterio diferente, considerando sus distintas características:

- **Ping:** No requiere cache ya que no se consulta a API externa
- **Spaceflight_news:** Se consulta a la API externa de antemano y se guarda en caché las noticias por un tiempo de 10 segundos. Una vez transcurridos esos 10 segundos, se vuelve a popular el caché con la información que nos da la API externa; en este sentido, nos inclinamos por *Active Population*. Si bien la documentación de esta API dice que [cada 10 minutos chequea por información nueva](#), decidimos usar ese tiempo para evitar dar noticias que podrían haber quedado viejas por más de 10 segundos. Siempre va a guardar las últimas 5 noticias y en caso de que las tenga en caché, va a devolver esas sin necesidad de consultar a la API externa.
- **Quote:** Decidimos no implementar este endpoint con caché ya que, al ser aleatorio y depender directamente de la aleatoriedad de la API externa, no vale la pena cachear la información obtenida.

- **Dictionary:** A medida que las palabras son solicitadas son guardadas en caché por un tiempo de 10 horas. Si se solicita una palabra que ya está en caché se devuelve la información sin necesidad de consultar a la API externa. En este caso, nos inclinamos por *Lazy Population*, teniendo en cuenta que se puede consultar por varias palabras distintas y no sabemos cuáles se suelen pedir con más frecuencia.

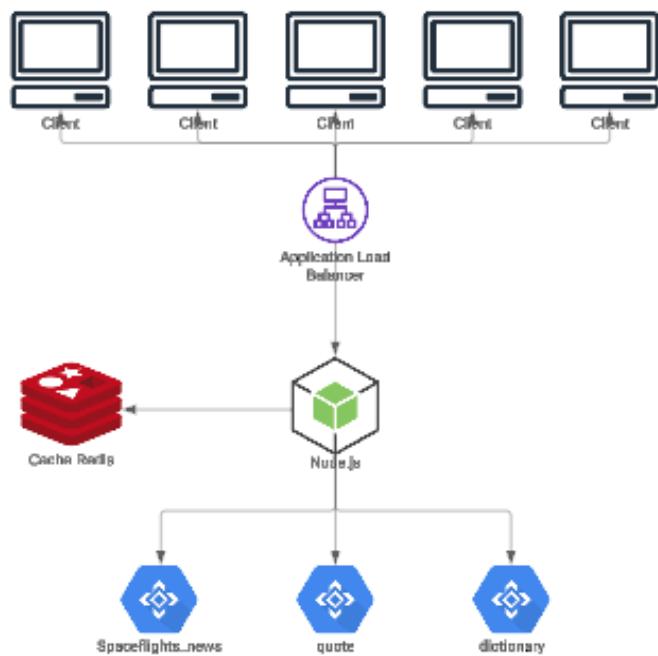
En cuanto a la implementación de *Active Population*, se hizo uso de la función `setInterval()` en Java que permite ejecutar otra instrucción cada un cierto tiempo:

```
const intervalID = setInterval(fetchSpaceflight, 10 * 1000);
await fetchSpaceflight();
```

De esta manera, pudimos realizar consultas de manera periódica a la API externa sin necesidad de que un cliente realice un pedido por dicha información previamente.

Por otro lado, para implementar *Lazy Population* solo requirió hacer un chequeo de la información en caché antes de consultar a la API externa, y en caso de tener lo necesario para responder la request evitar dicha consulta:

```
// check if word is in cache
const cachedWord = await client.get(`dictionary:${word}`);
if (cachedWord) {
  return res.json(JSON.parse(cachedWord));
}
```



Atributos de Calidad

- **Performance:** El caché debería mejorar notablemente el tiempo de respuesta al guardar la información en memoria y evitar tener que consultar a una API externa por cada request.
- **Escalabilidad:** Debería poder manejar más requests y optimizar recursos al ahorrar requests a APIs externas. Vemos que hace al sistema más escalable ante una mayor carga.

Análisis por Endpoint

ping

El endpoint /ping con caché no trajo ninguna conclusión ya que por obvias razones no se usó cache, y por ende se trataba de una situación exactamente igual a la del caso base.

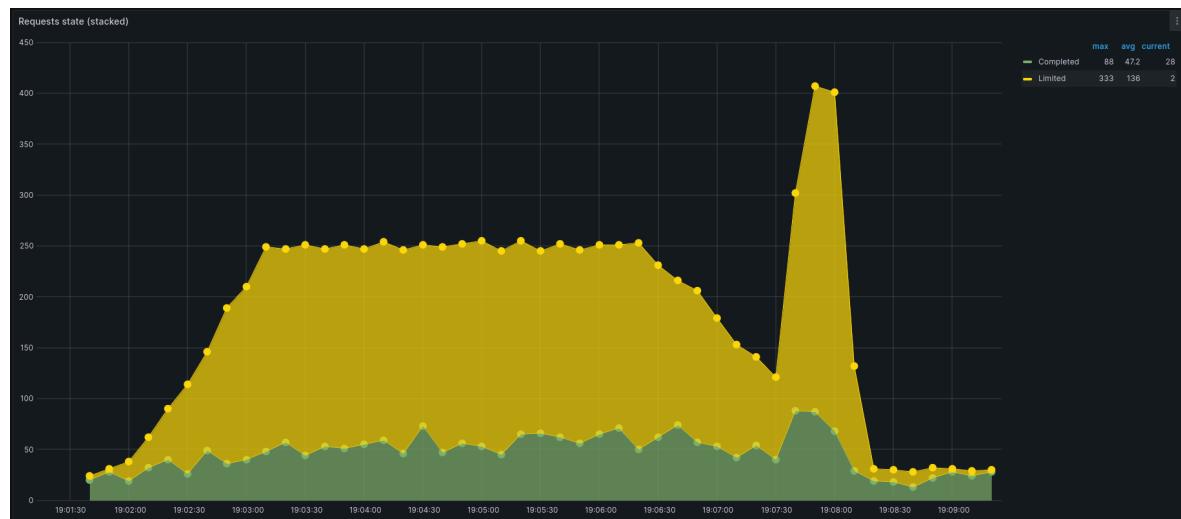
dictionary

Caso 1 (10000 palabras)

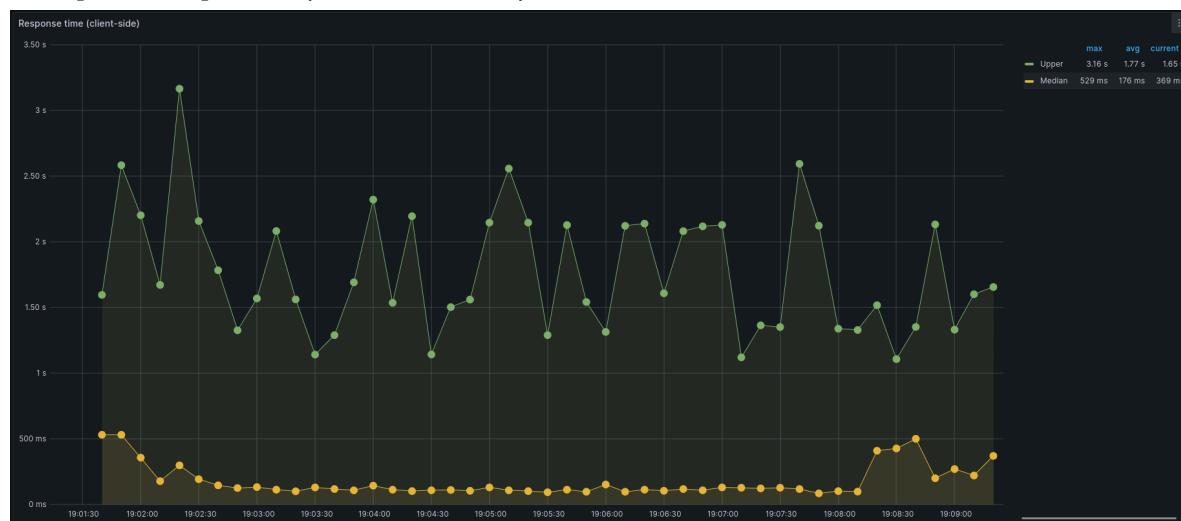
Escenarios desplegados



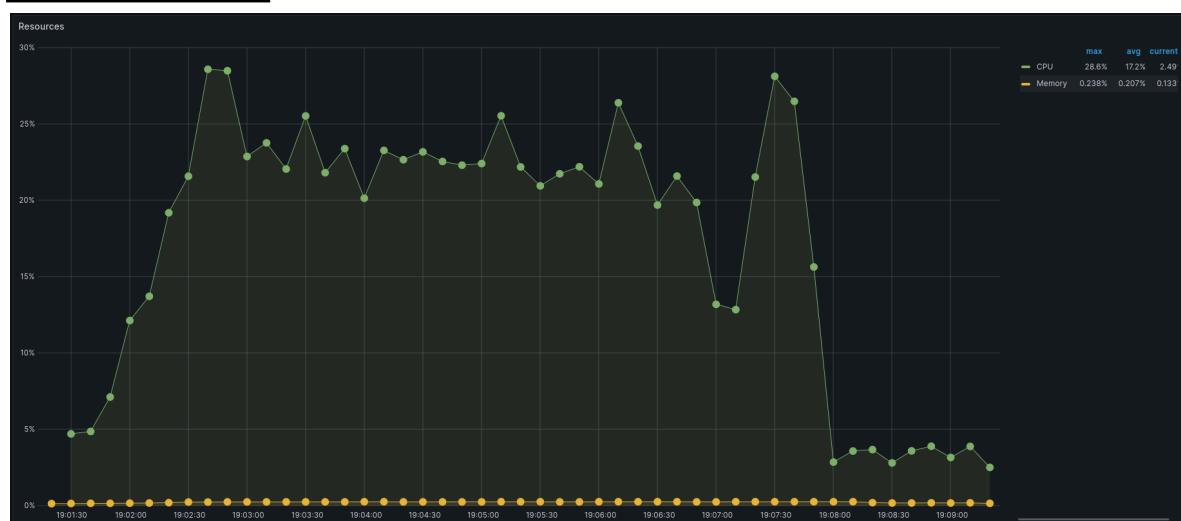
Estado de las requests



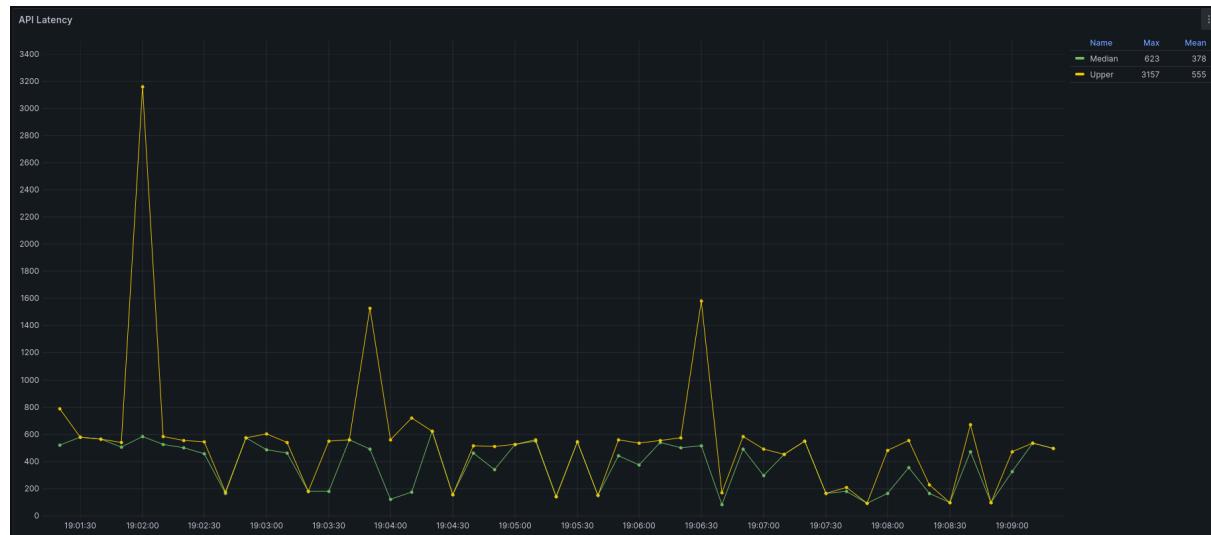
Tiempo de respuesta (lado del cliente)



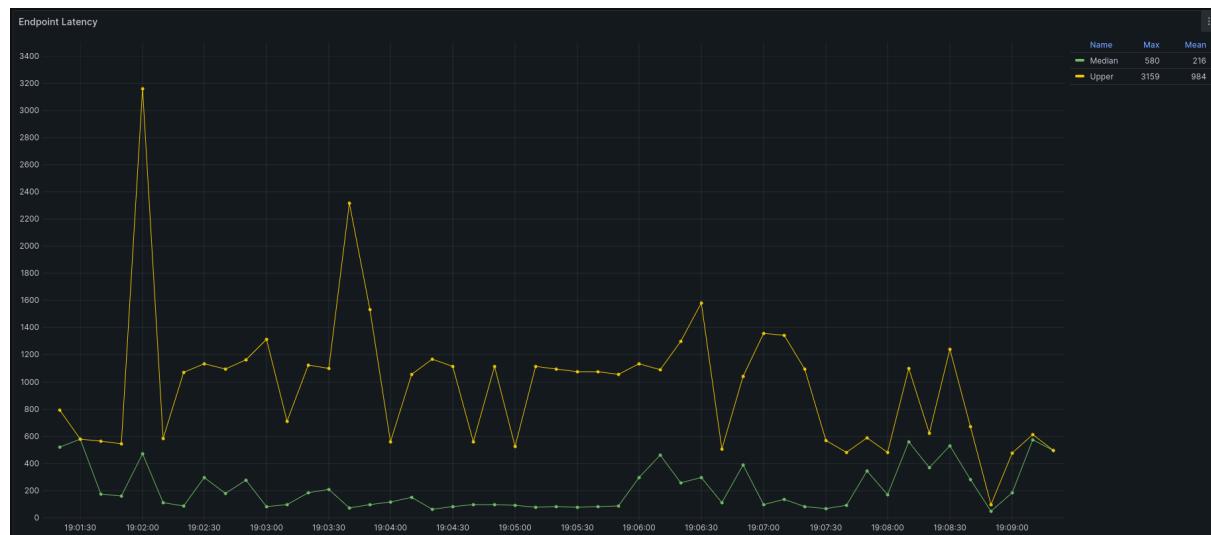
Recursos utilizados



Latencia de la API externa



Latencia del Endpoint



Comentarios

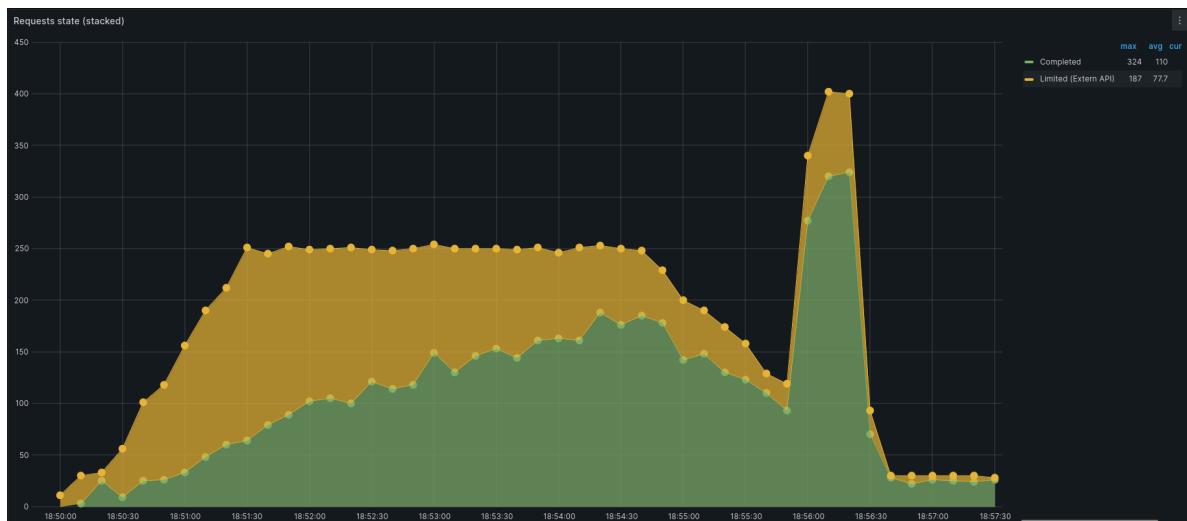
Aquí tuvimos una leve sorpresa. El caché fue configurado con *Lazy Population* ya que de antemano no se sabe qué palabras se usarán en las siguientes requests, como tamaño se usó el default de la configuración que le permite a Redis usar todo el tamaño disponible en el sistema que necesite, un *TTL* de 10 horas (tal vez exagerado pero para que supere el tiempo de las pruebas) y la política *noeviction* para que cada palabra cacheada pueda volver a ser conseguida las veces que se necesite. Al usar este caché esperamos que se incremente de manera considerable la cantidad de requests completadas de manera exitosa debido a que no se tendría que hacer el llamado a la API y esta no nos bloquearía el pedido con su Rate Limiting. Aunque este no fue el caso, de todas maneras se puede ver una **leve mejora** en la cantidad de respuestas (en promedio se completaron 47 cuando en el caso base había un promedio de 35) y en la mediana del tiempo de respuesta (tuvo un promedio de 176 ms y en el caso base fue 473 ms). Esta sorpresa en

cuanto al nivel de la mejora puede deberse a ciertos factores. Especialmente, la cantidad de palabras usadas. Para estos ejemplos se usa una lista de 10000 palabras. Para el payload de Artillery se usó “randomize: true”, que si se usa un generador lo suficientemente uniforme lo más probable es que no se repita casi ninguna palabra ya que no se llega a hacer 10000 requests en toda la ejecución.

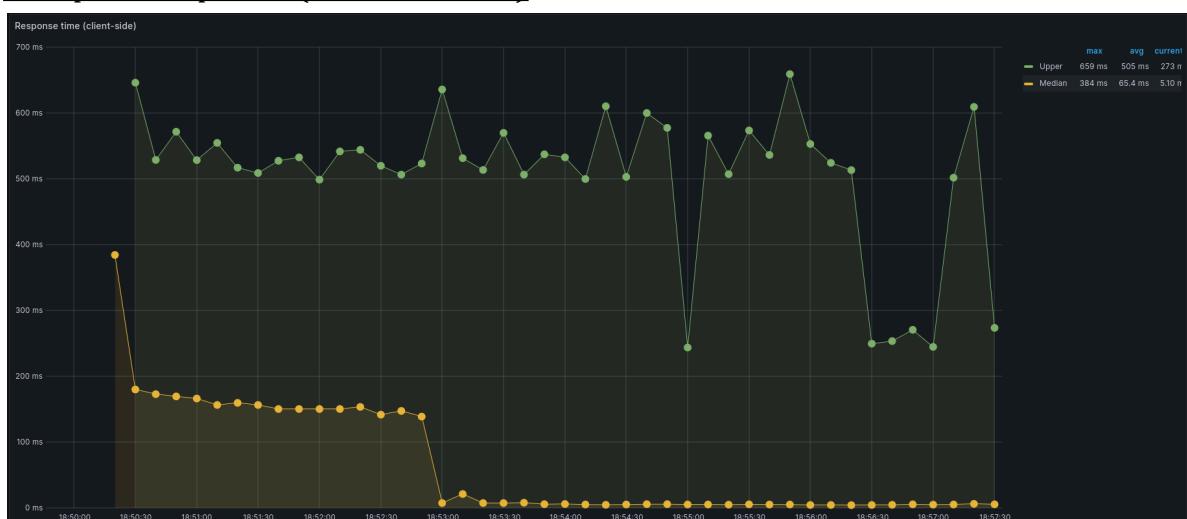
A partir de la sorpresa que nos llevamos con este caso, decidimos comprobar si nuestra suposición era correcta. Para eso, se redujo la cantidad de palabras a 500. Si a medida que pase el tiempo y se hagan requests, se completen más, entonces estamos en condiciones de decir que el problema a que no suceda lo esperado era efectivamente la cantidad de palabras. A continuación los gráficos del caso con menos palabras:

Caso 2 (500 palabras)

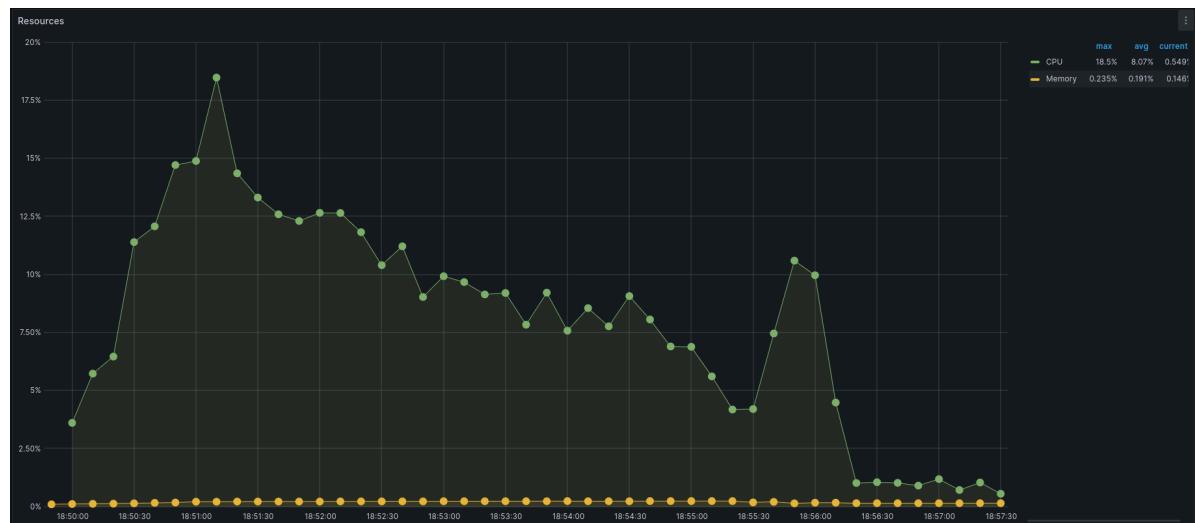
Estado de las requests



Tiempo de respuesta (lado del cliente)



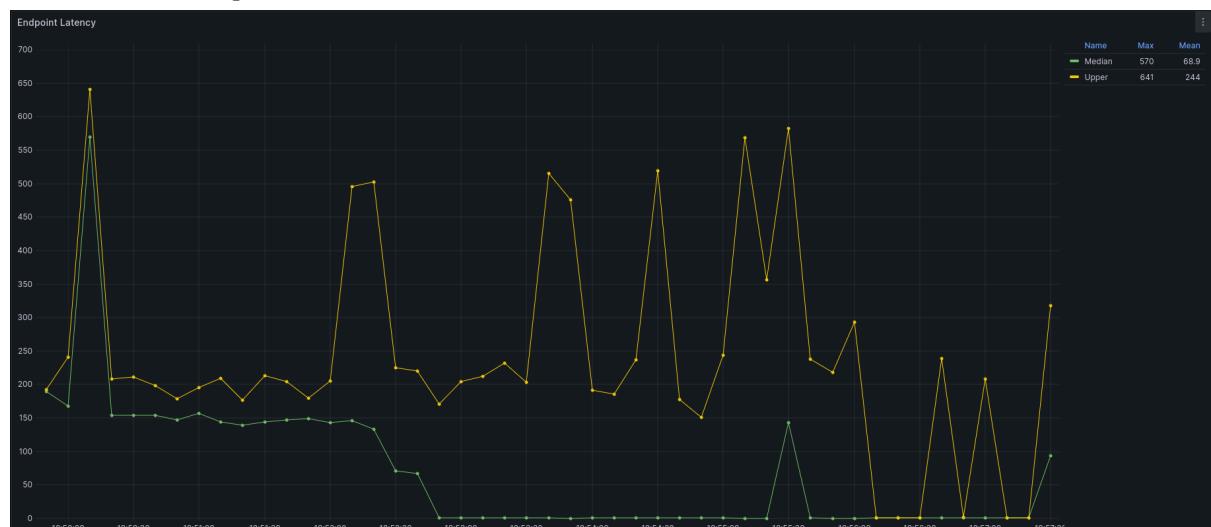
Recursos utilizados



Latencia de la API externa



Latencia del Endpoint



Comentarios

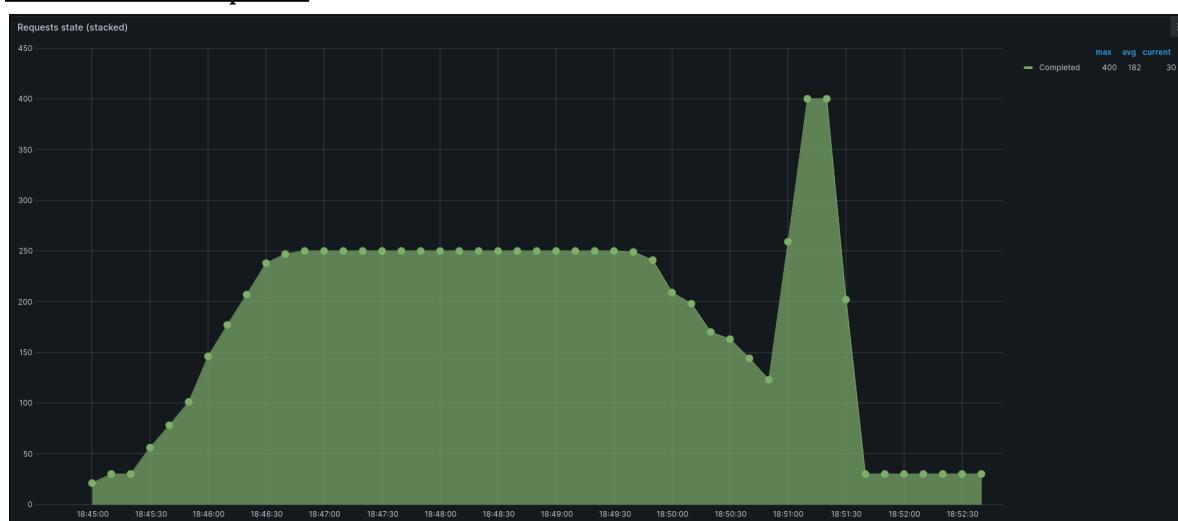
Podemos ver a partir de los resultados, que el problema se trataba de la cantidad de palabras. En el gráfico *Estado de las requests* vemos cómo a medida que pasa el tiempo, la diferencia entre requests exitosas y requests limitadas es cada vez menor. Esto se debe al uso del caché y que cada vez hay más palabras disponibles en él, generando menos uso de la API y por ende menos requests limitadas por parte de la misma. Esto genera que los tiempos de respuesta caigan a la mitad en comparación a cuando habían 10000 palabras y que el uso de CPU vaya disminuyendo a medida que se hace más uso del caché. Con respecto a las latencias, en el gráfico *API Latency* se puede ver los momentos que no se hizo uso de la API en lo absoluto ya que no hay valores y como el *Endpoint Latency* tiene también una caída de la mediana a medida que se usa menos la API.

spaceflight_news

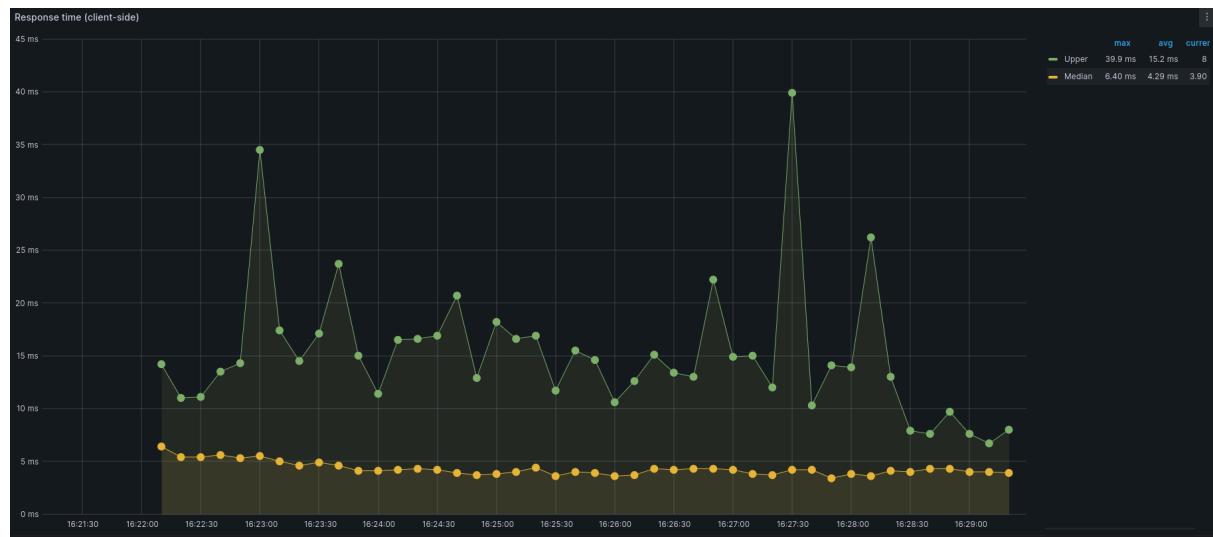
Escenarios desplegados



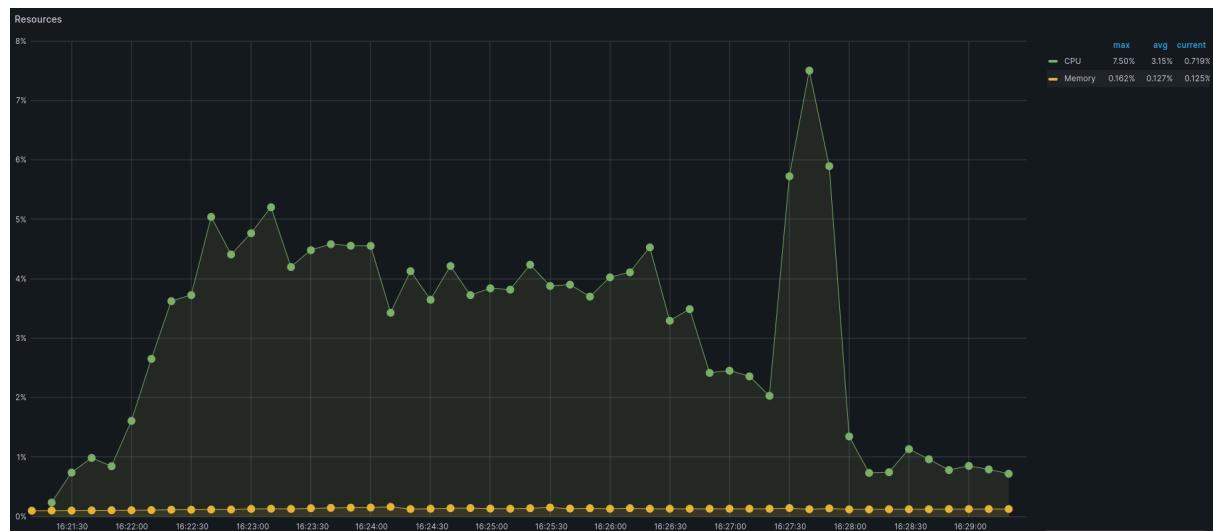
Estado de las requests



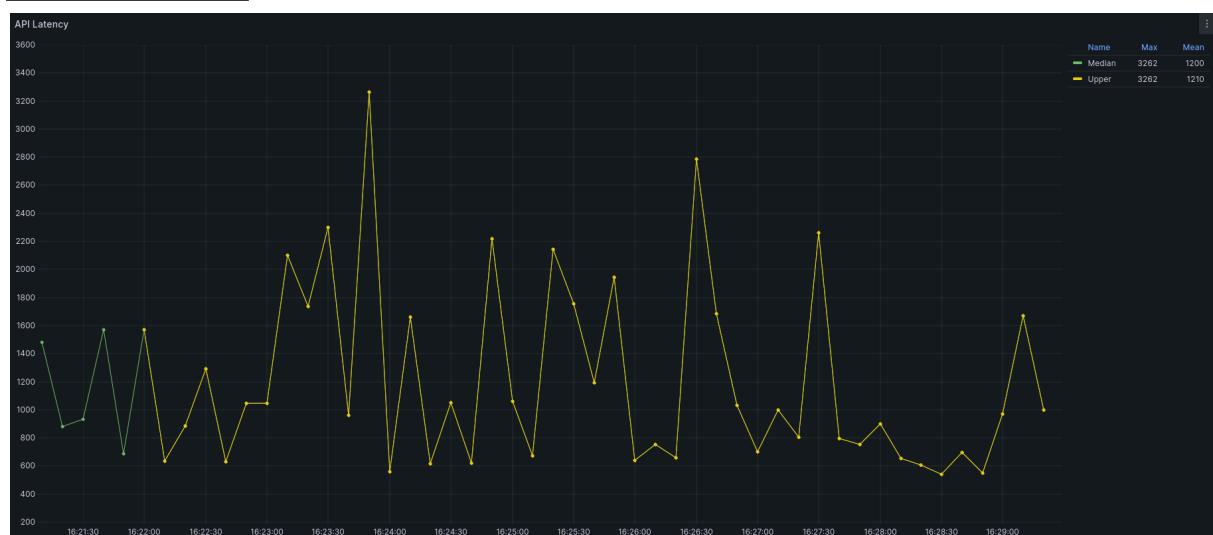
Tiempo de respuesta (lado del cliente)



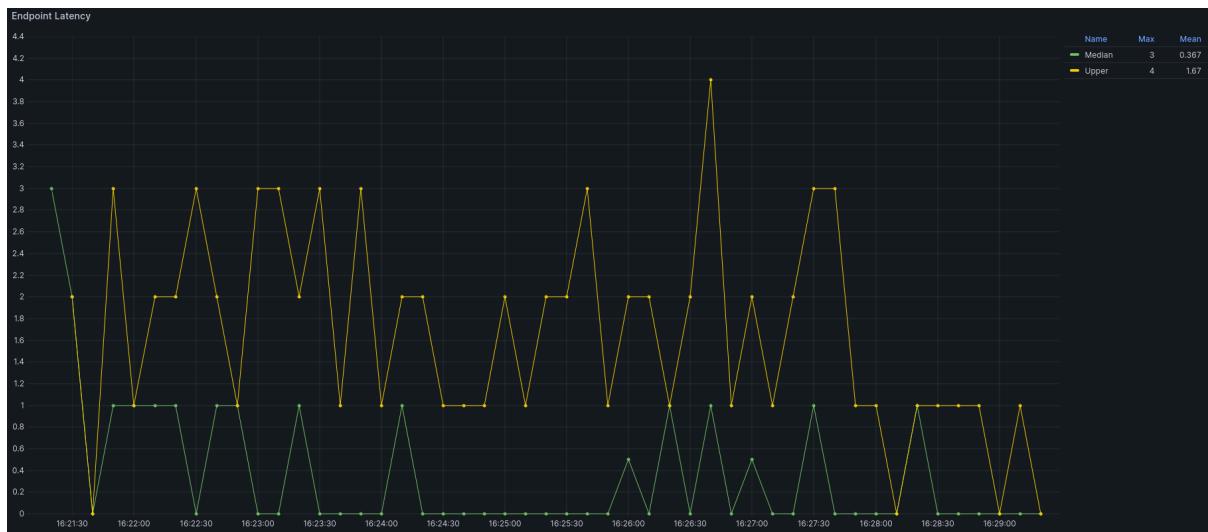
Recursos utilizados



Latencia de la API



Latencia del Endpoint



Comentarios

En este endpoint se decidió usar el caché con *Active Population* ya que las noticias a devolver se actualizan cada 10 minutos en la API entonces sabemos que por lo menos durante 10 minutos se devolverán los mismo datos, de todos modos se puebla el caché cada 10 segundos. Esto lo hacemos con el objetivo de evitar dar noticias que podrían haber quedado viejas por más de 10 segundos, lo cual no es muy costoso considerando que sólo implicaría una consulta a la API cada este intervalo. Como tamaño devuelta se usó el default de la configuración que le permite a Redis usar todo el tamaño disponible en el sistema que necesite, pero esta vez se usó un *TTL* de 20 segundos y la política *noeviction* para que las noticias cacheadas pueda volver a ser conseguida las veces que se necesite.

Hubo un **importante cambio gracias al caché**. De pasar a no responder prácticamente casi ningún request, se pudieron responder todos y con un tiempo de respuesta extremadamente bajo. Esto se debe al Active Population que se hace cada 10 segundos y que podemos ver en el gráfico API Latency.

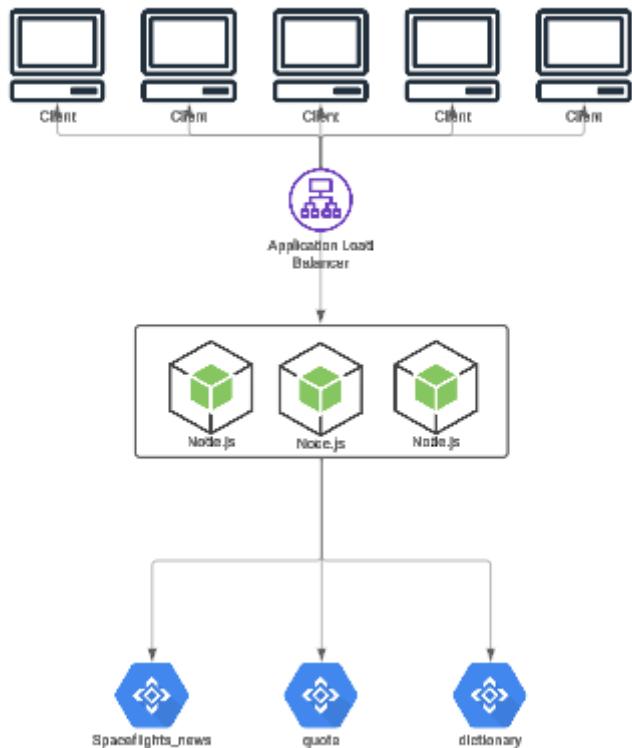
Respecto del tiempo de respuesta, en el caso base notábamos como empezaba alrededor de los 2 segundos y subía de manera drástica llegando a tardar casi 10 segundos en responder hasta que la disponibilidad del endpoint comenzaba a ser nula y no respondía requests. Con esta estrategia de caché vemos como eso no sucede. Ahora desde el primer request que tarda en el orden de los milisegundos y se mantiene de esa manera durante toda la ejecución. En cuanto a la latencia de la API, se puede ver que las consultas pueden llegar a tener un costo temporal de unos pocos segundos, pero esto no es problema considerando que la cantidad de consultas realizadas es muy pequeña.

Por el lado de los recursos, también vemos una mejora. El uso de CPU bajó de un promedio de 23.5% a uno de 3.15%, con un pico máximo de 7.50% cuando antes había llegado a un pico del 50%. Esto podría deberse a que en el caso base, como la API se demoraba decenas de segundos en responder, la cantidad de conexiones abiertas con distintos clientes al mismo tiempo llegaba a niveles máximos. A tal punto que como

mencionamos antes Nginx debía cerrar las conexiones por su cuenta. Con el caché, cada conexión se realiza en cuestión de pocos milisegundos, de tal manera que no se acumulan conexiones.

Replicación

Para esta táctica escalamos el servicio a 3 copias, convirtiendo a nginx en un load balancer.



Para lograr esto, no se debieron hacer grandes cambios en el código. Primero se tomó ventaja de la simplicidad del docker-compose y en él se agregaron 2 servicios más idénticos al de la app. Si en un futuro se deseara que el sistema sea realmente escalable, un simple script en el que se indique la cantidad de nodos necesarios debería únicamente agregar los servicios de la misma manera y con eso bastaría.

```
services:
  node:
    build: ./app
    container_name: app

  node2:
    build: ./app
    container_name: app2

  node3:
    build: ./app
    container_name: app3
```

Por último, se agregaron estas nuevas apps a la configuración de Nginx para que sepa entre cuales debe actuar como load balancer:

```
upstream arkitech {
  server app:3000;
  server app2:3000;
  server app3:3000;
}
```

Atributos de Calidad

- **Escalabilidad:** Distribuyendo la carga en una mayor cantidad de réplicas, el sistema escala y le debería permitir procesar una mayor cantidad de requests simultáneas.
- **Disponibilidad:** Agregar más réplicas mejora la disponibilidad del sistema ya que si un nodo falla, los otros pueden seguir respondiendo a requests.

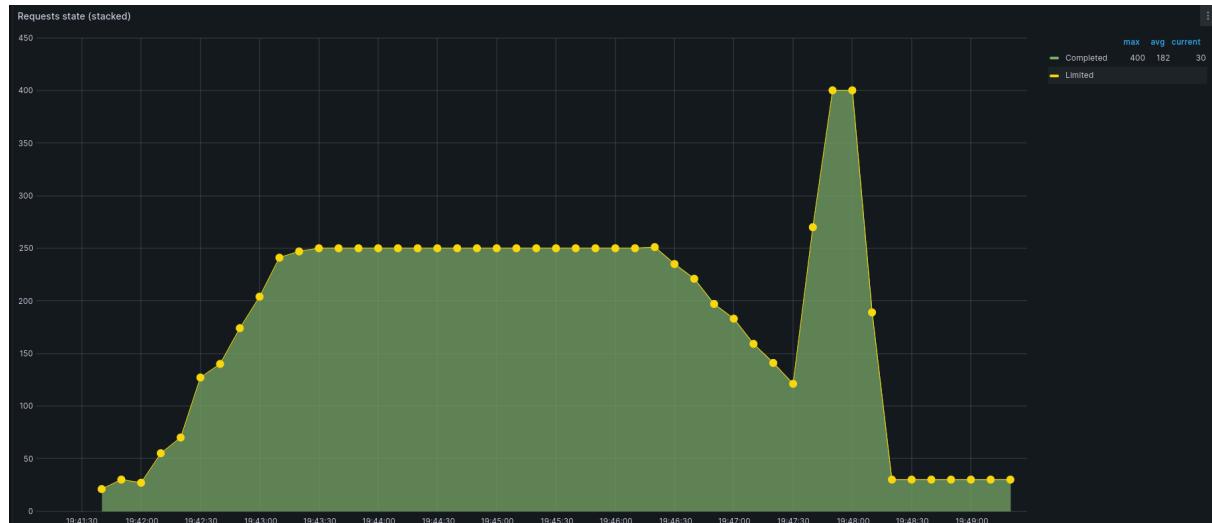
Análisis por Endpoint

ping

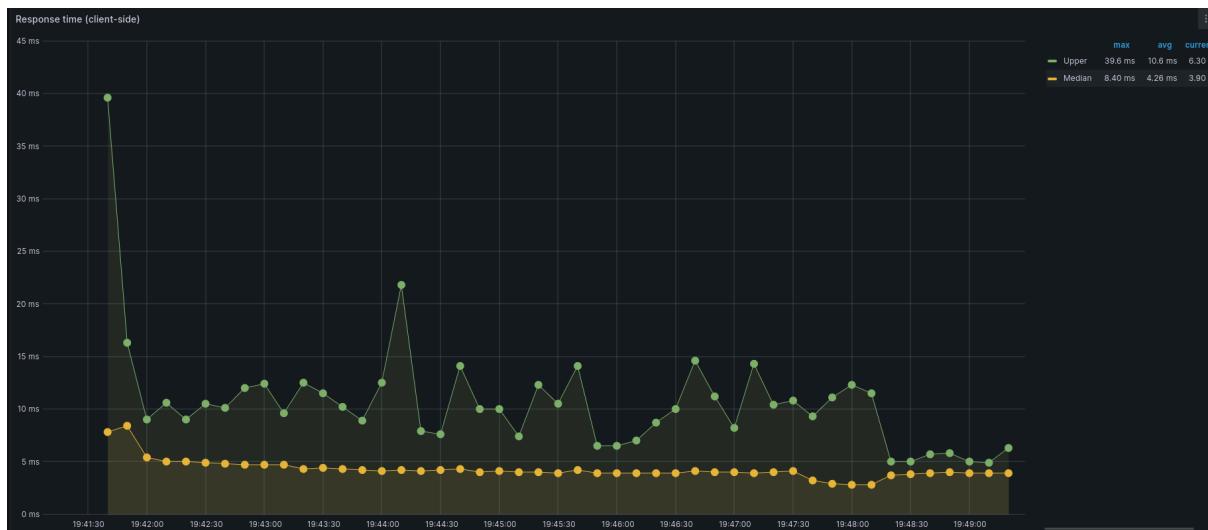
Escenarios desplegados



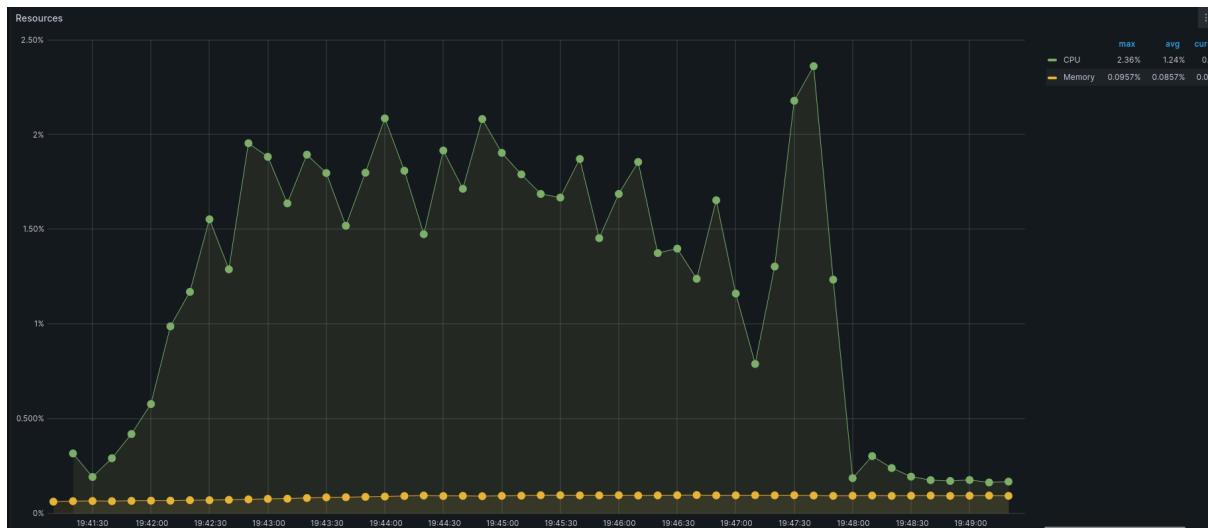
Estado de las requests



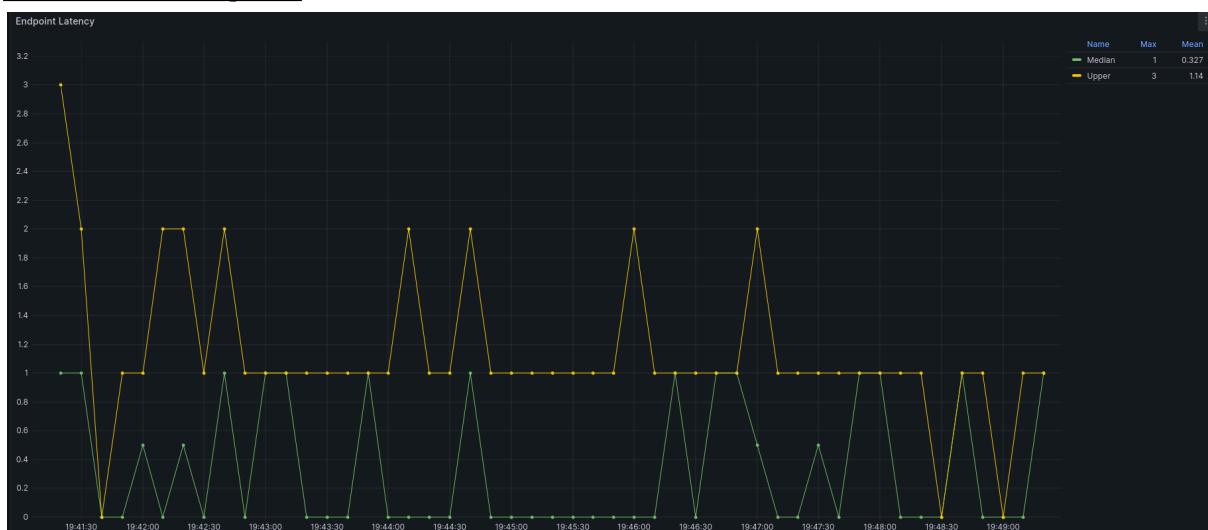
Tiempo de respuesta (lado del cliente)



Recursos utilizados



Latencia del Endpoint

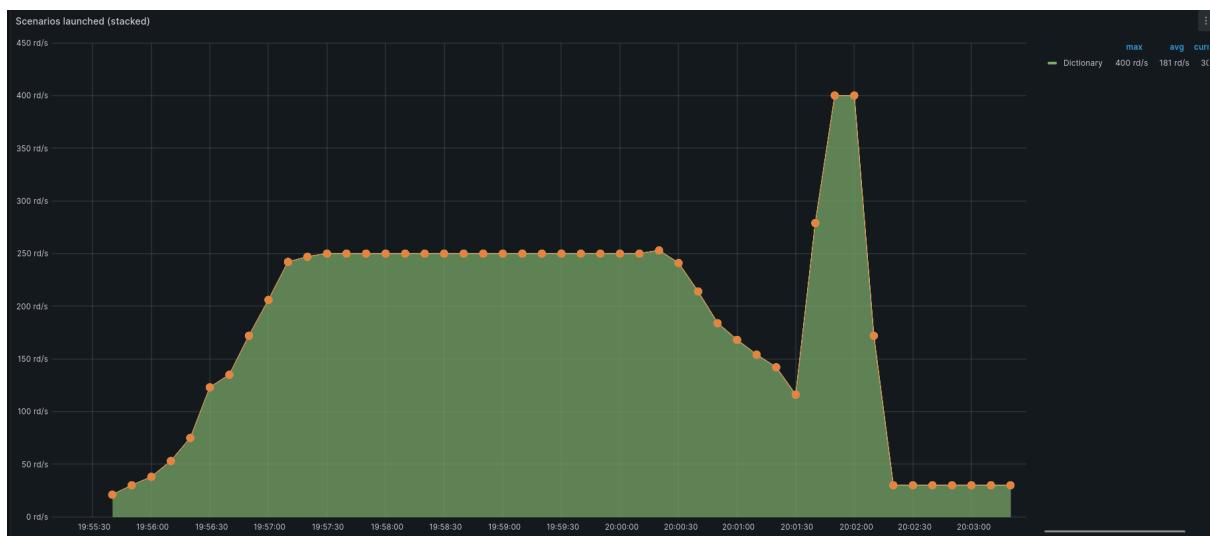


Comentarios

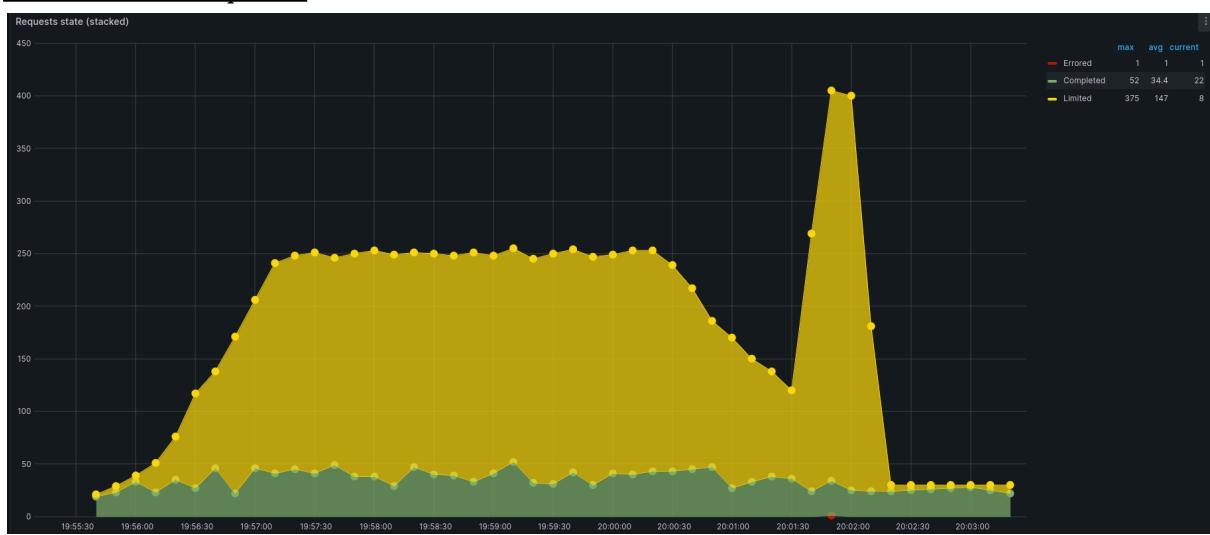
No sucedió lo que esperábamos. Al agregar 2 nodos más, suponíamos que incluso para este endpoint los tiempos disminuirían en comparación con el caso base, pero no fue así ya que podemos notar que se manejan los mismos tiempos. Esto puede deberse a varias cosas. En primer lugar, el procesamiento de ping es extremadamente simple y con la cantidad de requests por segundo que se generaron, incluso con un nodo se hizo todo de la manera más veloz posible. Otro factor podría ser Nginx. El nodo del reverse proxy podría estar actuando como cuello de botella y esto provocaría que no se pueda mejorar mucho más los tiempos para este caso, incluso aunque se agregaran más nodos.

dictionary

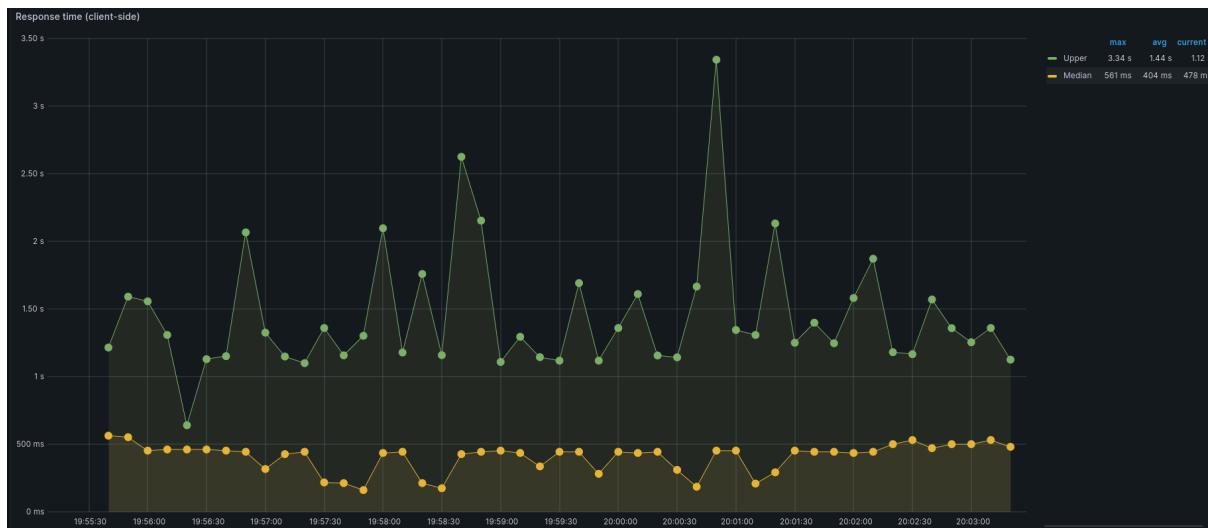
Escenarios desplegados



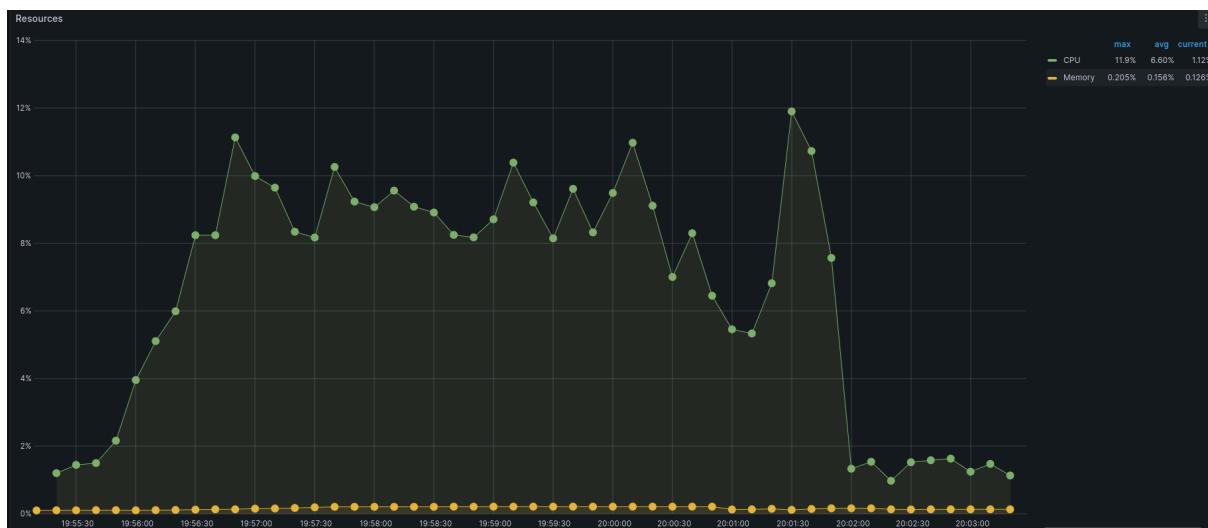
Estado de las requests



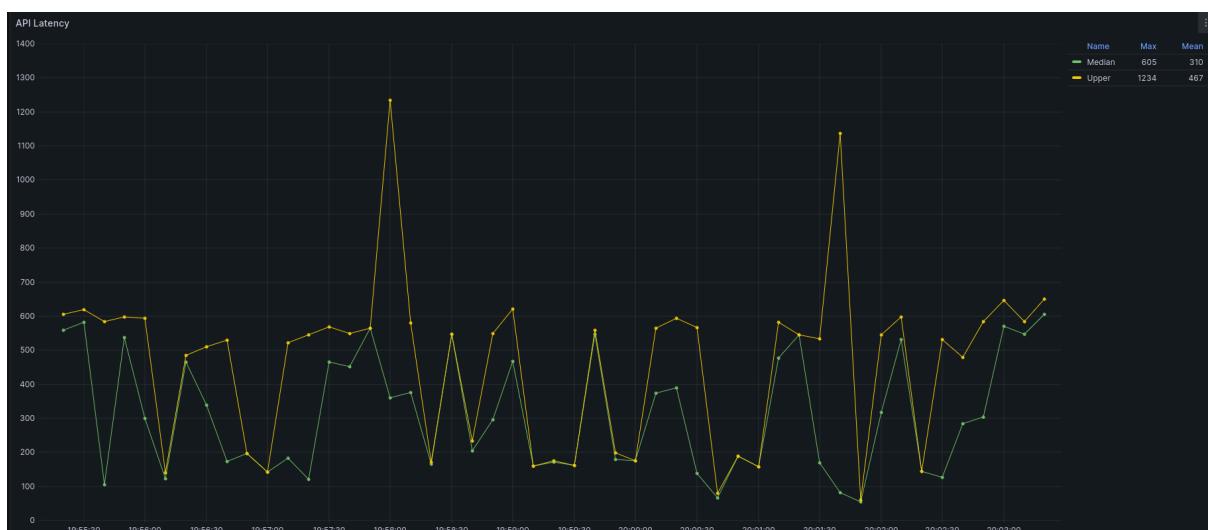
Tiempo de respuesta (lado del cliente)



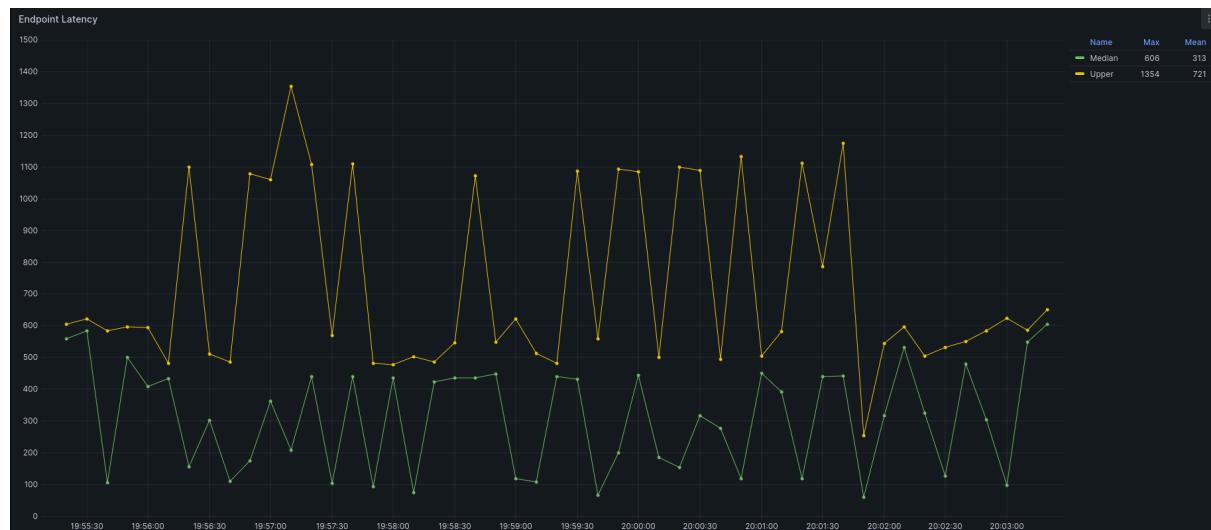
Recursos utilizados



Latencia de la API



Latencia del Endpoint



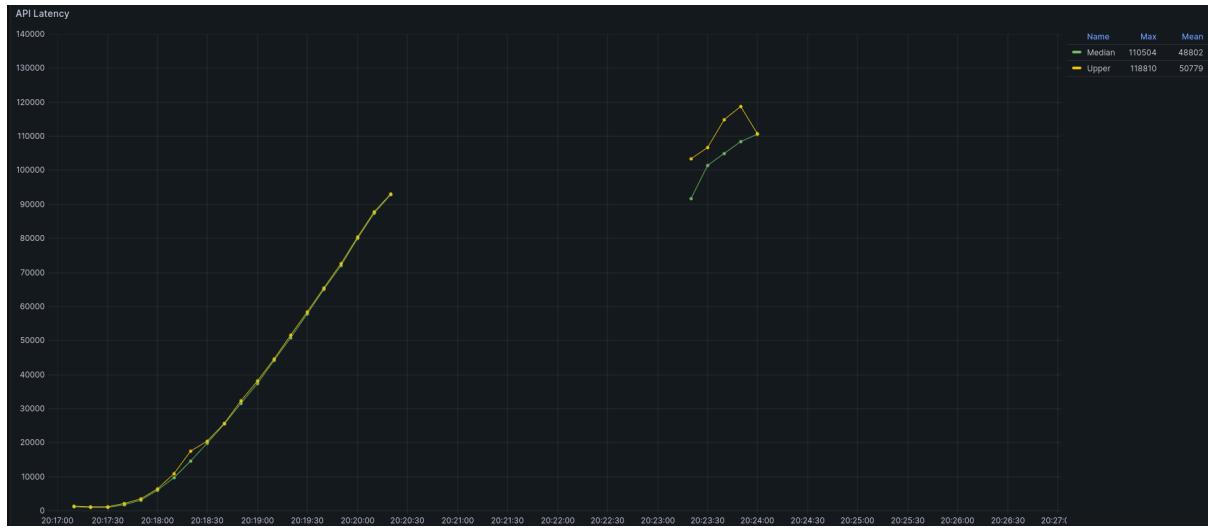
Comentarios

Podemos decir que se comportó según nuestras expectativas. Los requests completados y limitados se comportaron de la misma manera que en el caso base. Como ya sabemos, la API utilizada en este caso tiene un rate-limiting propio. Agregar más containers no sirve de nada ya que, por más que dentro de la red interna de Docker tengan diferentes IPs, en el 'exterior' el request de cualquier nodo tiene la misma IP, que será la del host. Con respecto a los recursos utilizados, a primera vista diríamos que se redujo a cerca de un tercio de lo que se usaba en el caso base. Pero debemos tener en cuenta que ese uso de recursos es para un container solo. En los demás, se usó lo mismo y por ende en total llegamos a los mismos valores que en el caso con 1 solo nodo. Del lado de las latencias, se puede ver una mejora. Aunque no podemos hablar de algo que realmente haga un cambio en los tiempos.

Lo que sí podemos concluir para este caso, es que al tener más nodos, se distribuye el cómputo y así el uso de recursos. De esta manera no sobre-cargaremos un mismo nodo y les daríamos lugar a procesar más cosas al mismo tiempo que las requests.

spaceflight_news

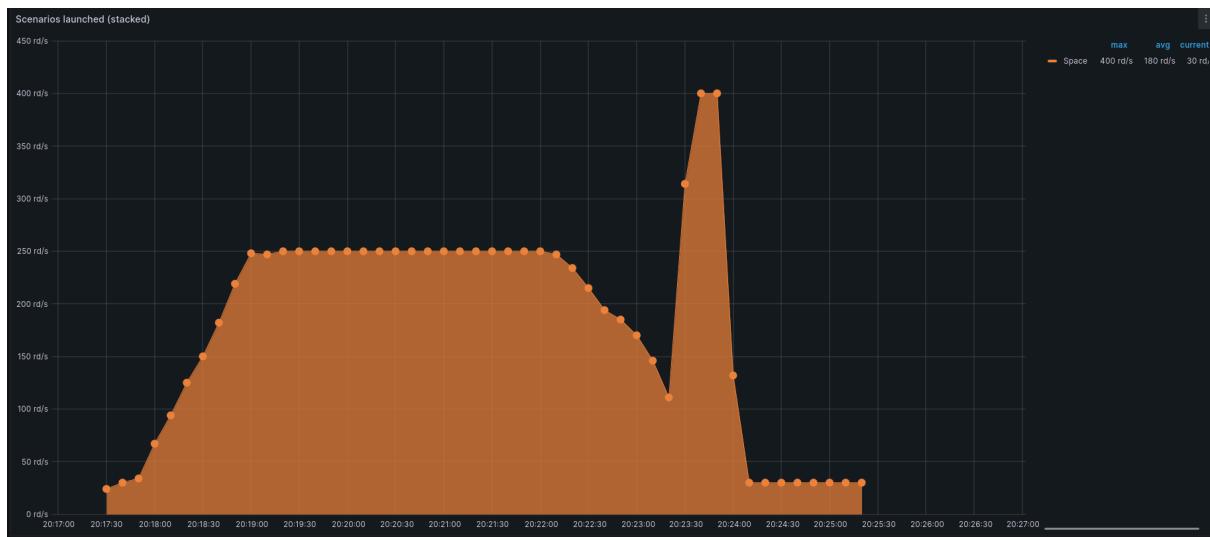
Latencia de la API



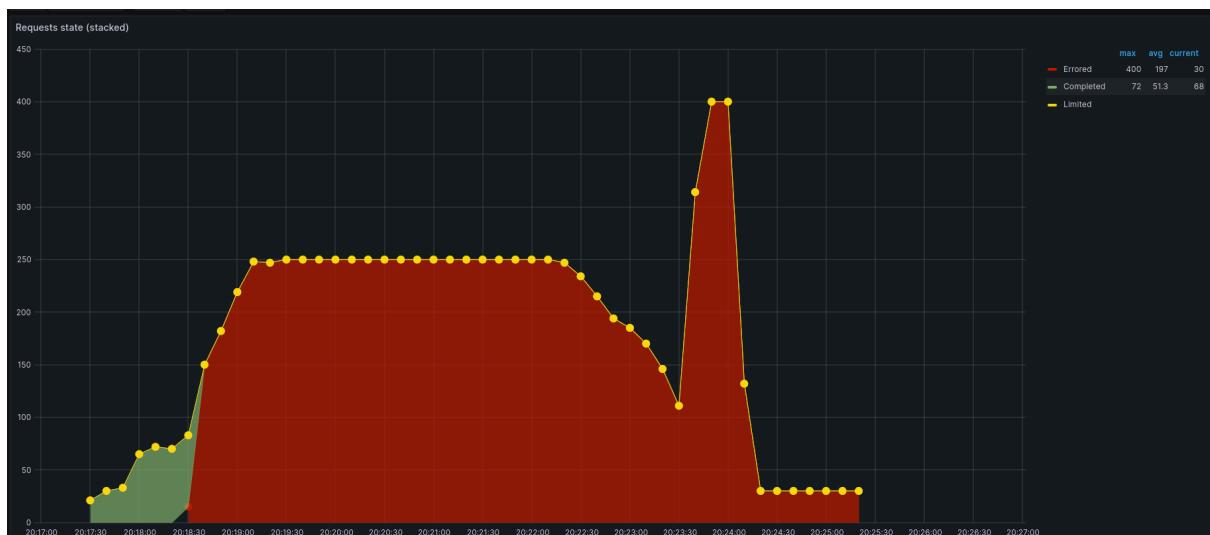
Latencia del Endpoint



Escenarios desplegados



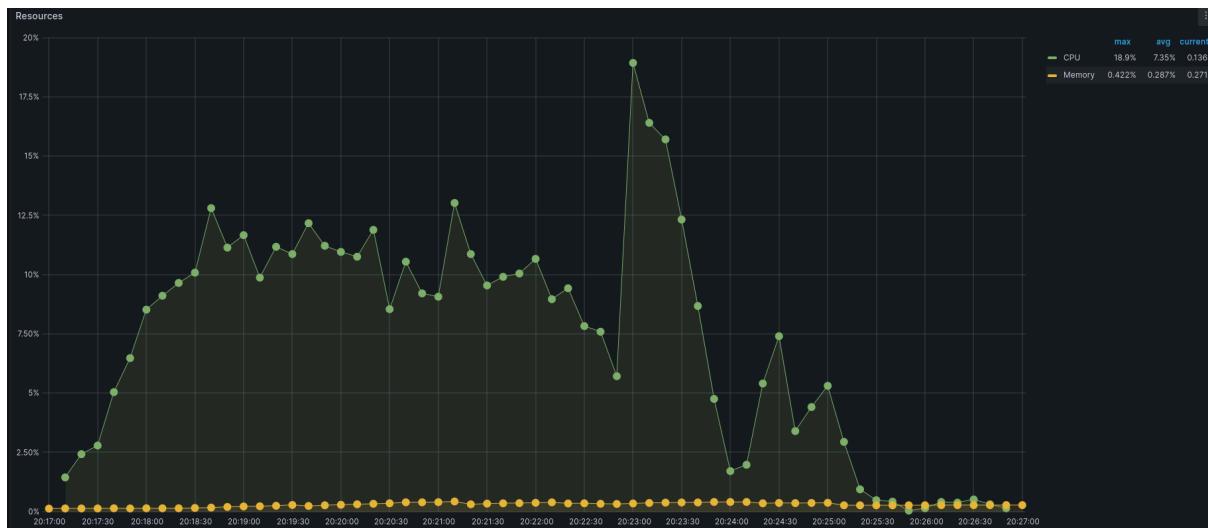
Estado de las requests



Tiempo de respuesta (lado del cliente)



Recursos utilizados

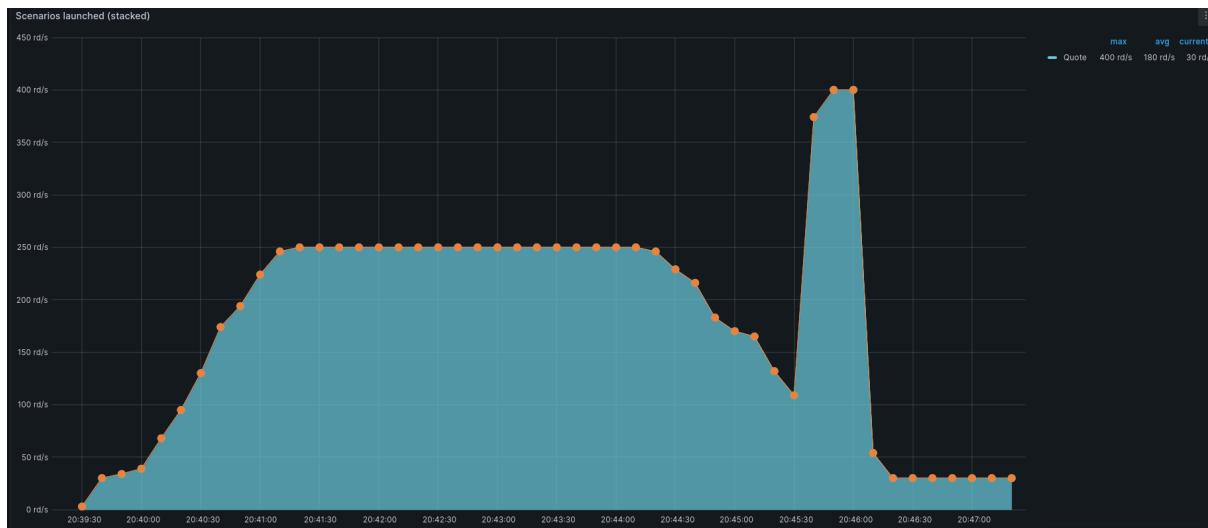


Comentarios

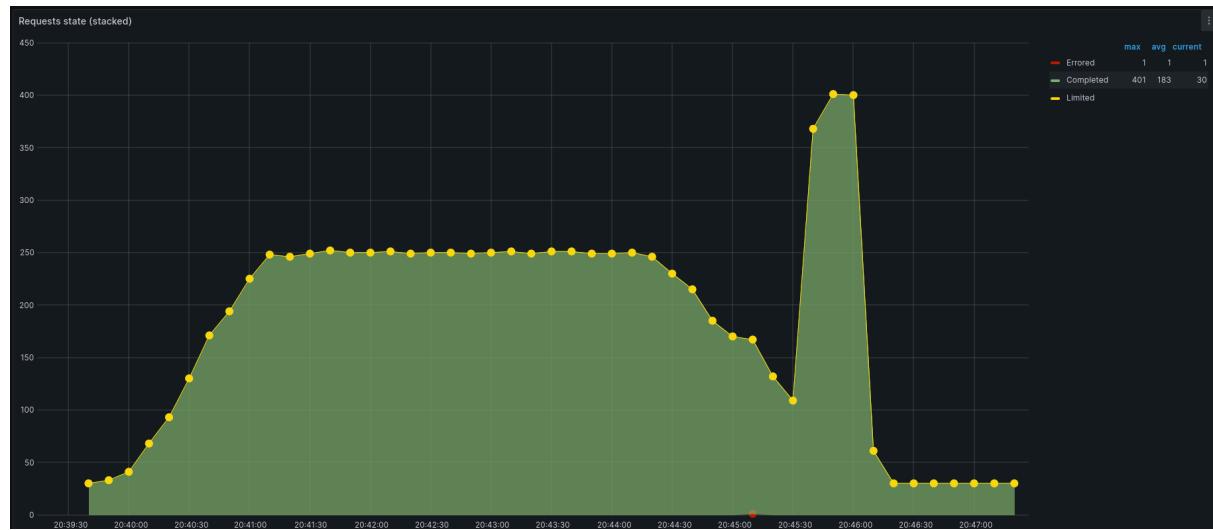
Aquí notamos devuelta cuanto puede llegar a tardar la API de noticias del espacio. Con este endpoint vuelve a suceder lo que ya esperábamos, una gran cantidad de requests fallidas. A medida que hay más requests, la API aumenta su latencia en la respuesta hasta tal punto que no vuelve a contestar. De esta manera los requests que llegan son cerrados por Nginx como comentamos anteriormente en casos de este mismo endpoint. La replicación de nodos no viene a solucionar este problema ya que lo único en lo que ayuda es en hacer más llamados a la API en un menor tiempo, algo que en realidad, empeora la situación.

quote

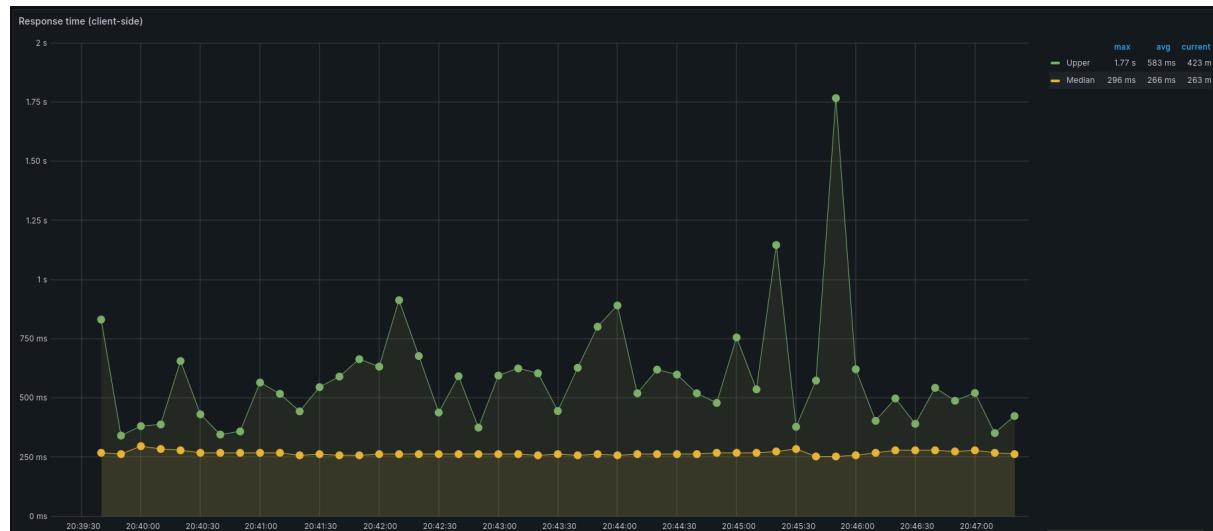
Escenarios desplegados



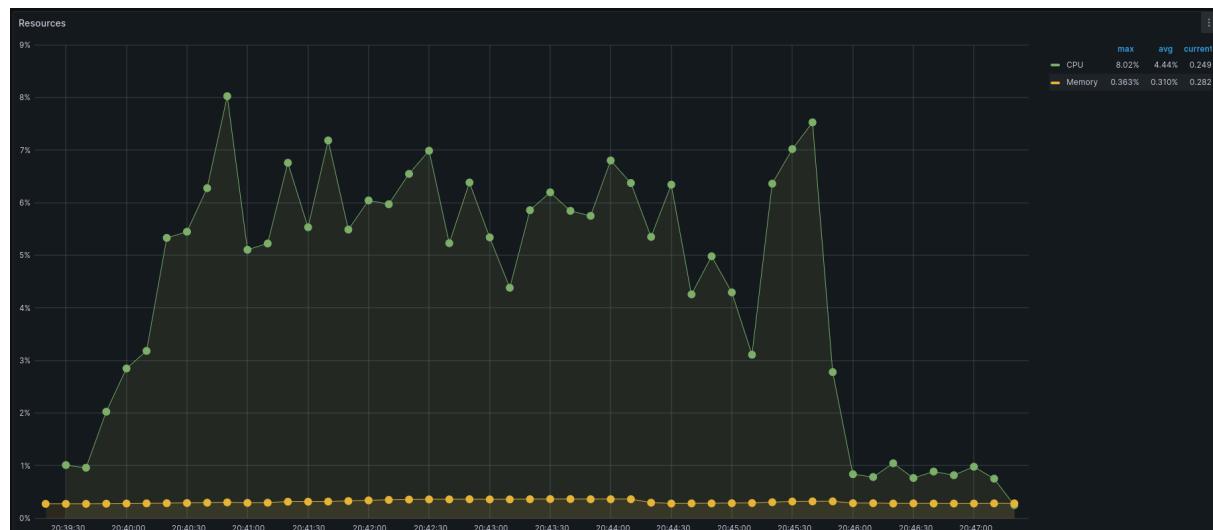
Estado de las requests



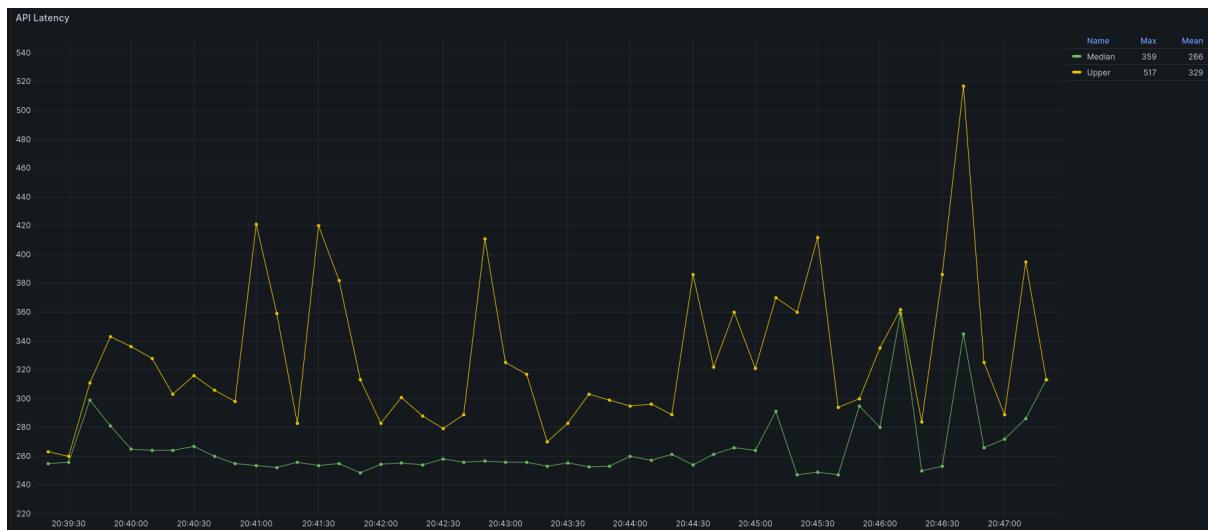
Tiempo de respuesta (lado del cliente)



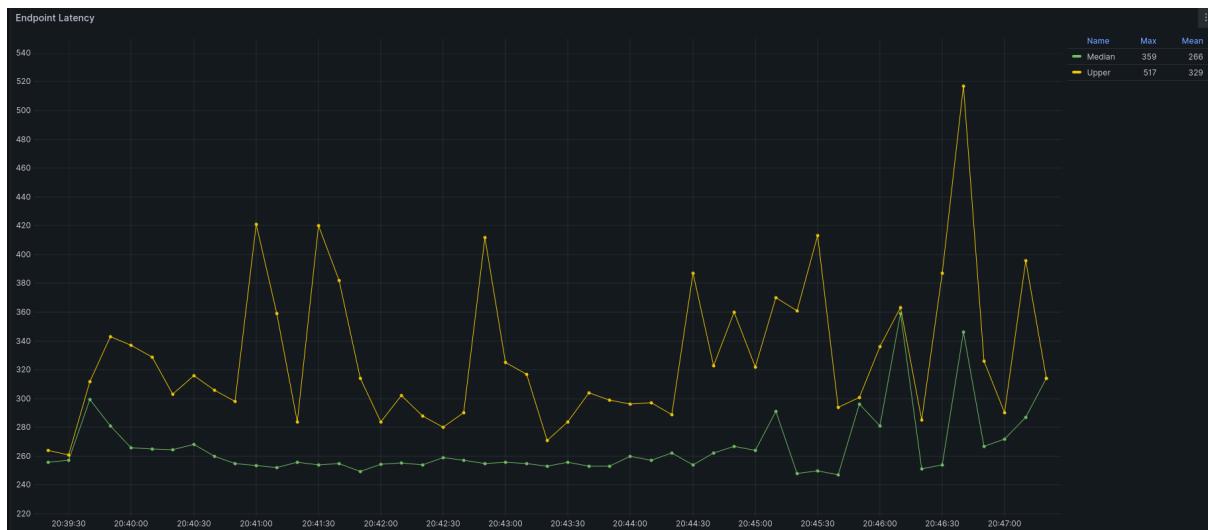
Recursos utilizados



Latencia de la API



Latencia del Endpoint



Comentarios

Fue un poco sorpresivo. Al haber más nodos y trabajar con una API sin rate-limiting y que consideramos veloz, suponíamos que los tiempos de respuesta y de latencia serían mucho menores en comparación con los demás casos. Mirando los gráficos, esto no fue así. Uno de los responsables podría ser el nodo de Nginx que genera un cuello de botella como mencionamos en caso anterior. Otro motivo podría ser el número de requests y que este no sea lo suficientemente alto como para notar una diferencia.

Rate limiting

Con esta táctica, buscamos limitar la cantidad de requests que puede consumir nuestra aplicación, con el objetivo de evitar cargas excesivamente grandes que puedan afectar a la performance de nuestra aplicación.

Para implementar esta táctica, optamos por hacer uso de las herramientas ofrecidas por Nginx para definir este límite. Nginx hace uso del “leaky bucket algorithm” (o, en español, el “algoritmo del balde con fugas”), que consiste en trazar una analogía con un balde que es llenado con agua y tiene una fuga en el fondo, siendo similar a una cola del tipo FIFO en la que las requests esperan por ser procesadas. El agua que se fuga corresponde a las requests que efectivamente son atendidas por nuestra aplicación, mientras el agua que rebalsa mientras se llena el balde corresponde a las requests que terminan siendo descartadas.

Con el uso de Nginx, esta táctica sólo requiere del agregado de algunas instrucciones en su archivo de configuración:

```
limit_req_zone $binary_remote_addr zone=mylimit:100k rate=10r/s; # 1 request every 100 ms

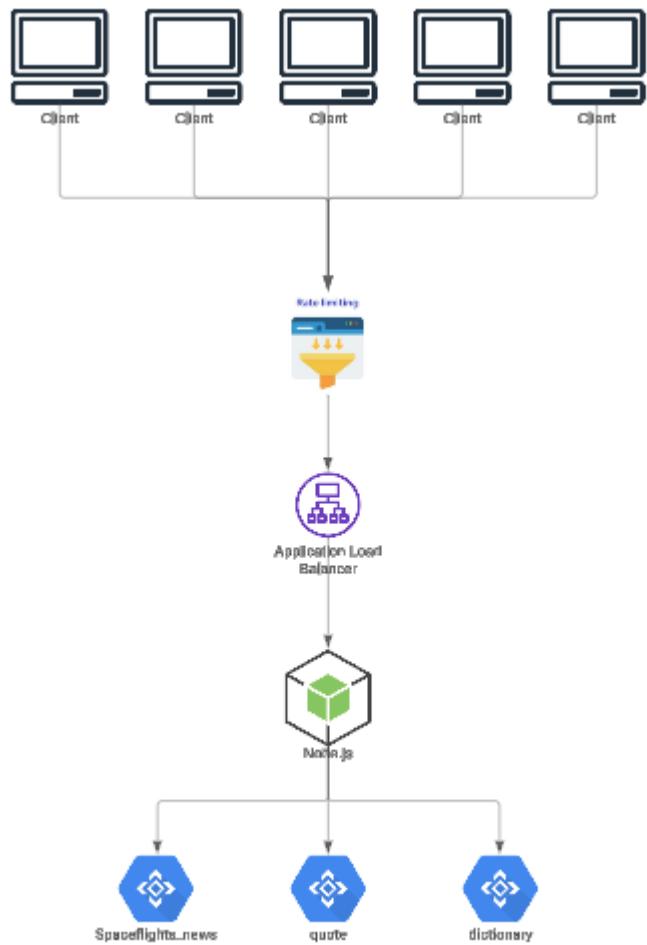
upstream arkitechs {
    server app:3000;
}

server {
    listen 80;

    location / {
        limit_req zone=mylimit; # We can also try with delay and bursts
        proxy_pass http://arkitechs;
    }
}
```

Como podemos ver, se agregan dos directivas:

- la primera es '*limit_req_zone*', que permite definir los parámetros para realizar dicho Rate Limiting, siendo estos los siguientes tres:
 - *key*, que nos permite definir la característica de las requests a bloquear; en este caso, usamos *\$binary_remote_addr* para bloquear las requests de cada dirección IP única
 - *zone*, que define la zona de memoria en la cual almacenar, para cada dirección IP, cuántas veces se intentó acceder a una request con límite; en este caso le ponemos el nombre *mylimit* y asignamos 100 kilobytes
 - *rate*, que define el máximo request rate; en este caso, optamos por un máximo de 10 requests por segundo
- la segunda es '*limit_req*', que aplica efectivamente el Rate Limiting a todas las requests llegadas a nuestro servidor utilizando la zona antes definida



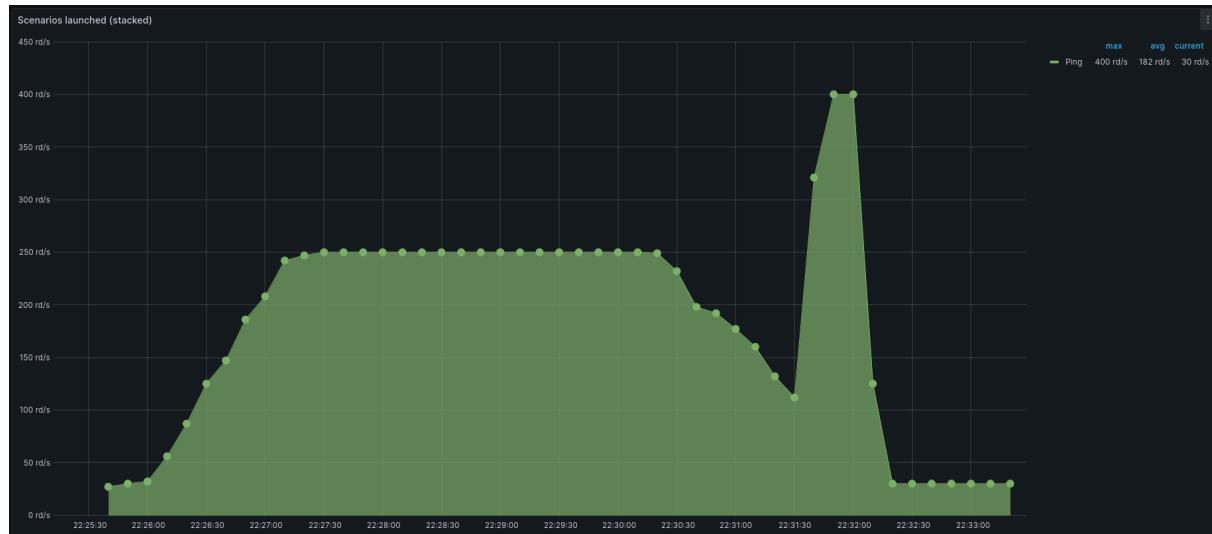
Atributos de Calidad

- **Performance:** Al limitar la cantidad de requests, esto ayuda a que la performance no se desplome cuando hay una carga que el sistema no puede soportar.
- **Disponibilidad:** Agregar Rate Limiting te permite establecer un límite propio con el cual podes tener mayor seguridad que el sistema va a responder ante determinada carga, evitando así una cantidad de requests que hagan caer a la aplicación.

Análisis por Endpoint

ping

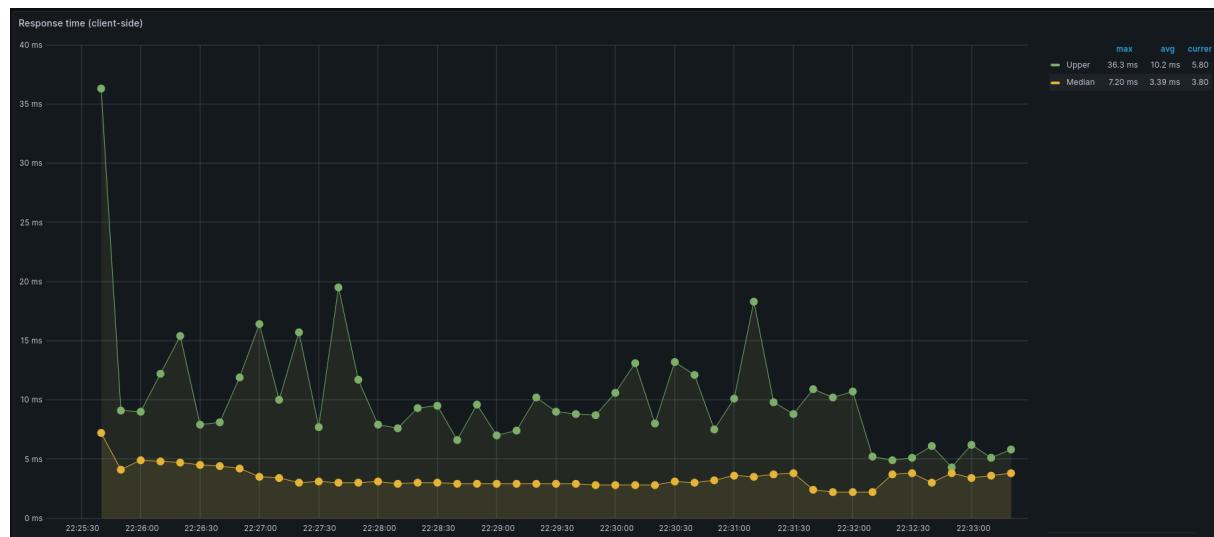
Escenarios desplegados



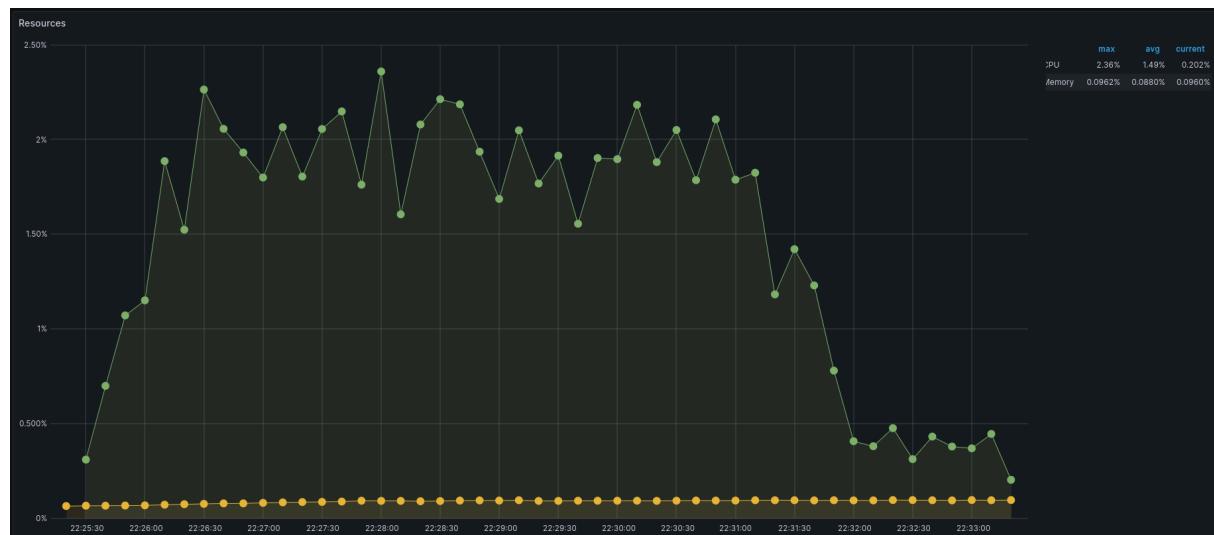
Estado de las requests



Tiempo de respuesta (lado del cliente)



Recursos utilizados



Latencia del Endpoint



Reporte de Artillery

```
Summary report @ 22:33:23(-0300)

http.codes.200: ..... 3306
http.codes.503: ..... 5278
http.request_rate: ..... 18/sec
http.requests: ..... 8584
http.response_time:
    min: ..... 0
    max: ..... 12
    median: ..... 1
    p95: ..... 3
    p99: ..... 4
http.responses: ..... 8584
vusers.completed: ..... 8584
vusers.created: ..... 8584
vusers.created_by_name.Ping: ..... 8584
vusers.failed: ..... 0
vusers.session_length:
    min: ..... 1.1
    max: ..... 36.3
    median: ..... 3
    p95: ..... 5.3
    p99: ..... 7.5
```

Comentarios

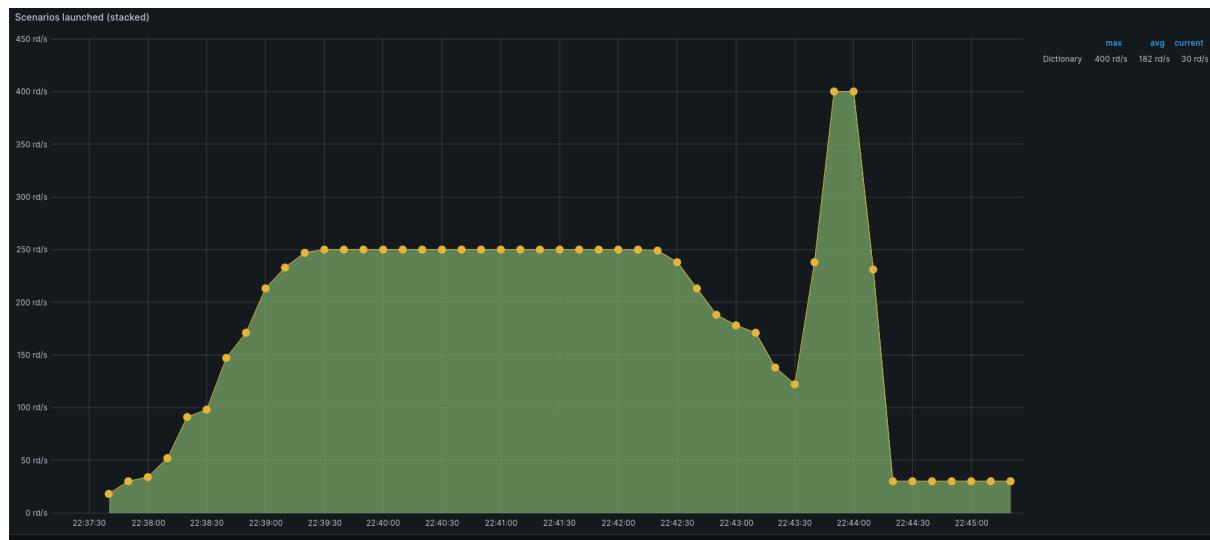
Vemos que una porción muy grande de las requests enviadas son limitadas, manteniéndose la cantidad de requests completadas cada **10 segundos** por debajo de las 90 y respetando nuestro límite impuesto.

Respecto al tiempo de respuesta percibido por el cliente, no notamos cambios sustanciales respecto del caso base, pues debemos recordar que este Endpoint incurre en un procesamiento despreciable, y para los escenarios planteados nuestro servidor no tiene dificultades a la hora de procesar tal carga. En este sentido, vemos que el gráfico de latencia del Endpoint tiene un patrón similar, pero con valores considerablemente menores. Esto no es de extrañar pues, a bajo procesamiento, lo que más tarda es la comunicación con el cliente.

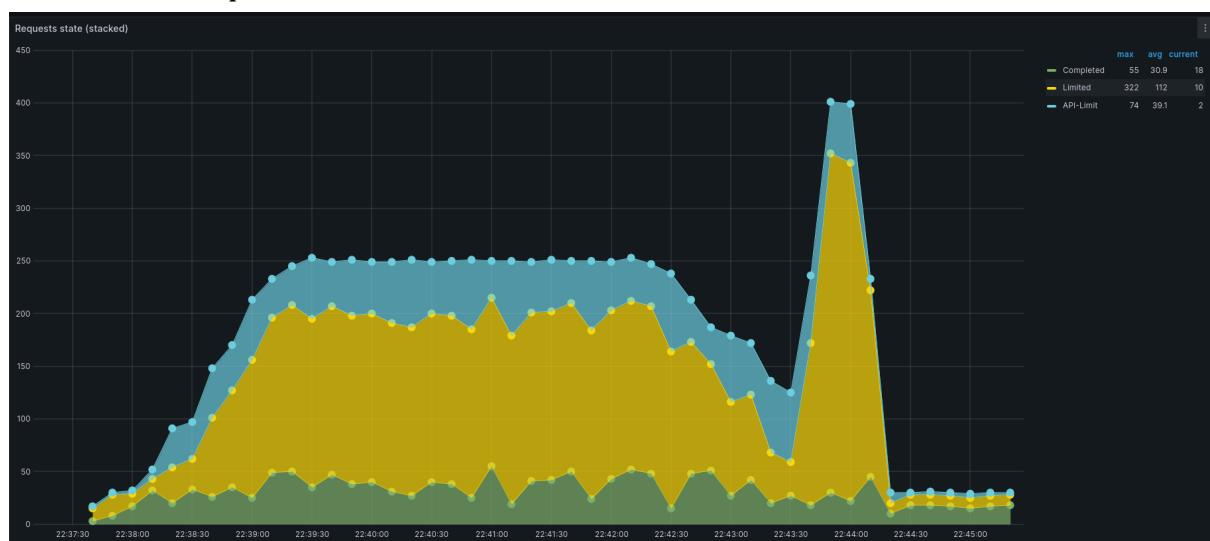
Sobre los recursos utilizados, nuevamente no hay diferencias con el caso base: la memoria se mantiene constante, y el uso del CPU varía de manera proporcional a la cantidad de requests completadas.

dictionary

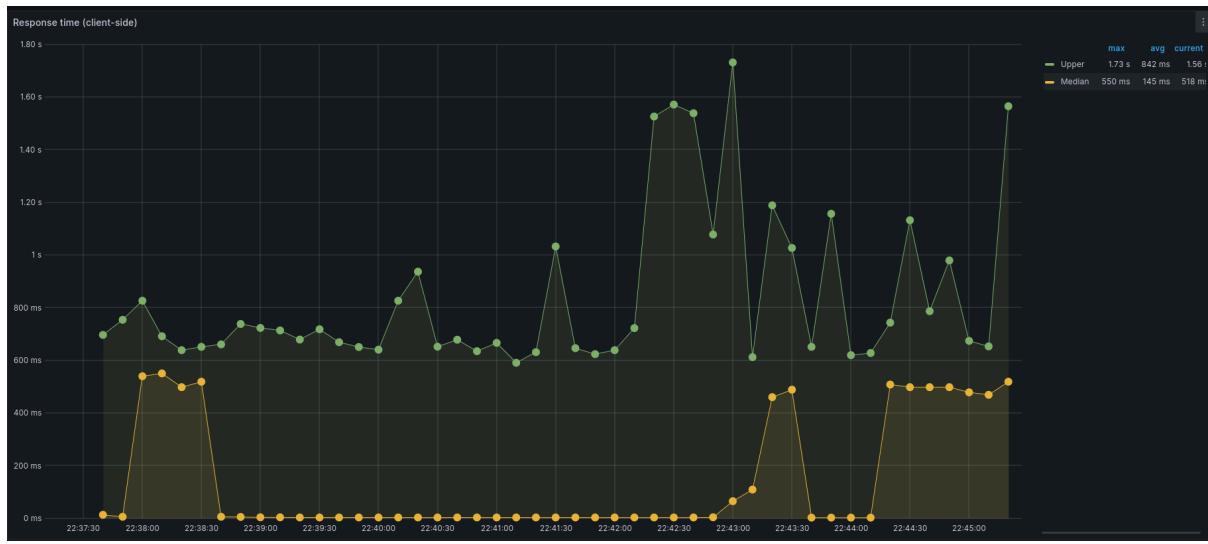
Escenarios desplegados



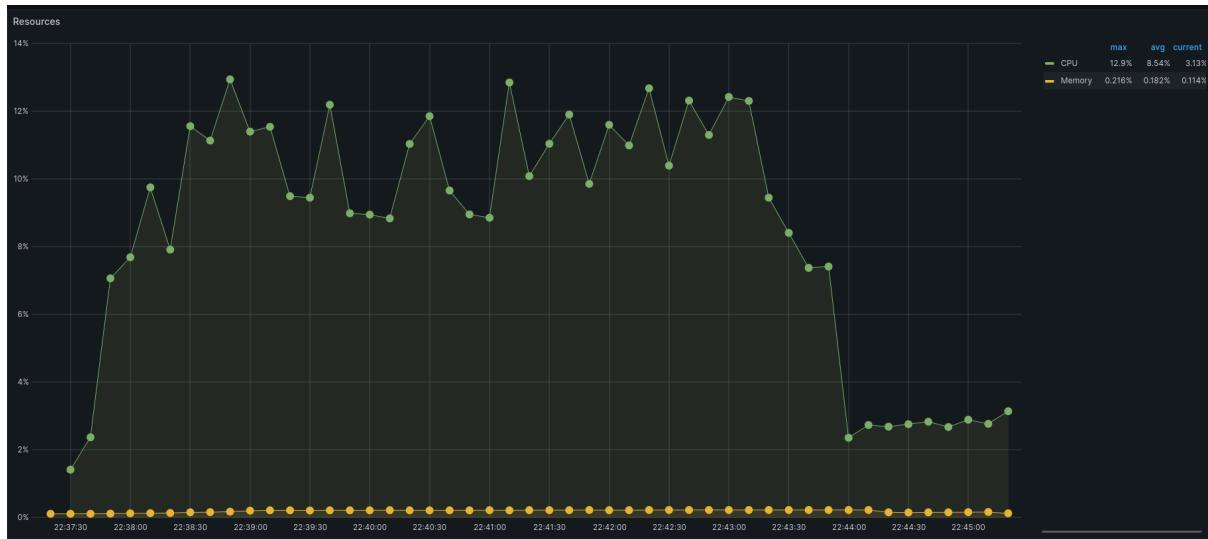
Estado de las requests



Tiempo de respuesta (lado del cliente)



Recursos utilizados



Latencia de la API externa



Latencia del Endpoint



Reporte de Artillery

```
-----  
Summary report @ 22:45:26(-0300)  
-----  
  
http.codes.200: ..... 1482  
http.codes.500: ..... 1840  
http.codes.503: ..... 5285  
http.request_rate: ..... 18/sec  
http.requests: ..... 8607  
http.response_time:  
    min: ..... 0  
    max: ..... 1729  
    median: ..... 1  
    p95: ..... 584.2  
    p99: ..... 645.6  
http.responses: ..... 8607  
vusers.completed: ..... 8607  
vusers.created: ..... 8607  
vusers.created_by_name.Dictionary: ..... 8607  
vusers.failed: ..... 0  
vusers.session_length:  
    min: ..... 1.1  
    max: ..... 1730.7  
    median: ..... 3  
    p95: ..... 584.2  
    p99: ..... 645.6
```

Comentarios

Para este Endpoint, obtuvimos resultados interesantes. Recordemos que, de por sí, la API externa utilizada implementa su propio Rate Limiting, por lo cual en el gráfico de estado de requests podemos diferenciar entre las requests limitadas; algunas pueden haber sido limitadas por nuestro servidor, otras por dicha API.

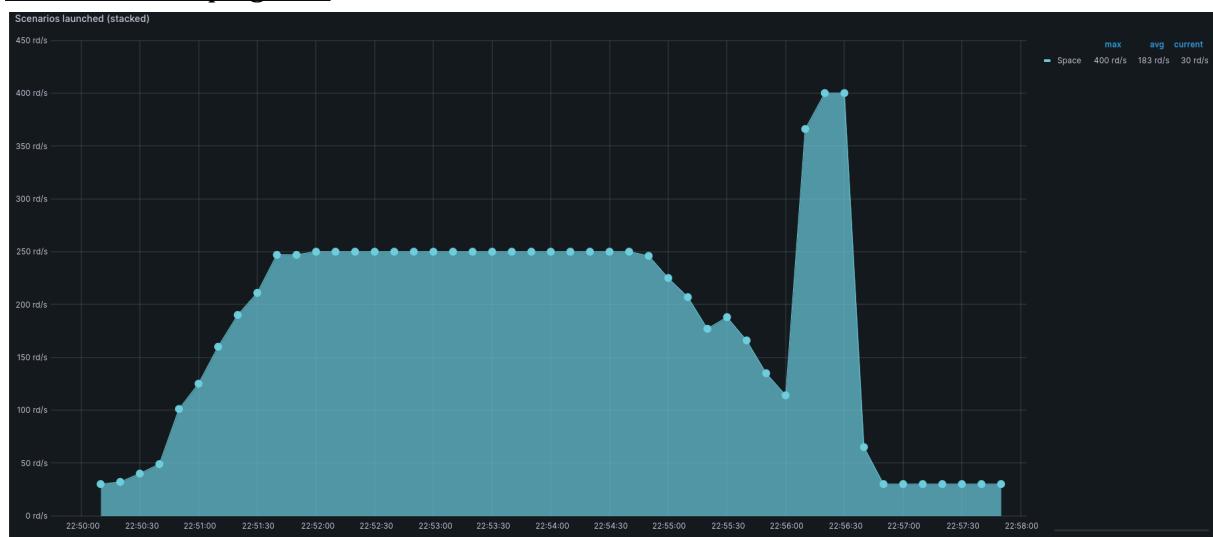
En cuanto al tiempo de respuesta percibido por el cliente, es interesante ver que la mediana se desploma completamente cuando alcanzamos aquellos puntos en los que las requests limitadas se incrementan; esto tiene mucho sentido, pues las requests limitadas incurren en un costo temporal mínimo correspondiente a simplemente

devolver el error informando que la request no será procesada. Sobre los valores máximos, podemos ver que, respecto al caso base, los mismos suelen mantenerse constantes al principio, probablemente gracias al Rate Limiting utilizado, pero a medida que se sostiene la carga empiezan a aparecer algunos picos, lo cual no es de extrañarse considerando que estamos enviando una carga constante. En cuanto a la latencia, tanto para API como para nuestro servidor, vemos que son valores relativamente constantes que suelen rondar los 700 milisegundos, lo que tiene sentido si consideramos que estamos mandando una cantidad constante de requests. Esto es bastante distinto en comparación a los valores obtenidos en el caso base, donde la latencia tenía a fluctuar mucho y llegaba a valores de hasta 2 segundos para la API.

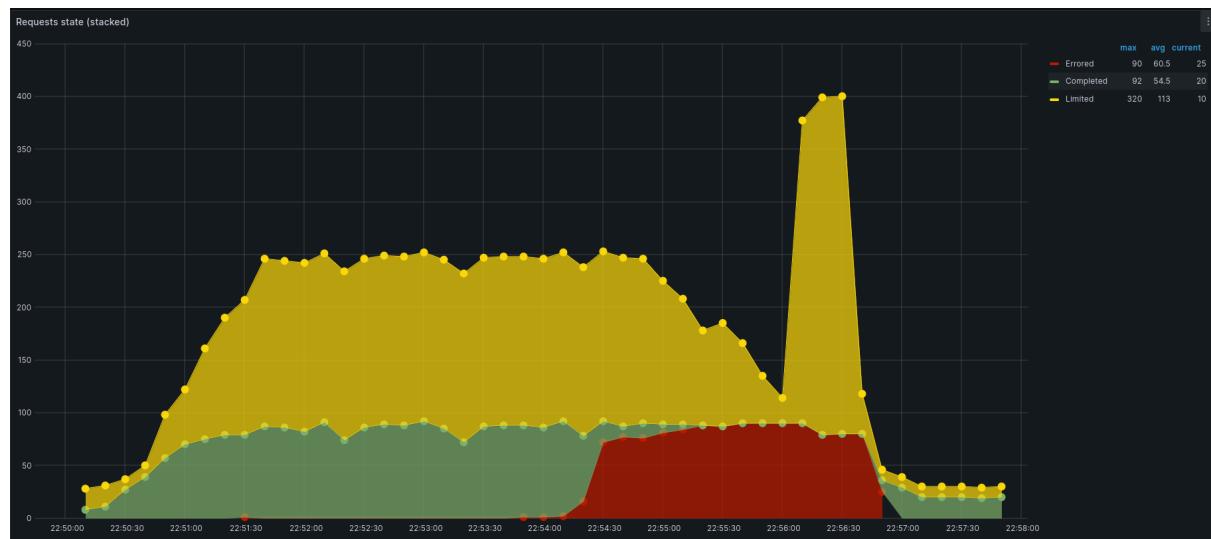
Sobre los recursos utilizados, nuevamente ocurre lo mismo que /ping: almacenamiento constante en memoria y un uso de la CPU correlativo a las requests que se ejecutan y no a las limitadas.

spaceflight_news

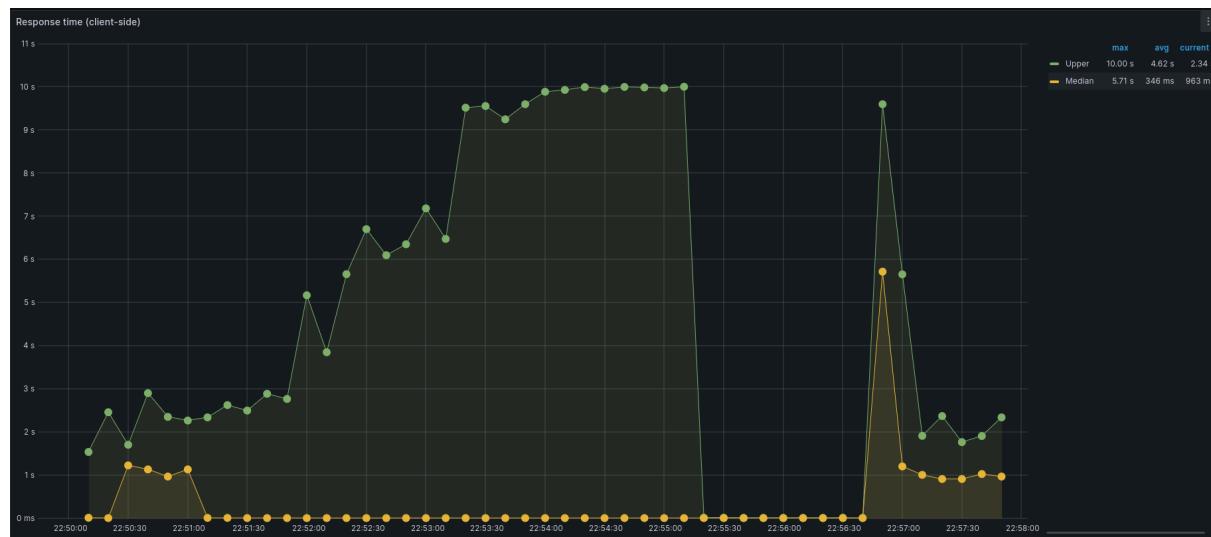
Escenarios desplegados



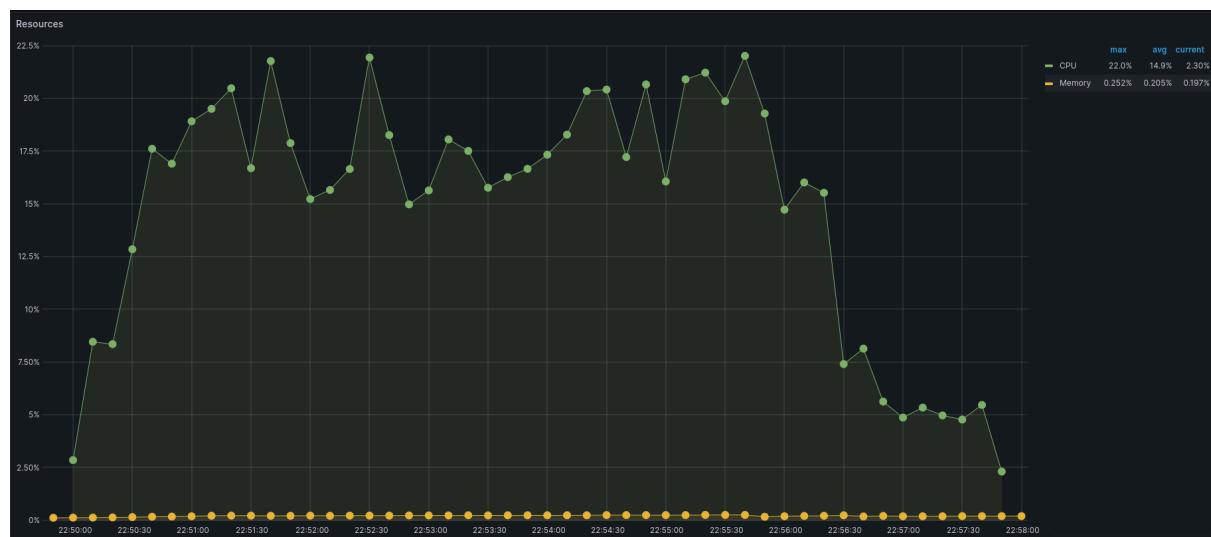
Estado de las requests



Tiempo de respuesta (lado del cliente)



Recursos utilizados



Latencia de la API externa



Latencia del Endpoint



Informe de Artillery

```
--  
Summary report @ 22:57:53(-0300)  
--  
  
errors.ETIMEDOUT: ..... 1210  
http.codes.200: ..... 2097  
http.codes.500: ..... 1  
http.codes.503: ..... 5308  
http.request_rate: ..... 18/sec  
http.requests: ..... 8616  
http.response_time:  
    min: ..... 0  
    max: ..... 9995  
    median: ..... 1  
    p95: ..... 8186.6  
    p99: ..... 9607.1  
http.responses: ..... 7406  
vusers.completed: ..... 7406  
vusers.created: ..... 8616  
vusers.created_by_name.Space: ..... 8616  
vusers.failed: ..... 1210  
vusers.session_length:  
    min: ..... 1.1  
    max: ..... 9996.9  
    median: ..... 3.1  
    p95: ..... 8186.6  
    p99: ..... 9801.2
```

Comentarios

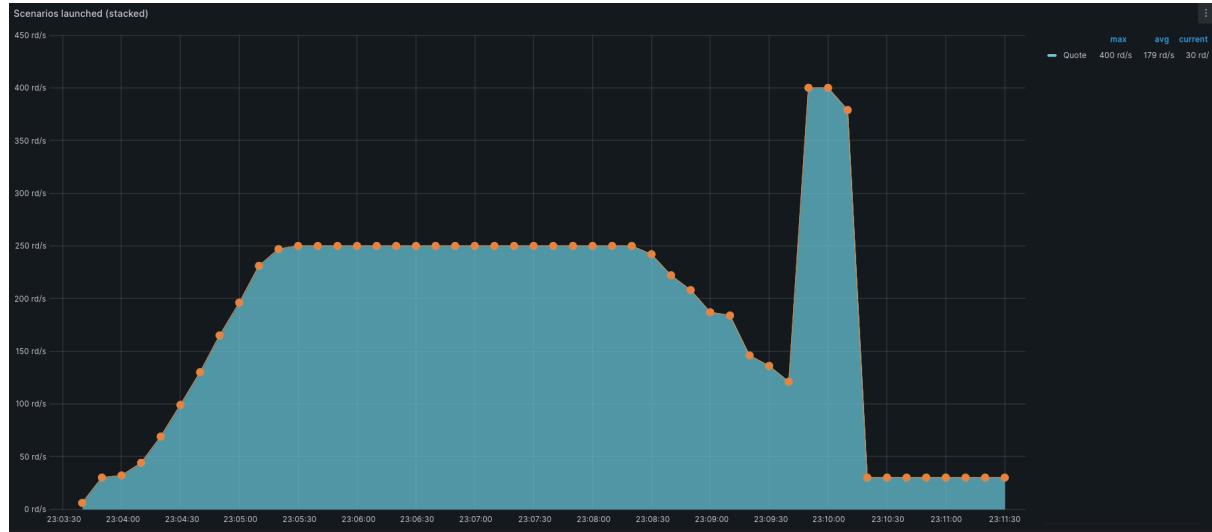
Para este Endpoint, vemos una leve mejoría respecto del caso base. En cuanto al estado de las requests, es interesante ver que la cantidad de **errores disminuyó considerablemente**, y la API no se volvió inutilizable luego del período de 'Ramp', como sí había pasado anteriormente.

Respecto al tiempo de respuesta percibido por el cliente, vemos que aumenta considerablemente a lo largo del tiempo, pero no tanto así como ocurría en el caso base, llegando a los **10 segundos sólo después de unos 4 minutos**, y no luego de poco más de 1 minuto. En cierto punto, dicho tiempo se desploma completamente, algo que se condice con la predominancia de requests limitadas por sobre las completadas. Respecto a los gráficos de latencia, es interesante notar cómo crecen de forma menos brusca en comparación al caso base, y para el caso de la latencia de la API dicho gráfico se mantiene continuo. Esto es así pues el Rate Limiting contribuye a evitar el Gateway Timeout obtenido por axios en varios casos donde la API se sobrecarga. Una vez que la carga disminuye, ambas latencias se reducen considerablemente.

Respecto a los recursos utilizados, la API incurre en el mismo patrón que el caso base, pero con un porcentaje de utilización mucho menor, pues hay una menor cantidad de requests enviándose al mismo tiempo.

quote

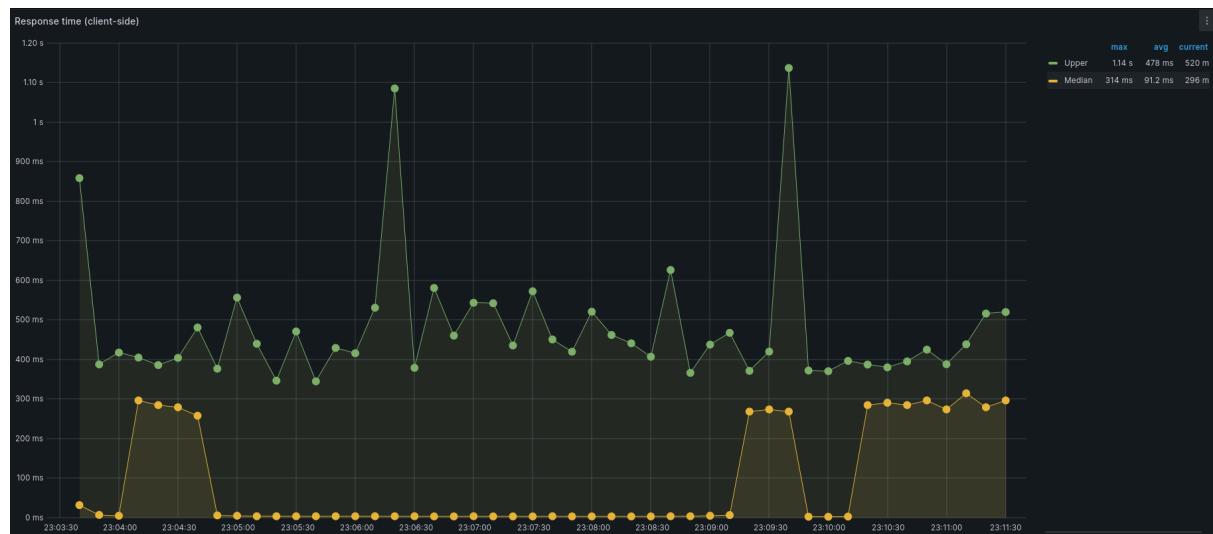
Escenarios desplegados



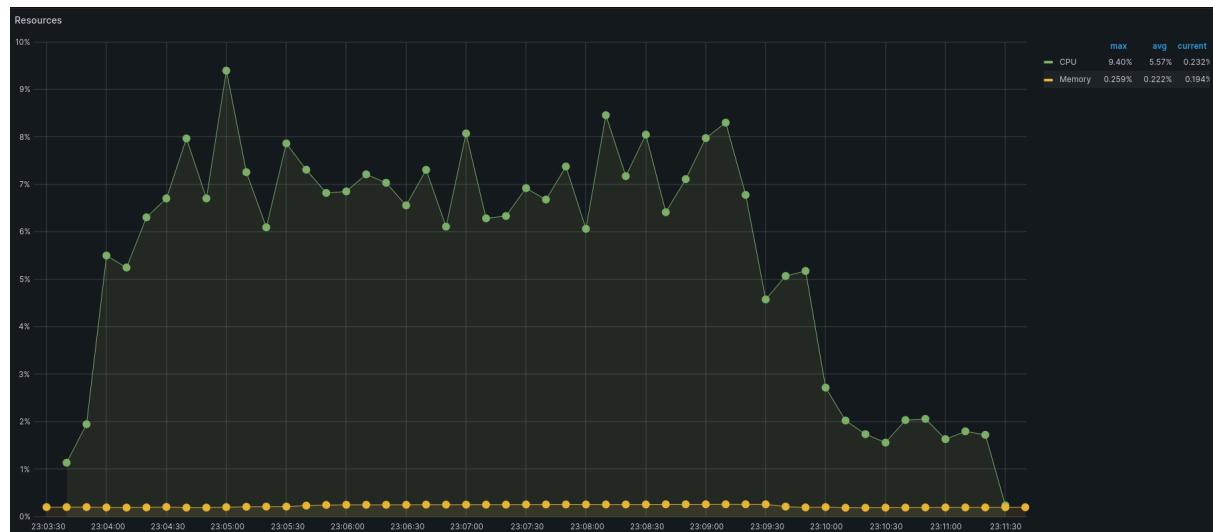
Estado de las requests



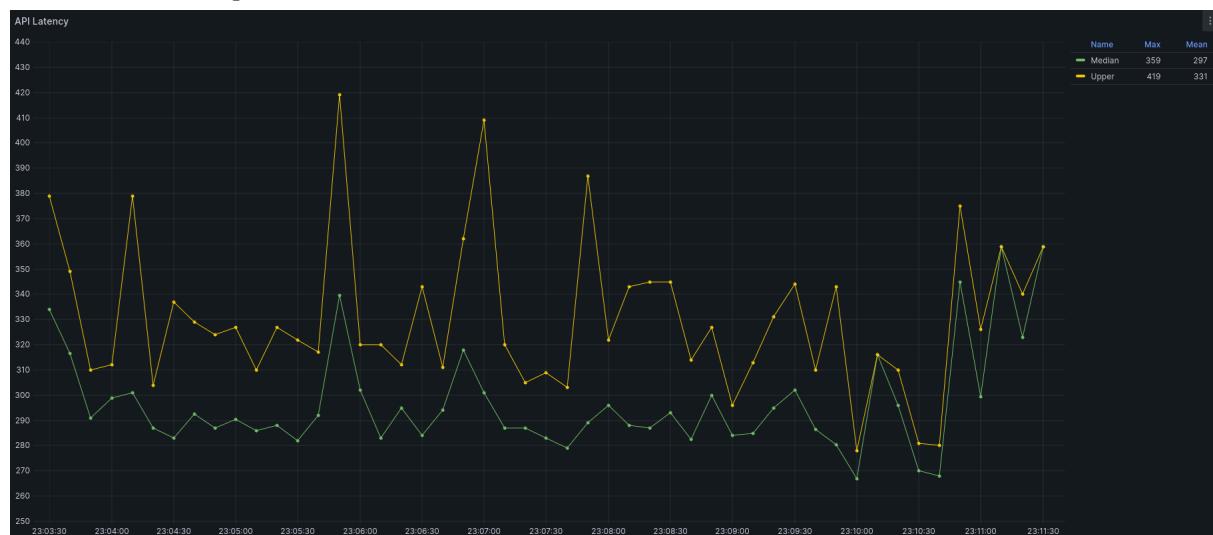
Tiempo de respuesta (lado del cliente)



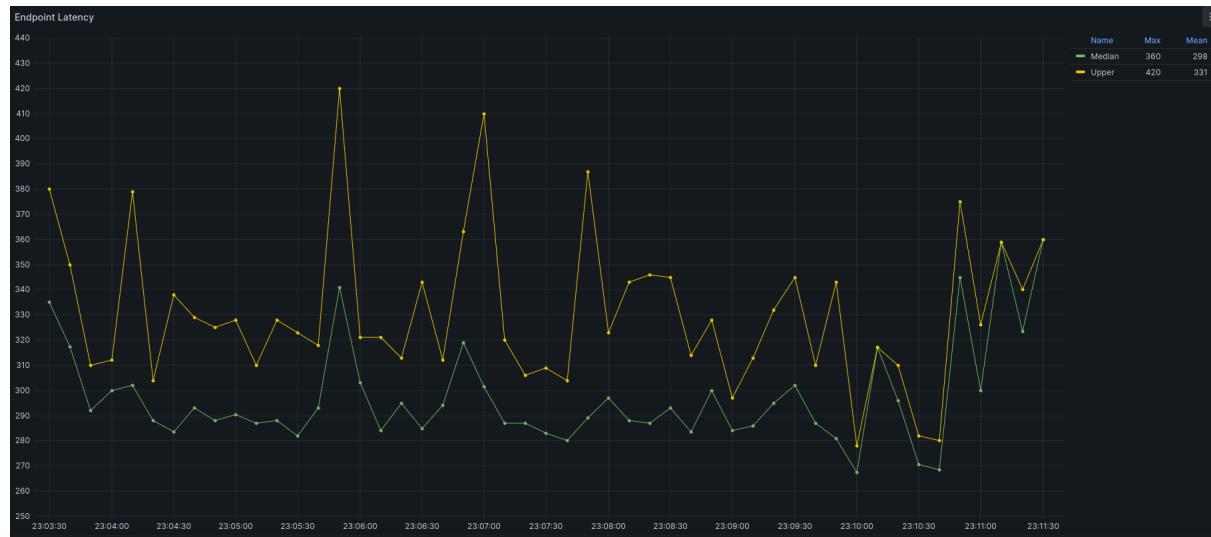
Recursos utilizados



Latencia del Endpoint



Latencia del Endpoint



Informe de Artillery

```
-----  
Summary report @ 23:11:30(-0300)  
-----  
  
http.codes.200: ..... 2754  
http.codes.503: ..... 3192  
http.request_rate: ..... 12/sec  
http.requests: ..... 5946  
http.response_time:  
    min: ..... 0  
    max: ..... 1134  
    median: ..... 1  
    p95: ..... 320.6  
    p99: ..... 376.2  
http.responses: ..... 5946  
vusers.completed: ..... 5946  
vusers.created: ..... 5946  
vusers.created_by_name.Quote: ..... 5946  
vusers.failed: ..... 0  
vusers.session_length:  
    min: ..... 1.2  
    max: ..... 1136.4  
    median: ..... 3.9  
    p95: ..... 327.1  
    p99: ..... 383.8
```

Comentarios

Para este endpoint, vemos que, sobre el estado de las requests, no hay ningún tipo de error ni Rate Limiting impuesto por la API externa, como bien se pudo apreciar desde el principio.

Respecto al tiempo de respuesta percibido por el cliente, vemos una situación similar al caso base, aunque en varios casos se desploma a raíz del tiempo tardado por aquellas requests limitadas. Independientemente de esos casos, la mediana se mantiene alrededor de los 300 milisegundos como en el caso base. Sobre la latencia, tanto de Endpoint como de la API externa, tampoco hay mejorías relevantes, aunque vemos que los picos de los valores máximos son mucho menores respecto al caso base. Sin Rate

Limiting, se llega a alcanzar picos de hasta 1 segundo, mientras que en este caso no se sobrepasan los 420 milisegundos, aunque la mediana se mantiene similar.

Respecto al uso de recursos, el porcentaje de utilización de CPU es menor que en el caso base pues se están completando menos requests.

En este sentido, podemos ver que esta API no se beneficia demasiado del Rate Limiting para los escenarios propuestos, pues ya desde un principio en el caso base no hubo ningún tipo de problema a la hora de responder las requests pedidas.

Conclusiones

Comparaciones

Ping

Estrategias	Comentarios
Caso base	<ul style="list-style-type: none">- Al tener procesamiento mínimo los tiempos son muy pequeños, tanto lo percibido por el cliente como la latencia del servidor- Todas las requests fueron respondidas correctamente- El uso del CPU respeta un patrón similar al escenario propuesto- Memoria constante ya que no se requiere almacenamiento
Caché	No requiere caché ya que no se consulta a API externa
Replicación	<ul style="list-style-type: none">- Se manejan los mismos tiempos aun así con 2 nodos más, posibles factores<ol style="list-style-type: none">1. Al ser tan simple con un nodo se hizo todo lo más veloz posible2. Trabas de nginx3. Cuello de botella en el nodo del reverse proxy y no es posible mejorarlo más incluso con más nodos
Rate Limiting	<ul style="list-style-type: none">- Se respeta el límite impuesto ⇒ 90 request completadas cada 10 segundos- Sin cambios significativos en tiempo de respuesta para el cliente- Tiempo invertido en la comunicación con el cliente ⇒ Latencia similar al base, pero en menor medida- Uso de memoria constante- Uso de CPU varía en proporción a la cantidad de requests completadas
Como es un endpoint con poca funcionalidad es preferible utilizar la estrategia de caso base, ya que si bien agregar replicación a la larga ayudaría, terminas agregando algo sobre un endpoint con funcionalidad nula y no valdría la pena.	

Dictionary

Estrategias	Comentarios
Caso base	<ul style="list-style-type: none">- En promedio 35 requests fueron completadas- Limitaciones por parte de la API con Rate Limiting (3 por segundo)- Tiempo de respuesta para el cliente en órdenes de segundos (promedio de 473 ms la mediana)- Latencia entre API y servidor son similares, servidor añade un mínimo procesamiento- CPU con patrón similar al escenario propuesto y memoria constante
Caché	<ul style="list-style-type: none">- En promedio 47 requests fueron completadas- La mediana del tiempo de respuesta (lado del cliente) tuvo un promedio de 176 ms- Se observa que inicialmente la mediana es alta y luego va bajando a medida que llegan nuevas (por estar cacheados)

Replicación	<ul style="list-style-type: none"> - Se comportó como esperábamos - Request completados y limitados igual que en caso base - Agregar más containers no sirve de nada ya que, si bien internamente tienen diferentes IPs, en el 'exterior' el request de cualquier nodo tiene la IP del host - Pareciera haber mejora pero solo sobre un nodo, en el conteo final se mantuvo igual que caso base - Mejora en las latencias pero no es significativa - Al tener más nodos se distribuye el cómputo y así el uso de los recursos ⇒ no sobrecargamos un nodo y damos lugar a procesar requests
Rate Limiting	<ul style="list-style-type: none"> - Algunas requests limitadas por la API otras por el Endpoint - + requests limitadas ⇒ - mediana del tiempo. Devolver el error conlleva un costo mínimo - Picos en tiempo de respuesta constantes al inicio. A medida que se sostiene la carga constante aparecen picos - Latencia del servidor y API alrededor de los 700 ms - Uso de memoria constante, y el uso del CPU está correlacionado con las requests completadas y no con las limitadas
La que mejor se adapta en este caso es el de caché, ya que vemos una mejora notable en cuanto a la funcionalidad del mismo, que en caso de ya estar cacheados (que es probable) respondería con rapidez	

Spaceflight News

Estrategias	Comentarios
Caso base	<ul style="list-style-type: none"> - Request rechazadas a partir de la mitad de la fase 'Ramp' - Error 499 causado por Nginx ya que cierra conexiones que tardan demasiado para evitar ralentizaciones - Tiempos de respuestas altos: Las requests completadas van de los 2 segundos a los 10 segundos antes de ser rechazadas por Nginx (en 1 min), luego la disponibilidad es nula - Latencia del servidor de 200 segundos ya que el servidor sigue intentando procesar la request - Error 504 en Axios cuando hay un timeout contra la API - CPU con patrón similar al escenario propuesto (pico de aprox. 50%) y memoria constante
Caché	<ul style="list-style-type: none"> - Se respondieron todos los request y con un tiempo de respuesta extremadamente bajo - Los tiempos de respuesta son del orden de los milisegundos desde la primer request y se mantienen constante - Aunque la latencia es de segundos no es problemáticos por la poca cantidad de requests - Mejora en el uso del CPU: El promedio bajó del 23,5% al 3,15%, con un pico máximo de 7,50%, ya que el caché evita la acumulación de conexiones abiertas
Replicación	<p>Sucede lo que esperábamos ⇒ una gran cantidad de requests fallidas</p> <ul style="list-style-type: none"> - + requests ⇒ + latencia, hasta tal punto de no contestar - Nginx cierra los nuevos request - Empeora la situación ya que esta estrategia ayuda en hacer más llamados a la API en menor tiempo
Rate Limiting	<ul style="list-style-type: none"> - Los errores disminuyeron considerablemente y ahora la API no se volvió inutilizable después de la fase 'Ramp' - Aunque aumenta el tiempo de respuesta llega a los 10 segundos en 4 minutos - Tiempo de respuesta cae abruptamente cuando las request limitadas superan a las completadas

	<ul style="list-style-type: none"> - Se evitan los Gateway Timeout y la latencia crece pero menos abruptamente, manteniéndose continua - Reducir la cantidad de request simultáneas ⇒ menor porcentaje de utilización de recursos
En este caso el caché es el que mejor se adaptaría, ya que consultando las noticias dentro de un rango de tiempo es muy probable que tengan concordancias y esto ayudaría a bajar los tiempos de respuesta. De todos modos, Rate Limiting podría ayudar a evitar que se colapse	

Quote

Estrategias	Comentarios
Caso base	<ul style="list-style-type: none"> - Requests respondidas con éxito, la API externa no aplica Rate Limiting - Mediana del tiempo es menor a 300 ms, pero algunas request alcanzan hasta 1 seg durante la fase 'Plain 2' debido a la alta carga de requests - Nula diferencia entre la latencia de la API externa y del Endpoint ⇒ servidor añade poca latencia - Picos en la latencia y el tiempo de respuesta del cliente correlacionados ⇒ tiempo de respuesta depende principalmente de la API - CPU con patrón similar al escenario propuesto y memoria constante
Caché	No fue implementado por su aleatoriedad
Replicación	<p>No se cumplió lo que pensábamos → Más nodos + API veloz, sin Rate Limiting ⇒ mejoras en tiempo de respuesta y latencia</p> <ul style="list-style-type: none"> - Este incumplimiento puede deberse a dos posibles factores: <ol style="list-style-type: none"> 1. Nodo de nginx como cuello de botella 2. Número de requests baja como para notar diferencia significativa
Rate Limiting	<ul style="list-style-type: none"> - Sin errores ni Rate Limiting por parte de la API - Mediana del tiempo de respuesta alrededor de 300 ms y caídas debido a algunas requests limitadas - Sin mejoras en latencia, pero los picos son menores (420 ms de pico) - Menor cantidad de requests completadas ⇒ Menor uso de la CPU - No se beneficia de esta estrategia ya que en el base no había problemas de respuesta
En este caso, para los escenarios que fueron testeados, notamos que el caso base es suficiente para satisfacer la demanda de pedidos. De todas maneras, es posible que, frente a cargas más altas, las tácticas de Replicación y Rate Limiting puedan ser muy beneficiosas.	

Lecciones aprendidas

- **Impacto de las tácticas en el rendimiento:** En este Trabajo Práctico fuimos capaces de identificar y comparar el impacto que cada una de las tácticas implementadas tenía sobre el sistema. Pudimos hacer un análisis exhaustivo de cada una de las métricas para definir si lo que pensábamos que iba a ocurrir con cada táctica era lo que realmente terminó sucediendo.
- **Limitaciones de APIs externas:** Uno de los mayores desafíos que tuvimos fue saber adaptarnos a las limitaciones que presentaban las APIs externas. Algunas tenían Rate Limiting propio lo que afectaba directamente a nuestra aplicación, ya que nos impone una cantidad máxima de requests que podía soportar nuestro endpoint. Sin embargo, fuimos capaces de detectarlo e incluirlo en los análisis. Otras como quotable, tenían serios problemas de disponibilidad, lo cual nos obligó a cambiar a otra API externa con una funcionalidad similar. Y también entendimos que la performance de nuestra aplicación iba a depender fuertemente de la performance de las API externas particulares que estuviéramos consumiendo.
- **Medición de métricas:** Entendimos la importancia de medir correctamente las métricas de nuestra aplicación. Esto nos permitió ver cómo se comportaba en distintos escenarios reales de carga y con diferentes tácticas. Fuimos capaces de comprender cada uno de los gráficos y analizarlos correctamente.
- **Importancia de informarse sobre las APIs externas:** Al leer detalladamente cada una de las documentaciones de las APIs externas utilizadas en este trabajo, entendimos que es crucial para poder tener todas las herramientas para poder tomar las mejores decisiones de implementación basados en cómo funcionan las APIs externas.

Referencias

A continuación se detallan las referencias utilizadas durante el desarrollo del Trabajo Práctico:

- Nodejs: <https://nodejs.org/>
- Nginx: <https://nginx.org/>
- Redis: <https://www.npmjs.com/package/redis>
- Grafana: https://docs.grafana.org/guides/getting_started/
- Artillery: <https://artillery.io/docs/>
- Generacion de carga: <https://queue-it.com/blog/load-vs-stress-testing/>
- Hotshots: <https://www.npmjs.com/package/hot-shots>
- Generacion dinamica de requests con Artillery:
<https://www.artillery.io/docs/reference/test-script#payload---loading-data-from-csv-files>
- Archivo CSV con palabras: <https://www.mit.edu/~ecprice/wordlist.10000>
- Rate Limiting con Nginx: <https://blog.nginx.org/blog/rate-limiting-nginx>