



# Trabajo Práctico N°1

[TB054] Arquitectura de Software  
Primer cuatrimestre de 2024

Integrantes:

- 107705 - Lucas Rafael Aldazabal
- 109386 - Juan Pablo Carosi Warburg
- 109635 - Mateo Daniel Vroonland
- 107490 - Bautista Boselli

# Índice

<b>Introducción.....</b>	<b>3</b>
Sobre la aplicación.....	3
Sobre las pruebas.....	4
Sobre las tácticas.....	5
Sobre la Ejecución.....	6
<b>Arquitecturas.....</b>	<b>7</b>
Caso base.....	7
Cache.....	8
Replicación.....	9
Rate limiting.....	10
<b>Dictionary.....</b>	<b>11</b>
Caso Base.....	11
Cache.....	13
Replicación.....	15
Rate Limiting.....	16
Conclusión.....	18
<b>Spaceflight News.....</b>	<b>19</b>
Caso Base.....	19
Cache.....	21
Replicación.....	23
Rate Limiting.....	25
Conclusión.....	27
<b>Random quote.....</b>	<b>28</b>
Caso Base.....	28
Cache.....	30
Replicación.....	32
Rate Limiting.....	33
Conclusión.....	34
<b>Táctica “Ideal”.....</b>	<b>35</b>
Dictionary.....	36
Spaceflight News.....	37
Random Quote.....	38
<b>Conclusiones Generales.....</b>	<b>39</b>

# Introducción

En el siguiente informe se describen y comparan distintas tácticas de arquitectura de software contra un caso base de una aplicación Node.js cuyo funcionamiento consiste en un servicio HTTP que representa una API que consume otras APIs para retornar una determinada información a los usuarios que la consultan.

Las tácticas utilizadas son:

- Cache
- Replicación
- Rate-Limiting

Cada una de esta se implementa por separado manteniendo el funcionamiento de la aplicación y se las compara a cada una contra el caso base.

Se crean escenarios personalizados generando una determinada carga en la aplicación y se analiza cómo las distintas tácticas impactan en los atributos de calidad mediante la visualización de mediciones obtenidas al generar la carga.

## Sobre la aplicación

La aplicación cuenta con los siguientes endpoints a los cuales se les puede realizar peticiones.

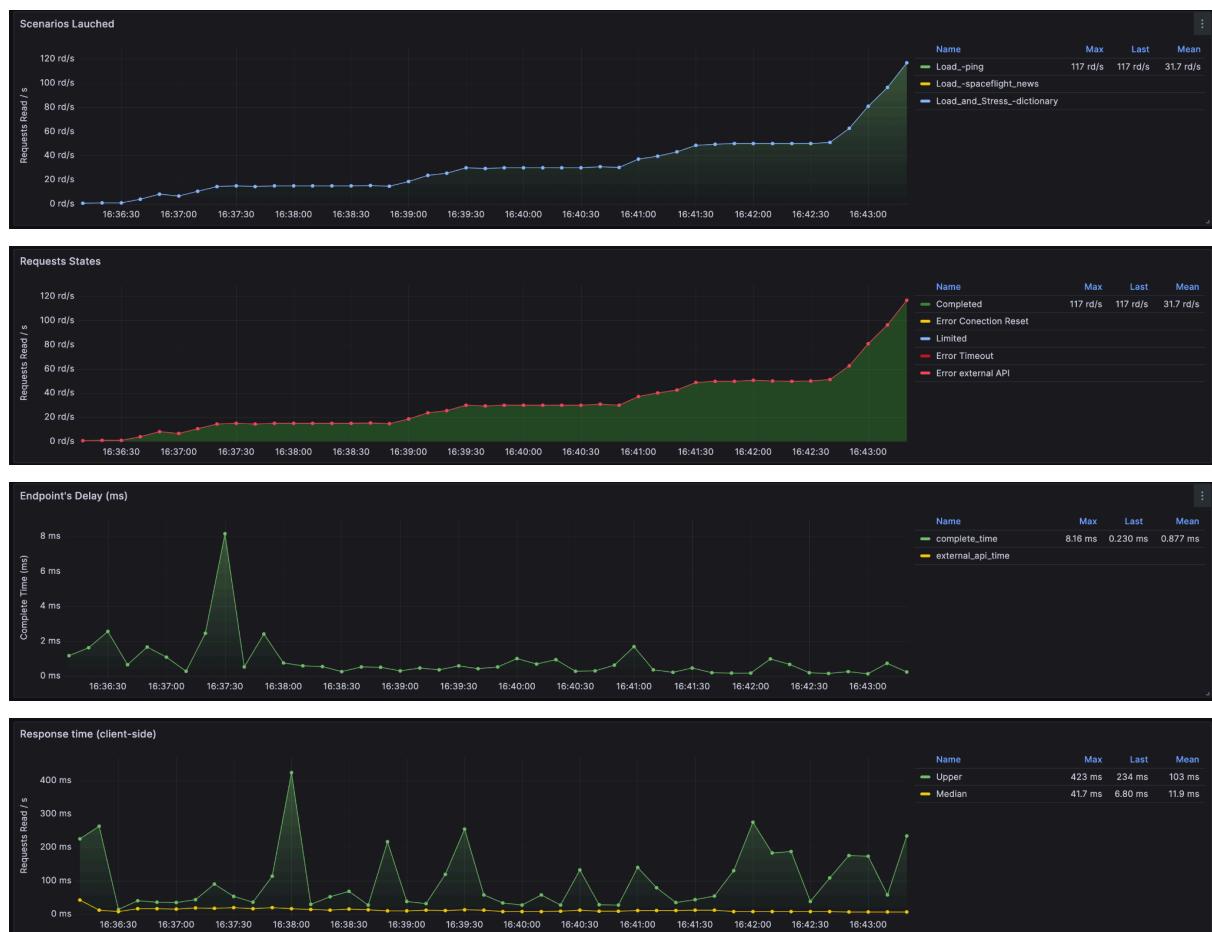
- **Ping:** Se responde Pong y se lo utiliza como healthcheck y como baseline contra los demás endpoints
- **Dictionary:** Se le pasa una palabra y responde con su fonética y significado. Al generarle carga se busca que la palabra sea distinta en cada petición
- **Spaceflight News:** Devuelve los títulos de las ultimas 5 noticias relacionadas con la actividad espacial
- **Quote:** Devuelve una cita aleatoria de algún autor

## Sobre las pruebas

Se crea un escenario para generar carga distinta para cada endpoint donde se utiliza una variante de Load Testing, combinando Load Testing con Stress Testing. Todos arrancan con una fase de warm up donde se envían unas pocas requests por segundo durante cierto tiempo.

Luego tienen varias etapas similares donde siempre se va aumentando o manteniendo la carga. Estas etapas consisten de una fase de ramp, donde progresivamente se aumenta la cantidad de requests enviadas, seguida de una fase de load testing, donde se mantiene la tasa de requests enviadas constante durante un periodo de tiempo determinado. Y así sucesivamente las etapas siguen aumentando la cantidad de requests sobre la etapa anterior. Finalmente, se realiza un ramp abrupto al final que sirve como spike y demanda el TP fuertemente.

Este escenario personalizado varía con cada endpoint pero no con la táctica utilizada, a fin de poder comparar cómo afectan estas a un determinado endpoint bajo las mismas condiciones. Este test también nos permite visualizar el funcionamiento de un mismo endpoint en múltiples circunstancias en vez de solo una al combinar distintos tipos de testing.



En las imágenes se ejecutó el escenario con el endpoint Ping. Todos los análisis se realizan en un dashboard personalizado de grafana con los mismos 4 gráficos. Estos son:

- 1- Scenarios Launched: que indica a lo largo del tiempo cuantos requests fueron enviados
- 2- Requests State: que indica a lo largo del tiempo cuántos de los requests fueron respondidos con código de status 200 (exito) y cuantos con errores ya sean por timeouts o por errores de respuesta por parte de la API externa.
- 3- Endpoint's delay: son las métricas propias pedidas en la consigna y se ven en un mismo gráfico, cuánto tiempo se tardó en consultar a la API externa y por otro lado cuánto se tardó en consultar la API externa sumado al procesamiento propio.
- 4- Response time(client-side): que indica cuanto tiempo tardo el cliente en recibir una respuesta de la aplicación node.

En el primer gráfico se pueden observar las distintas etapas de ramp seguidas de fases de load constante a una misma tasa con un spike abrupto al final como se describió inicialmente. Como este endpoint es más bien sencillo, no falla ninguna de las requests enviadas. Es importante aclarar que entre cada endpoint varía los valores pero no la forma, es decir, las fases descriptas previamente se mantienen entre los escenarios de cada endpoint pero por ejemplo puede cambiar a qué valor se realiza el ramp y a que valor se realiza la fase load constante, esto varía según las características y capacidades de cada endpoint.

## Sobre las tácticas

Las tácticas se implementan sobre cada endpoint de manera distinta, aprovechando las características de cada uno, para así poder visualizar un mayor impacto de esas tácticas sobre los distintos endpoints en las mediciones a diferencia del impacto con una misma implementación común a cada endpoint.

## Sobre la Ejecución

Para ejecutar la aplicación es necesario levantar el docker compose en una consola.

Este se puede realizar con el comando:

```
$ docker compose up
```

Para ejecutar los distintos escenarios con las distintas tácticas primero es necesario ir a la branch correspondiente a la táctica que se quiere ejecutar. Esto se hace parado en la terminal en el repositorio y ejecutando el siguiente comando reemplazando la palabra branch por la correspondiente a la táctica que se quiere analizar:

```
$ git checkout <branch>
```

Se dispone de las siguientes branches:

- base
- cache
- replicating
- limiting
- ideal

Una vez que ya se está sobre la branch correspondiente a la táctica que se quiere visualizar, ejecutar los siguientes comandos reemplazando la palabra escenario por la correspondiente al escenario que se quiere probar para generar escenarios de carga con artillery:

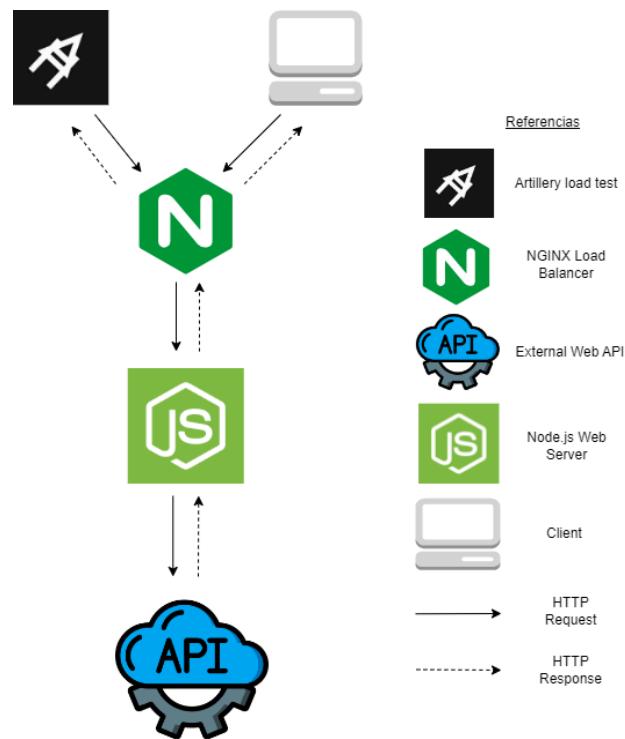
```
$ cd perf  
$ npm run scenario <escenario>
```

Se dispone de los siguientes escenarios:

- dictionary
- spaceflight
- quotes
- ping

# Arquitecturas

## Caso base

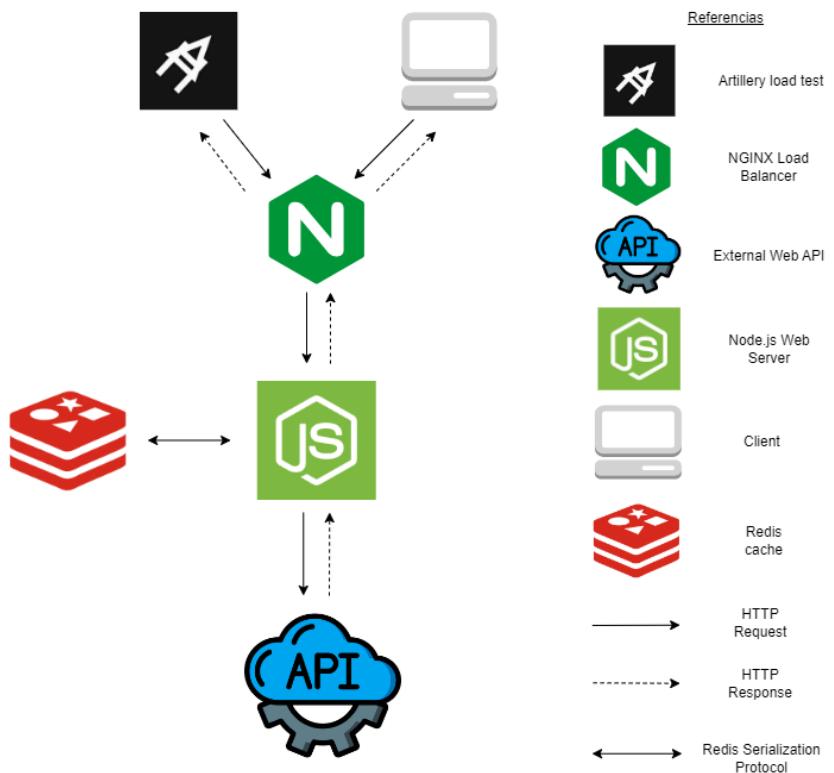


Este es el diagrama de componentes y conectores de la arquitectura del caso base. Partiendo de esta estructura luego se agregan componentes para realizar cada táctica.

La misma consiste de:

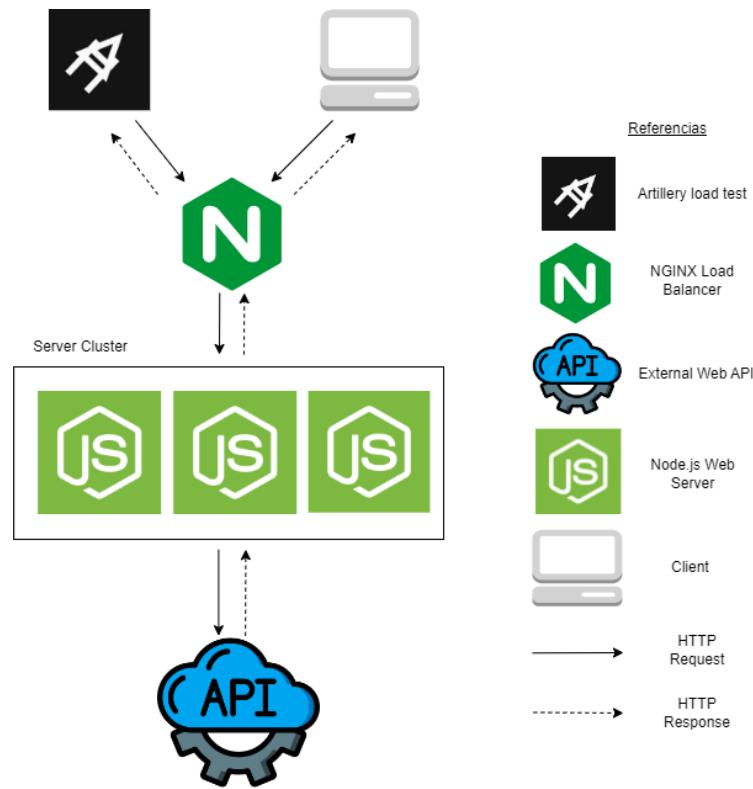
- Artillery: para simular la carga que va a recibir el sistema
- Nginx: punto de entrada de los clientes a la API, redirige las peticiones al servidor
- Node.js Web Server: se encarga de manejar las peticiones del artillery y comunicarse con las APIs externas

## Cache



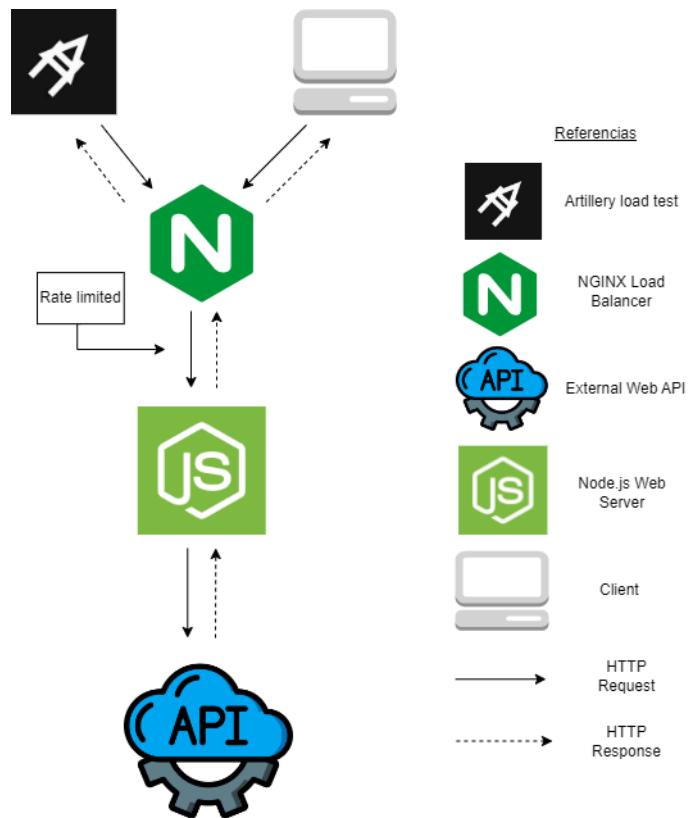
Para el diagrama de esta táctica se agrega un nuevo componente, una caché de redis, la cual va a interactuar solo con el servidor mediante el redis serialization protocol. De esta manera, el servidor va a guardar información en memoria, evitando tener que recurrir a la API externa en múltiples ocasiones.

## Replicación



Para la siguiente táctica se escala horizontalmente el servidor a un total de tres réplicas, formando un clúster de servidores. El load balancer se encarga de distribuir las requests entre las tres réplicas.

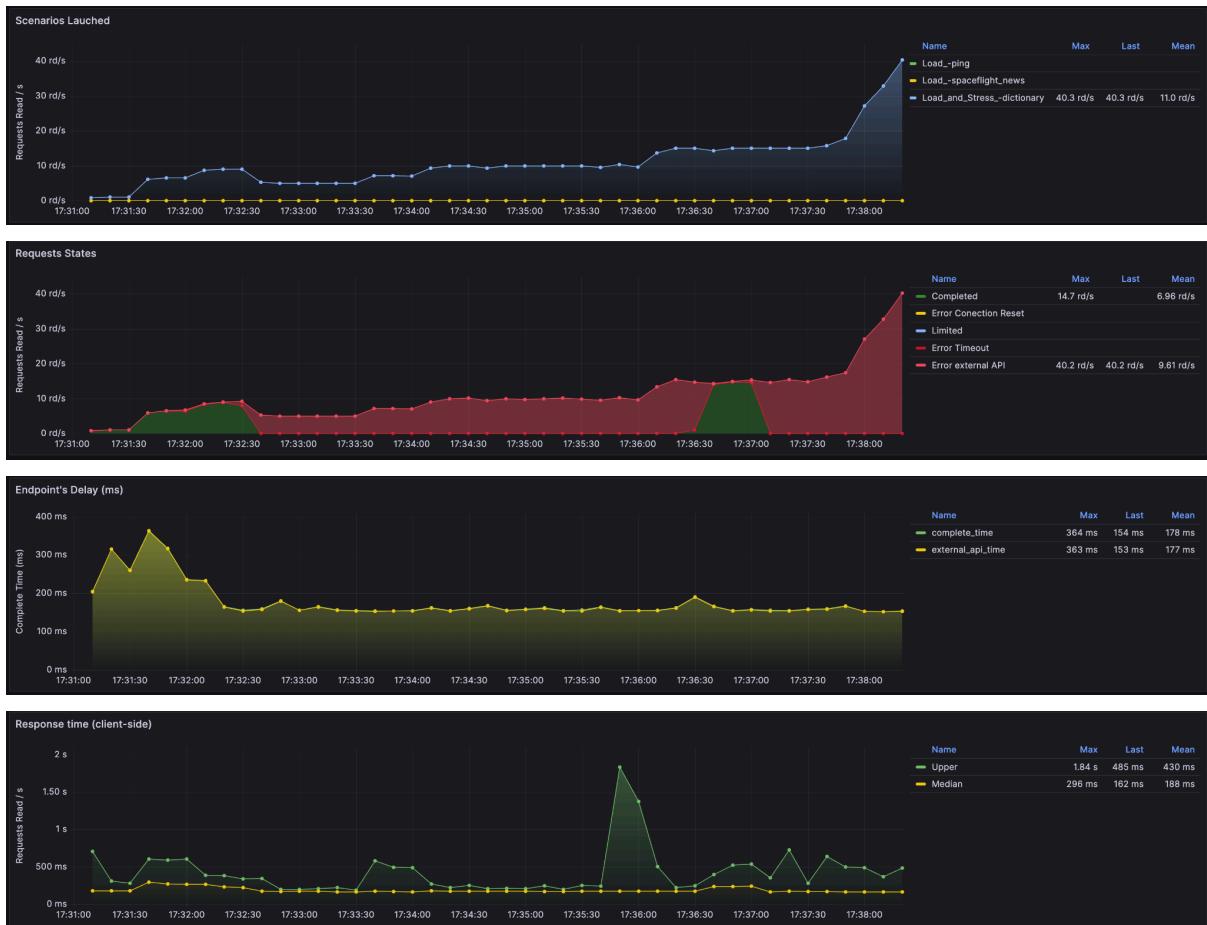
## Rate limiting



Este es el diagrama de componentes y conectores de la arquitectura para la táctica de rate limiting. El componente encargado de limitar es el servidor ya que se implementa el rate limiting mediante el uso del rate limiter que provee Express. Se implementó de esta manera para poder asignar a cada endpoint un rate-limiting específico y no utilizar uno global para todos.

# Dictionary

## Caso Base



### Observaciones:

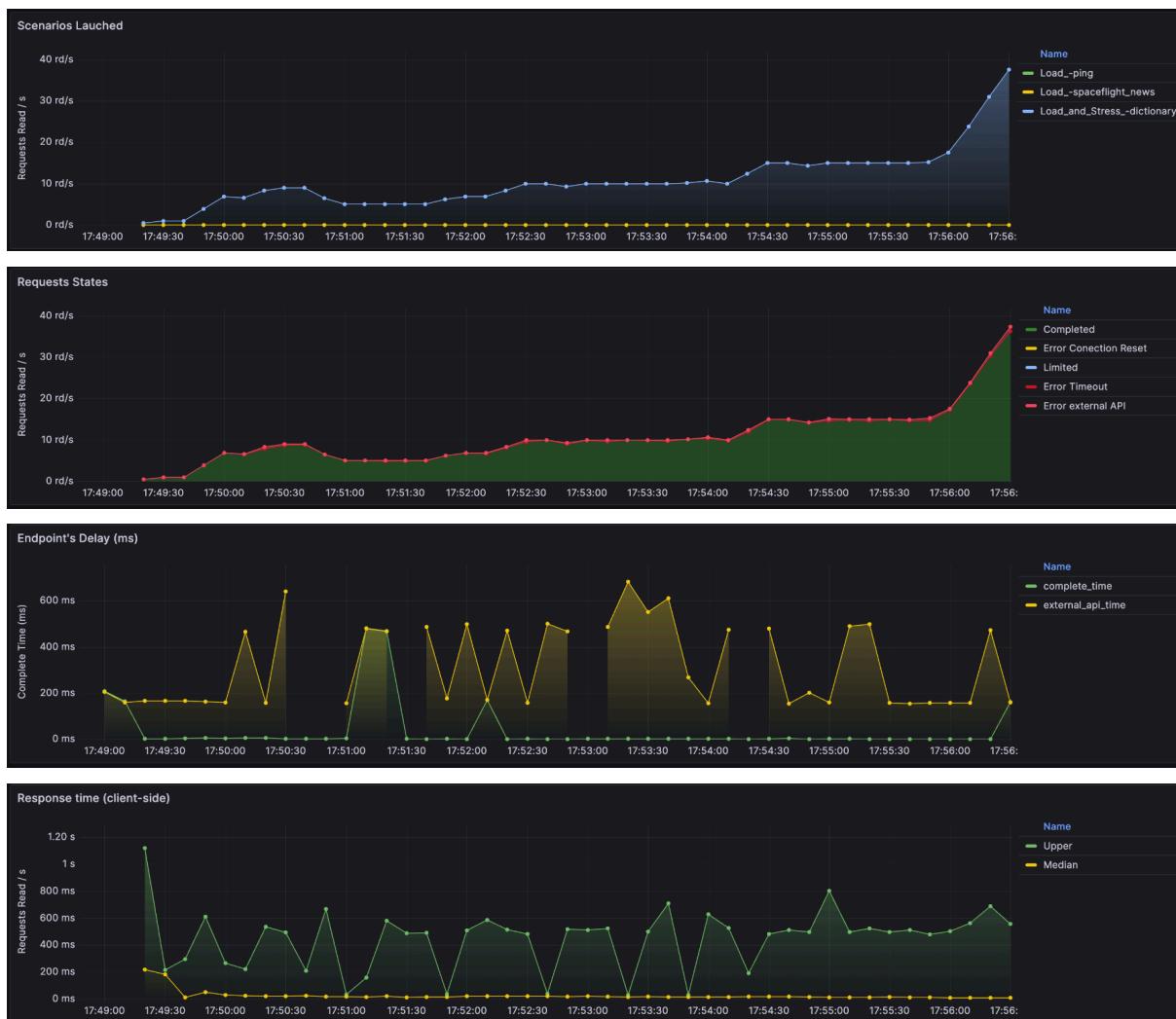
- Se observa que la API externa es cuello de botella, se puede ver ya que casi no aumentan los Endpoint's delays y porque no aumenta el response time salvo alguna excepción, por lo que el servidor no suele tener problemas para responder al cliente.
- Se puede apreciar en el gráfico de los requests states una window de 5 minutos en la API externa, tras completar una cantidad de requests inicialmente durante un minuto aproximadamente, da timeout durante aproximadamente 4 minutos antes de volver a aceptar más requests, la disponibilidad es muy baja para la cantidad de request que permite la API externa.
- Complete time y external API time son similares para el caso base ya que lo que demora es la API externa y no un procesamiento propio

**Posibles Mejoras:**

- Cache podría servir para ahorrar requests a la API externa en palabras repetidas
- Limitar la cantidad de requests por usuario podría servir para tener mayor disponibilidad a diferentes usuarios
- La replicación sería útil si el rate limiting es por cada servidor, de esta forma triplicando la cantidad de requests posibles en la window

## Cache

Se implementa cache con lazy population ya que no hay forma sencilla de predecir qué palabra busca el usuario a continuación, su uso acelera la respuesta si se llega a repetir una palabra previamente buscada. De esta manera, a lo largo del tiempo el servicio implementado tendría cada vez más palabras cacheadas, acercándose al caso ideal que nombra Roy Fielding donde la base está en local. Sin embargo, como toda decisión trae un trade off, en este caso es que se tiene un programa más grande en memoria debido a que el caché nunca es liberado. De todas formas, esto no debería ser un problema en sistemas no acotados en este aspecto, ya que el diccionario entero no debería suponer más de 1gb de ram (basándonos en el peso de algunos diccionarios que encontramos en diferentes páginas como [wikipedia](#) entre otras), y de ser necesario puede liberarse la cache desde el endpoint /reset-cache.



### Observaciones:

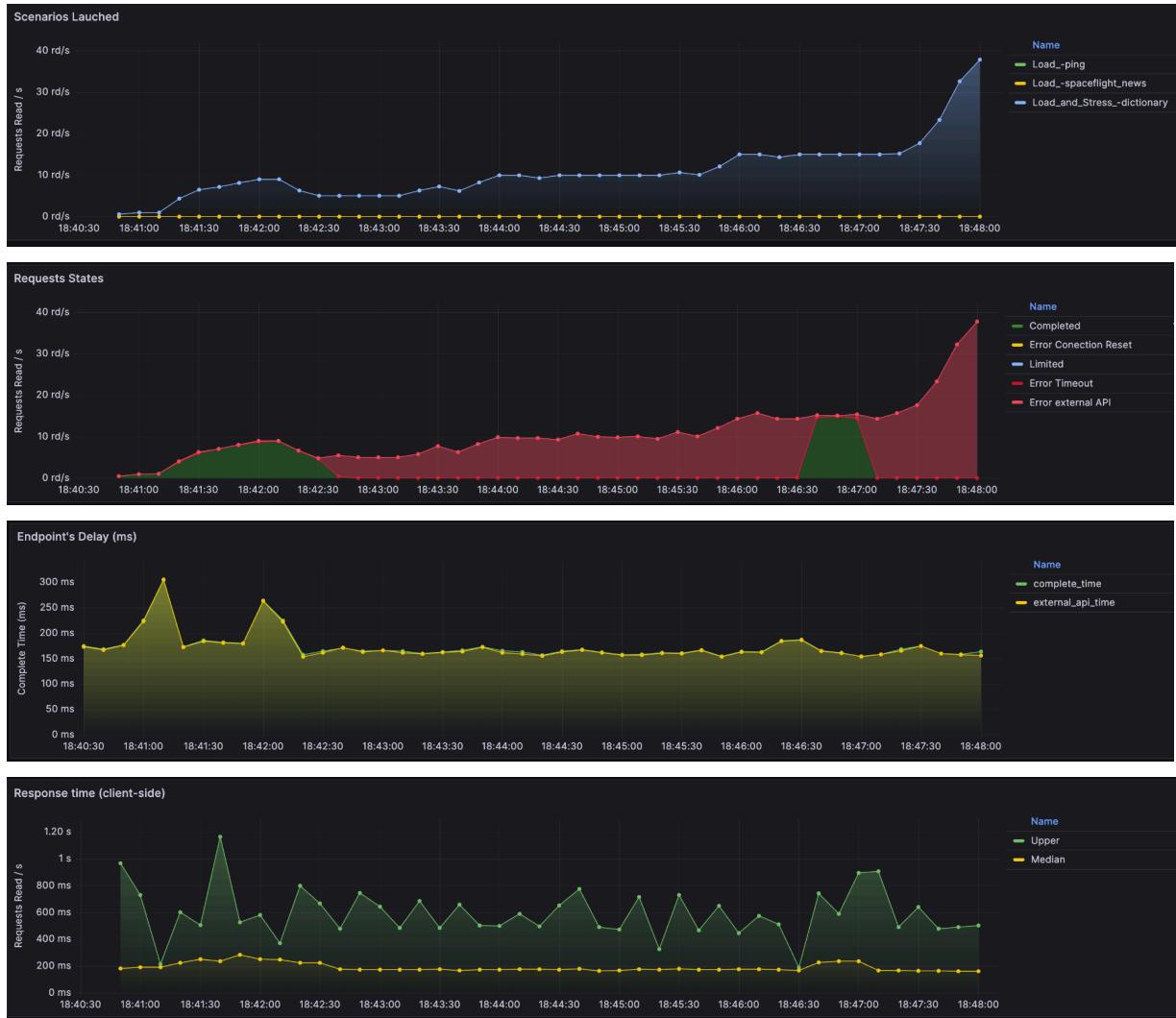
- Como era de esperar, el cuello de botella mejora sustancialmente con el caché, donde las palabras pedidas repetidas (almacenadas en

redis) se reutilizan reduciendo la cantidad de requests a la API externa.

- Se puede apreciar en el gráfico de las requests states que esta táctica permite el manejo con éxito de la gran mayoría de las requests realizadas sin disminuir la tasa de envíos de requests que se utiliza en nuestro escenario.
- Se puede ver que el delay del endpoint al recurrir a la API externa es entrecortado ya que no siempre necesita acceder a ella debido a que algunas palabras se encuentran en la caché. También se ve que inicialmente cuando no hay ninguna palabra en la caché, ambos tiempos coinciden ya que se van almacenando esas palabras en la misma.
- Se aprecia que la media del response time, a pesar de su upper (que se da al consultarse la API externa), se mantiene bajo debido a los hits en la caché
- En este caso, se prueba con 100 palabras, lo cual permite que en una sola window (aproximadamente 450 requests) se llene todo el cache y nunca se tengan Timeouts, pero de todas formas si se prueba con mayor cantidad de palabras, a la larga se terminará llenando la caché de todas formas y no habría demasiada diferencia en el mediano plazo con los resultados obtenidos.

# Replicación

Para implementar la replicación se utiliza la opción `replicas: 3` en el docker-compose y luego se agrega el contenedor en nginx.

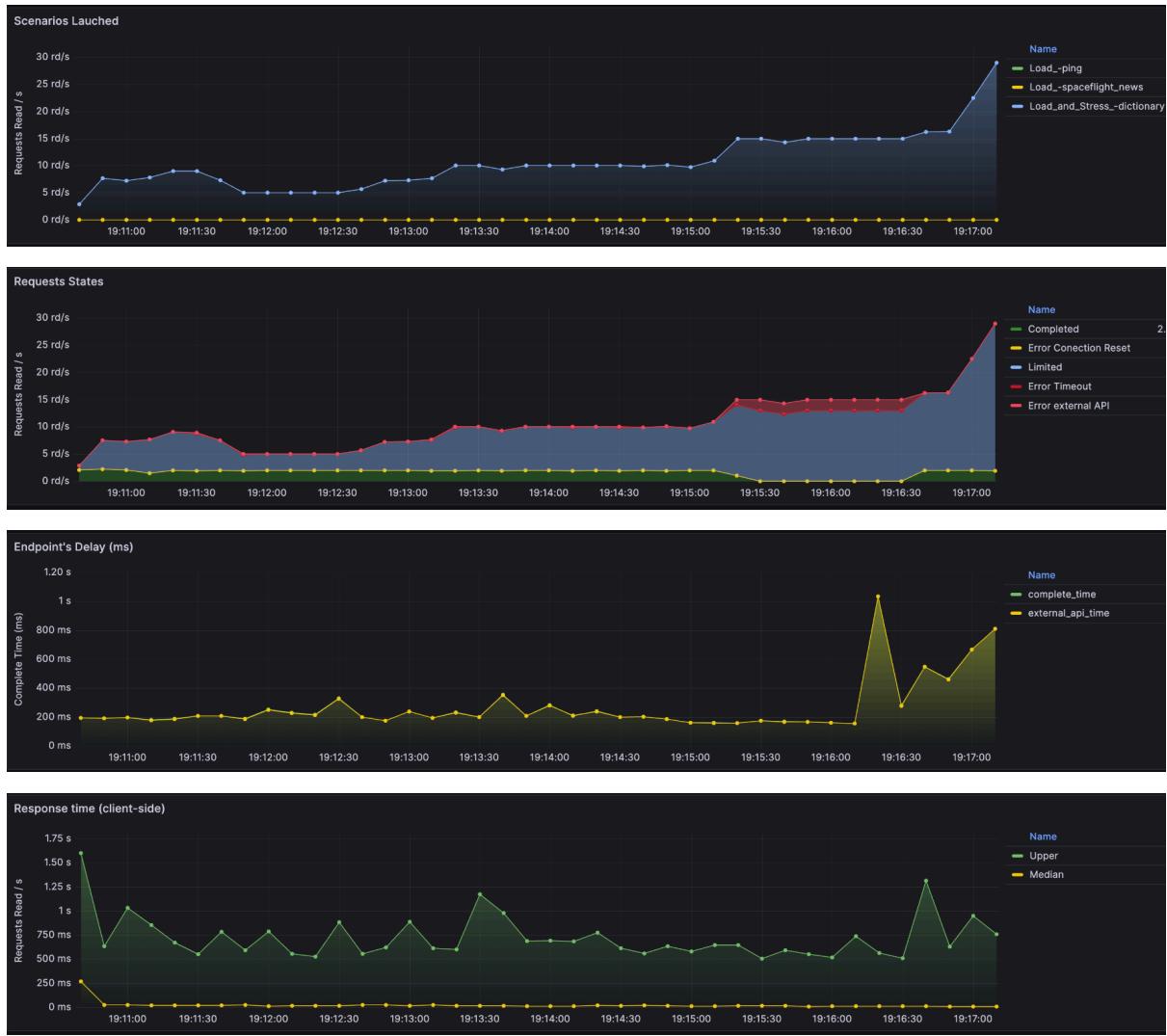


## Observaciones:

- Al contrario de lo hipotetizado en la sección de posibles mejoras del caso base, se visualiza que no hay mejoras en performance ni disponibilidad al replicar los servidores ya que la API externa toma a los requests según su IP y no por el servidor del que provienen.
- Complete time y external API time son similares para este caso ya que lo que demora es la API externa y no un procesamiento propio.
- Ni el response time ni el delay de los endpoints mejora debido a que la aplicación node no estaba siendo cuello de botella, por lo que replicarlo no resulta en una ventaja en performance.
- La forma de que replicar funcione para este caso es si los servidores estuvieran en redes distintas y las requests proveniesen de distintas IPs, de esta forma se podría recibir el triple de requests.

# Rate Limiting

Viendo que la API externa es el cuello de botella, y que soporta hasta 450 requests por cada 5 minutos, se decidió poner un límite de 20 requests por cada 10 segundos, lo que permitiría que el test no consuma la window entera en cuestión de segundos y aumentar la disponibilidad del endpoint.



## Observaciones:

- Se puede ver a simple vista en el gráfico de los requests states, que el uso de rate limiting aumenta en gran medida la disponibilidad en comparación con el caso base aunque esto se hace a costa de manejar picos de menor cantidad de requests ya que una parte es limitada por el rate limiter implementado.
- En este caso se utilizó rate limiting de 20 requests cada 10 segundos, pero si se usara menor cantidad de requests en ese tiempo se podría obtener disponibilidad aun mejor.

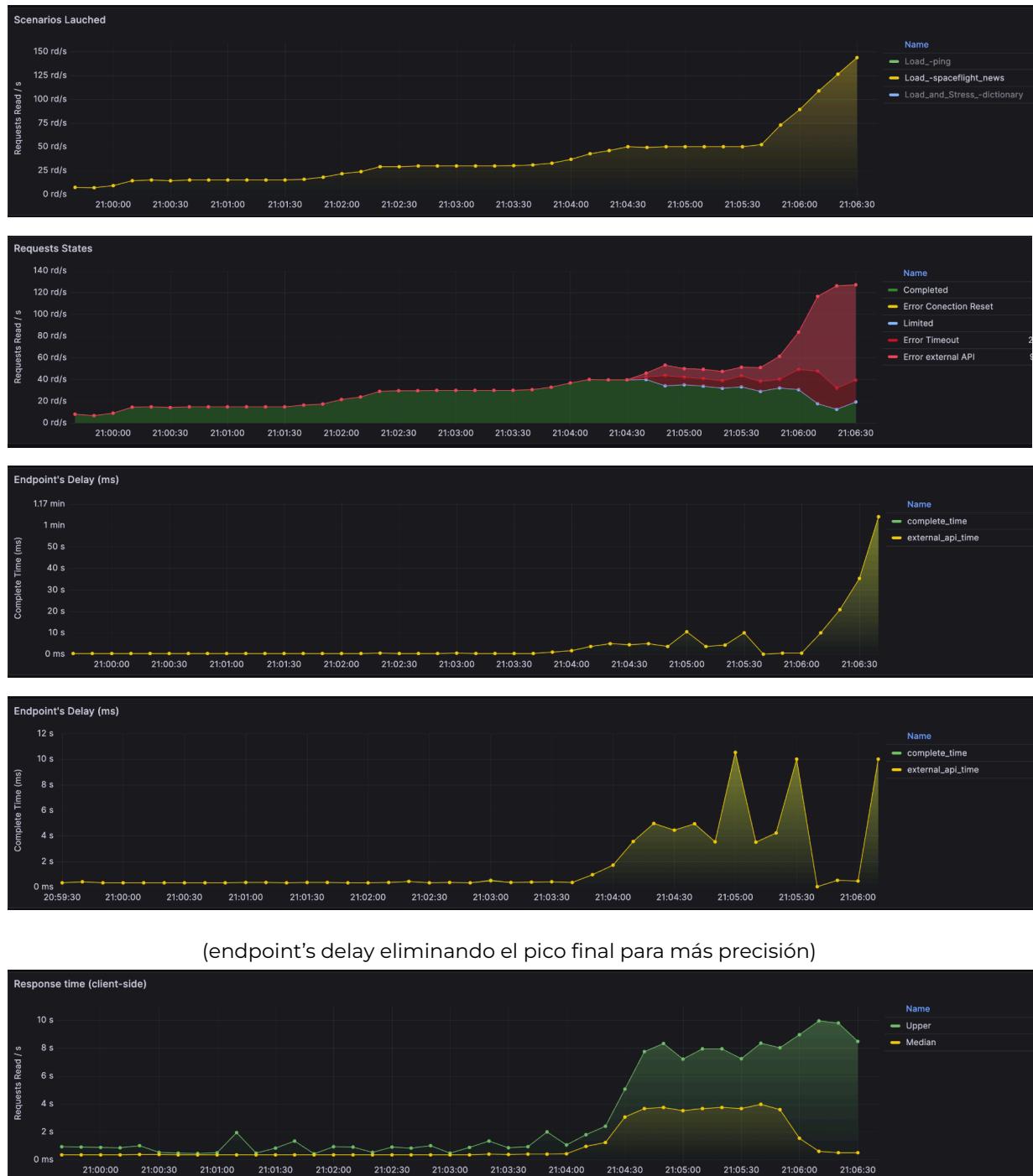
- Viendo el repositorio oficial de la API se ve que en ella también se usa un rate limiter, pero de 450 requests en 5 minutos ([link al código fuente](#)). Con el rate limiter implementado se estaría promediando las 600 en ese lapso, por ese motivo se tiene disponibilidad durante aproximadamente 4 minutos y al final se la pierde obteniendo así unos errores provenientes de la API externa.
- Se puede apreciar que el delay de la API no varía notablemente con respecto al del caso base, esta táctica no resulta en beneficios o detrimientos fuertes en este aspecto
- Mientras que el Upper en el response time del client-side se mantiene similar al caso base, ya que está conformado por las requests que no se ven afectadas por el rate-limiting, se ve que la media si baja drásticamente debido a la cantidad de timeouts del rate-limiting que ahorran el request a la API externa, resultando en que el promedio baje

## **Conclusión**

El uso del caché presentó una gran mejora en los tests en todas las métricas pero, como fue ya mencionado, esto es en parte debido a que no se usa un conjunto muy grande de palabras, por lo que agregar un rate limiting bastante generoso (que se encuentre por encima del límite de las 450 requests por cada 5 minutos que implementa la API externa) podría ser una combinación ideal para permitir que muchas requests se vayan cargando en la cache, pero evitando una sobrecarga demasiado grande de palabras distintas en un periodo corto de tiempo. Para esto se debería hacer un análisis estadístico en producción de cuántas palabras son repetidas y cuantas distintas para en base a eso poder estimar un límite equilibrado.

# Spaceflight News

## Caso Base



### Observaciones:

- Se puede ver que la cantidad de requests completadas con éxito en el gráfico de requests states ronda entre las 30 - 40 por segundo y luego a medida que se aumenta la tasa de requests enviadas, el endpoint colapsa.

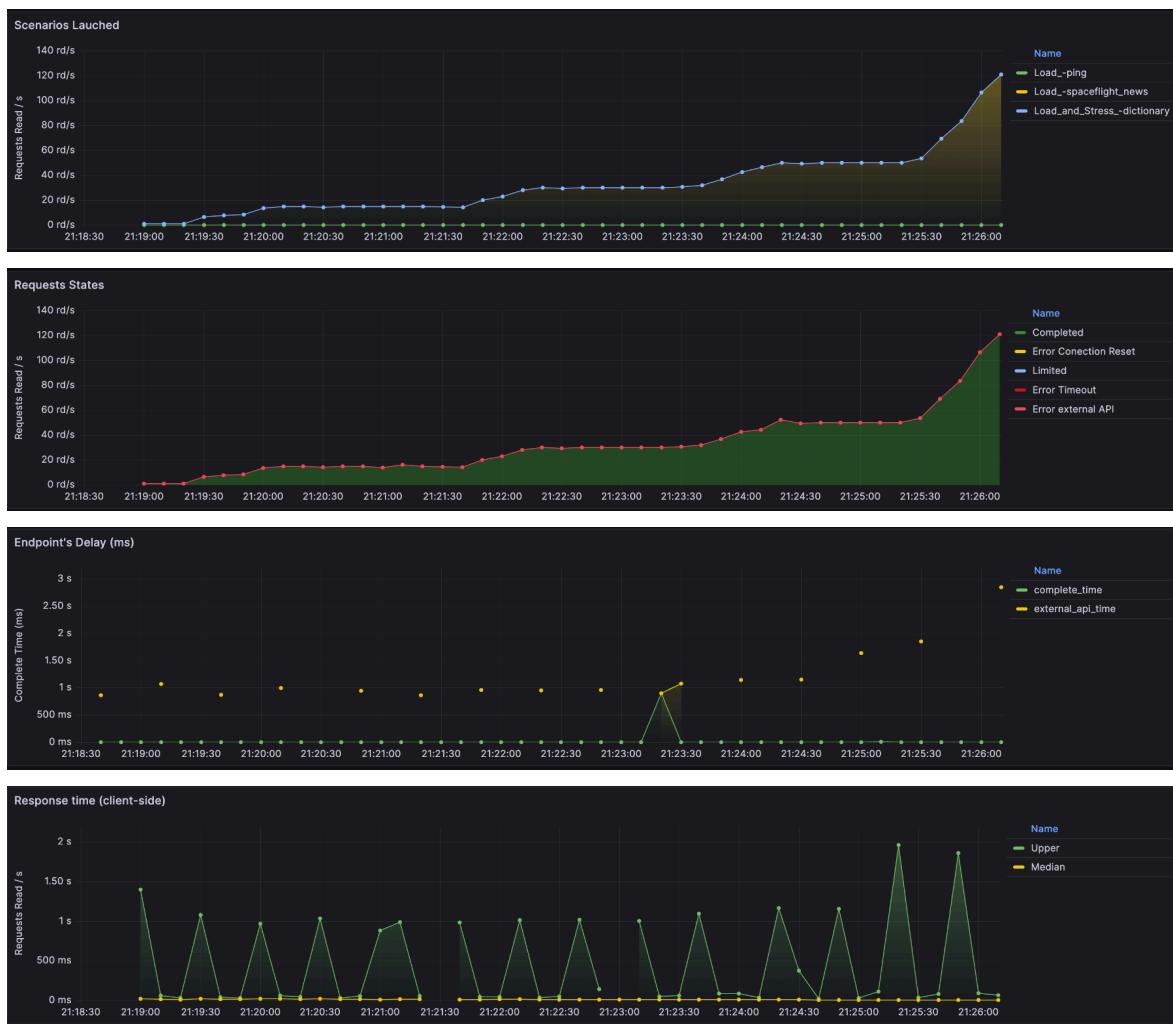
- Se puede notar que el response time aumenta tanto en upper como de media cuando empieza la etapa de ramp a 50 requests por segundo. También como en el ramp final donde ya el endpoint devuelve errores a la mayoría de las requests la media baja considerablemente.
- Viendo el conjunto de gráficos, se concluye que los problemas al pasar 50 requests/segundo pueden tener múltiples causas paralelamente. Por un lado se ve que la API externa deja de responder rápidamente haciendo que el response time a las requests enviadas aumente de manera exponencial, pero también se nota que se cuenta con Timeouts del cliente provenientes de la aplicación implementada, lo cual lleva a inferir que también se cuenta con un cuello de botella en el procesamiento de tantas requests (bloqueando el event loop) y por el procesamiento de la información.

### **Posibles mejoras:**

- Con caché se podría reducir la cantidad de requests a la API externa y además ahorrar el procesamiento de las respuestas del lado de la aplicación node implementada, aligerando dos posibles cuellos de botella.
- Con replicación se podría agregar un nivel de concurrencia al servidor de node, de esta manera alivianando el event loop y repartiendo las requests a procesar en múltiples servidores
- El rate limiting puede llegar a ser útil ya que como se puede apreciar en el gráfico, a partir de los 30-40 requests por segundo es donde comienzan los problemas, manteniendo los valores por debajo de eso podría evitar el colapso del servidor.

## Cache

Para spaceflight news se utiliza lazy population donde se guardan los últimos 5 títulos de noticias de la API externa en la cache por cierto tiempo. Esto aprovecha que no hay noticias de actividad espacial apareciendo constantemente y ante requests muy seguidas se utiliza la caché para responder rápidamente. Se determinó usar 30s antes de que expiren los títulos en la caché para poder realizar los escenarios de una duración adecuada y poder apreciar la diferencia en las mediciones, pero siendo que no suelen salir más de 3 noticias por día ni son noticias de carácter urgente, podría ser mucho mayor en producción.



### Observaciones:

- Podemos ver que con la caché se evitan un número considerable de consultas a la API externa lo que se traduce en cada request sea menos demandante permitiendo aumentar la cantidad de requests por segundo que puede manejar con éxito alcanzando las 150 req/s sin fallar ni picos en el response time promedio del client-side.

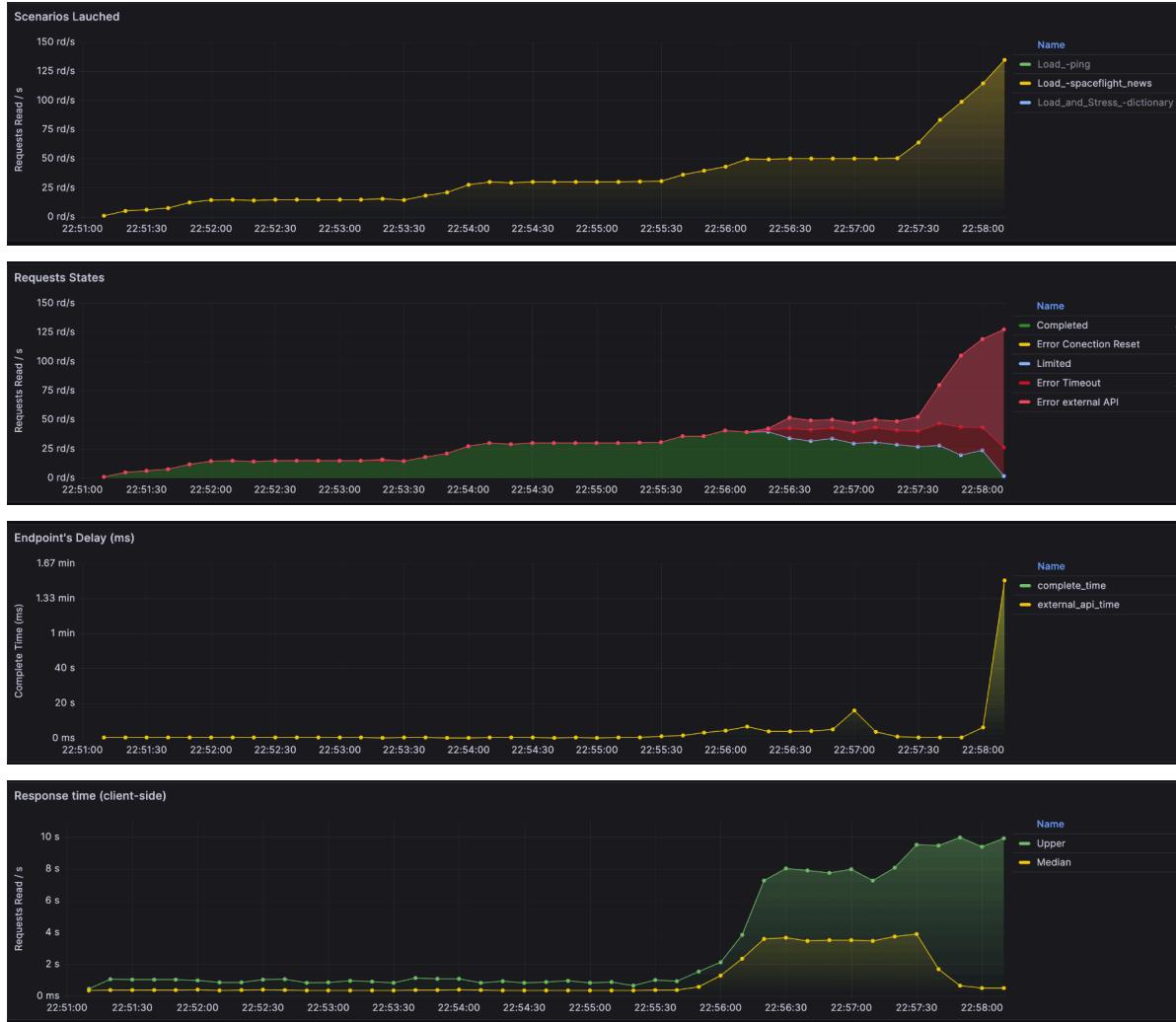
- No solo mejoró la cantidad de requests que se soportan, sino que también logra mejorar sustancialmente el response time promedio y upper con respecto al caso base, este último teniendo picos cada 30s debido a que cada este tiempo se vence el caché y al llegar la siguiente request se vuelve a consultar la API externa para obtener los últimos 5 títulos en la caché.
- A pesar de mejorar notablemente la disponibilidad y el promedio de endpoint's delay, se sigue notando una alza de este último valor al final del escenario a pesar de hacer únicamente una request cada 30s, por lo que no debería saturarse la API externa. Esto indica que queda otro cuello de botella además de la API externa, que puede ser tanto los recursos del servidor node como el propio event loop de javascript que no puede resolver las peticiones concurrentemente.

#### **Posibles mejoras:**

- Si se realiza replicación podría ayudar a bajar el endpoint's delay a medida que aumentan los escenarios lanzados gracias a tener más event loops trabajando en paralelo.
- Escalar verticalmente también podría ayudar a resolver estas requests en menor tiempo, mejorando el endpoint's delay.

# Replicación

Para implementar la replicación se utiliza la opción `replicas: 3` en el docker-compose y luego se agrega el contenedor en nginx.



## Observaciones:

- Se puede ver que en esta táctica el endpoint delay es similar al caso base y por lo tanto no es efectiva al mejorar los resultados de esa medición. Lo mismo se puede ver en el response time y el resto de métricas, por lo que esta táctica no mejora las mediciones en general y trae mayor complejidad al sistema ya que en vez de tener un único servidor de la aplicación de node ahora se tienen 3 y se está haciendo que nginx tenga que actuar como load balancer.
- Contrario a una de las hipótesis hechas en el caso de caché al ver que disminuyendo las requests a la API externa, aumentar la cantidad de event loops, y por ende tener más hilos principales manejando los callbacks de las promises, no se logró mejorar por este lado contra el caso base, por lo cuál se infiere que el cuello de botella del node

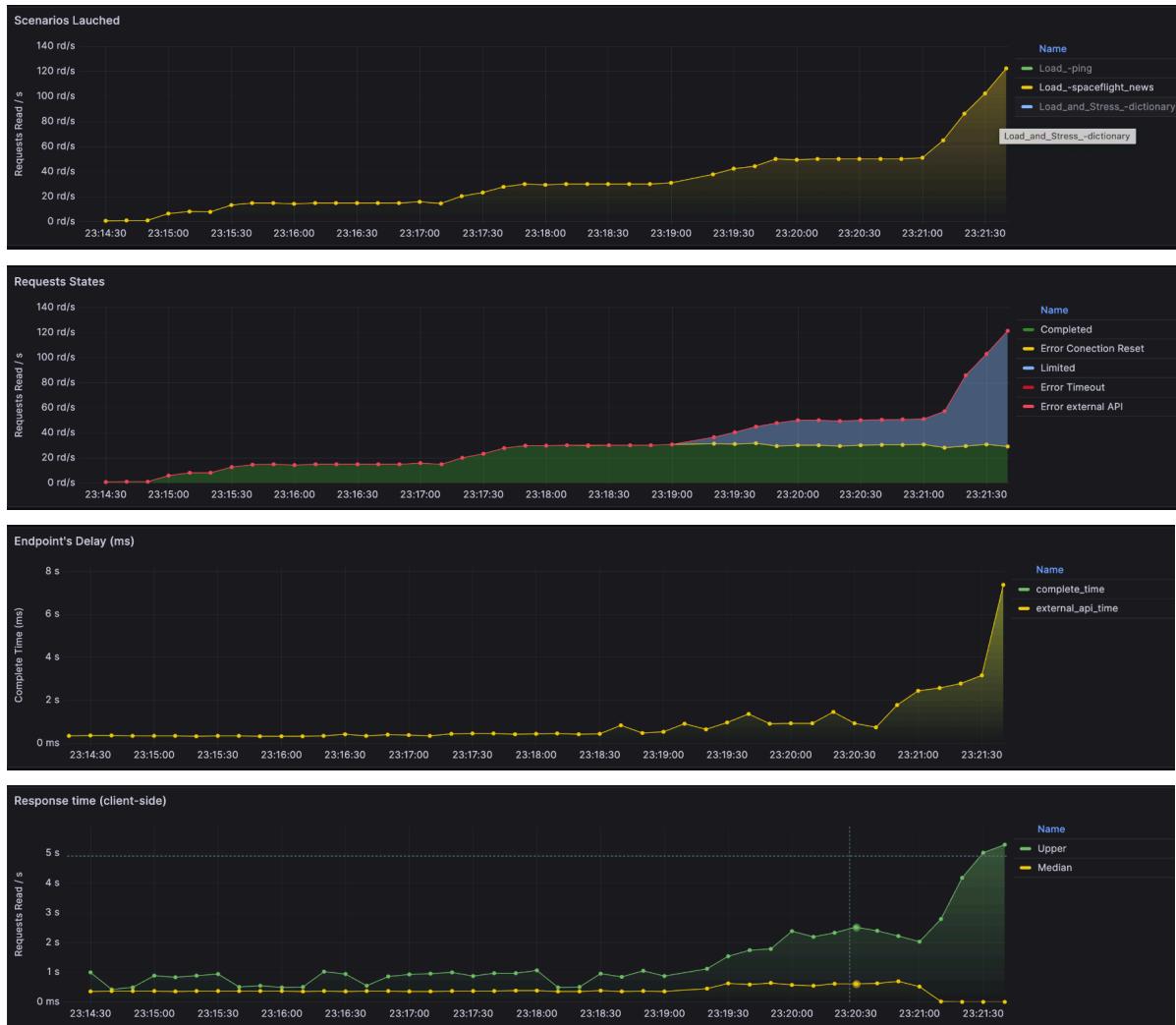
viene por los recursos (que aunque se repliquen los servidores, siguen siendo los mismos recursos pero divididos entre varias instancias).

**Posibles mejoras:**

- Ya habiendo descartado todas las opciones, la última restante es escalar verticalmente, para que de esta manera el servidor pueda manejar un volumen más grande de requests intentando mitigar el cuello de botella por recursos.
- Aunque de manera aislada no se haya detectado una mejora, quizás combinar replicación con caché podría dar mejores resultados al disminuir la exigencia de procesamiento, y por ende de recursos, pero manteniendo múltiples hilos para distribuir los requests.

# Rate Limiting

Observando que las requests fallidas comienzan a partir de los 30-40 requests por segundo, se decide realizar un rate limiter de 30 requests por segundo, lo que se espera que mejore los tiempos de respuesta al limitar la cantidad de requests realizados.



## Observaciones:

- Como se puede ver, con rate-limiting tampoco se obtienen grandes ventajas contra el caso base salvo que las respuestas sean más controladas eliminando tanto los timeouts como los errores provenientes de la API externa.
- El aumento del upper response time (no limitado) y del delay de la API externa, confirma aún más que el cuello de botella también viene de los recursos del servidor node ya que, al igual que en la táctica de caché, reducir (o capar) consultas al endpoint no evita una caída de rendimiento general del endpoint.

- A pesar que la media del response time sea efectivamente más baja, esto no implica una mejora en el sistema, lo que está sucediendo es que el servidor responde con estado 429 en lugar de lo que se pidió. Esto implica que en definitiva no se completan más rápido, en cambio se responde un mensaje de error más rápido.

### **Posibles mejoras:**

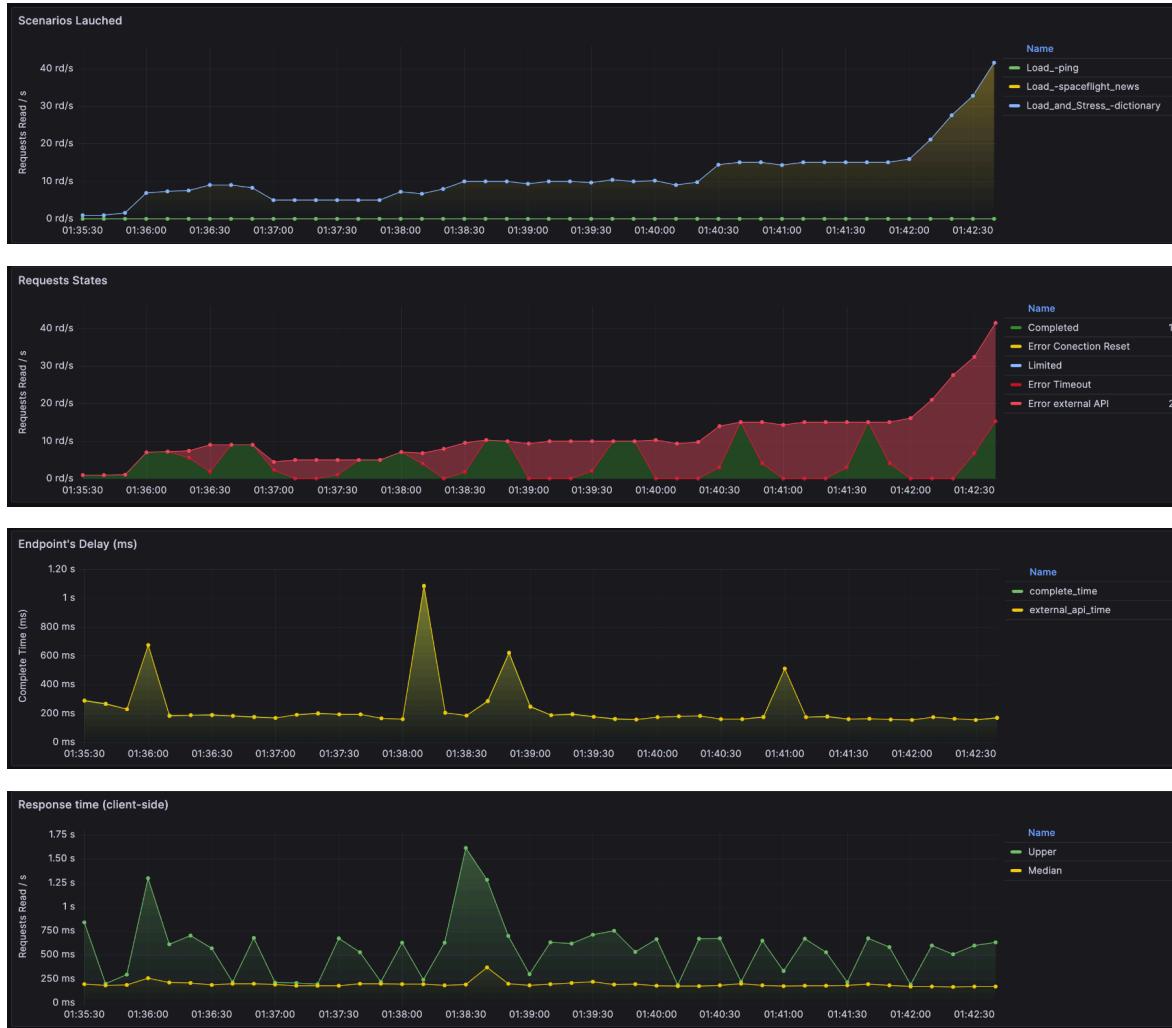
- Una posible mejora al sistema puede llegar a ser implementar el rate limiter vía NGINX, lo cual probablemente sea más performante que realizarlo en JavaScript (con la biblioteca de rate-limiter de express). Esta última teoría se basa en que es mucho más costoso realizar este tipo de operaciones en una librería de JavaScript en comparación a la implementación de NGINX escrita principalmente en C.

## **Conclusión**

A diferencia del diccionario, acá no se encuentra ningún rate limiting agresivo de parte de la API externa, por lo que los cuellos de botella fueron distintos y consisten en limitaciones de recursos tanto en la API externa como del TP. Utilizar rate limiting fue útil para erradicar las limitaciones provenientes de la API, pero para el TP no fue suficiente e igualmente hubo problemas de performance. Por otro lado, el caché fue la mejor alternativa permitiendo que mejore tanto la exigencia a la API externa como al TP además de aumentar la cantidad de respuestas exitosas, aunque se puede alcanzar el límite de sus recursos de todas formas. Por otro lado, no se ven mejoras con replicación, porque aunque debería alivianar el trabajo de cada hilo del node del TP, sigue el cuello de botella de la API externa y los recursos que quedan divididos en cada réplica del servidor. La solución ideal sería un uso agresivo del caché, acompañado por un rate limiting en el NGINX tal que evite que saturen los recursos del node, y una replicación que permite tener múltiples hilos además de facilitar el escalado horizontal con nuevo hardware. De todas maneras, se alcanzó a triplicar la cantidad de requests del Dictionary por lo que no hay realmente una problema de performance, sino que no está capado por servicios de terceros y la limitante ocurre en el servicio del TP.

# Random quote

## Caso Base



### Observaciones:

- Se puede ver que el endpoint delay de la API externa es similar al tiempo de eso sumado al del procesamiento propio del servidor ya que el tiempo de procesamiento propio es mínimo en comparación al de la API externa por lo que la diferencia no es apreciable.
- Se nota al consultar la API con muchos requests que la disponibilidad es bastante baja con múltiples caídas apreciables en el gráfico de requests states
- Se puede notar que la media del response time es bastante uniforme sin demasiados picos aun en el ramp abrupto final característico del escenario de pruebas utilizado en este informe, esto muestra que no hay cuello de botella en el servicio

implementado, sino que está dado por el rate limiting de la API externa.

- La misma observación se obtiene al ver que el delay del endpoint también se mantiene estable aun aumentando abruptamente la cantidad de requests recibidas.

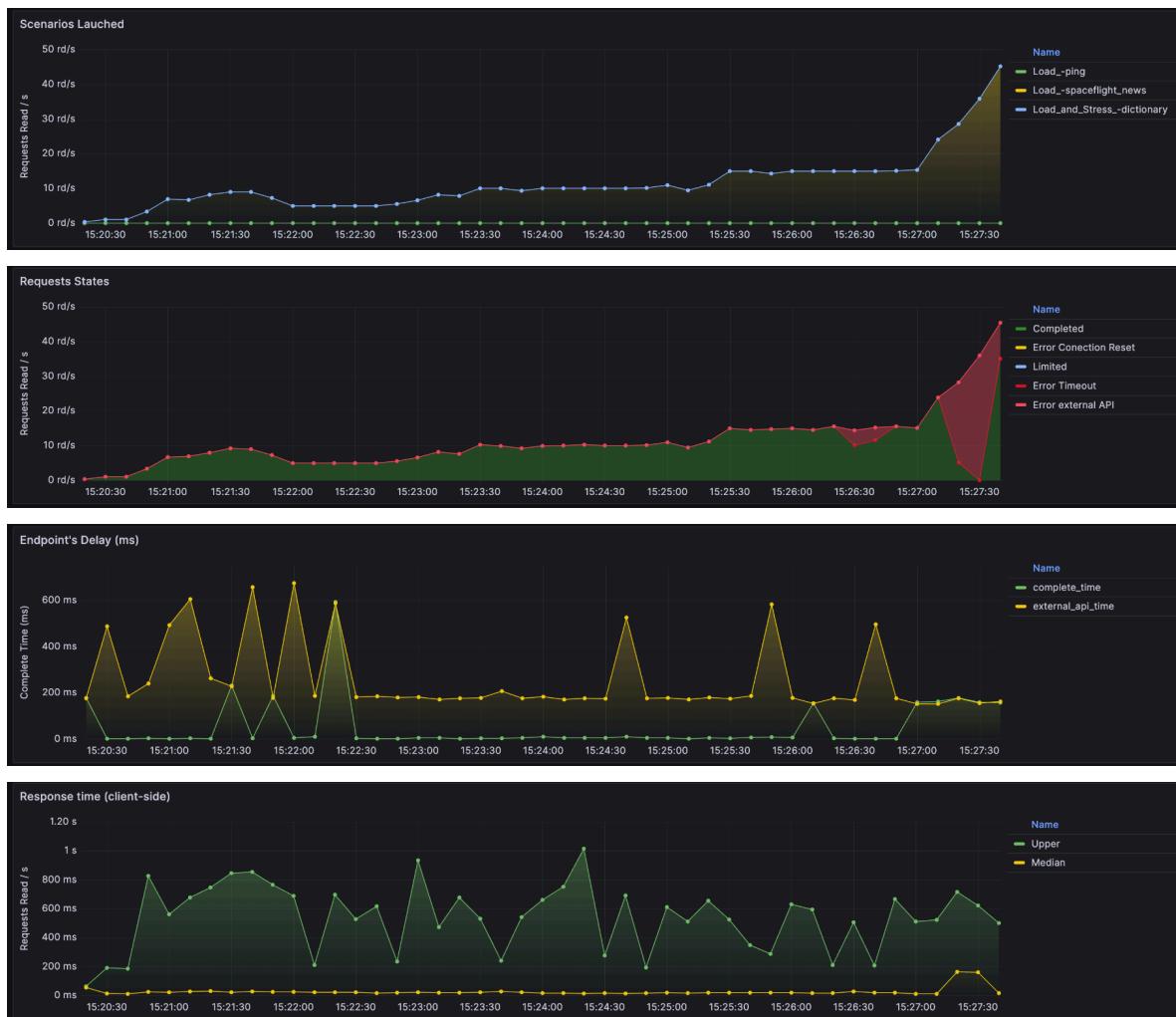
### **Posibles mejoras:**

Estas son similares a las del endpoint Dictionary puesto que ambas cuentan con un rate limiting propio dando lugar a que los cuellos de botella sean similares.

- Cache podría servir para ahorrar requests a la API externa si se implementa con active population, permitiendo de antemano ya tener una quote preparada para el siguiente usuario que haga el request lo que reduciría el response time para esa request. Además, si se utiliza la caché de batches se reducen también la cantidad de consultas a la API externa
- Limitar la cantidad de requests por usuario podría servir al igual que para Dictionary, para tener mayor disponibilidad a diferentes usuarios y evitar tener tantas caídas.
- La replicación sería útil si el rate limiting es por cada servidor, de esta forma se triplicaría la cantidad de requests posibles en la window, aunque si sucede como para Dictionary que se limita por IP y no por servidor esta táctica no resultara demasiado útil.

## Cache

Para las quotes se utiliza active population de a batches. Como se sabe que las quotes siempre deben ser distintas de la anterior, se guarda en la caché un batch de cierta cantidad de quotes distintos y a medida que el usuario solicita una se le responde con una de las guardadas en la caché que posteriormente es eliminada. Cuando el usuario solicita la última guardada del batch, la aplicación fetchea un nuevo batch de la API externa luego de haber respondido. Trabajar con batches permite no acceder tantas veces a la API externa. Para esta prueba se hicieron batches de 10 quotes, pero podría incrementarse para poder soportar aún más consultas.



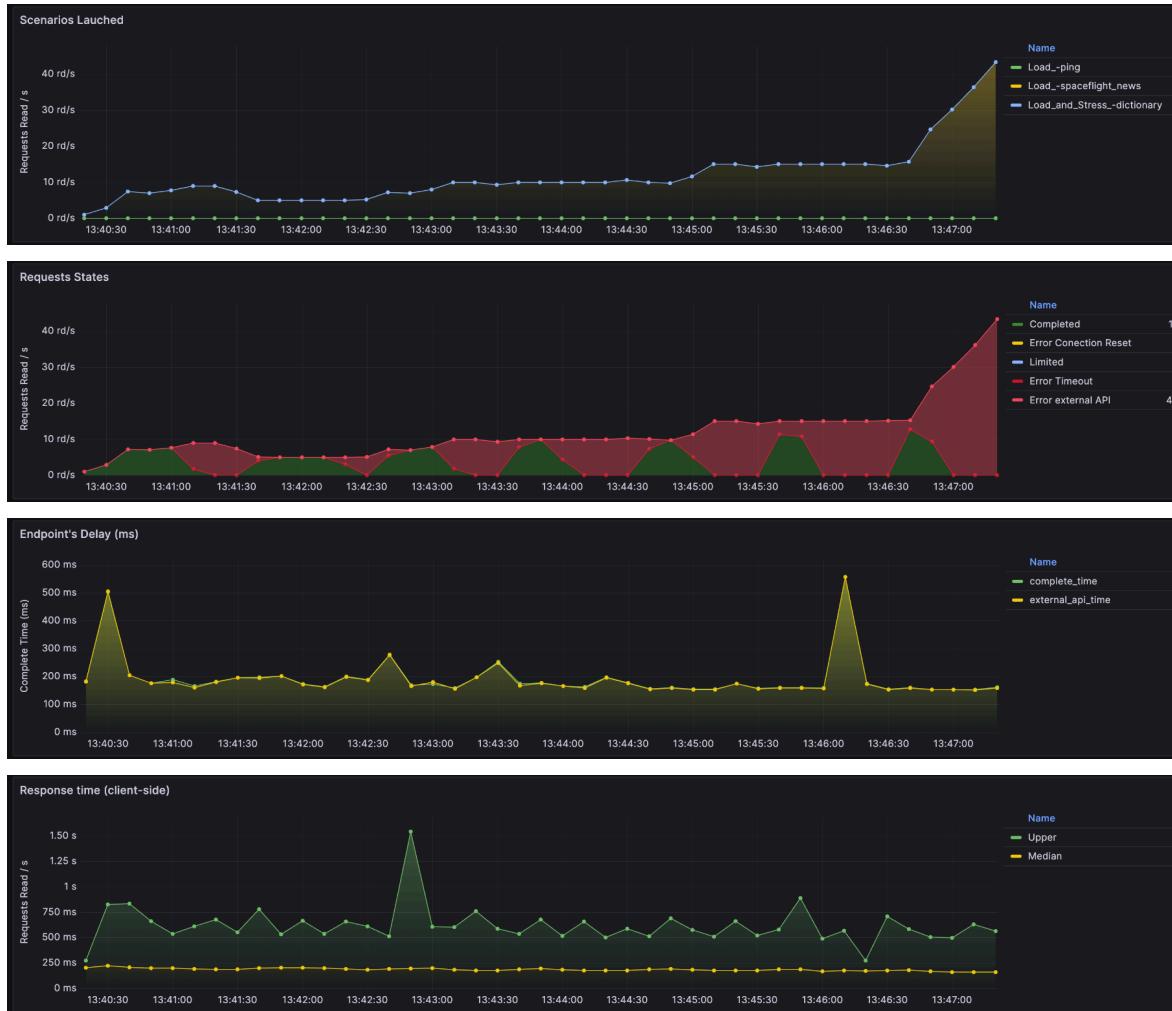
## Observaciones:

- Se puede ver en el gráfico de Response Time como la utilización de una caché reduce abruptamente el response time medio, manteniendo los uppers donde se refleja el caso de que la caché se vació y se debe volver a consultar a la API externa por un nuevo batch de quotes.

- Se aprecia también como el uso de caché de batches aumenta la disponibilidad ya que al reducir la cantidad de consultas a la API externa, requiere un mayor esfuerzo llegar al rate limiting propio de esta. Sin embargo, en el pico final la cantidad de requests es tan alta que se pierde disponibilidad, una posible mejora en este aspecto es hacer los batches más grandes a costa de ocupar mayor cantidad de memoria pero reduciendo aún más las consultas a la API externa.
- Se nota viendo el gráfico de endpoint's delay como las consultas a la API externa quedan corriendo de fondo para obtener nuevos batches para nuestro caché, mientras que el complete time es sumamente bajo debido a que se toman quotes dentro del caché, devolviendo la response al instante y disminuyendo la latencia. También se ve que en ciertos casos, cuando no hay ningún quote en la caché, ambos tiempos coinciden ya que se van almacenando esos quotes en la misma.

# Replicación

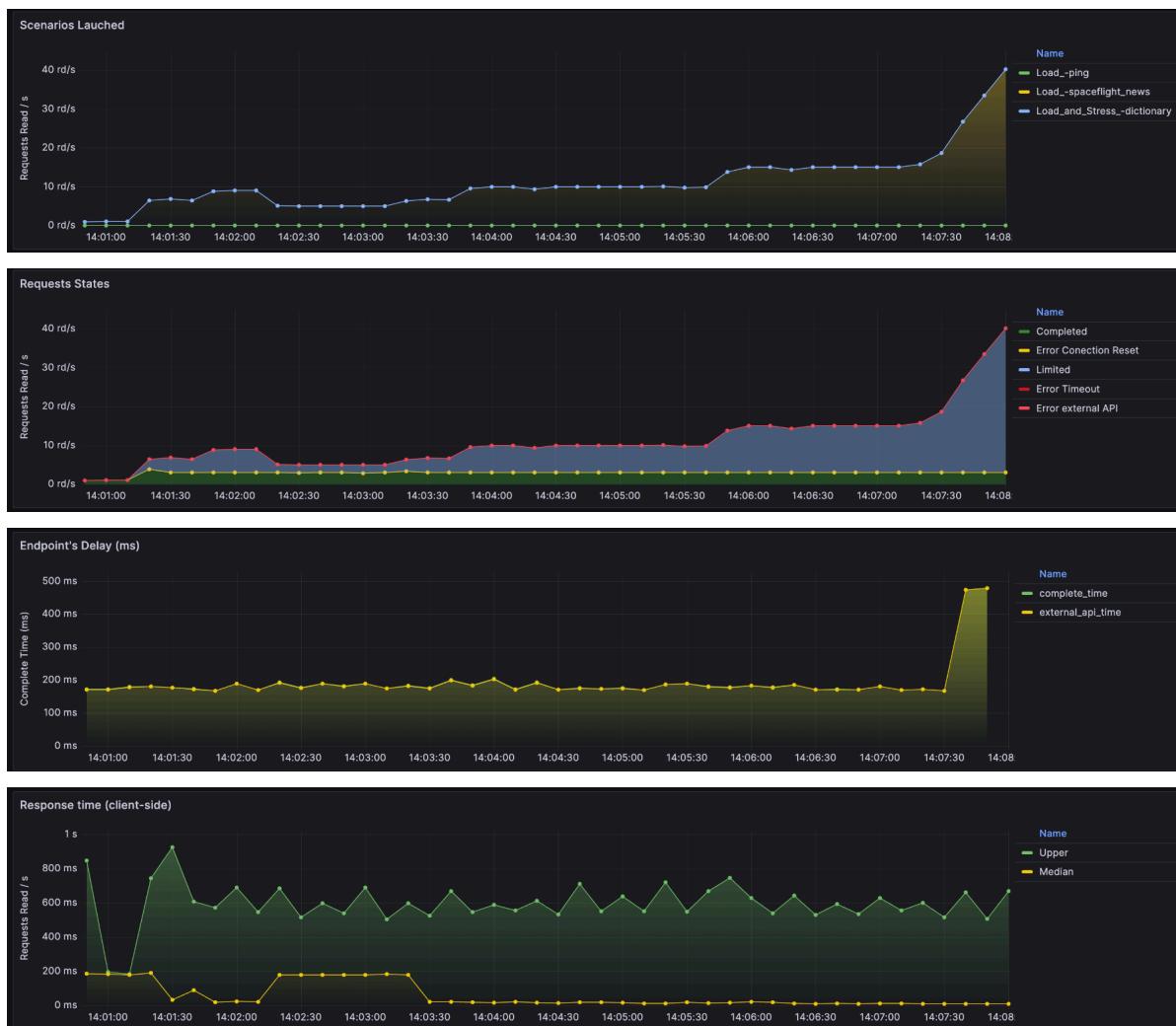
Para implementar la replicación se utiliza la opción `replicas: 3` en el docker-compose y luego se agrega el contenedor en nginx.



## Observaciones:

- Se puede observar que en este caso el complete time es muy similar al external API time ya que lo que demora es la API externa, prácticamente no hay procesamiento dentro del servidor.
- En este caso también vuelve a suceder el mismo inconveniente que con el endpoint de dictionary, donde la API externa posee un rate limiter que limita la cantidad de requests. La replicación sería de utilidad en el caso donde los servidores estuvieran en redes distintas y las requests provengan de IPs variadas, tal como se menciona anteriormente en el endpoint de dictionary. Como resultado la disponibilidad aun usando esta táctica se mantiene baja y no resulta beneficiosa en este aspecto.

# Rate Limiting



## Observaciones:

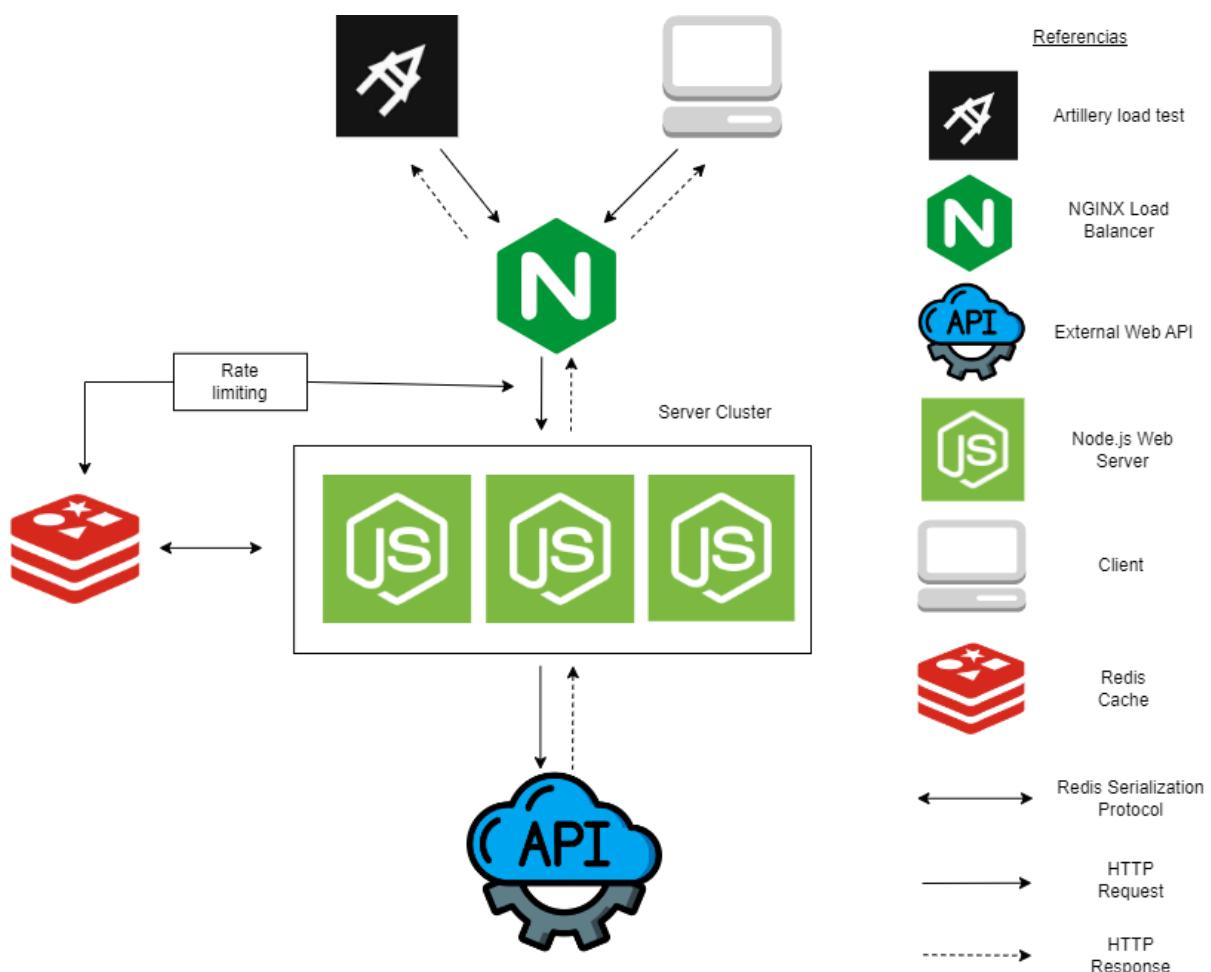
- Se puede apreciar en los gráficos que al limitar la cantidad máxima por cliente se evita el rate limiting por parte de la API externa, mejorando a niveles generales la disponibilidad a diferentes clientes y evitando caídas del endpoint.
- Se eligió el monto de 30 requests cada 10 segundos ya que como se comenta en [la documentación](#) la API limita a 180 requests por minuto, es decir, 3 requests por segundo. De esta manera se pueden maximizar la cantidad de requests a la API externa antes de ser limitados por la misma.

## Conclusión

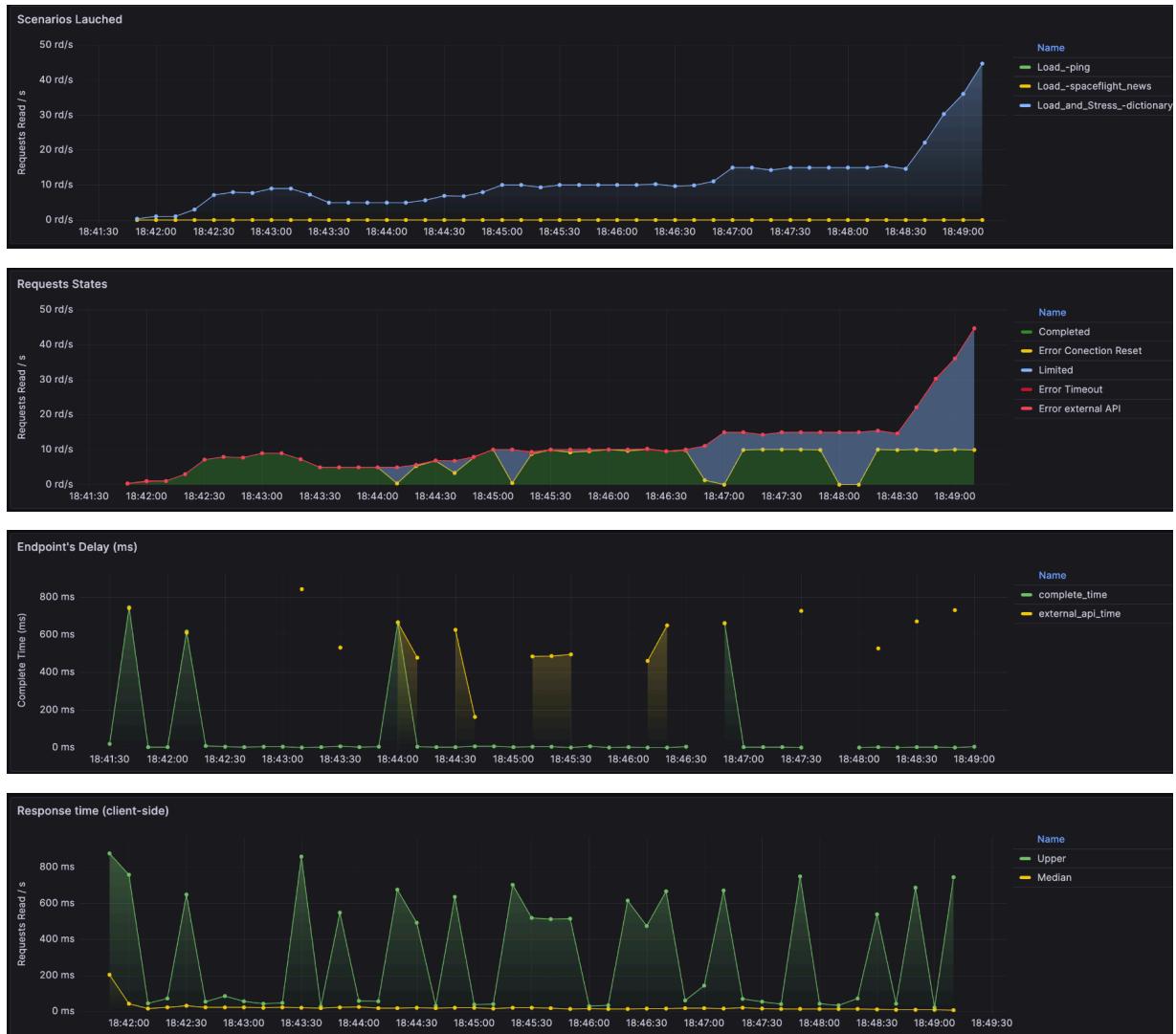
En resumen, este endpoint tiene condiciones similares al Dictionary debido al rate limiting propio que ambos implementan, por lo que el principal cuello de botella es el mismo, que es la API externa, y no el servidor node ni los recursos que tiene asignados. Por lo tanto, las tácticas que más benefician este endpoint son las mismas, de la misma manera que las que menos aportan también son las mismas. Se concluye entonces que una implementación ideal posible sería una combinación de cache con rate limiting. Esta cache idealmente funciona con active population usando los batches más grandes posibles que permita almacenar la memoria del sistema y un rate limit bastante superior al de la API externa (180 requests por minuto) pero que previene sobrecargas demasiado fuertes en el TP, manteniendo así disponibilidad del endpoint.

## Táctica “Ideal”

Como se fue observando entre los distintos servicios y tácticas, generalmente una combinación de tácticas sería lo ideal para mejorar el servicio. Por esto, se decidió agregar una táctica final combinando replicación en 3 instancias, cache con redis (de la misma forma en la que se implementó en cada endpoint) y rate limiting con express-rate-limit, conectado a rate-limit-redis de forma que todas las instancias usen el redis para almacenar los rates de cada IP, pero en este caso aumentando el límite de cada servicio a valores que maximicen el de cada uno teniendo en cuenta sus response states en caché.



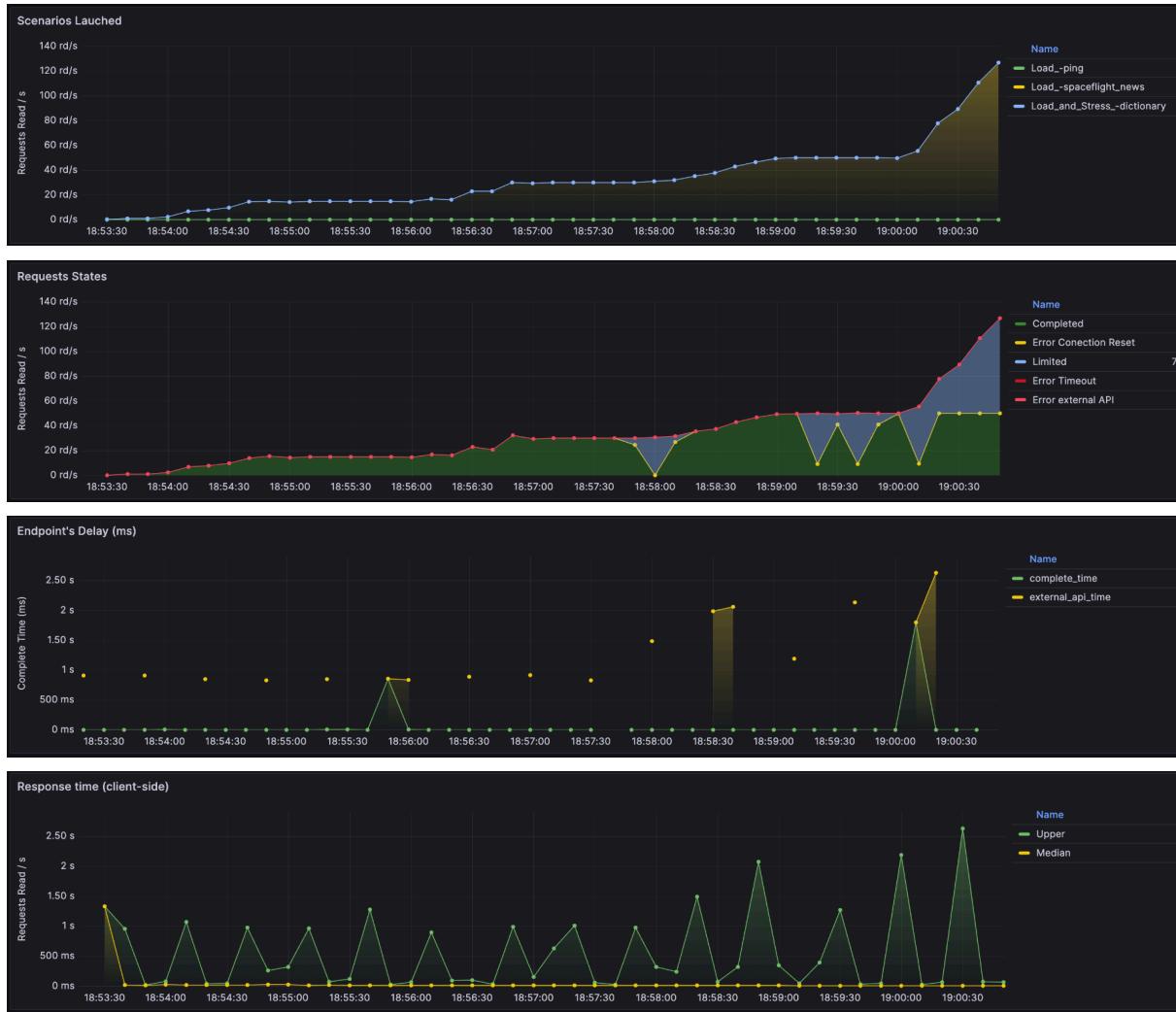
# Dictionary



## Observaciones:

- Aunque a simple vista podría parecer que el resultado parece peor que la táctica de caché, en este caso se hicieron las pruebas con 500 palabras, por lo que tener el rate limiting ayuda a evitar la observación que se había hecho en esa táctica donde se menciona que mayor variedad de palabras podría llevar a timeouts de la API externa, esto con el rate limiting del lado del servicio implementado se mejora.
- No se ve una mejora en performance proveniente de la replicación, lo que coincide con la conclusión de que no había cuello de botella en el servicio implementado.
- Esta versión ofrecerá una mejor disponibilidad en caso de que aumente todavía más la cantidad de consultas que cualquiera de las anteriores, haciéndola la más robusta.

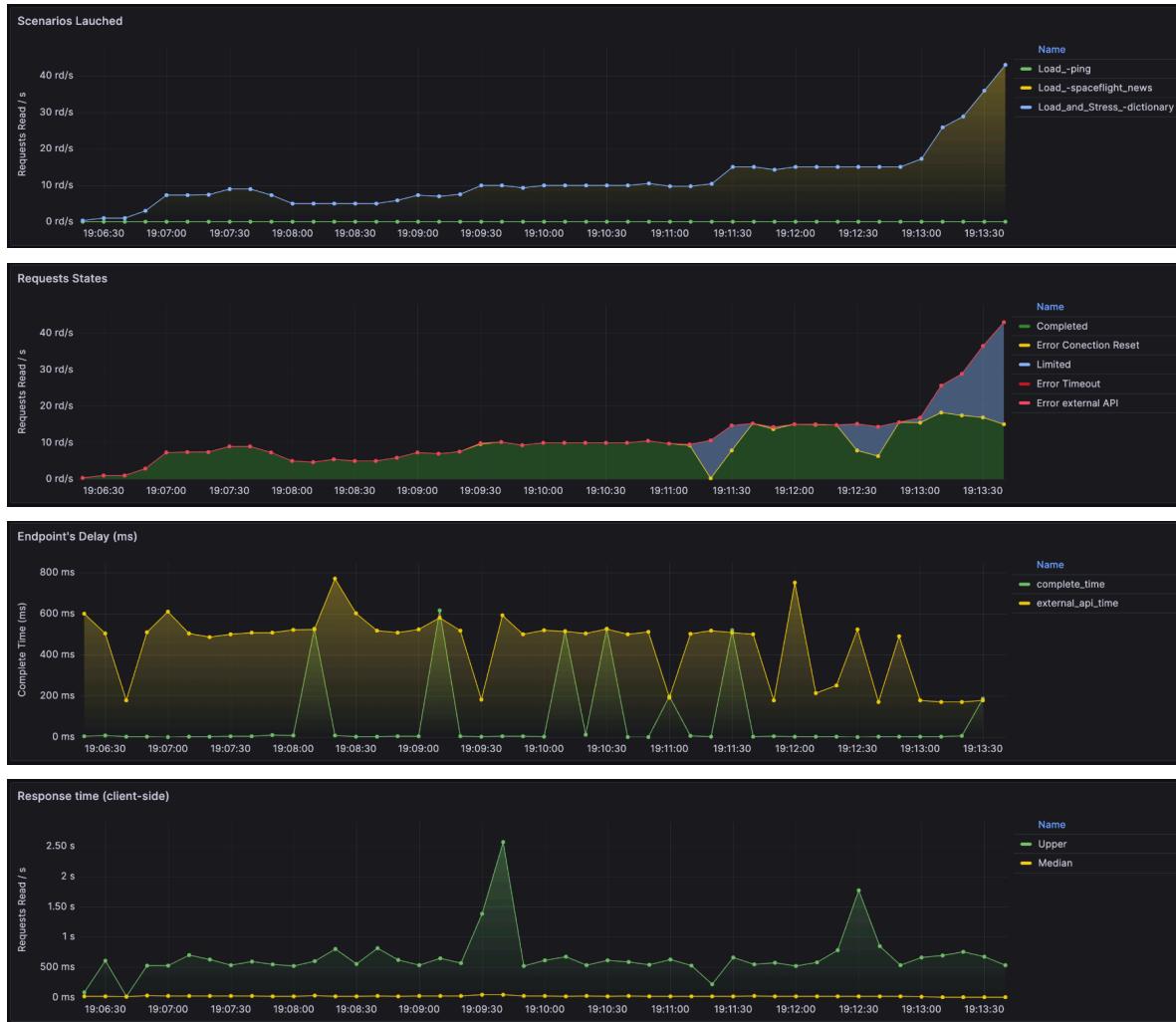
# Spaceflight News



## Observaciones:

- Se puede ver como esta combinación mantiene las ventajas del uso de la cache para este endpoint que da mayor throughput de requests y simultáneamente el rate limit junto a la replicación permite que delay de procesar la API externa no sea demasiado alto ya que aliviana los event loops.
- Para el rate limit de este endpoint el external API time llegaba a valores de 7 segundos mientras que con esta táctica combinada se mantuvo relativamente bajo, no llegando a pasar de los 3
- Gracias al uso de cache la response time promedio se mantuvo relativamente baja y los picos en el upper response time no fueron tan altos como con rate limit resultando en una implementación mejor que con las técnicas por separado.
- Para este servicio, fueron claves las 3 tácticas, coincidiendo con lo concluido en su conclusión donde se vio que ninguna era del todo efectiva de manera aislada.

# Random Quote



## Observaciones:

- Comparando con caché únicamente la disponibilidad aumento ya que a la hora de tener una alta demanda los requests se distribuyen mejor a lo largo del tiempo, reduciendo los momentos donde servicio implementado es inaccesible para todos los posibles usuarios.
- Combinando las 3 tácticas se aprovecha al máximo la velocidad que otorga la caché, pero limitando el uso excesivo del servicio implementado.
- También es apreciable cierta mejora en el upper response time comparado a las tácticas anteriores, esto se puede atribuir a la replicación, la cual anteriormente no presenta mejora alguna, pero al reducir cierto ruido con caché y rate limiting, la misma demuestra ser más útil.

## Conclusiones Generales

Lo que se puede observar entre todas las tácticas y todos los servicios, es que no existe una táctica ideal que solucione todo, sino que hay que hacer una combinación de ellas para cada caso.

Respecto al caché, esta táctica es la que más beneficios trajo aumentando la performance, disponibilidad y aprovechando mejor los límites que nos imponen las APIs externas. Por otro lado, no se puede abusar del caché ya que no siempre el contenido es fácilmente cacheable, o bien porque este expira (como los títulos de la actividad espacial) o necesita altas tasas de regeneración (ej. los quotes) o bien por variedad de consultas (ej. el diccionario). De todas formas, permite bajar mucho la exigencia tanto del servicio implementado como de APIs externas.

Respecto al Rate Limiting, esta táctica fue de gran utilidad para evitar que se sobrecarguen los servicios y, en casos de APIs externas con su propio rate limiting, ayudó a esparcir el cupo a lo largo del tiempo evitando picos que consuman toda la window dejando los endpoints con baja disponibilidad. A diferencia del cache, es más una manera de mitigar la sobrecarga que de optimizar el servicio implementado ya que aunque el tipo de response mejora porque se deja de tener errores y se puede responder requests con éxito durante un mayor porcentaje de tiempo, estas no aumentan respecto a no tener rate limiting, sino que se distribuyen mejor. En base a la experiencia al utilizar estas APIs, se concluye en que todo endpoint externo debería implementarlas justamente para evitar que se saturen los recursos causando un aumento de los costos, o una caída del sistema.

Una gran estrategia que se puede concluir con estas dos primeras tácticas, es que una combinación de cache y rate limiting puede ser muy útil, usando el cache para minimizar consultas a la API externa y los recursos del TP, junto a un rate limiting que evite que se sobrepase las capacidades conseguidas con el cache.

Respecto a la replicación, fue la que menos beneficios trajo pero se concluye que esto es debido al tipo de endpoints que tiene el TP donde la limitación está dada más por las APIs externas o la cantidad de recursos que por la cantidad de hilos ocupados haciendo procesamientos demasiado exigentes en el main thread. Quizás un endpoint que calcule fibonacci para un número recibido por query params donde los tests

hicieran pruebas random hubiera podido representar mejor la distribución de recursos entre múltiples instancias sin aumentar recursos ya que cuando una request fuera con un número grande, el node que lo responda iba a bloquearse y el nginx configurado con least connections para las réplicas hubiera repartido las siguientes a los otros nodes menos exigidos. Para los casos del TP, se vio la siguiente analogía a la hora de utilizar la táctica de replicación: una ducha donde sale mala lluvia, quizás cambiar la flor de la ducha hubiera permitido distribuir mejor los chorros y que mejore con la misma presión de agua, pero al probar replicar vimos que simplemente faltaba mejorar el termotanque (recursos).

A lo largo del TP se analizó como distintas tácticas afectan a distintos aspectos de atributos de calidad, y para medir los impactos de estas, se utilizaron distintas métricas que permitían visualizar de múltiples formas estos impactos de las táticas sobre las métricas. A partir de todo esto, viendo como las tácticas impactan positivamente en ciertos atributos y negativamente en otros, se concluye que no hay una “silver bullet” que permite solucionar o mejorar todos los sistemas, sino que se debe analizar caso a caso y saber priorizar para cada sistema que atributos de calidad son los más importantes y en base a ello utilizar la combinación de tácticas que más permiten mejorar esos atributos de calidad prioritarios siempre teniendo en cuenta las características del sistema.